Technical University of Crete

School of Electrical and Computer Engineering

Machine Learning in the "Settlers of Catan" Strategic Board Game

---

Diploma Thesis

Diamantis Rafail Papadam

COMMITTEE:

- Professor **Georgios Chalkiadakis** (Supervisor)
- Professor **Michail G. Lagoudakis**
- Professor **Thrasyvoulos Spyropoulos**

Chania, March 2024

Πολυτεχνείο Κρήτης

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Μηχανική Μάθηση στο Στρατηγικό Παίγνιο «Άποικοι του Κατάν»

Διπλωματική Εργασία

Διαμαντής Ραφαήλ Παπαδάμ

ΕΠΙΤΡΟΠΗ:

- Καθηγητής **Γεώργιος Χαλκιαδάκης** (Επιβλέπων)
- Καθηγητής **Μιχαήλ Γ. Λαγουδάκης**
- Καθηγητής **Θρασύβουλος Σπυρόπουλος**

Χανιά, Μάρτιος 2024

ABSTRACT

Despite recent deep neural network superhuman performance in many strategic board games, such as Chess and Go, there does not yet exist an algorithm that beats "Settlers of Catan" professional human players. Towards this direction, we present a combination of modern machine learning with traditional tree-based adversarial search algorithms and achieve performance close to the state-of-the-art in initial settlement placement. In particular, we use a generalization of the classic Minimax search algorithm, known as Max^n, with the novelty that the evaluation function at the leaf nodes is the result of a forward pass in a trained convolutional neural network. Our work consists of two distinct parts that can work independently. The first is the Max^n algorithm implementation that could use any evaluation function. The second is the neural network, which acts as an evaluation function and could be plugged into any adversarial search algorithm. After 10000 simulated games, which is a sufficient number for the demanding strategic board game "Settler of Catan", we achieve performance close to the state-of-the-art; with the advantage that, in contrast to the state-of-the-art one, our approach's runtime is acceptable by human players.

ΠΕΡΙΛΗΨΗ

Παρά το γεγονός ότι προσφάτως πολλοί αλγόριθμοι στηριζόμενοι στη χρήση βαθέων νευρωνικών δικτύων έχουν κατορθώσει να πετύχουν επιδόσεις ανώτερες των ανθρωπίνων σε πολλά στρατηγικά παίγνια, όπως το Σκάκι ή το Go, δεν υπάρχει ακόμη κάποιος αλγόριθμος που να νικάει επαγγελματίες ανθρώπους στο πολυπρακτορικό στρατηγικό παίγνιο «Άποικοι του Κατάν». Στην παρούσα διπλωματική εργασία, παρουσιάζουμε έναν συνδυασμό σύγχρονης μηχανικής μάθησης με κλασικές μεθόδους δενδρικής αναζήτησης υπό αντιπαλότητα και πετυχαίνουμε απόδοση λίγο χαμηλότερη από την καλύτερη που υπάρχει στην βιβλιογραφία. Συγκεκριμένα, χρησιμοποιούμε μία γενίκευση του κλασικού αλγορίθμου Minimax, με την επονομασία Max^n, όπου η συνάρτηση αξιολόγησης που εφαρμόζεται στα φύλλα του δένδρου είναι ένα εκπαιδευμένο συνελικτικό νευρωνικό δίκτυο. Η εργασία μας αποτελείται από δύο μέρη, τα οποία δύνανται να λειτουργήσουν ανεξάρτητα το ένα από το άλλο. Το πρώτο μέρος είναι η υλοποίηση του αλγορίθμου Max^n, ο οποίος μπορεί να χρησιμοποιήσει οποιαδήποτε συνάρτηση αξιολόγησης ορίσουμε. Το δεύτερο μέρος είναι το νευρωνικό δίκτυο, το οποίο δρα ως συνάρτηση αξιολόγησης και θα μπορούσε να ενσωματωθεί σε οποιονδήποτε αλγόριθμο αναζήτησης υπό αντιπαλότητα για να προσφέρει την πληροφορία της αξιολόγησης τερματικών καταστάσεων. Έπειτα από 10000 προσομοιωμένα παιχνίδια, που αποτελούν ένα ικανό πλήθος για την εξαγωγή εμπειρικών αποτελεσμάτων στο στρατηγικό παίγνιο «Άποικοι του Κατάν», πετυχαίνουμε επίδοση κοντά στην καλύτερη που υπάρχει στην βιβλιογραφία, με το πλεονέκτημα ότι σε αντίθεση με τη μέθοδο που πετυχαίνει αυτήν την επίδοση, η δική μας μέθοδος έχει χρόνο εκτέλεσης που είναι αποδεκτός από ανθρώπους.

ACKNOWLEDGEMENTS

It has been a very fruitful journey as a student at the Technical University of Crete. I learned a lot during this period of my life and I am very thankful to all the teachers and personnel of the School of Electrical & Computer Engineering. I would like to especially thank Professor Georgios Chalkiadakis, for his intriguing courses on artificial intelligence that I had the pleasure of attending, as well as his supervision and feedback on this thesis. My sincere thanks to Professor Michail Lagoudakis for his equally intriguing and engaging courses on artificial intelligence and computer science. Likewise, I want to express my gratitude towards all teachers who inspired me with their love for their field of study combined with an eagerness to effectively answer questions. Especially, Professor Minos Garofalakis, Professor Michail Paterakis, Associate Professor Vasilis Samoladas, and Instructor Dr. Sofia Tsakiridou. Most importantly, I am deeply grateful to my parents for their irreplaceable support and self-sacrifice during rough financial times, which is the main factor that enabled me to successfully complete my studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	**Meaning**
AI	Artificial Intelligence
Adam	Adaptive Momentum Estimation
CNN(s)	Convolutional Neural Network(s)
DNN(s)	Deep Neural Netowrk(s)
GPU(s)	Graphics Processing Unit(s)
LSTM	Long Short-term Memory
MAS	Multi-Agent Systems
MCTS	Monte Carlo Tree Search
MLP(s)	Multi-Layer Perceptron(s)
MSE	Mean Squared Error
RL	Reinforcement Learning
SGD	Stochastic Gradient Descent
SoC	Settlers of Catan

Chapter 1

Introduction

1.1 Background

Deep Neural Networks (DNNs) [1] have an unparalleled capability of approximating complex functions by mapping multiple input parameters to multiple outputs, which has allowed the efficient solution of many challenging problems. The input parameters interact in complex ways within a DNN, in a manner that is not currently feasible by any other methodology. For instance, by using supervised learning, as defined in Section 3.2, one can plug in labeled inputs and the network adjusts its weights and biases to try and match the expected outputs. Some examples, where DNNs were used to make big breakthroughs, include image recognition [2] and real-time object detection [3].

Regarding the scope of our current research, DNNs have been used to achieve superhuman performance in many strategic board games, for example, Go [4] and Chess [5].

More recently, the Settlers of Catan (SoC) strategic board game has become an increasingly popular benchmark in Artificial Intelligence (AI) and Multi-Agent Systems (MAS) research [6], [7], [8] because it combines the following intriguing characteristics:

- Stochasticity
- Large state space
- Non-zero-sum game
- Imperfect information
- Multi-agent environment

We thoroughly present the game of SoC in Chapter 2. There are two distinct phases: the initial phase (Section 2.3) and the main phase (Section 2.4). In this thesis, we focus on the initial phase which is of vital importance for winning a SoC game. As we will see in Section 4.1, we produce data in the form of *".soclog"* files by running a few million simulations with the use of the JSettlers2 framework [9]. Moreover, in Sections 4.2-4.3, we use the data to create three datasets, and in Sections 4.4-4.5, we train different model architectures to predict the final scores given the completed initial phase in a supervised learning manner. Finally, in Section 4.6, we implement the Max^n algorithm [10] which is presented in Subsection 3.1.2, in the domain of SoC, and use our trained neural network to perform terminal state evaluation. This combination of Max^n with the neural network evaluator constitutes our final agent.

1.2 Contributions

Our main contribution to the bibliography regarding AI research in SoC is the use of a Convolutional Neural Network (CNN) [11] to evaluate how good the initial placement is for each player. This new addition to the literature can be used not only to pave new paths for future work, but also to improve existing research, such as adversarial search algorithms that might benefit from our neural network evaluator [12], [13] or methods that do not take initial placement into account at all [7], [14].

Furthermore, we showcase the feasibility of the Max^n algorithm [10] which, to the best of our knowledge, has never been used for initial settlement placement in SoC. Even though this algorithm can work independently to perform initial settlement placement, we use it in combination with a trained CNN, which performs the evaluation at the leaf nodes before the tree backpropagation takes place. By making generous pruning with a simple heuristic function, we were able to achieve results close to the state-of-the-art [12] without resorting to Monte Carlo Tree

Search (MCTS)-based approaches [15], even though that constitutes a promising future direction in our view. The main advantage of the method that we present in this thesis compared to the state-of-the-art is that its runtime is acceptable by human players. Our method overrides the initial settlement placement strategy of the JSettlers version 2.6.10 agents [9], and retains their strategy during the main phase of the game.

Finally, our three datasets which are illustrated in Section 4.3, are made open source in the following Google Drive: https://drive.google.com/drive/folders/1HvrxDFylTSBYPAy_wK8nPuEz3Np6L_5k. All three datasets consist of pairs of input tensors that represent the completed initial settlement placement phase, and output tensors that represent the final scores. The input tensor content is shown in Table 4.3, and the output tensor content is presented in the last row of Table 4.2. For dataset $\mathcal{D}_2$, we also make public the raw data of 2,000,000 simulated SoC games before preprocessing.

1.3 Thesis Structure

The rest of our thesis is structured in the following way. In Chapter 2, we describe the domain of SoC in detail. In Chapter 3, we present the bibliography that was used as a foundation for our work, as well as the bibliography regarding related work in the domain of strategic board games, especially SoC. In Chapter 4, we demonstrate our methodology in detail. In Chapter 5, the empirical results that illustrate our findings can be found in the form of graphs, figures, and tables. Moreover, the aforementioned results are explained in detail and useful insights are derived. Finally, in Chapter 6 we summarize our work, draw conclusions, and propose future directions.

Chapter 2

The Settlers of Catan Strategic Board Game

In this Chapter, we provide a high-level overview of how the game of SoC is played. In Section 2.1, we present the game components, and in Section 2.2, we explain how the board is set up. Moreover, in Section 2.3, we explain the initial placement phase of the game, which is the focus of this thesis, and in Section 2.4, we describe the main phase of the game. Furthermore, in Section 2.5, we present the scoring system of the game. Finally, in Section 2.6, we describe the edge cases that can come up during a SoC game.

2.1 Game Components

In SoC, players strategically build roads, settlements, and cities on a hexagonal board that consists of terrain hexes. By collecting resources and trading with other players, they compete for victory. The game consists of two phases, the initial placement phase and the main phase, both of which will be presented in subsequent Sections. The board consists of number tokens placed on terrain hexes, which are surrounded by sea frame pieces that contain harbors. The players can own 5 types of resources, roads, settlements, cities, and development cards. In this Section, we will discuss the game components in detail. As a starting point, we outline the main game components in Table 2.1 below.

Piece Type	**Amount**	**Description**
Terrain Hexes	19	Contain resources that players collect
Sea Board Frames	6	Hold the board together and contain harbors
Harbor Pieces	9	Allow for cheap trades with the bank
Number Tokens	18	Activate hexes based on the dice sum
Resource Cards	95	Used to buy property and development cards
Developments Cards	25	Five types of cards with unique properties
Building Cost Cards	4	Contain information about purchasing costs
Special Cards	2	Indicate longest road and largest army
Roads	60	Purchasable property placed in board edges
Settlements	20	Purchasable property placed in board intersections
Cities	16	Purchasable property placed in board intersections
Dice	2	Rolled each time a player's turn begins
Robber	1	Blocks the activation of hexes

Table 2.1: Game Components

In order to gain a better understanding of SoC, we ought to look at some piece types in greater detail.

2.1.1 Terrain Hexes

There are 5 unique resource types in the game, Lumber, Brick, Wool, Grain, and Ore. Since resources are produced from terrain hexes, there are also 5 unique terrain hex types as shown below.

Terrain Hex Type	**Amount**	**Description**
Lumber Hex	4	Produces Lumber
Brick Hex	3	Produces Brick
Wool Hex	4	Produces Wool
Grain Hex	4	Produces Grain
Ore Hex	3	Produces Ore

Table 2.2: Terrain Hex Types

2.1.2 Harbor Pieces

Owning settlements or cities at harbors allows for convenient trading rates with the bank, as explained in Subsection 2.4.3. Building property at harbors actually constitutes a very viable strategy and, in some board setups, even a good one. Let us take a look at the different harbor types.

Harbor Piece Type	**Amount**	**Description**
3:1 Ports	4	Allow for 3 to 1 trades with the bank
2:1 Lumber Port	1	Allows for 2 to 1 trades with the bank
2:1 Brick Port	1	Allows for 2 to 1 trades with the bank
2:1 Wool Port	1	Allows for 2 to 1 trades with the bank
2:1 Grain Port	1	Allows for 2 to 1 trades with the bank
2:1 Ore Port	1	Allows for 2 to 1 trades with the bank

Table 2.3: Harbor Types

2.1.3 Number Tokens

Number tokens are placed on terrain hexes randomly. The only terrain hex that does not contain a number token is the desert hex where the robber initially lies.

When the dice sum is equal to a number token, then the terrain hex containing that number token is activated and produces resources that are collected by players who own settlements or cities adjacent to that hex.

Below, we see the 18 number tokens in detail.

Number Token	**Amount**	**Description**
2	1	Activates when $dice\_sum = 2$
3	2	Activates when $dice\_sum = 3$
4	2	Activates when $dice\_sum = 4$
5	2	Activates when $dice\_sum = 5$
6	2	Activates when $dice\_sum = 6$
8	2	Activates when $dice\_sum = 8$
9	2	Activates when $dice\_sum = 9$
10	2	Activates when $dice\_sum = 10$
11	2	Activates when $dice\_sum = 11$
12	1	Activates when $dice\_sum = 12$

Table 2.4: Number Tokens

2.1.4 Property

There are 3 types of property that players can buy: roads, settlements, and cities. Each player can buy up to 15 roads, 5 settlements, and 4 cities. Whenever a settlement is upgraded to a city, the settlement goes back to the owner and can be built elsewhere in the future.

2.1.5 Development Cards

Development cards truly spice up the game by adding another layer of unpredictability and thus allowing for more complicated strategies. For an explanation of their mechanics please refer to Subsection 2.4.1.

There are 3 main types of development cards, 14 knight cards, 6 progress cards, and 5 victory point cards. The progress cards have 3 sub-categories.

All types of development cards are presented below.

Development Card	**Amount**	**Description**
Knight	14	Player moves the robber to any terrain hex and collects a random resource from a player that has at least one settlement or city adjacent to that hex
Road Building	2	Player builds two roads for free
Year of Plenty	2	Player gets two resources of his choosing for free
Monopoly Cards	2	Player steals all resources of the type he chooses from all players
Victory Point	5	Player gets 1 secret point

Table 2.5: Development Card Types

2.2 Board Setup

In the most interesting version of the game, all of the pieces are placed randomly with the following algorithm:

1. Connect the sea board frame pieces.
2. Randomly place each of the 9 ports on the available positions on the frame.
3. Randomly place the terrain hexes inside the frame.
4. Randomly place the number tokens on the hexes (excluding the desert hex).
5. Place the robber on the desert hex.

Below is a random board setup from the online SoC environment *colonist.io* [16].

Figure 2.1: Board Setup

2.3 Initial Phase

In the initial phase of the game, the players begin by placing two settlements and two roads each. In order to decide who plays first, all of the players roll the dice and whoever achieves the highest sum wins. After the starting player has placed a settlement in an intersection of his choice, followed by a road adjacent to the settlement, the rest of the player do the same in a clockwise order.

Once this is done and each player has placed a single settlement and a single road, the second round of the initial phase begins with the player who placed last, now placing first and the rest of the players following a counter-clockwise order until the previously first player places his second settlement and road last. Along with the placement of the second settlement, each player receives the resources that are adjacent to the chosen intersection for that settlement.

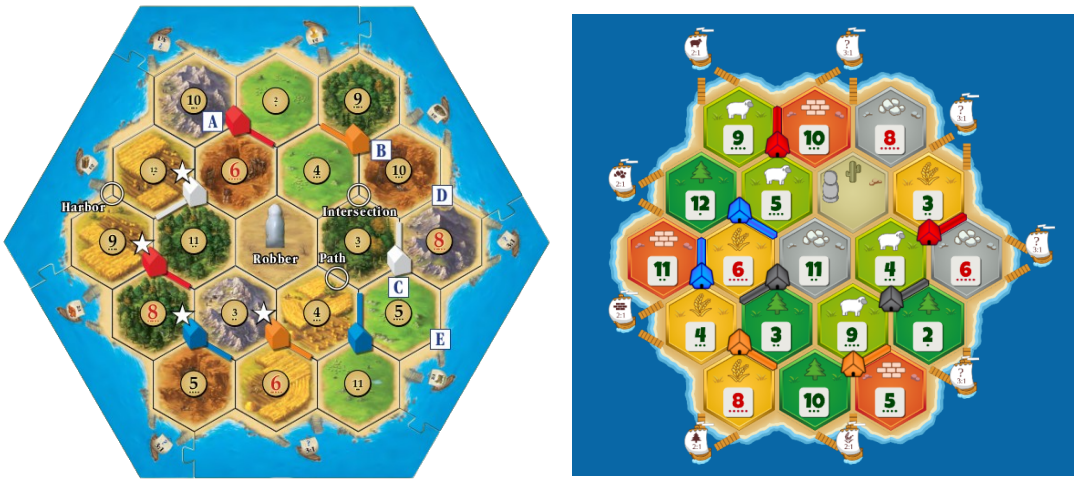


Figure 2.2: Completed Initial Phase

On the left we see an example in *Catan official rules* [17]
On the right we see an example in *colonist.io* online environment [16]

2.4 Main Phase

After the initial phase is complete, the main phase of the game begins with the player that placed his first settlement first and his second settlement last (i.e. the player that achieved the highest dice sum before the start of the initial phase). Beginning at the start of the main phase and until the end of the game, the players take turns in a clockwise order.

To describe the actions that can be taken during a player's turn in the main phase of the game, we shall take a top-down approach.

Action	Optional	Description
Development Card	YES	Player plays a development card
Roll Dice	NO	Player rolls the dice
Trade	YES	Player trades with other players or with the bank
Build	YES	Player builds settlements/cities/roads

Table 2.6: Action Space during the Main Phase of the game

2.4.1 Using Development Cards

Playing a development card is the only action a player can optionally take before rolling the dice. However, he can also wait and use such a card after rolling the dice. The strategy would depend on the type of card, for example, knight cards are usually played before rolling the dice, whereas monopoly cards are usually played after rolling the dice. The different types of development cards are presented in detail in Table 2.5.

A development card can be purchased during the building phase of a player's turn with the cost that is presented in Table 2.8. One notable point is that a development card cannot be used at the same turn that it was acquired. Hence, it is very usually the case that players buy a development card right before ending their turn so that they have an ace up their sleeve for their next turn.

2.4.2 Rolling the Dice

In contrast to other player actions that are optional, each player is to roll the dice every time his turn comes. The sum of the dice determines which terrain hexes become activated and produce resources. Every player that has settlements or cities adjacent to activated hexes, collects the corresponding amount of the hex's resource (i.e. 1 resource for each settlement & 2 resources for each city).

In the special case where the sum of the two dice is 7, no hexes are activated. Additionally, if any player has 8 or more resource cards in his possession, he has to select half of them (rounded down) and discard them. Moreover, the player who rolled the dice gets to move the robber to whichever terrain hex he desires and steal one random resource from a player that has at least one settlement or city adjacent to that hex. Finally, while the robber is at some terrain hex, that hex does not produce resources even when it is activated. An interesting aspect of the dice rolling system that should be mentioned is that not all dice sums are equally probable. Each die corresponds to a uniform distribution over

the set $\{1, 2, 3, 4, 5, 6\}$. Since the dice outcomes are independent, each of their combinations has a probability equal to $\frac{1}{6} \times \frac{1}{6} = \frac{1}{36}$. The equally probable cells are shown in Table 2.7 below:

Die 1\Die 2	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Table 2.7: Dice Sum Scenarios

Therefore, the distribution that describes the probability of the dice sum is the following:

- $P(\{\text{dice\_sum} = 2\}) = 1/36 \approx 0.03$
- $P(\{\text{dice\_sum} = 3\}) = 2/36 \approx 0.06$
- $P(\{\text{dice\_sum} = 4\}) = 3/36 \approx 0.08$
- $P(\{\text{dice\_sum} = 5\}) = 4/36 \approx 0.11$
- $P(\{\text{dice\_sum} = 6\}) = 5/36 \approx 0.14$
- $P(\{\text{dice\_sum} = 7\}) = 6/36 \approx 0.17$
- $P(\{\text{dice\_sum} = 8\}) = 5/36 \approx 0.14$
- $P(\{\text{dice\_sum} = 9\}) = 4/36 \approx 0.11$
- $P(\{\text{dice\_sum} = 10\}) = 3/36 \approx 0.08$
- $P(\{\text{dice\_sum} = 11\}) = 2/36 \approx 0.06$
- $P(\{\text{dice\_sum} = 12\}) = 1/36 \approx 0.03$

2.4.3 Trading

Once the player has rolled the dice, he can propose trades to any other player and any other player can propose trades to him. There are no limitations in trading with other players, meaning that any type and amount of resources can be traded with another type and amount of resources provided that both trading players agree on it.

Besides trading with other players, the current player can also trade with the bank. The default ratio for bank trades is 4:1, meaning that 4 resources of the same type may be traded with a single resource of any other type.

In the special case where the current player has built a settlement or a city at a port, he can enjoy better trading deals with the bank. In particular, for 3:1 ports, the player can trade 3 resources of a single type for 1 resource of any different type. Moreover, if the player owns a settlement or a city in a 2:1 port, he has the ability to trade 2 resources of the type specified by the port for 1 resource of any different type.

2.4.4 Building

Once the player has rolled the dice, optionally used a development card and possibly traded with other players or with the bank, it is time to make purchases and complete his turn.

In Table 2.8 shown below, the buying options are listed:

Property	**Cost**
Road	1 Lumber, 1 Brick
Settlement	1 Lumber, 1 Brick, 1 Wool, 1 Grain
City	2 Grain, 3 Ore
Development Card	1 Wool, 1 Grain, 1 Ore

Table 2.8: Purchasing Options

A road must be built so that it is located at a hex edge, unlike settlements and cities that must be located at intersections. An additional constraint is that in order to build a road, it has to be adjacent to another property owned by the same player (i.e. another road, settlement, or city).

A settlement can be built at an intersection when a player owns a road that is adjacent to it, provided that no other settlement or city exists at a distance of one edge away.

A city can be built where settlements already exist. In other words, building a city is effectively upgrading the current settlement so that it produces 2 resources instead of 1 whenever an adjacent hex gets activated.

2.5 Score System

The first player to get to 10 points wins the SoC game.

Property	**Points**	**Description**
Settlement	+1	Own a settlement
City	+2	Own a city
Victory Point	+1	Own a victory point development card
Longest Road	+2	Get the longest road (at least 5 consecutive road pieces)
Largest Army	+2	Use the most knight development cards (at least 3 knight development cards)

Table 2.9: Point System

If a player already has the longest road, for example by owning 7 consecutive road pieces, then to receive the longest road two bonus points one would need to build a road of at least 8 consecutive road pieces.

The same idea applies to the largest army points. The first player to use 3 knight development cards receives two bonus points and for this bonus to be taken away from him, another player needs to create an army that surpasses his army in size.

2.6 Edge Cases

In this Section, we shall explore some rare but important to know edge cases that can come up during a SoC game.

Given the limited number of resource cards, specifically 95, it may be the case that after a dice roll and hex activation, there are not enough resources to be distributed to the players. In such a case, there are two scenarios. If more than one player is affected by the lack of resources, then no player gets any resources. On the other hand, if only a single player is affected, he gets as many resources as are available (possibly zero).

Each time a development card is used, it gets discarded and does not go back to the pile of development cards. This means that if all 25 development cards are used then no player can buy a development card.

If the road building card is used by a player who has already built the maximum amount of roads, which is 15, then the card has no effect. If it is used by a player who has built 14 roads, then he can build 1 road for free.

A very interesting edge case is when your longest road is broken by another player who builds a settlement in an unoccupied intersection along that road. If your reduced connected road ties in length with the connected road of one or more other players, then you retain the 2 bonus points for the longest road. On the other hand, if your reduced connected road is smaller than others, and two or more of the other players tie for the longest road, then the 2 bonus points are deducted from you and they are not given to any player until someone's road becomes the single longest road.

If you reach 10 points while it is not your turn, for example in the case where another player splits the longest road and you become the new longest road owner, then you do not immediately win the game. To win the game, you need to have 10 points during your turn.

Chapter 3

Literature Review

In this Chapter, we conduct a literature review in two directions: Firstly, we present work that was used as a foundation for this thesis, and secondly, we present related work, a part of which is used to benchmark our approach.

Particularly, in Section 3.1, we present the Max^n algorithm in detail by building a generalization on the simpler version, known as minimax. Furthermore, in Section 3.2, we present the theoretical background of supervised learning. Finally, in Section 3.3, we present related work in strategic board games with a focus on the domain of SoC.

3.1 Generalization of Minimax (Max^n)

In this Section, we present a generalization of minimax, known as Max^n, which is a main component of our thesis. In Subsection 3.1.1, we present the minimax algorithm, and in Subsection 3.1.2, we generalize it to the Max^n algorithm.

3.1.1 The Minimax Algorithm

We shall begin by briefly describing the Minimax algorithm [18] which constitutes the foundation on top of which the Max^n generalization [10] was developed. Minimax works in perfect information, sequential, two-player, zero-sum games.

Before proceeding any further, we ought to define rationality in the context of game theory. As described in [19] and is widely accepted in the field of game theory, a rational agent is one who is acting optimally in order to maximize its expected utility given the information it has. In the case of a two-player, zero-

sum game, maximizing the utility for one player is equivalent to minimizing the utility for the other player. Hence, instead of describing the score by a tuple of two numbers (i.e. the score for each player), we transform the scoring system and use only one number centered around zero. If the score of the game is below zero then the so-called minimizing player is winning, otherwise the maximizing player is winning or we have a draw if the score is exactly zero. Consider two players, p_0 and p_1, along with two a priori known functions $a_{p_0}(s)$ and $a_{p_1}(s)$. The former function defines the set of states reachable by player p_0 from state s, while the latter defines the set of states reachable by player p_1 from state s. Assuming, without loss of generality, that p_0 is the maximizing player and p_1 is the minimizing player, as well as that both players are rational, we can calculate the minimax score of the game in the following brute-force manner:

$$V(s,p) = \begin{cases} u(s), & \text{if } s \text{ is a terminal state} \\ \max_{s' \in a_{p_0}(s)}\{V(s', p_1)\}, & p = p_0 \\ \min_{s' \in a_{p_1}(s)}\{V(s', p_0)\}, & p = p_1 \end{cases}$$

where:

- $p \in \{p_0, p_1\}$ is the current player.
- s is the current state of the game.
- s' is the new state reached by some action.
- $u(s)$ is the expected utility in terminal state s.
- $V(s,p)$ is the minimax score in state s and player p's turn.

We note that there is an extension to this algorithm, known as alpha-beta pruning [20], but it is outside the scope of this thesis.

3.1.2 The Max^n Algorithm

The Max^n algorithm [10] constitutes a generalization of Minimax that can be used in perfect information, sequential, non-zero-sum games with more than 2 players. In this case, the codomain of the Max^n function is not a single value but rather a tuple of n values given that we have n players. In other words, the codomain is generally speaking $\mathbb{R}^n$, while the domain can vary depending on how many dimensions and what type of data is needed to describe the state space. In any case, we can say that Max^n is a function $V : \mathcal{S} \to \mathbb{R}^n$, where $\mathcal{S}$ is the set of all possible game states.

For the sake of simplicity and better understanding, let us begin by exploring the scenario where we have $n = 4$ players, which is actually the case for this thesis. In this specific case, using similar notation as in Subsection 3.1.1, we can calculate the Max^n score, or in particular the Max^4 score, given the game state and who's turn it is (in practice we can encode who's turn it is inside the game state) as follows:

$$V(s,p) = \begin{cases} u(s), & \text{if } s \text{ is a terminal state} \\ \max_{s' \in a_{p_0}(s)}\{V(s', p_1)_0\}, & p = p_0 \\ \max_{s' \in a_{p_1}(s)}\{V(s', p_2)_1\}, & p = p_1 \\ \max_{s' \in a_{p_2}(s)}\{V(s', p_3)_2\}, & p = p_2 \\ \max_{s' \in a_{p_3}(s)}\{V(s', p_0)_3\}, & p = p_3 \end{cases}$$

where:

- $p \in \{p_0, p_1, p_2, p_3\}$ is the current player.
- s is the current state of the game.
- s' is the new state reached by some action.

- $u(s)$ is the tuple of expected utilities in terminal state s.
- $V(s, p)$ is the Max^4 score in state s and player p's turn.
- $\max_{s' \in a_{p_i}(s)}\{V(s', p_j)_i\}$ means that we are trying to maximize the i-th element of the tuple in the codomain of V, which corresponds to the score for the i-th player. We do that over the set of possible actions for player p_i, where each action leads to a new state s' as given by the function $a_{p_i}(s)$.

Having established a solid understanding by presenting the case where $n = 4$, we will now illustrate the general case of the recursive Max^n algorithm:

$$V(s, p_i) = \begin{cases} u(s), & \text{if } s \text{ is a terminal state} \\ \max_{s' \in a_{p_i}(s)}\{V(s',\ p_{(i+1) \bmod n})_i\}, & \text{otherwise} \end{cases}$$

It is worth mentioning that a generalization of the alpha-beta pruning algorithm [21] exists for Max^n as well. Nevertheless, it was not used in this thesis although it constitutes a promising future direction, especially in terms of time efficiency.

3.2 Supervised Learning

Over the years, many ways to estimate functions based on input-output pairs have been studied, but in the last decade, the most effective way in complex environments such as SoC is modeling the problem with the use of DNNs due to their ability to effectively represent nonlinear functions [7], [8], [14], [22]. The main reason that the use of DNNs became feasible is the increase in the computational power of hardware, mainly that of Graphics Processing Units (GPUs).

Most approaches that aim to improve the AI performance in SoC use Reinforcement Learning (RL) [7], [14], [22]. However, we use supervised learning to learn a function that evaluates the completed initial settlement placement and use it at the leaf nodes of the Max^n algorithm for terminal state evaluation. For this purpose, let us explore the theoretical background behind supervised learning.

As described in the book by Stuart Russell and Peter Norvig [19], in supervised learning the agent observes multiple input-output pairs and learns a function that maps from input to output.

In mathematical terms, given a training set of N input-output samples:

$$(\boldsymbol{x}, \boldsymbol{y}) = \begin{bmatrix} (x_1, y_1) \\ (x_2, y_2) \\ \vdots \\ (x_N, y_N) \end{bmatrix}$$

where each pair was generated by an unknown function $f(x) = y$, we have to find a function h that approximates f as closely as possible.

To fully describe a supervised learning process of a neural network, we need to define three main components:

1. **Model Architecture**
2. **Learning Algorithm** (**Optimizer** in the terminology of PyTorch)
3. **Loss Function** (**Criterion** in the terminology of PyTorch)

The model architectures we experimented with are outside the scope of this Section, but they can be found in Section 4.4.

Regarding the learning algorithm, a well-established option is Stochastic Gradient Descent (SGD) [23]:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \gamma \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t)$$

In order to speed up convergence, momentum can be added to the optimizer. PyTorch allows for convenient incorporation of Nesterov's momentum [24]:

$$\boldsymbol{u}_{t+1} = \mu \boldsymbol{u}_t - \gamma \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t + \mu \boldsymbol{u}_t)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{u}_{t+1}$$

A more modern learning algorithm that usually converges faster but does not consistently achieve better performance, is the Adaptive Momentum Estimation (Adam) algorithm [25]:

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1$^{\text{st}}$ moment vector)
  $v_0 \leftarrow 0$ (Initialize 2$^{\text{nd}}$ moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Finally, mini-batch processing can be used to speed up the training process. As a loss function for our linear regression problem, we used the Mean Squared Error (MSE) over mini-batches. Specifically, for each mini-batch $\mathcal{B}$, the loss is computed as follows:

$$L(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{i=1}^{|\mathcal{B}|} (h_{\boldsymbol{\theta}}(x^{(i)}) - y^{(i)})^2$$

3.3 Related Work

In this Section, we present research from the bibliography of strategic board games that is related to this thesis. In Subsection 3.3.1, we make some remarks regarding the field of strategic board games in general, and in Subsection 3.3.2, we review the SoC literature.

3.3.1 Strategic Board Games

We shall begin our journey in the literature of AI in strategic board games with a very fascinating event that shook the world back in 1997. As you might imagine, we are referring to the loss of the former world chess champion, Garry Kasparov, to Deep

Blue [26], which is an algorithm based on alpha-beta pruning that ran on specially designed hardware by IBM. This was the first time in chess history [27] that an AI agent beat the human world champion.

Over the last decade, DNNs have pushed the state-of-the-art in many strategic board games by a lot (e.g. Chess, Go). They have not only been incorporated in traditional approaches [4] but have also been used independently [5].

An extensive survey that covers the field of strategic board games, is conducted by Fernando Fradique Duarte, Nuno Lau, et. al. [28]. For the remainder of this Section, we will focus on SoC.

3.3.2 Settlers of Catan

The first AI SoC agent to be able to compete against humans at a decent level was developed by Thomas R.S. as part of his Ph.D. thesis in 2003 [29]. The agent was named JSettlers as a combination of the names "Java" and "Settlers of Catan". In the author's approach, the core of his agent's decision-making process is an algorithm that estimates the building speed of available types of property (i.e. settlement, city, road, and development card), based on the hexes and ports that a player has access to. This building speed estimate algorithm is used for both the initial and the main phases of the game. For the initial free placement phase, JSettlers agents consider pairs of settlements, and for each of them calculate the sum of building speeds of each property type. After the calculations have taken place, the player decides to build the pair that minimizes the building speed sum, starting with the settlement that is adjacent to the number tokens that are most likely to be activated. For the main phase of the game, the decision algorithm has a lot of special cases, but generally speaking, during the early stages of the game, building speed estimates are calculated for actions that maximize resource production, whereas, during the late stages of the game, building speed estimates are calculated for actions that yield the most points.

In 2007, the JSettlers2 framework [9] was built by using Thomas' work as a foundation and it has increasingly been used over the past decade for MAS research.

In [30], the authors use ant colony optimization [31] and make an effort to generate balanced SoC maps where no player has an advantage from the beginning of the game due to the random map initialization. We think that this is an interesting research direction that has not been explored a lot, perhaps due to the lack of a systematic way to evaluate the proposed map. In [32], the authors attempt to create agents that are more interesting for humans to interact with compared to the baseline JSettlers agents.

To move to research that focuses on the strategic aspect of the game, an approach that managed to beat the JSettlers agents, even though trades were disallowed and the rules were changed to make it a 2-player game, is that of Quentin Gendre and Tomoyuki Kaneko [22].

A more noteworthy effort that did not simplify the rules of the game, is that of Panousis K. P. [33]. In this approach, an MCTS approach that considers the complete main phase of the game was developed. Additionally, the author tested 3 approaches to solve the exploration-exploitation dilemma [34]. The first two approaches, namely Upper Confidence Bound for Trees (UCT) and Bayesian UCT are well-known. On the other hand, the third approach, myopic value of perfect information (myopic-VPI) [35], was used in MCTS for the first time. None of the approaches was able to beat the JSettlers agents, nevertheless, the author's ideas and empirical findings proved to be useful in more recent research for SoC.

One notable example is the work of Karamalegos E. [13]. As part of his thesis, the author improved the 3 aforementioned approaches that balance exploration and exploitation within the MCTS algorithm. Additionally, to handle the full set of legal actions of the game, he merged these methods that handle the main phase of SoC with an approach that still constitutes the state-of-the-art for the initial phase of SoC [12]. His empirical results show an improvement compared to the work of Panousis [33], but the created agent was not able to beat the JSettlers agents either.

In [8], Apostolidou M. uses supervised learning to predict player moves. To train her models, she uses not only game state information but also linguistic data. Furthermore,

the author develops an MLP architecture, a Long Short-term Memory Neural Network (LSTM) architecture, as well as combinations of the two. Even though she managed to reach an 88% prediction accuracy, her research was not concerned with the development of a way to test the trained models against JSettlers agents, for example by trying to predict an expert player's move.

In [36], the trading policy of the JSettlers agent is improved after thorough consideration of experimentally collected data. From our research in the literature, we confirmed the claim made in the original JSettlers Ph.D. thesis [29] that the trading aspect of SoC is where the proposed agents have the most room for improvement. In fact, several approaches have significantly improved the trading strategy [14], [37], [7].

Remarkably, in [7], the authors use a novel approach based on reinforcement learning to achieve the aforementioned improvement. To calculate the Bellman Q-value function, they use the concept of Q-decomposition [38], and create an architecture that consists of $|\mathcal{A}|$ parallel LSTM neural networks, where $\mathcal{A}$ is the set of legal actions. Once the game state is fed into these networks and the outputs are produced, the final Q-value is computed as the maximum over all Q^i, where $i \in \{1, 2, ..., |\mathcal{A}|\}$. Moreover, at each iteration, the LSTM weights Q^i, are updated $\forall\, i \in \{1, 2, ..., |\mathcal{A}|\}$ and not simply for $\underset{i}{argmax}\{Q^i\}$. Finally, the complete algorithm is able to train itself online during gameplay and the authors report a superior performance to that of the JSettlers agents.

Hereafter, the approaches that we will explore are either part of the STAC project [39] which has been funded by the European Research Council, or heavily based on it. The most notable approaches are the ones that consider the full action space during the main phase of the game. In [6], Mihai S. Dobre and Alex Lascarides, who has made significant contributions to the literature of AI in SoC, developed an MCTS algorithm that considers the full set of legal actions of the game after the initial placement phase. They improved the traditional MCTS algorithm in two ways. Firstly, they broke the sampling policy into two steps; sampling over action types, and sampling over specific actions within the chosen action type. They empirically show that this two-step process

significantly speeds up the algorithm. Secondly, they implement tree parallelization [40] and make their approach much more time-efficient.

In [41], the same authors extend the definition of the Partially Observable Monte Carlo Planning algorithm (POMCP) [42] to multi-agent environments. This approach also considers the full action space during the main phase of the game. Similarly to their MCTS approach, they strongly dominate baseline STAC agents which we will present shortly.

To conclude our literature review, we will present approaches that deal with the same problem as us; the initial settlement placement. In [43], Markus Guhe and Alex Lascarides, improve the initial placement strategy as well as the strategy for choosing the next action during the main phase of the game. They show the dominance of their agent against JSettlers agents but do not show the effects of the initial placement improvement alone. Their created agent was named STAC agent in future publications of Alex Lascarides et. al. and has been used as a baseline ever since. In [12], Alex Lascarides and Mihai Sorin Dobre, further improve the initial placement state-of-the-art with an MCTS-based algorithm seeded with a human corpus. Their new agent achieves a 30.43% win ratio after playing against three STAC agents over 10,000 games. To the best of our knowledge, this is the state-of-the-art for initial settlement placement but unfortunately, we do not have a way to run our agent against it. As a final note, the few approaches that handle the full set of game rules, such as [6], do not report results where only the initial settlement placement is taken into account, therefore we have no way to compare our results to them or know whether they're better than [12].

In version 2.5.00 of JSettlers2 [9], released in 2022, certain components from the STAC project [39] were incorporated into the JSettlers agent. However, the specific parts integrated have not been specified. In our research, we used version 2.6.10 which is currently the newest version. By making the assumption that the initial placement strategy of the JSettlers agents in this version is at least as good as that of the STAC agent in [43], we indirectly compare our results to the state-of-the-art [12] and manage to get close to it.

Chapter 4
Methodology

In this Chapter, we explain our methodology in detail. As a first step, we create our own datasets through a few million simulations using the JSettlers2 framework [9]. Furthermore, we train 6 different model architectures in a supervised learning manner to predict the final scores given the completed initial phase. Since there is stochasticity during a SoC game, we want our network to be able to predict the expected final score for each player. As a final step, we choose the most suitable of these 6 created DNN architectures, and merge the trained model that acts as an evaluator with the Max^n algorithm [10] which decides where to place a settlement. In our implementation of Max^n, which is a tree-based adversarial search algorithm presented in Section 3.1, each node explores a subset of available moves according to a heuristic function. Once a leaf node is reached, our trained DNN performs evaluation, allowing Max^n to backpropagate the result to the root and choose the best move.

Specifically, in Section 4.1, we present how we produced data that can be used to train a neural network. In Section 4.2, we showcase how we preprocessed the data to maximize the effectiveness of our neural networks. In Section 4.3, we describe different datasets that we created from the preprocessed data. In Section 4.4, we showcase the architectures of the different models that we trained, and in Section 4.5, we describe the training process. Finally, in Section 4.6, we present the pseudocode of the Max^n implementation in the domain of SoC and explain the communication process between the server that contains the trained DNN and the aforementioned algorithm.

Our agent overrides the initial placement strategy of the JSettlers version 2.6.10 agent that is available in the JSettlers2 framework [9], while it retains its strategy during the main phase of the game.

4.1 Data Creation

One of the necessary elements for any supervised learning algorithm is data. In the scope of our research, we used the JSettlers2 framework [9] and simulated 3,500,000 games of SoC with 2 types of agents:

- Built-in JSettlers Agent. (places initial settlements heuristically)
- Custom Random Agent. (places initial settlements uniformly at random)

Agent	**Initial Phase Strategy**	**Main Phase Strategy**
JSettlers Agent	JSettlers version 2.6.10 [9], based on [29]	JSettlers version 2.6.10 [9], based on [29]
Random Agent	Uniformly random settlement placement	JSettlers version 2.6.10 [9], based on [29]

Table 4.1: Agent Types for Data Creation

In particular, we created:

- 2,500,000 Games Consisting of 4 Random Agents.
- 500,000 Games Consisting of 3 Random Agents & 1 JSettlers Agent.
- 500,000 Games Consisting of 4 JSettlers Agents.

Using available functionality, we stored these games in *".soclog"* files. Afterwards, through parsing, we extracted the data that is relevant to our work. Below, in Table 4.2, we present the features which fully represent the completed initial phase of the game, as well as the final scores:

Feature	**Number**	**Domain**	**Description**
Hex Layout	37	$\{0, 1, ..., 213\}$	Desert Hexes $\rightarrow$ 0 Water Hexes $\rightarrow$ 6 Resource Hexes $\rightarrow \{1, ..., 5\}$ 3:1 Port Hexes $\rightarrow \{7, ..., 15\}$ 2:1 Port Hexes $\rightarrow \{16, ..., 213\}$
Number Layout	37	$\{-1, 0, ..., 9\}$	$-1 \rightarrow$ No Number Tile $0 \rightarrow$ Number Tile 2 $1 \rightarrow$ Number Tile 3 $2 \rightarrow$ Number Tile 4 $3 \rightarrow$ Number Tile 5 $4 \rightarrow$ Number Tile 6 $5 \rightarrow$ Number Tile 8 $6 \rightarrow$ Number Tile 9 $7 \rightarrow$ Number Tile 10 $8 \rightarrow$ Number Tile 11 $9 \rightarrow$ Number Tile 12
Roads	72	$\{0, 1, ..., 4\}$	$0 \rightarrow$ No Road Built $1 \rightarrow$ Road Belongs to Player 1 $2 \rightarrow$ Road Belongs to Player 2 $3 \rightarrow$ Road Belongs to Player 3 $4 \rightarrow$ Road Belongs to Player 4
Settlements	54	$\{0, 1, ..., 4\}$	$0 \rightarrow$ No Settlement Built $1 \rightarrow$ Settlement Belongs to Player 1 $2 \rightarrow$ Settlement Belongs to Player 2 $3 \rightarrow$ Settlement Belongs to Player 3 $4 \rightarrow$ Settlement Belongs to Player 4
Final Scores	4	$\{2, 3, ..., 12\}$	The final player scores. Even though 10 points are needed to win, the highest possible score is 12 points.

Table 4.2: Extracted Data from *".soclog"* Files

4.2 Data Preprocessing

Having extracted all the meaningful data, as explained in Section 4.1, we now have to preprocess it in a way that the neural network can utilize it efficiently. Notice that:

- *Hex Layout* → Categorical Data (13 Values)
- *Number Layout* → Numerical Data
- *Roads* → Categorical Data (5 Values)
- *Settlements* → Categorical Data (5 Values)
- *Final Scores* → Numerical Data

To begin with, we wanted the network to be able to understand spatial information. For this reason, we represented *Hex Layout* and *Number Layout* as 7×7 matrices, where any cell that is not part of the initial 37 tiles is set to zero.

Additionally, we used one-hot encoding for the categorical data, which heavily increased the total number of features, but also the accuracy of our neural network. We do not show any results without one-hot encoding, because they were significantly worse, as it is well-established to be the case. Hence, we focus our attention on more meaningful findings.

Finally, we empirically found that transforming *Number Layout* tiles by mapping them to their corresponding probabilities of being activated after a dice roll works best for the neural network:

- $-1 \to 0$ (This hex does not have a number tile)
- $0 \to 3$ $P(\{\text{dice sum equals } 2\}) = 1/36 \approx 0.03$
- $1 \to 6$ $P(\{\text{dice sum equals } 3\}) = 2/36 \approx 0.06$
- $2 \to 8$ $P(\{\text{dice sum equals } 4\}) = 3/36 \approx 0.08$

- $3 \rightarrow 11$ $P(\{\text{dice sum equals } 5\}) = 4/36 \approx 0.11$
- $4 \rightarrow 14$ $P(\{\text{dice sum equals } 6\}) = 5/36 \approx 0.14$
- $5 \rightarrow 14$ $P(\{\text{dice sum equals } 8\}) = 5/36 \approx 0.14$
- $6 \rightarrow 11$ $P(\{\text{dice sum equals } 9\}) = 4/36 \approx 0.11$
- $7 \rightarrow 8$ $P(\{\text{dice sum equals } 10\}) = 3/36 \approx 0.08$
- $8 \rightarrow 6$ $P(\{\text{dice sum equals } 11\}) = 2/36 \approx 0.06$
- $9 \rightarrow 3$ $P(\{\text{dice sum equals } 12\}) = 1/36 \approx 0.03$

The only feature that received no preprocessing is that of the final scores. Intuitively speaking, we want our neural network to take the completed initial placement phase as input, and output the expected final scores. In this way, the initial placement is evaluated for each player.

To help the DNN learn $E[s_1, s_2, s_3, s_4 \mid \textit{initial placement}]$, where s_1, s_2, s_3, s_4 are the final player scores, both Random Agent and JSettlers Agent use the same strategy after the initial placement phase. We hope that the stochastic elements of SoC will be averaged out by the large number of training samples.

The input format of the neural network is shown below, in Table 4.3:

Feature	**Number**	**Domain**
Hex Layout	$49 \times 12 = 588$	$\{0, 1\}$
Number Layout	49	$\{0, 3, 6, 8, 11, 14\}$
Roads	$72 \times 4 = 288$	$\{0, 1\}$
Settlements	$54 \times 4 = 216$	$\{0, 1\}$

Table 4.3: Preprocessed Input for the DNN

There are 1141 neurons in the input and 4 neurons in the output.

4.3 Dataset Creation

As presented in Section 4.1, we have simulated a total of 3,500,000 SoC games. In order to test different DNN architectures, we created 2 additional datasets that constitute a subset of the total games, as shown below in Table 4.4:

Dataset	**Total Size**	**Description**
$\mathcal{D}_1$	1,000,000	Games with 4 Random Agents
$\mathcal{D}_2$	2,000,000	1,000,000 games with 4 Random Agents 500,000 Games with 3 Random Agents & 1 Jsettlers Agent 500,000 Games with 4 Jsettlers Agents
$\mathcal{D}_3$	3,500,000	All created games presented in Section 4.1

Table 4.4: Datasets used for Training

Dataset $\mathcal{D}_1$ was created to test the different architectures in games where the initial placement is uniformly random. Dataset $\mathcal{D}_2$ was created as a representative subset of $\mathcal{D}_3$, to allow for faster hyperparameter optimization.

4.4 Model Architectures

We created and tested 6 DNN architectures, 3 Multi-Layer Perceptrons (MLPs), and 3 CNNs.

- Small MLP (1,432,836 parameters)
- Medium MLP (6,762,452 parameters)
- Large MLP (17,067,316 parameters)
- Small CNN (1,401,460 parameters)
- Medium CNN (4,646,524 parameters)
- Large CNN (16,528,512 parameters)

In Figures 4.1-4.6, we share visualizations of the above architectures by making use of the *draw_graph* module from the *torchview* package [44]. Particularly, in Figures 4.1-4.3, the MLP architectures are illustrated, and in Figures 4.4-4.6, the CNN architectures are illustrated.

The small MLP architecture, shown in Figure 4.1, consists of 2 fully connected hidden layers. The first hidden layer has 1024 neurons, and the second hidden layer has 256 neurons.

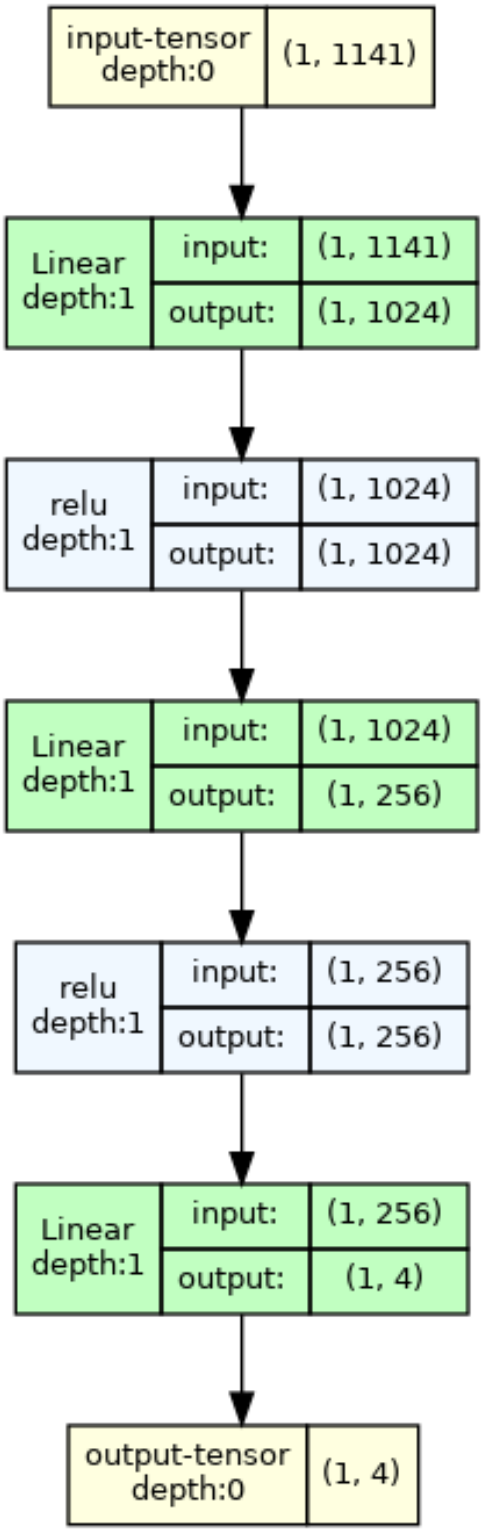


Figure 4.1: Small MLP

The medium MLP architecture, shown in Figure 4.2, consists of 3 fully connected hidden layers. The first hidden layer has 3000 neurons, the second hidden layer has 1024 neurons, and the third hidden layer has 256 neurons.

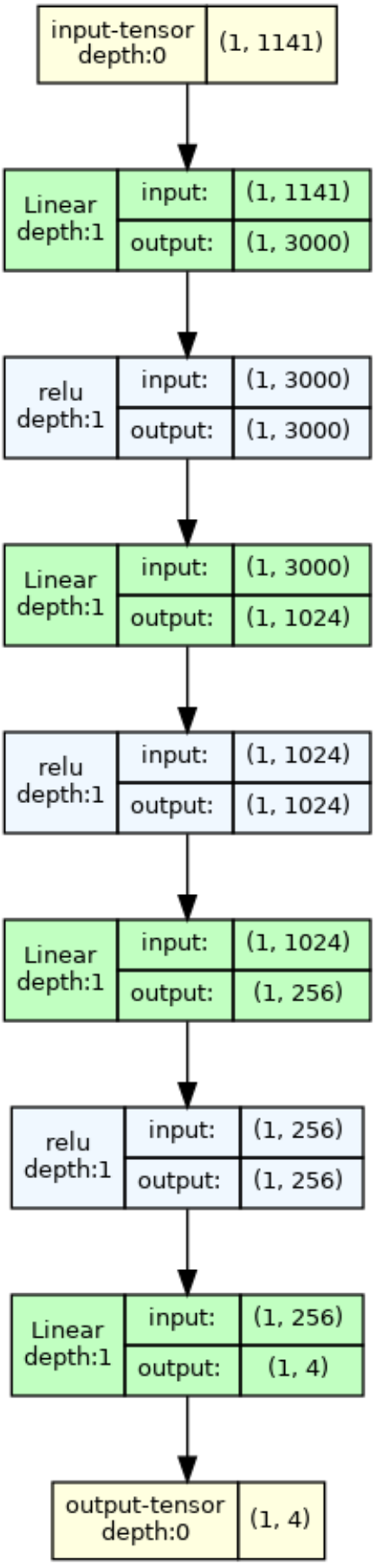


Figure 4.2: Medium MLP

The large MLP architecture, shown in Figure 4.3, consists of 4 fully connected hidden layers. The first hidden layer has 5000 neurons, the second hidden layer has 2048 neurons, the third hidden layer has 512 neurons, and the fourth hidden layer has 128 neurons.

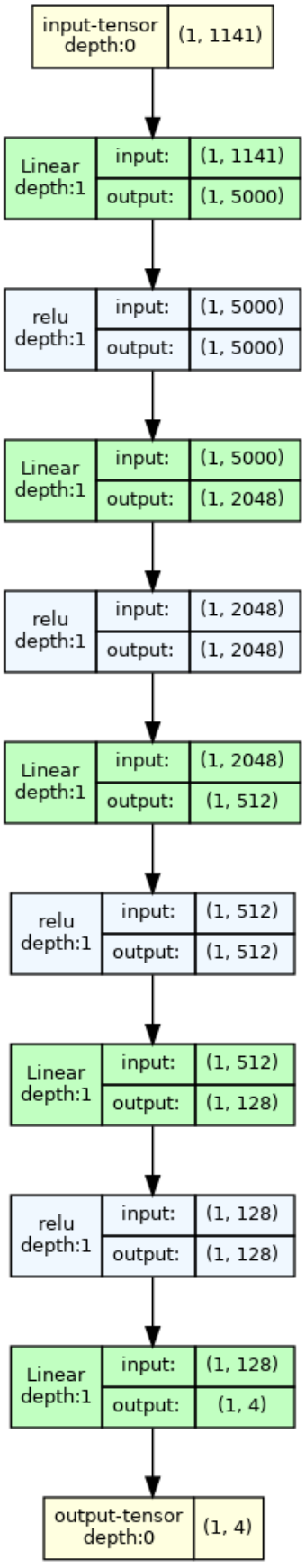


Figure 4.3: Large MLP

Before moving to the CNN architectures, we note that they all have a similar structure to process the input, which is shown in Table 4.3. In particular, convolutions are applied separately in the neurons that correspond to the hex layout and in the neurons that correspond to the number layout. After the convolutions have taken place, the results are concatenated with the unaffected road and settlement encodings, and fully connected hidden layers are then used to compute the final output.

The small CNN architecture, shown in Figure 4.4, consists of 2 convolutional layers for the hex layout, 2 convolutional layers for the number layout, and 2 fully connected hidden layers to compute the final output.

Regarding the hex layout, the first convolutional layer consists of 1 input channel, 16 output channels, and a kernel of size $(5, 5, 12)$. Moreover, $padding = (2, 2, 0)$ is performed in the first 2 dimensions to retain their size after the convolution. The second convolutional layer consists of 16 input channels, 32 output channels, and a kernel of size $(5, 5, 1)$.

Regarding the number layout, the first convolutional layer consists of 1 input channel, 16 output channels, and a kernel of size $(5, 5)$. Moreover, $padding = (2, 2)$ is performed to retain the size after the convolution in both dimensions. The second convolutional layer consists of 16 input channels, 32 output channels, and a kernel of size $(5, 5)$.

The results from the convolutional layers, along with the road and settlement encodings, are then concatenated into a layer of 1080 neurons. The first fully connected hidden layer that follows has 1024 neurons, and the second hidden layer has 256 neurons.

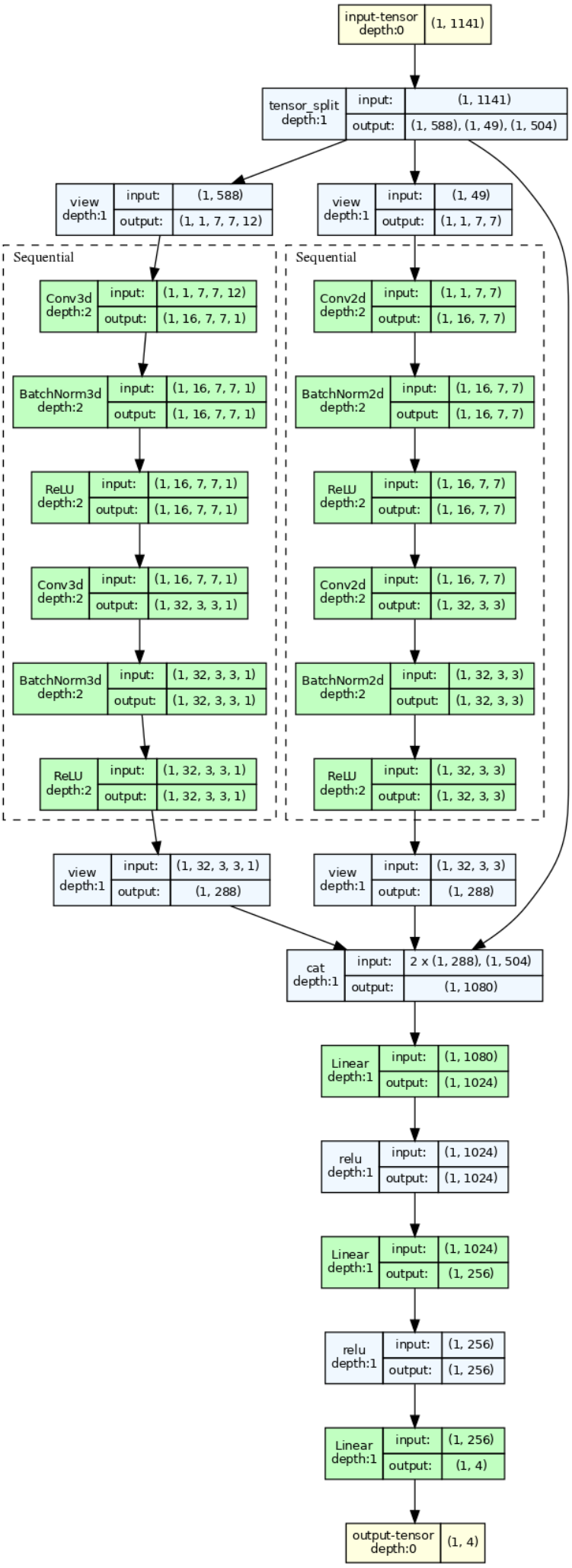


Figure 4.4: Small CNN

The medium CNN architecture, shown in Figure 4.5, consists of 4 convolutional layers for the hex layout, 4 convolutional layers for the number layout, and 3 fully connected hidden layers to compute the final output.

Regarding the hex layout, the first convolutional layer consists of 1 input channel, 8 output channels, and a kernel of size $(5, 5, 12)$. Moreover, $padding = (2, 2, 0)$ is performed in the first 2 dimensions to retain their size after the convolution. The second convolutional layer consists of 8 input channels, 16 output channels, and a kernel of size $(5, 5, 1)$. Moreover, $padding = (2, 2, 0)$ is performed. The third convolutional layer consists of 16 input channels, 32 output channels, and a kernel of size $(5, 5, 1)$. Moreover, $padding = (2, 2, 0)$ is performed. The fourth convolutional layer consists of 32 input channels, 64 output channels, and a kernel of size $(5, 5, 1)$.

Regarding the number layout, the first convolutional layer consists of 1 input channel, 8 output channels, and a kernel of size $(5, 5)$. Moreover, $padding = (2, 2)$ is performed to retain the size after the convolution in both dimensions. The second convolutional layer consists of 8 input channels, 16 output channels, and a kernel of size $(5, 5)$. Moreover, $padding = (2, 2)$ is performed. The third convolutional layer consists of 16 input channels, 32 output channels, a kernel of size $(5, 5)$, and $padding = (2, 2)$ is also performed. The fourth convolutional layer consists of 32 input channels, 64 output channels, and a kernel of size $(5, 5)$.

The results from the convolutional layers, along with the road and settlement encodings, are then concatenated into a layer of 1656 neurons. The first fully connected hidden layer that follows has 2048 neurons, the second hidden layer has 512 neurons, and the third hidden layer has 128 neurons.

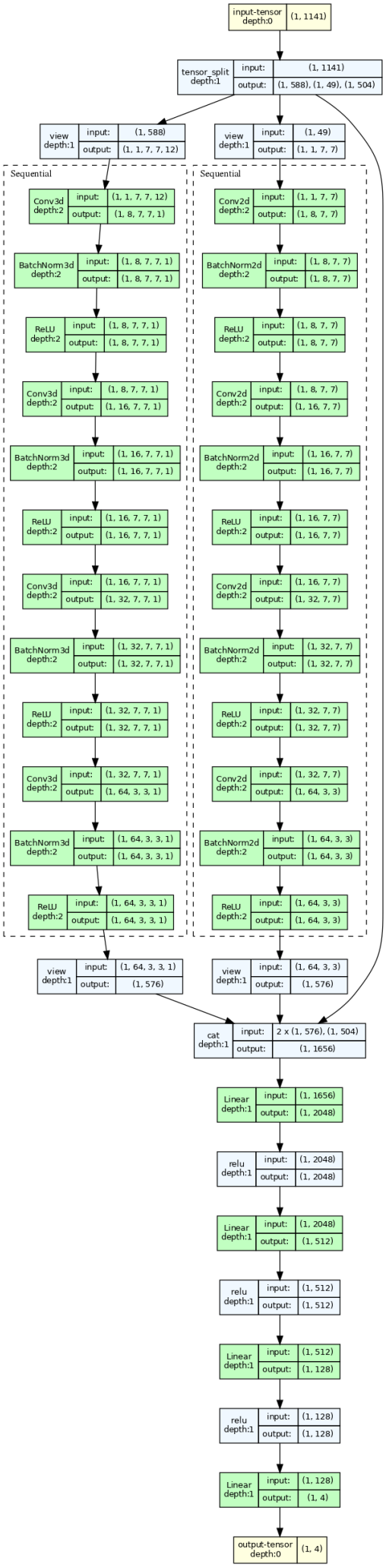


Figure 4.5: Medium CNN

The large CNN architecture, shown in Figure 4.6, consists of 6 convolutional layers for the hex layout, 6 convolutional layers for the number layout, and 4 fully connected hidden layers to compute the final output.

Regarding the hex layout, the first convolutional layer consists of 1 input channel, 4 output channels, and a kernel of size $(5, 5, 12)$. Moreover, $padding = (2, 2, 0)$ is performed in the first 2 dimensions to retain their size after the convolution. The second convolutional layer consists of 4 input channels, 8 output channels, and a kernel of size $(5, 5, 1)$. Moreover, $padding = (2, 2, 0)$ is performed. The third convolutional layer consists of 8 input channels, 16 output channels, and a kernel of size $(5, 5, 1)$. Moreover, $padding = (2, 2, 0)$ is performed. The fourth convolutional layer consists of 16 input channels, 32 output channels, and a kernel of size $(5, 5, 1)$. Moreover, $padding = (2, 2, 0)$ is performed. The fifth convolutional layer consists of 32 input channels, 64 output channels, and a kernel of size $(5, 5, 1)$. Moreover, $padding = (2, 2, 0)$ is performed. The sixth convolutional layer consists of 64 input channels, 128 output channels, and a kernel of size $(5, 5, 1)$.

Regarding the number layout, the first convolutional layer consists of 1 input channel, 4 output channels, and a kernel of size $(5, 5)$. Moreover, $padding = (2, 2)$ is performed to retain the size after the convolution in both dimensions. The second convolutional layer consists of 4 input channels, 8 output channels, and a kernel of size $(5, 5)$. Moreover, $padding = (2, 2)$ is performed. The third convolutional layer consists of 8 input channels, 16 output channels, a kernel of size $(5, 5)$, and $padding = (2, 2)$ is also performed. The fourth convolutional layer consists of 16 input channels, 32 output channels, and a kernel of size $(5, 5)$. Moreover, $padding = (2, 2)$ is performed. The fifth convolutional layer consists of 32 input channels, 64 output channels, and a kernel of size $(5, 5)$. Moreover, $padding = (2, 2)$ is performed. The sixth convolutional layer consists of 64 input channels, 128 output channels, and a kernel of size $(5, 5)$.

The results from the convolutional layers, along with the road and settlement encodings, are then concatenated into a layer of 2808 neurons. The first fully connected hidden layer that follows has 4096 neurons, the second hidden layer has 1024 neurons, the third hidden layer has 256 neurons, and the fourth hidden layer has 64 neurons.

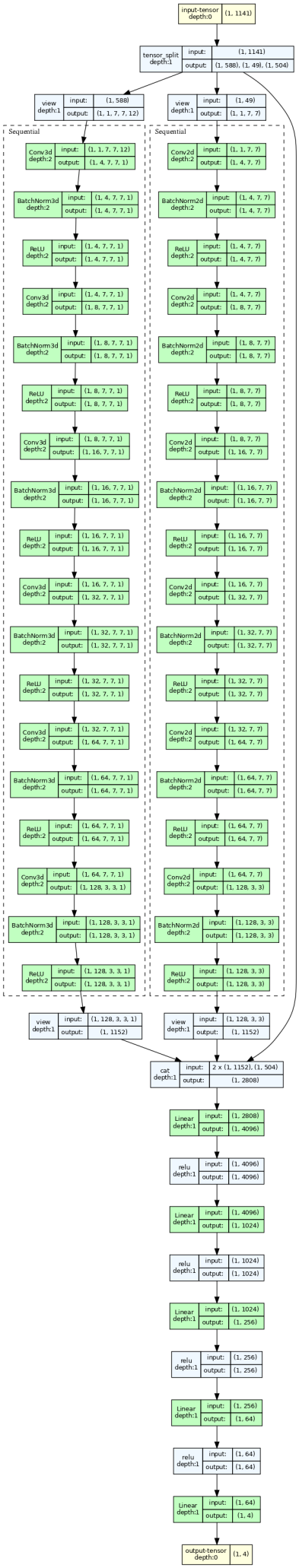


Figure 4.6: Large CNN

4.5 Training

As described in Section 4.2, the intuition behind training the neural network to predict the final scores, is that it hopefully learns: $E[s_1, s_2, s_3, s_4 | \textit{initial placement}]$, which represents the expected score for each player given the initial placement.

For the training process of the neural network, we followed an 80-10-10 split ratio, randomly allocating 80% of the dataset for the training set, 10% for the validation set, and 10% for the test set.

Throughout our empirical results, inference on the test set resulted in practically the same loss as inference on the validation set. This is perhaps due to the large amount of data that was used to train our model. In any case, we do not report any test set results and the reader can safely assume that they do not differ significantly from the validation set results.

As a loss function throughout all training experiments, we used the MSE, which is defined at the end of Section 3.2. We treated the scores in the output not as labels that need to be classified, but as float numbers. The reason for that is that they have interpretable order (e.g. a score of 4 is less good than a score of 7). Training the DNN in our case, constitutes a regression problem.

Regarding the hardware, we used an *"NVIDIA GeForce RTX 2080 Ti GPU"*, an *"NVIDIA GeForce RTX 1080 Ti GPU"*, and an *"AMD Ryzen 5 7600X CPU"*, often running in parallel. Some of the experiments were executed using PyTorch 1.12.1 and the rest were executed using PyTorch 2.0.1. For all of the experiments, Python 3.10 was used.

In Subsection 4.5.1 we present our strategy for selecting a model architecture, and in Subsection 4.5.2 we discuss the hyperparameter tuning process in detail.

4.5.1 Selecting the Most Suitable Model Architecture

Generally speaking, there is a trade-off between training speed and model efficiency. Put simply, a model with very few parameters can be trained very quickly

but it will lack the ability to effectively learn the training data. On the other hand, a very large model can learn the training data effectively, but it requires a long time to train. In order to find a sweet spot between these two extremes, we are interested in a model architecture that can learn the training data at the point where overfitting begins, while having as few parameters as possible. When we select the final model parameters, we consider the best epoch in the validation set where overfitting has not begun.

To compare the 6 different architectures from Section 4.4, we need a benchmark. For this purpose, we trained each architecture in all datasets $\mathcal{D}_1, \mathcal{D}_2,$ and$\mathcal{D}_3$. We used the Adam Optimizer with initial *learning rate* of 0.001 and a *batch size* of 1024. The results are shown in Figures 5.1, 5.2, 5.3.

We observe that all six of our model architectures are capable of learning the training data. Consequently, we ought to choose the architecture with the least amount of parameters. That architecture is the small CNN with $1,401,460$ parameters, which is shown in Figure 4.4.

This concludes our straightforward approach concerning the selection of a model architecture. In the next Subsection, we will describe our strategy for hyperparameter tuning that aims to maximize the accuracy of the chosen architecture in the validation set.

4.5.2 Hyperparameter Optimization

Stochastic Gradient Descent

As a starting point, we experimented with hyperparameter optimization using the SGD optimizer [45]. We chose a fixed *batch_size* of 1024, a fixed number of *epochs* of 100 and tuned the *learning_rate*, the *weight_decay*, and the *momentum*.

Initially, we performed a grid search with the following values:

- *learning_rate* $\in \{0.0001,\ 0.0005,\ 0.001,\ 0.005,\ 0.01\}$
- *weight_decay* $\in \{0,\ 0.0001,\ 0.0005,\ 0.001,\ 0.005\}$
- *momentum* $\in \{0,\ 0.2,\ 0.4,\ 0.6,\ 0.8,\ 0.9,\ 0.95,\ 0.99\}$

Our goal is to choose a combination of values in the hyperparameter space: *learning_rate* $\times$ *weight_decay* $\times$ *momentum*, to maximize the accuracy in the output space: $[0, 1]$.

There are a total of $5 \cdot 5 \cdot 8 = 200$ configurations in the hyperparameter space. Results are shown in Figure 5.4. The highest accuracy we got is 57.983%, with *learning_rate* $= 0.001$, *weight_decay* $= 0.005$, and *momentum* $= 0.95$.

After performing a grid search, which is not the most efficient way to optimize hyperparameters, we experimented with Bayesian optimization which is mainly attributed to Jonas Mockus [46]. Thankfully, this optimization technique is already implemented in wandb.ai, so all we need to do is define the hyperparameter space and let the algorithm work until the accuracy stops improving.

We used the 200 already created experiments and fed them into the algorithm before running an additional 400 experiments. The much larger and continuous hyperparameter space is defined by:

- *learning_rate* $\in [0.01,\ 0.0001]$
- *weight_decay* $\in [0.01,\ 0.0001]$
- *momentum* $\in [0,\ 0.99]$

We have a total of $200 + 400 = 600$ experiments which is a very small subset of the infinitely large hyperparameter space. Nevertheless, Bayesian optimization instills confidence that the accuracy cannot be significantly improved by choosing any other point in the hyperparameter space. Results are shown in Figure 5.5. The highest accuracy we got is 58.072%, with *learning_rate* $= 0.00447$, *weight_decay* $= 0.00485$, and *momentum* $= 0.66834$.

After the extensive hyperparameter space search, we were able to find a subspace where the final accuracy is the highest by filtering the sweep as shown in

Figure 5.5b. We proceeded to run a random search of 100 experiments in the empirically optimal subspace:

- $learning\_rate \in [0.002,\ 0.006]$
- $weight\_decay \in [0.003,\ 0.0055]$
- $momentum \in [0.65,\ 0.9]$

We were not able to increase the accuracy any further as shown in Figure 5.6. The highest accuracy we got is 58.065%, with $learning\_rate = 0.00281$, $weight\_decay = 0.00529$, and $momentum = 0.81512$.

Finally, we tried one more approach to increasing the accuracy even further. We took the hyperparameters that yielded the highest accuracy so far, specifically $learning\_rate = 0.00447$, $weight\_decay = 0.00485$, $momentum = 0.66834$ and ran 100 additional experiments where the only variable is the random initialization of weights and biases that is automatically done by PyTorch. Surprisingly, we were not able to increase the accuracy any further with this approach either. The results are shown in Figure 5.7. The highest accuracy we were able to get is 58.054%.

Adaptive Momentum Estimation

After extensive testing with the use of the SGD optimizer, we proceeded to experiment with the Adam optimizer [47]. Throughout the period of one month, we ran seven different hyperparameter optimization sweeps which can be found here: https://wandb.ai/diamantis-rafail-papadam/thesis_adam_2m_hpo/sweeps, with a fixed number of *epochs* equal to 50. Since we did not follow a particularly systematic approach, we had to bring everything together and consolidate our findings.

For this purpose, we created a new sweep that contains all of these 939 total experiments and ran an additional 561 experiments using Bayesian optimization. The hyperparameter space is defined by:

- $batch\_size \in \{64,\ 128,\ 256,\ 512,\ 1024,\ 2048\}$
- $learning\_rate \in [0,\ 0.1]$
- $weight\_decay \in [0,\ 0.1]$
- $dropout \in [0,\ 0.9]$

We ought to mention that the Adam optimizer automatically adjusts the *learning_rate*, hence when we report setting the *learning_rate* to some value, we are actually referring to the initial learning rate with which the algorithm begins. There are a total of $939 + 561 = 1500$ experiments. Results are shown in Figure 5.8. The highest accuracy we got is 58.285%, with $batch\_size = 256$, $learning\_rate = 0.00014$, $weight\_decay = 0.00087$, and $dropout = 0.33129$.

By filtering the experiments, as shown in Figure 5.8b, we found the empirically optimal subspace:

- $batch\_size = 256$
- $learning\_rate \in [0.000005,\ 0.0002]$
- $weight\_decay \in [0.0007,\ 0.001]$
- $dropout \in [0.3,\ 0.5]$

Similarly to our strategy with the SGD optimizer, we ran 100 experiments by sampling the hyperparameter space uniformly at random. As shown in Figure 5.9, we were able to slightly increase the accuracy to 58.292%, with $batch\_size = 256$, $learning\_rate = 0.00018$, $weight\_decay = 0.00075$, and $dropout = 0.44091$.

Finally, we wanted to find out whether we could push the accuracy a bit further. Hence, we froze the hyperparameters to $batch\_size = 256$, $learning\_rate = 0.00018$, $weight\_decay = 0.00075$, $dropout = 0.44091$, which are the values that yielded the best result thus far, and ran 100 experiments with random weight and bias initializations. The results are illustrated in Figure 5.10. The ultimate accuracy we achieved is 58.333%. Hopefully, an agent equipped with this pre-trained DNN can beat the plan-based heuristic initial settlement placement of JSettlers version 2.6.10 agents [9] which, as discussed in Subsection 3.3.2, constitute an improvement on Thomas' Ph.D. thesis [29]. This comparison will take place in Section 5.3.

4.6 Integration of CNN with Max^n

In this Section, we present the integration of our trained CNN evaluator with the Max^n algorithm. In Subsection 4.6.1, we showcase the communication process between the CNN and Max^n, and in Subsection 4.6.2, we present the complete algorithm that combines the two.

4.6.1 Socket Creation

As specified in Table 4.2, we preprocess $37 + 37 + 72 + 54 = 200$ "primitive" features. By using one-hot encoding and numerical transformations, we increase the features to 1141 as shown in Table 4.3.

To allow the Max^n algorithm [10] to communicate with our CNN, we created a socket. Thus, when a tree leaf is reached by Max^n in Java, the completed initial phase game state, composed of 200 features, is sent to a Python server through port 1999 of localhost.

The Python server preprocesses these features to create the CNN input of 1141 features and runs a forward pass. Once the forward pass is complete, it sends the result of 4 float numbers, representing the expected final scores for each player, back to the Java Max^n algorithm for backpropagation.

4.6.2 Our Final Settlers of Catan Agent

Before evaluating our final agent, which is named "Diamantis", we present the pseudocode which constitutes the core of its decision-making process:

Algorithm 1: Max^n with CNN as Evaluation Function

```
Input: int settlements_rem, int depth, SOCGame game
Output: int[] scores, int move

Function maxn(settlements_rem, depth, game):
    int player ← −1;
    if settlements_rem = 0 then
        return {evaluateBoard(game), null};
    else if settlements_rem = 8 or settlements_rem = 1 then
        player ← 0;
    else if settlements_rem = 7 or settlements_rem = 2 then
        player ← 1;
    else if settlements_rem = 6 or settlements_rem = 3 then
        player ← 2;
    else
        player ← 3;
    end

    // Getting a list of legal settlement spots for player.
    settlement[] potential ← game.getLegalSettlements(player);
    // Sorting & pruning moves with a heuristic function.
    settlement[] moves ← prunePotential(potential, depth, player, game);

    int best ← −1;
    int[] scores ← {0, 0, 0, 0};
    settlement move ← null;
    for settl in moves do
        game.putPiece(settl);
        // JSettlers heuristic to decide where to build road.
        road = planRoad(player);
        game.putPiece(road);

        local_score = maxn(settlements_rem − 1, depth + 1, game)[0];

        game.removePiece(road);
        game.removePiece(settl);
        if local_score[player] > best then
            best ← local_score[player];
            scores ← local_score;
            move ← settl;
        end
    end
    return {scores, move};
end
```

Algorithm 2: Evaluate Completed Initial Settlement Placement with CNN

Input: SOCGame *game*
Output: int[] *scores*

Function *evaluateBoard*(*game*)**:**

```
int[] scores ← new int[4];
int[] NN_input ← new int[200];
NN_input[0 : 37] ← game.getHexLayout();
NN_input[38 : 74] ← game.getNumLayout();
NN_input[75 : 146] ← game.getRoads();
NN_input[147 : 200] ← game.getSettlements();
// Send data through socket and receive evaluation.
float[] NN_output ← send_data_to_NN(NN_input);
for i ← 0 to 3 do
    // We consider 5 decimal digits.
    scores[i] ← (int)(100000 · NN_output[i]);
end
return scores;
```

end

Algorithm 3: Prune Potential Settlements

Input: settlement[] *potential*, int *depth*, int *player*, SOCGame *game*
Output: settlement[] *pruned*

Function *prunePotential*(*potential*, *depth*, *player*, *game*)**:**

```
int size ← min( ⌈30 / depth^1.46⌉, potential.size() );
Hashtable<settlement, int> nodes;
for settl in potential do
    nodes.add(settl, 0);
end
// JSettlers heuristic for settlement evaluation.
scoreNodesForSettl(nodes, player, game, 10, 5, 10);
// Keep size best settlement spots in sorted order.
settlement[] pruned ← sort_and_prune(nodes, size);
return pruned;
```

end

Algorithm 4: JSettlers Heuristic to Score Intersections for Settlement Placement

```
Input: Hashtable<settlement, int> nodes, int player, SOCGame game,
       int numWeight, int miscPortWeight, int portWeight
Output: void
Function scoreNodesForSettl(nodes, player, game, numWeight, miscPortWeight, portWeight):
    // score according to numbers.
    int[] numRating ← {0, 0, 3, 6, 8, 11, 14, 17, 14, 11, 8, 6, 3};
    SOCBoard board ← game.getBoard();
    int maxScore ← 80;
    Enumeration<int> keys ← nodes.keys();
    while keys.hasMoreElements() do
        int key ← keys.nextElement();
        int oldScore ← nodes.get(key);
        int score ← 0;
        for hex in board.getAdjacentHexes(key) do
            int num = board.getNumFromHex(hex);
            score ← numRating[num];
            if player.hasNumber(num) = false then
                score ← score + numRating[num];
            end
        end
        int newScore ← numWeight · ((100 · score)/maxScore);
        nodes.put(key, oldScore + newScore);
    end
    // score according to 3:1 ports.
    keys ← nodes.keys();
    if player.hasSettlAtMiscPort() = false then
        while keys.hasMoreElements() do
            int key ← keys.nextElement();
            int oldScore ← nodes.get(key);
            int score ← 0;
            if board.isMiscPort(key) then
                score ← 100;
            end
            int newScore ← miscPortWeight · score;
            nodes.put(key, oldScore + newScore);
        end
    end
    // score according to 2:1 ports.
    // JSettlers function to calculate rarity for each of the 5 resources.
    int[] resourceEstis = estimateResourceRarity();
    for int portType ← SOCBoard.CLAY_PORT to SOCBoard.WOOD_PORT do
        keys ← nodes.keys();
        if player.hasSettlAtPort(portType) = false and resourceEstis[portType] > 33 then
            int estimatedPortWeight ← (resourceEstis[portType] · portWeight)/56;
            while keys.hasMoreElements() do
                int key ← keys.nextElement();
                int oldScore ← nodes.get(key);
                int score ← 0;
                if board.isPort(key) then
                    score ← 100;
                end
                int newScore ← estimatedPortWeight · score;
                nodes.put(key, oldScore + newScore);
            end
        end
    end
end
```

Algorithm Explanation

Whenever a leaf node is reached, Algorithm 1 calls Algorithm 2 for initial placement evaluation by the neural network. On the other hand, when Algorithm 1 is in a non-leaf node, it calls Algorithm 3 to prune the Max^n tree according to the current depth and recursively moves to the next settlement placement nodes. Let us take a more detailed look at each algorithm.

Algorithm 1, which uses Algorithms 2-4 as sub-routines, begins by checking the number of settlements that are left to be placed. If this number is zero, then it calls Algorithm 2 which performs the evaluation with the use of our trained CNN. On the other hand, if this number is above zero, it figures out which player's turn it is. Furthermore, it retrieves a list of all available settlement placements for that player and calls Algorithm 3 to prune those settlement placements according to the current depth. In its main part, it recursively calls itself for each availability in the pruned options and chooses the settlement placement that maximizes the score for the current player. Finally, it returns the four scores, one for each player, that resulted from the chosen settlement placement along with that placement.

Algorithm 2, creates an array with the neural network input before preprocessing, as shown in the first 4 rows of Table 4.2, and sends it to a python server through localhost, as described in Subsection 4.6.1. The server preprocesses the input, performs the forward pass, and responds with the result which is an array of the 4 predicted float scores. Finally, Algorithm 2, multiplies each element with 100000 and keeps the integer part, thus accounting for 5 decimal digits. The computed array of 4 integers after the multiplication, is returned by the initial settlement placement evaluation algorithm.

Algorithm 3, begins by computing the size of the pruned settlement placement options. This size is given by $\lceil \frac{30}{d^{1.46}} \rceil$, which results in a pruned size of 30, 11, 7, 4, 3, 3, 2, 2, for $d = 1, 2, 3, 4, 5, 6, 7, 8$ respectively. In reality, we

take the $min\left(\lceil\frac{30}{d^{1.46}}\rceil, potential.size()\right)$, in case the enumerator is chosen to be greater than 54, which is the number of available intersections when the board is empty. Moreover, a hashtable where each potential settlement is mapped to an initial score of 0 is created, and Algorithm 4 is called to perform the scoring. Furthermore, a trivial function that we do not present, is used to sort the potential settlement placements according to the calculated scores and return the *size* options with the highest scores. Finally, the array of the *size* most promising initial settlement placement options is returned by our algorithm.

Algorithm 4, is a built-in algorithm in the JSettlers2 framework [9] which we made use of. Initially, it calculates the score of potential settlement placements according to the hex numbers adjacent to them. Notice that the *numRating* array corresponds to the probabilities at the end of Subsection 2.4.2 multiplied by 100. Furthermore, if the *player* does not already have a settlement adjacent to a 3:1 port, the algorithm increases the score for each potential settlement placement that is adjacent to such a port. Finally, for each 2:1 port, if the player does not have a settlement adjacent to that port and the resource rarity that is specified by the port type is greater than the threshold of 33, the potential settlement placements that are adjacent to that port have their scores increased.

Algorithm Details & Result Preview

After careful consideration of preliminary empirical results that we do not share in this thesis, as well as inspiration from [12], we decided to use our algorithm only for the second settlement placement. The reason is that in the first settlement placement, the state space is huge; hence our pruning heuristic, which is very simplistic in our approach, plays a very important role. Conversely, when the state space is smaller, there is less pruning; hence we do not heavily rely on the effectiveness of the pruning heuristic.

By running our algorithm only for the second settlement placement, we were able to beat the JSettlers version 2.6.10 plan-based heuristic settlement placement [9], which we safely assumed to be at least as good as the one in [43], as explained in the end of Subsection 3.3.2. To clearly illustrate our agent's dominance and make it evident that it does not simply exploit a weakness of JSettlers agents, we also incorporated Random agents in the final evaluation, as we will see in Subsection 5.3.

Regarding the speed of our algorithm, it is able to evaluate approximately 450 leaf nodes per second. This may sound slow but keep in mind that the bottleneck is the forward pass through the CNN which is done by an *NVIDIA GeForce RTX 2080 Ti*, as well as the fact that Max^n communicates with the CNN via a localhost socket as explained in Subsection 4.6.1. In most Catan tournaments, players have 1-3 minutes per initial settlement placement. Nevertheless, our algorithm is capable of reaching a decision within 21 seconds in the worst case.

We can easily calculate the runtime of our approach since we know the number of leaf nodes by the heuristic pruning shown in Algorithm 3, and we also know that approximately 450 leaf nodes can be evaluated per second. Since we employ our method only for the second settlement placement, there are four possible cases as presented below:

- We are the 1^{st} player, hence we place last in the second round.
  - $\#\text{leaf\_nodes} = \prod_{d=1}^{1} \lceil \frac{30}{d^{1.46}} \rceil = 30$
  - $\text{runtime} = \frac{30 \text{ leaf nodes}}{450 \text{ leaf nodes per second}} \approx 67ms$
- We are the 2^{nd} player, hence we place third in the second round.
  - $\#\text{leaf\_nodes} = \prod_{d=1}^{2} \lceil \frac{30}{d^{1.46}} \rceil = 30 \times 11 = 330$
  - $\text{runtime} = \frac{330 \text{ leaf nodes}}{450 \text{ leaf nodes per second}} \approx 733ms$

- We are the 3^{rd} player, hence we place second in the second round.
  - $\#\text{leaf\_nodes} = \prod_{d=1}^{3} \lceil \frac{30}{d^{1.46}} \rceil = 30 \times 11 \times 7 = 2310$
  - $\text{runtime} = \frac{2310 \text{ leaf nodes}}{450 \text{ leaf nodes per second}} \approx 5.1s$
- We are the 4^{th} player, hence we place first in the second round.
  - $\#\text{leaf\_nodes} = \prod_{d=1}^{4} \lceil \frac{30}{d^{1.46}} \rceil = 30 \times 11 \times 7 \times 4 = 9240$
  - $\text{runtime} = \frac{9240 \text{ leaf nodes}}{450 \text{ leaf nodes per second}} \approx 20.5s$

We showcase our empirical results with the chosen DNN architecture (i.e. Small CNN shown in Figure 4.4) trained with two different methods. Results using the second-best CNN training session with the SGD optimizer are shown in Figure 5.11. Results using the best CNN training session with the Adam optimizer are shown in Figure 5.12. The former can be found in this link: https://wandb.ai/diamantis-rafail-papadam/thesis_sgd_2m_hpo/runs/gu4mfgqk, while the latter can be found here: https://wandb.ai/diamantis-rafail-papadam/thesis_adam_2m_hpo/runs/hgz7c9k6. By using either trained neural network, our agent's dominance is illustrated in the aforementioned figures.

The final results can be downloaded, in the form of *".soclog"* files, from the following Google Drive: https://drive.google.com/drive/folders/1f_hT1byUH9VMt9amPhFCZh7-FVAclCgA along with a Python script that can be used to parse them and print the win ratios along with the average scores for each player.

Chapter 5

Results

In this Chapter, we illustrate our empirical results and make observations that highlight important insights. Specifically, in Section 4.5.1, we choose the most suitable model architecture based on our empirical results. In Section 4.5.2, we perform hyperparameter optimization on the chosen architecture with an SGD optimizer and an Adam optimizer. Finally, in Section 5.3, we integrate two ultimate neural network evaluators, one trained with SGD and one trained with Adam, to the Max^n algorithm and derive the final results which we compare with the state-of-the-art.

5.1 Experimental Selection of the Most Appropriate Network Architecture

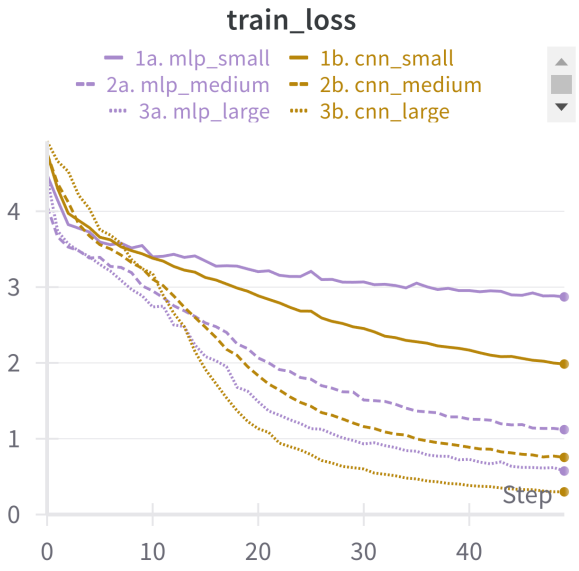
In Section 4.4, we presented six architectures which we put to the test and compared empirically in Figures 5.1 - 5.3, where the horizontal axis represents the epochs and the vertical axis represents the figure caption. By looking at these figures we can make some remarkable observations that are dataset-independent. As seen in Figures 5.1a, 5.1c, 5.2a, 5.2c, 5.3a, 5.3c, when the network parameters are increased, the DNN learns the training data in less epochs. However, by looking at Figures 5.1b, 5.1d, 5.2b, 5.2d, 5.3b, 5.3d, we see that the performance of all of the architectures peaks at more or less the same loss and accuracy in the validation set.

Ideally, we hoped that at least one architecture would fail to learn the training data, for example, the small MLP. Had this been the case, we would present two extremes: the inability to learn the training data on one end (small MLP), and the severe overfitting on the other end (large CNN). In the former case, it would

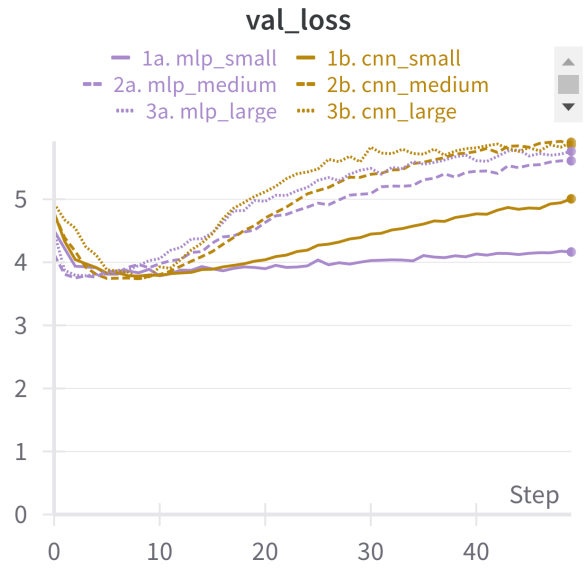
be quick to make a forward pass on the network but the accuracy would be low. In the latter case, assuming that we stopped the training early enough, the accuracy would be high but the inference time would be longer than necessary. However, we were not able to experimentally show both extremes because even the small MLP was able to begin overfitting in all datasets. It seems that the large amount of training data enabled even a simple architecture such as the small MLP, which is depicted in Figure 4.1, to be effective. In any case, this does not affect our findings but simply makes our empirical results less general.

As a final observation, we notice that the small CNN uses fewer parameters than the small MLP, the medium CNN uses fewer parameters than the medium MLP, and the large CNN uses fewer parameters than the large MLP. Nevertheless, by looking at Figures 5.1a, 5.1c, 5.2a, 5.2c, 5.3a, 5.3c, it is evident that all CNNs learn faster than the corresponding MLPs which have more parameters. This can be attributed to the fact that hex and number patterns are location invariant in SoC, which enables the weight-sharing property of CNNs to be effective. An additional advantage of the CNNs over the MLPs is that they allow the GPU to efficiently parallelize computations and perform a forward pass more quickly.

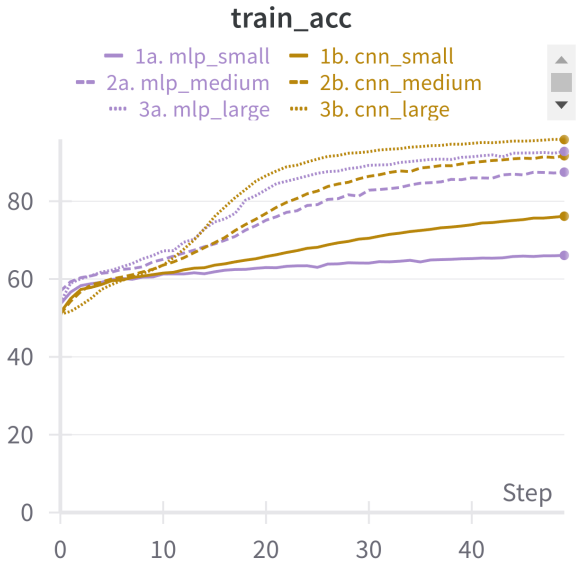
With this extensive assessment in mind, as we also mentioned in Subsection 4.5.1, we chose the small CNN as the most suitable architecture to perform hyper-parameter optimization and further improve our baseline results.



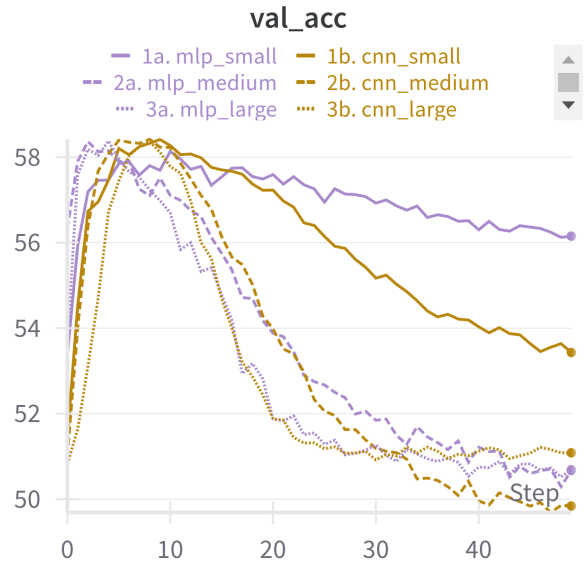
(a) Training Loss



(b) Validation Loss



(c) Training Accuracy (%)



(d) Validation Accuracy (%)

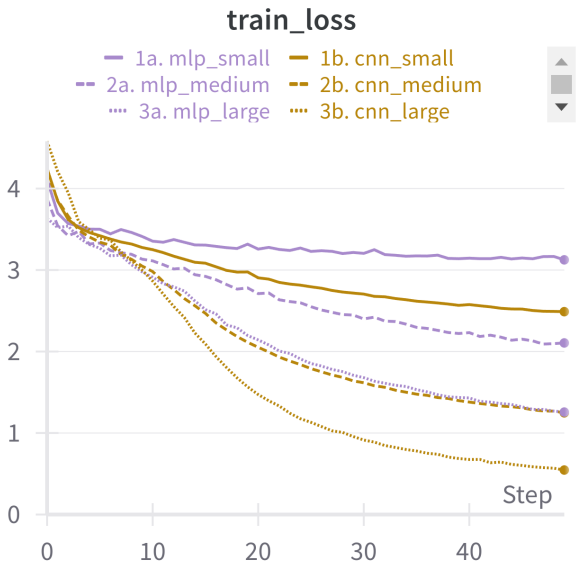
Figure 5.1: DNN Comparison in Dataset $\mathcal{D}_1$

$optimizer$ = Adam

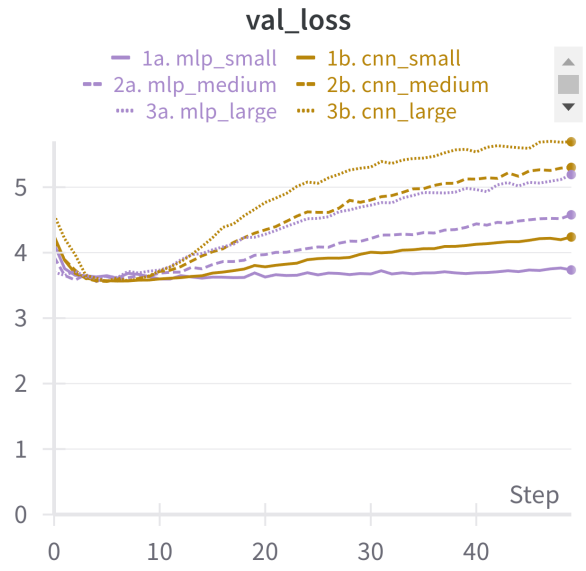
$batch\_size$ = 1024

$initial\_learning\_rate$ = 0.001

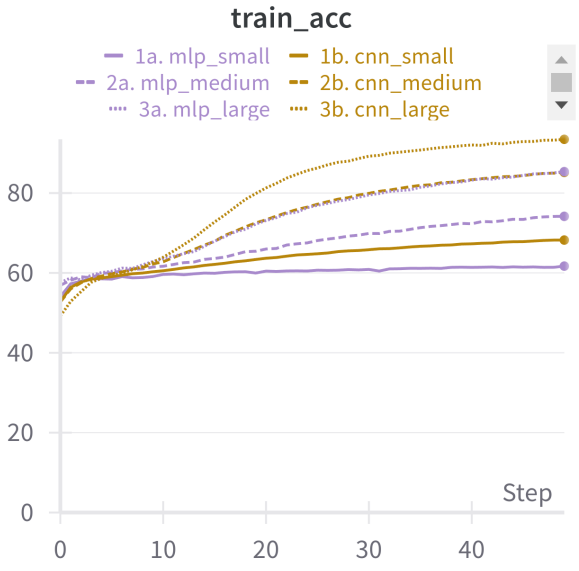
https://wandb.ai/diamantis-rafail-papadam/thesis_adam_1m/workspace



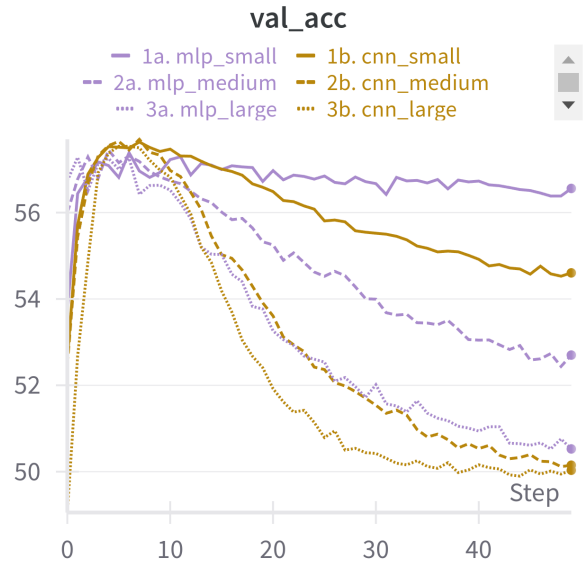
(a) Training Loss



(b) Validation Loss



(c) Training Accuracy (%)



(d) Validation Accuracy (%)

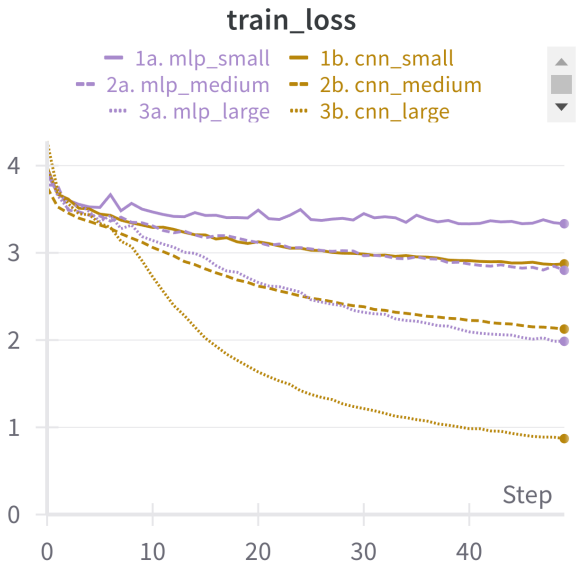
Figure 5.2: DNN Comparison in Dataset $\mathcal{D}_2$

$optimizer$ = Adam

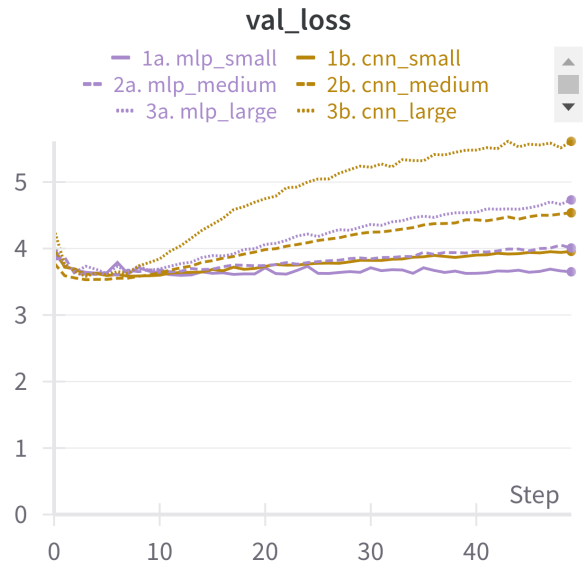
$batch\_size$ = 1024

$initial\_learning\_rate$ = 0.001

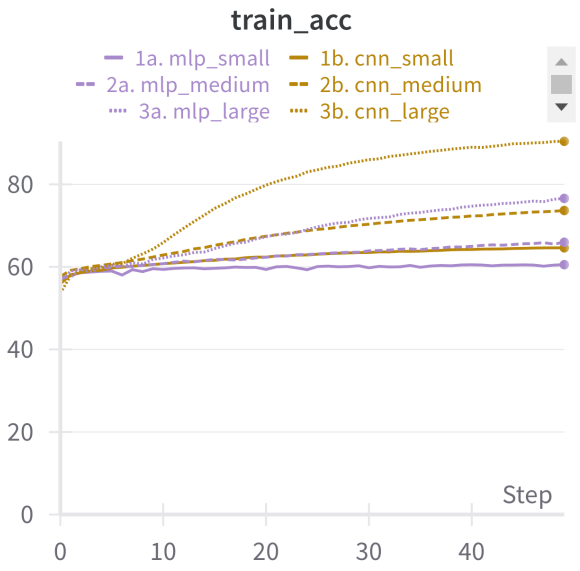
https://wandb.ai/diamantis-rafail-papadam/thesis_adam_2m/workspace



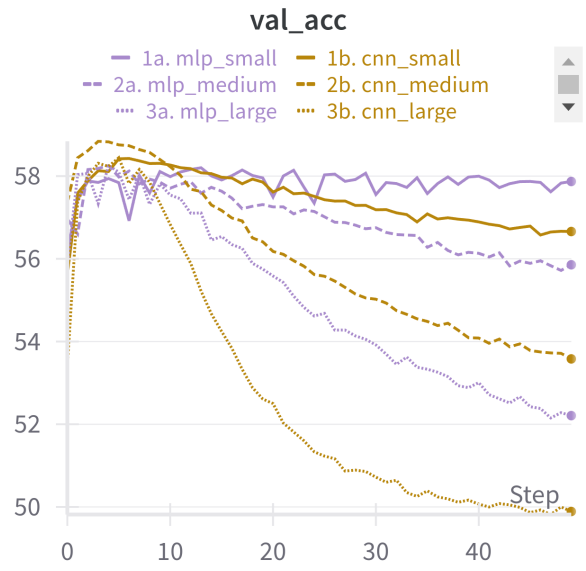
(a) Training Loss



(b) Validation Loss



(c) Training Accuracy (%)



(d) Validation Accuracy (%)

Figure 5.3: DNN Comparison in Dataset $\mathcal{D}_3$

$optimizer$ = Adam

$batch\_size$ = 1024

$initial\_learning\_rate$ = 0.001

https://wandb.ai/diamantis-rafail-papadam/thesis_adam_3.5m/workspace

5.2 Hyperparameter Optimization

In this Section, we will highlight important observations based on the empirical results depicted in Figures 5.4 - 5.10. In each of these Figures, we see 3 types of results:

(a) Training Date versus Highest Accuracy in the validation set.

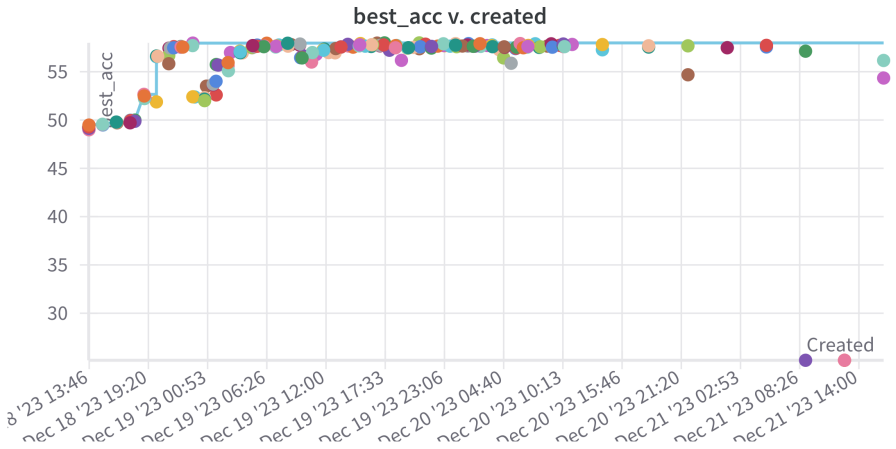
(b) Hyperparameter Space Visualization.

(c) Hyperparameter Statistics Produced by wandb.ai.

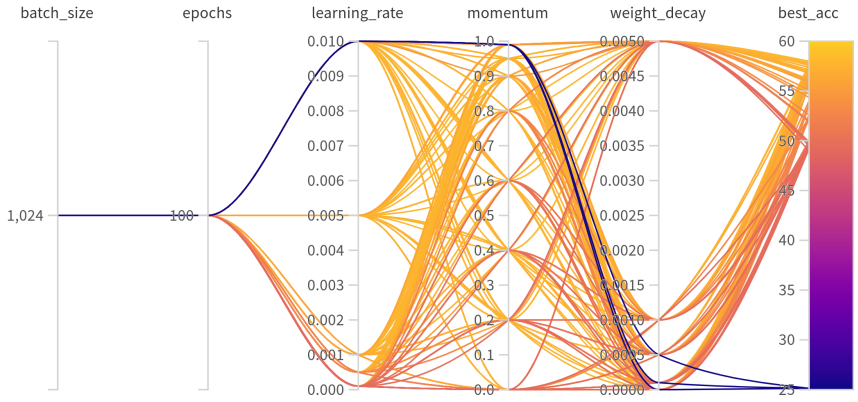
We are particularly interested in graphs of type (b), while graphs of type (c) are also useful since they quantify the importance of each hyperparameter and show how it is correlated with the highest accuracy on the validation set.

As thoroughly elaborated in Subsection 4.5.2, we performed hyperparameter optimization twice. In our first attempt, shown in Figures 5.4-5.7, we used the SGD optimizer and tuned the *learning rate*, the *weight decay*, and the *momentum*. In our second attempt, shown in Figures 5.8-5.10, we used the Adam optimizer and tuned the *batch size*, the *learning rate*, the *weight decay*, and the *dropout*.

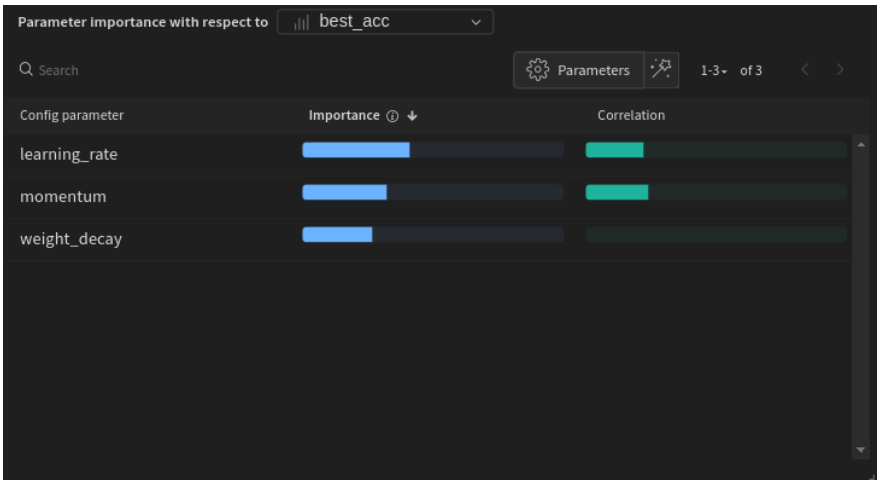
Our strategy, regardless of the optimizer, can be summarized in two steps. In the first step, we integrated any experiments that we had previously run into a Bayesian optimization algorithm within the wandb.ai framework, where we conducted several hundred additional experiments to efficiently explore the hyperparameter space. In the second step, we empirically located the neighborhood of the global maximum accuracy within the hyperparameter space. Furthermore, we deployed 200 final experiments in the narrowed-down hyperparameter space to try and push the accuracy a little higher. The first 100 were performed with random samples from the empirically global maximum neighborhood, and the latter 100 by using the best hyperparameters found in all previous trials while randomly varying the weight and bias initialization. This proved to be effective with the Adam optimizer but ineffective with the SGD optimizer.



(a) Training Date versus Highest Accuracy in the validation set



(b) Hyperparameter Space Visualization



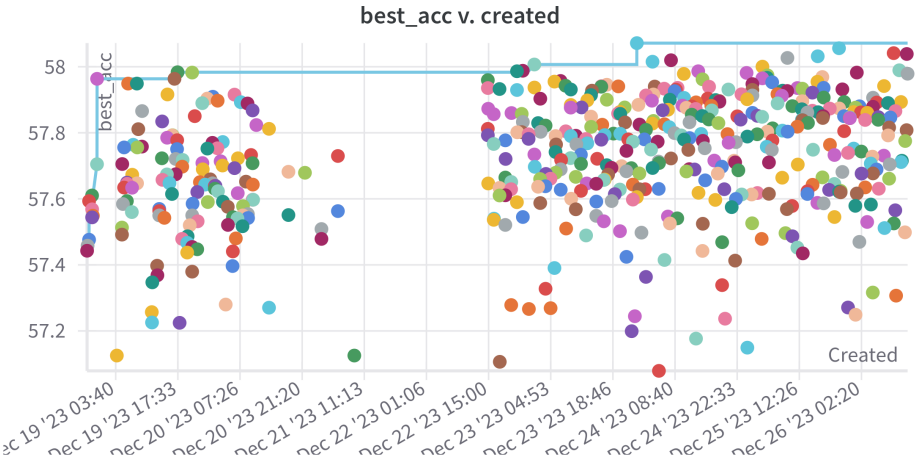
(c) Hyperparameter Statistics Produced by wandb.ai

Figure 5.4: Grid Search using Small CNN in $\mathcal{D}_2$ (SGD Optimizer)

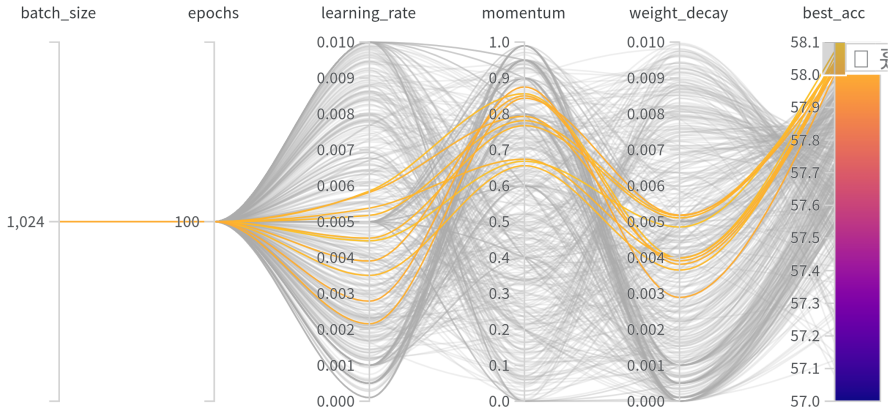
$$number\_of\_experiments = 200$$

$$best\_acc = 57.983\%$$

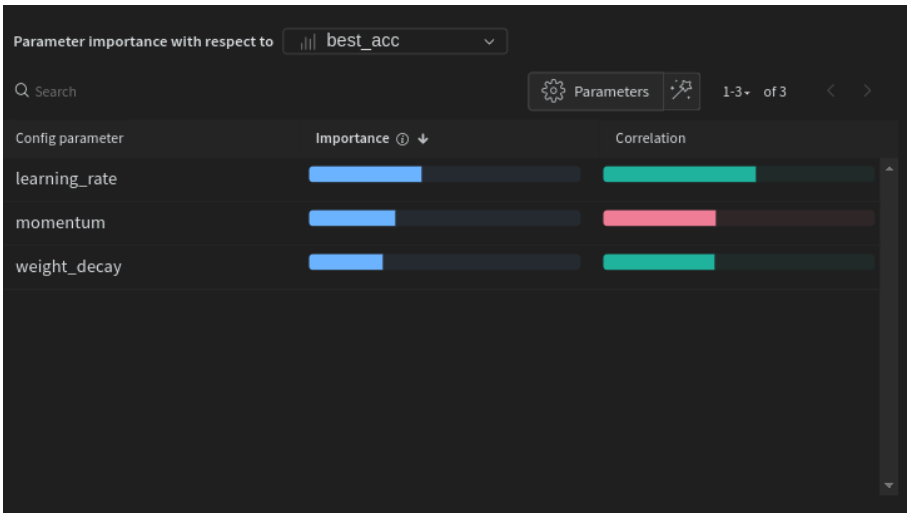
https://wandb.ai/diamantis-rafail-papadam/thesis_sgd_2m_hpo/sweeps/kb53di9s



(a) Training Date versus Highest Accuracy in the validation set



(b) Hyperparameter Space Visualization (**filter**: *best_acc* > 58)



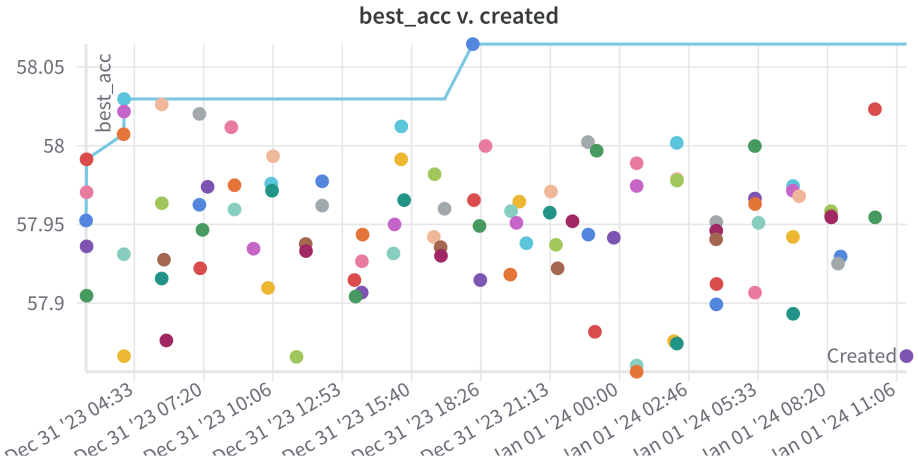
(c) Hyperparameter Statistics Produced by wandb.ai

Figure 5.5: Bayesian Search using Small CNN in $\mathcal{D}_2$ (SGD Optimizer)

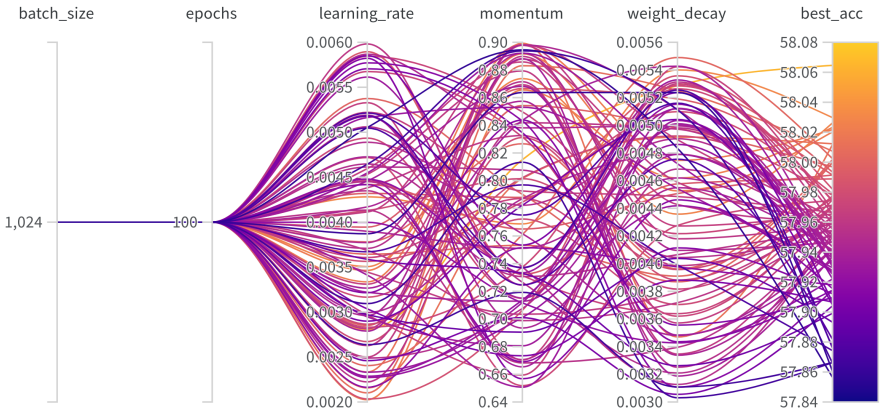
$$number\_of\_experiments = 600$$

$$best\_acc = 58.072\%$$

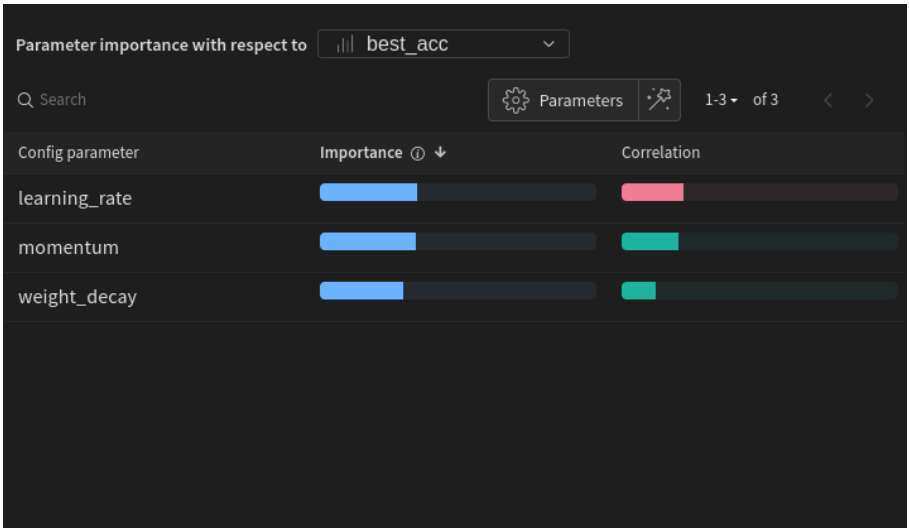
https://wandb.ai/diamantis-rafail-papadam/thesis_sgd_2m_hpo/sweeps/4fwa08e9



(a) Training Date versus Highest Accuracy in the validation set



(b) Hyperparameter Space Visualization



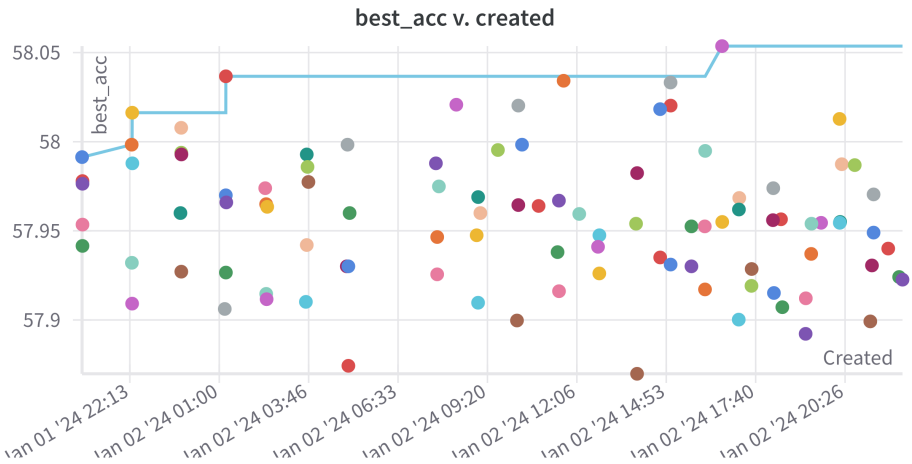
(c) Hyperparameter Statistics Produced by wandb.ai

Figure 5.6: Random Search using Small CNN in $\mathcal{D}_2$ (SGD Optimizer)

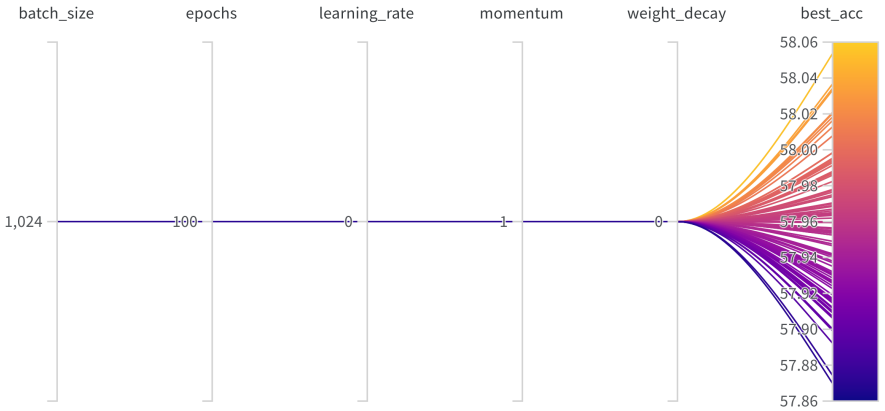
$$number\_of\_experiments = 100$$

$$best\_acc = 58.065\%$$

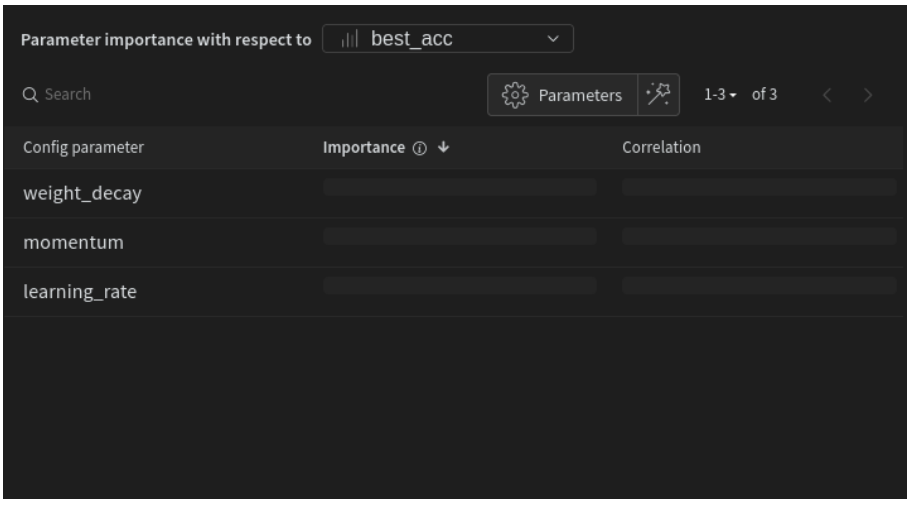
https://wandb.ai/diamantis-rafail-papadam/thesis_sgd_2m_hpo/sweeps/yo3j14bd



(a) Training Date versus Highest Accuracy in the validation set



(b) Hyperparameter Space Visualization



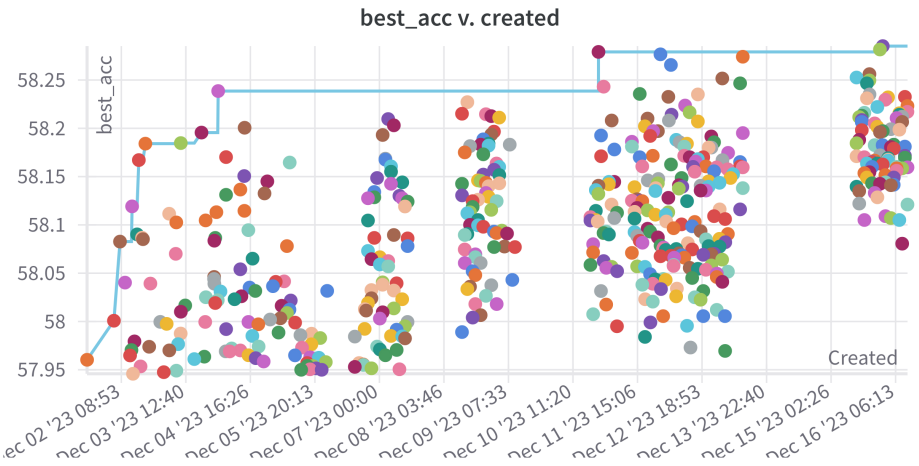
(c) Hyperparameter Statistics Produced by wandb.ai

Figure 5.7: Repetitive Search using Small CNN in $\mathcal{D}_2$ (SGD Optimizer)

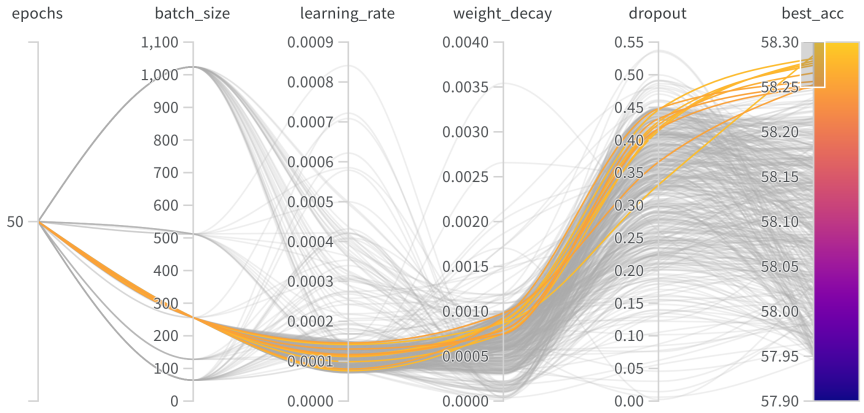
$number\_of\_experiments = 100$

$best\_acc = 58.054\%$

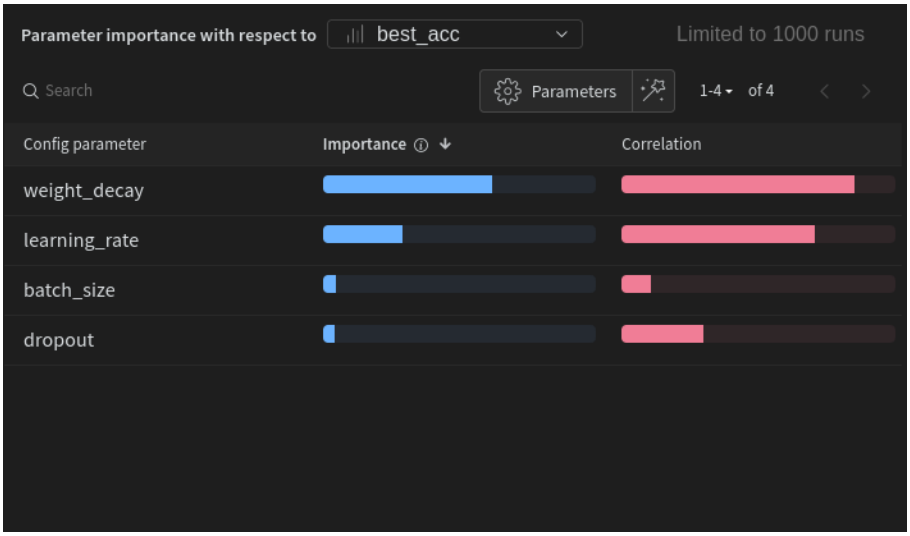
https://wandb.ai/diamantis-rafail-papadam/thesis_sgd_2m_hpo/sweeps/p6zqtyrv



(a) Training Date versus Highest Accuracy in the validation set



(b) Hyperparameter Space Visualization (**filter**: *best_acc* > 58.25)



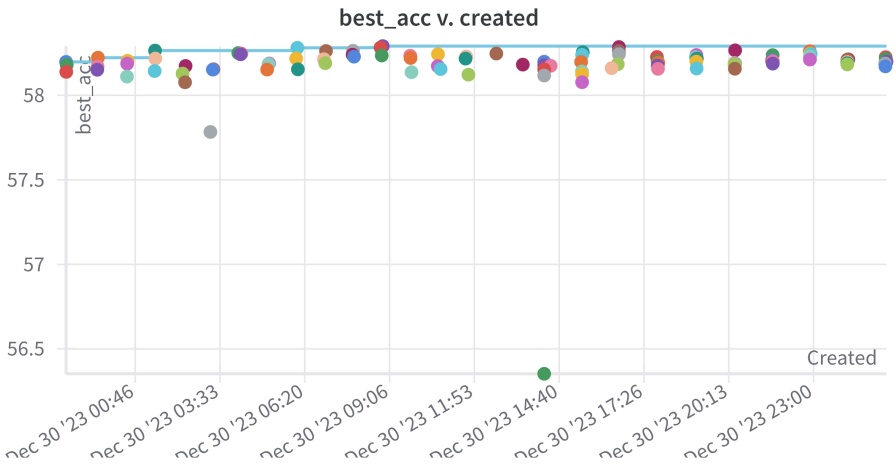
(c) Hyperparameter Statistics Produced by wandb.ai

Figure 5.8: Bayesian Search using Small CNN in $\mathcal{D}_2$ (Adam Optimizer)

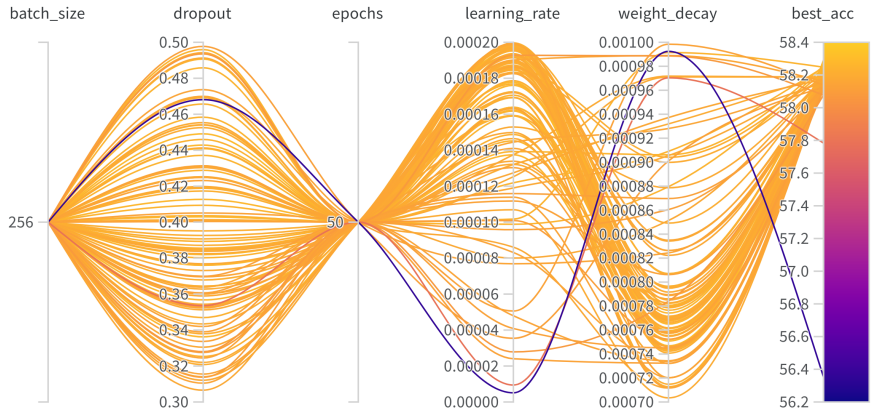
$$number\_of\_experiments = 1500$$

$$best\_acc = 58.285\%$$

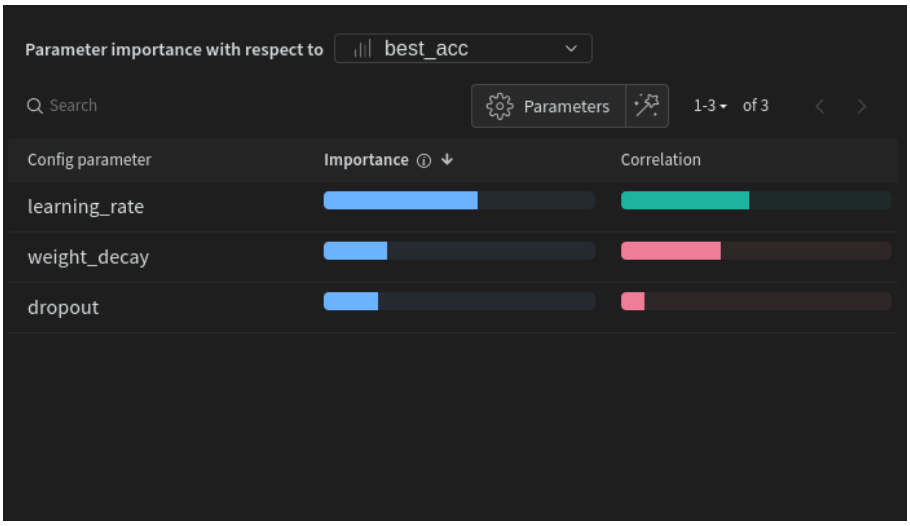
https://wandb.ai/diamantis-rafail-papadam/thesis_adam_2m_hpo/sweeps/ru18d64d



(a) Training Date versus Highest Accuracy in the validation set



(b) Hyperparameter Space Visualization



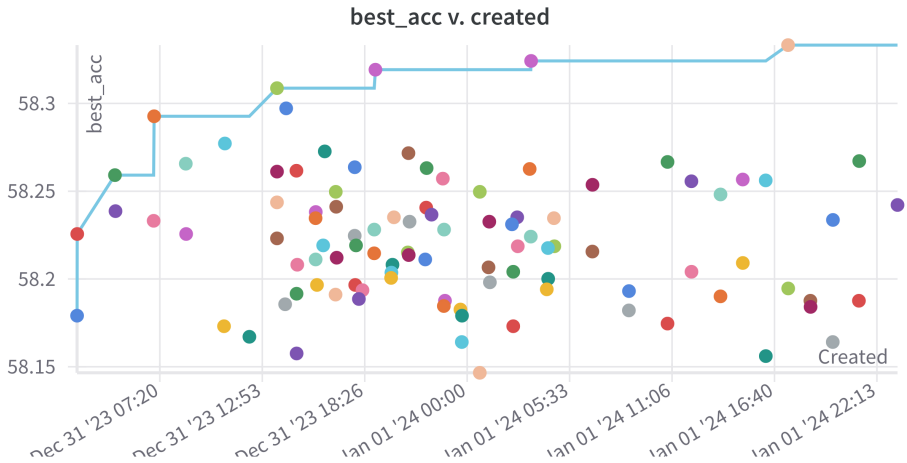
(c) Hyperparameter Statistics Produced by wandb.ai

Figure 5.9: Random Search using Small CNN in $\mathcal{D}_2$ (Adam Optimizer)

$number\_of\_experiments = 100$

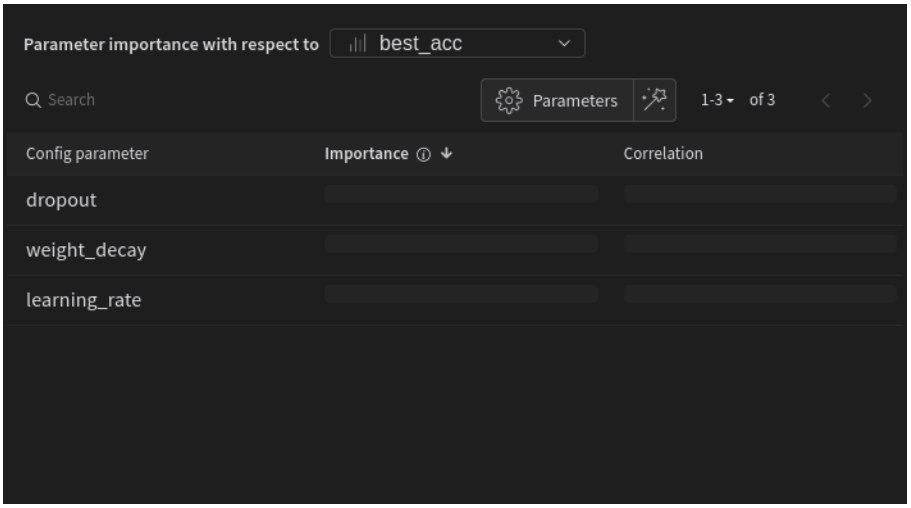
$best\_acc = 58.292\%$

https://wandb.ai/diamantis-rafail-papadam/thesis_adam_2m_hpo/sweeps/1cwb5njr



(a) Training Date versus Highest Accuracy in the validation set

(b) Hyperparameter Space Visualization



(c) Hyperparameter Statistics Produced by wandb.ai

Figure 5.10: Repetitive Search using Small CNN in $\mathcal{D}_2$ (Adam Optimizer)

$$number\_of\_experiments = 100$$

$$best\_acc = 58.333\%$$

https://wandb.ai/diamantis-rafail-papadam/thesis_adam_2m_hpo/sweeps/a9oqjql1

5.3 Final Results

To achieve the final results, we benchmarked our agent against 4 combinations of Random and JSettlers agents. As depicted in Figures 5.11, 5.12, our agent is dominant in all game formats regardless of the optimizer that was used to train the CNN evaluator.

Regarding the effectiveness of our method, we see that our agent beats the JSettlers version 2.6.10 agents [9]; hence it is also superior to the original JSettlers agents of Thomas's Ph.D. thesis [29]. The drawback of our method compared to any heuristic-based approach, is that our agent requires more time to take an action. However, the 21 seconds that our agent needs in the worst-case scenario is considered a small amount of time for initial settlement placement in human games.

Through a closer examination of our final results, we noticed something that we did not expect. As illustrated in Subsection 4.5.2 and in the links at the end of Subsection 4.6.2, the SGD optimizer achieved a lower accuracy in the validation set compared to the Adam optimizer. Nevertheless, it performs slightly better in all game formats.

After some thought, we concluded that there are two possible explanations. The first is that a decrease of $3.555 - 3.534 = 0.021$ in the loss, and an increase of $58.333\% - 58.065\% = 0.268\%$, in the accuracy of our validation set, do not translate well on new, unseen data. The second possible explanation is that we were "unlucky", and if we had run more than $10,000$ simulations per game format, the Adam-trained CNN would perform better than the SGD-trained CNN, even though that is unlikely.

Statistics over 10,000 Simulated Games		
Player	**Win Ration**	**Average Score**
Diamantis Agent	27.6%	7.65
JSettlers Agent	24.9%	7.54
JSettlers Agent	23.8%	7.47
JSettlers Agent	23.6%	7.5

(a) 1 Diamantis vs. 3 JSettlers

Statistics over 10,000 Simulated Games		
Player	**Win Ration**	**Average Score**
Diamantis Agent	35.6%	7.96
JSettlers Agent	31.6%	7.79
JSettlers Agent	30.7%	7.77
Random Agent	2.1%	4.21

(b) 1 Diamantis vs. 2 JSettlers vs. 1 Random

Statistics over 10,000 Simulated Games		
Player	**Win Ration**	**Average Score**
Diamantis Agent	49.3%	8.48
JSettlers Agent	43.5%	8.27
Random Agent	3.8%	4.54
Random Agent	3.4%	4.43

(c) 1 Diamantis vs. 1 JSettlers vs. 2 Random

Statistics over 10,000 Simulated Games		
Player	**Win Ration**	**Average Score**
Diamantis Agent	78.7%	9.54
Random Agent	7.1%	4.92
Random Agent	7.1%	4.92
Random Agent	7%	4.9

(d) 1 Diamantis vs. 3 Random

Figure 5.11: Ultimate Results of CNN evaluator trained with SGD

Statistics over 10,000 Simulated Games		
Player	**Win Ration**	**Average Score**
Diamantis Agent	27.3%	7.66
JSettlers Agent	24.7%	7.52
JSettlers Agent	24.3%	7.49
JSettlers Agent	23.7%	7.46

(a) 1 Diamantis vs. 3 JSettlers

Statistics over 10,000 Simulated Games		
Player	**Win Ration**	**Average Score**
Diamantis Agent	34.4%	7.91
JSettlers Agent	32.1%	7.82
JSettlers Agent	31.2%	7.78
Random Agent	2.2%	4.24

(b) 1 Diamantis vs. 2 JSettlers vs. 1 Random

Statistics over 10,000 Simulated Games		
Player	**Win Ration**	**Average Score**
Diamantis Agent	49.2%	8.49
JSettlers Agent	43.2%	8.29
Random Agent	3.9%	4.51
Random Agent	3.7%	4.51

(c) 1 Diamantis vs. 1 JSettlers vs. 2 Random

Statistics over 10,000 Simulated Games		
Player	**Win Ration**	**Average Score**
Diamantis Agent	77.8%	9.5
Random Agent	7.7%	4.95
Random Agent	7.3%	4.96
Random Agent	7.2%	4.96

(d) 1 Diamantis vs. 3 Random

Figure 5.12: Ultimate Results of CNN evaluator trained with Adam

In Table 5.9, we compare our agent's performance to that of the state-of-the-art, showcasing that our approach works at a reasonable speed while achieving results near the state-of-the-art. With the safe assumption made at the end of Subsection 3.3.2, that the baseline agents in the state-of-the-art [12] are in the worst-case for us, of equal strength to our baseline JSettlers version 2.6.10 agents, we derive the results that are presented in the aforementioned table. The authors of [12], achieve a 30.43% win ratio over 10000 games against 3 STAC agents by making use of a human-generated corpus of data, and a 28.74% under the same settings without making use of such data.

In our approach, by using the CNN that was trained with an SGD optimizer, we achieve a 27.6% win ratio over 10000 simulations against 3 JSettlers agents of the latest version. According to Section V of the aforementioned state-of-the-art paper, the authors performed a Z-test and showed that with 10000 simulations and $p < 0.01$, win ratios between $24 - 26\%$ are not considered significant. Therefore, our result of 27.6% means that our agent is clearly superior to the latest version of JSettlers agents. Regarding the comparison with the state-of-the-art, we stress that the main advantage of our method is that our agent's runtime is acceptable to human players.

Even though the state-of-the-art uses less capable hardware, our runtime is more than 10 times faster. In particular, the authors of [12] use an *"Intel-I5 hyper-threaded processor"*. On the other hand, we use an *"AMD Ryzen 5 7600X CPU"* for Max^n, and an *"NVIDIA GeForce RTX 2080 Ti GPU"* for the leaf node evaluation by the CNN.

Player	**Win Ratio versus Baseline**	**Reported Runtime** (not the same hardware)
Diamantis Agent	27.6%	$\approx$ 21 seconds
State-of-the-art Agent without human-generated corpus	28.74%	$\approx$ 300 seconds
State-of-the-art Agent seeded with human-generated corpus	30.43%	$\approx$ 300 seconds

Table 5.9: Comparison of our final agent to the state-of-the-art

Chapter 6
Conclusions

6.1 Summary

In this thesis, inspired by [4], we incorporated DNNs in the traditional adversarial search Max^n algorithm [10] and achieved initial settlement placement performance close to the state-of-the-art in the domain of SoC.

At first, we made an introduction to the field of AI within the context of strategic board games and presented our contributions. Furthermore, we presented the domain of SoCf in detail along with some remarks on the probability theory aspect of the game. Moreover, we conducted a literature review. Initially, we presented the theoretical background behind Max^n and supervised learning. Subsequently, we explored the bibliography regarding the AI world of strategic board games by focusing on the domain SoC. For the main part of our thesis, we shared our methodology in detail, from data collection, design of model architectures, and the training process to the implementation of Max^n, as well as the integration of modern DNNs into this traditional, tree-based, adversarial search algorithm. Finally, we shared graphs, figures, tables, and online resources that showcase our empirical results. Furthermore, we made noteworthy observations with the purpose of deepening the understanding regarding our research and deriving meaningful insights. As a main insight, we empirically showcased the effectiveness of our approach since without requiring a runtime that's intolerant to humans like the state-of-the-art does, we manage to get close to it.

To conclude, in this thesis, we illustrated the effectiveness of using adversarial search algorithms in combination with modern DNNs by achieving nearly state-of-the-art performance. In the next Section, we will explore promising future directions that, in our view, have the potential to further improve our results.

6.2 Future Directions

There are several future work ideas that can be explored by using our work as a foundation. Below, we present the ones that we find the most promising.

An obvious way to further improve our win ratio over the plan-based heuristic agents is to merge our work with agents that do not improve the initial placement aspect of the game at all, for example, ones that only improve trading [7], [14].

Another promising future direction is to acquire data from professional human games and use it to enrich one of our datasets or create a new one. Furthermore, by retraining the CNN evaluator on the new dataset with a similar process to that of Subsection 4.5.2, we would most likely acquire a noticeably better evaluator for the terminal states of Max^n.

A straightforward improvement to our existing work is the implementation of the generalization of a-b pruning for Max^n [21]. This would certainly improve the running time of our agent, and if we reduced the pruning done by the Algorithm 3 heuristic, possibly its win ratio over other agents as well.

Another direction worth investigating is the experimentation with different heuristic functions used for greedy pruning. Finding a more suitable evaluation function than the one that already exists in the JSettlers2 framework [9], as presented in Algorithm 4, has the potential to further improve our results and perhaps even allow for more aggressive pruning.

The complete replacement of Max^n with a different adversarial search algorithm, such as Monte Carlo Tree Search (MCTS), also constitutes a future direction that is worth researching.

Last, but not least, a very promising future work idea is to modify the model architecture to allow our agent to account for all game states and not only those of the initial placement phase. This would open the possibility of improving other strategy aspects as well, like trading or purchasing property and development cards. Such a DNN could be trained either in a supervised learning manner, with the data that we have already created, or with the use of RL after many game simulations.

REFERENCES

[1] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017. [Online]. Available: https://doi.org/10.1109/JPROC.2017.2761740

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," vol. 25, 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems*, vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf

[4] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016. [Online]. Available: https://doi.org/10.1038/nature16961

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. [Online]. Available: https://doi.org/10.1126/science.aar6404

[6] M. S. Dobre and A. Lascarides, "Exploiting action categories in learning complex games," in *2017 Intelligent Systems Conference (IntelliSys)*, 2017, pp. 729–737. [Online]. Available: https://doi.org/10.1109/IntelliSys.2017.8324210

[7] K. Xenou, G. Chalkiadakis, and S. Afantenos, "Deep reinforcement learning in strategic board game environments," in *Multi-Agent Systems*, M. Slavkovik, Ed. Cham: Springer International Publishing, 2019, pp. 233–248. [Online]. Available: https://doi.org/10.1007/978-3-030-14174-5_16

[8] M. Apostolidou, "Exploiting linguistic data for modeling players' behaviour in strategic board games," Senior Undergraduate Engineering Diploma Thesis (MEng), School of Electrical and Computer Engineering, Technical University

of Crete, 2020. [Online]. Available: https://doi.org/10.26233/heallink.tuc.86478

[9] J. Monin, P. Bilnoski *et al.*, "Jsettlers2: A java implementation of settlers of catan," 2011. [Online]. Available: https://github.com/jdmonin/JSettlers2

[10] C. Luckhart and K. B. Irani, "An algorithmic solution of n-person games," in *AAAI Conference on Artificial Intelligence*, 1986. [Online]. Available: https://api.semanticscholar.org/CorpusID:23411351

[11] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, 2017, pp. 1–6. [Online]. Available: https://doi.org/10.1109/ICEngTechnol.2017.8308186

[12] M. S. Dobre and A. Lascarides, "Online learning and mining human play in complex games," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2015, pp. 60–67. [Online]. Available: https://doi.org/10.1109/CIG.2015.7317942

[13] E. Karamalegkos, "Monte carlo tree search in the "settlers of catan" strategy game," Senior Undergraduate Engineering Diploma Thesis (MEng), School of Electrical and Computer Engineering, Technical University of Crete, 2016. [Online]. Available: https://doi.org/10.26233/heallink.tuc.66891

[14] H. Cuayáhuitl, S. Keizer, and O. Lemon, "Strategic dialogue management via deep reinforcement learning," 2015. [Online]. Available: https://arxiv.org/abs/1511.08099

[15] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas *et al.*, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. [Online]. Available: https://doi.org/10.1109/TCIAIG.2012.2186810

[16] G. Yilmaz, D. Yilmaz *et al.* (2019) Colonist.io alternative to settlers of catan. [Online]. Available: https://colonist.io/about

[17] Catan GmbH. (2020) Catan - the official rules. [Online]. Available: https://www.catan.com/understand-catan/game-rules

[18] J. von Neumann, "Zur theorie der gesellschaftsspiele," *Mathematische Annalen*, vol. 100, no. 1, pp. 295–320, 1928. [Online]. Available: https://doi.org/10.1007/BF01448847

[19] S. Russell and P. Norvig, *Artificial Intelligence: a modern approach*, 4th ed., 2020.

[20] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975. [Online]. Available: https://doi.org/10.1016/0004-3702(75)90019-3

[21] R. E. Korf, "Multi-player alpha-beta pruning," *Artificial Intelligence*, vol. 48, no. 1, pp. 99–111, 1991. [Online]. Available: https://doi.org/10.1016/0004-3702(91)90082-U

[22] Q. Gendre and T. Kaneko, "Playing catan with cross-dimensional neural network," in *Neural Information Processing: 27th International Conference, ICONIP 2020, Bangkok, Thailand, November 23–27, 2020, Proceedings, Part II 27*. Springer, 2020, pp. 580–592. [Online]. Available: https://doi.org/10.1007/978-3-030-63833-7_49

[23] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400 – 407, 1951. [Online]. Available: https://doi.org/10.1214/aoms/1177729586

[24] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. Dasgupta and D. McAllester, Eds., vol. 28, no. 3. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 1139–1147. [Online]. Available: https://proceedings.mlr.press/v28/sutskever13.html

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014. [Online]. Available: https://doi.org/10.48550/arXiv.1412.6980

[26] M. Campbell, A. Hoane, and F. hsiung Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002. [Online]. Available: https://doi.org/10.1016/S0004-3702(01)00129-1

[27] D. Heath, D. Allum, and P. Square, "The historical development of computer chess and its impact on artificial intelligence." *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*, vol. 63, 1997. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/2908791.2908804

[28] F. F. Duarte, N. Lau, A. Pereira, and L. P. Reis, "A survey of planning and learning in games," *Applied Sciences*, vol. 10, no. 13, p. 4529, 2020. [Online]. Available: https://doi.org/10.3390/app10134529

[29] R. S. Thomas, *Real-time decision making for adversarial environments using a plan-based heuristic*. Northwestern University, 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:107498331

[30] R. D. Saraiva, A. Grichshenko, L. J. P. d. Araújo, B. Amaro Junior, and G. N. de Carvalho, "Using ant colony optimisation for map generation and improving game balance in the terra mystica and settlers of catan board games," in *Proceedings of the 15th International Conference on the Foundations of Digital Games*, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3402942.3409778

[31] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, 2006. [Online]. Available: https://doi.org/10.1109/MCI.2006.329691

[32] L. R. Carneiro, C. A. Delgado, and J. C. da Silva, "Social analysis of game agents: How trust and reputation can improve player experience," in *2019 8th Brazilian Conference on Intelligent Systems (BRACIS)*, 2019, pp. 485–490. [Online]. Available: https://doi.org/10.1109/BRACIS.2019.00091

[33] K. P. Panousis, "Real-time planning and learning in the "settlers of catan" strategy game," Senior Undergraduate Engineering Diploma Thesis (MEng), School of Electrical and Computer Engineering, Technical University of Crete, 2014. [Online]. Available: https://doi.org/10.26233/heallink.tuc.18113

[34] P. Perick, D. L. St-Pierre, F. Maes, and D. Ernst, "Comparison of different selection strategies in monte-carlo tree search for the game of tron," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 2012, pp. 242–249. [Online]. Available: https://doi.org/10.1109/CIG.2012.6374162

[35] R. Dearden, N. Friedman, and D. Andre, "Model-based bayesian exploration," *arXiv preprint arXiv:1301.6690*, 2013. [Online]. Available: https://doi.org/10.48550/arXiv.1301.6690

[36] M. Guhe and A. Lascarides, "The effectiveness of persuasion in the settlers of catan," in *2014 IEEE Conference on Computational Intelligence and Games*, 2014, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CIG.2014.6932861

[37] H. Cuayáhuitl, S. Keizer, and O. Lemon, "Learning to trade in strategic board games," in *Computer Games*, Springer. Cham: Springer International Publishing, 2016, pp. 83–95. [Online]. Available: https://doi.org/10.1007/978-3-319-39402-2_7

[38] S. J. Russell and A. Zimdars, "Q-decomposition for reinforcement learning agents," in *Proceedings of the 20th international conference on machine learning (ICML-03)*, 2003, pp. 656–663. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/3041838.3041921

[39] STAC - strategic conversation. [Online]. Available: https://www.irit.fr/STAC/

[40] G. M. B. Chaslot, M. H. Winands, and H. J. van Den Herik, "Parallel monte-carlo tree search," in *Computers and Games: 6th International Conference, CG 2008, Beijing, China, September 29-October 1, 2008. Proceedings 6*. Springer, 2008, pp. 60–71.

[41] M. Dobre and A. Lascarides, "Pomcp with human preferences in settlers of catan," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 14, no. 1, 2018, pp. 17–23. [Online]. Available: https://doi.org/10.1609/aiide.v14i1.13014

[42] D. Silver and J. Veness, "Monte-carlo planning in large pomdps," in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23. Curran Associates, Inc., 2010. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2010/file/edfbe1afcf9246bb0d40eb4d8027d90f-Paper.pdf

[43] M. Guhe and A. Lascarides, "Game strategies for the settlers of catan," in *2014 IEEE Conference on Computational Intelligence and Games*, 2014, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CIG.2014.6932884

[44] M. Kurttutan and H. Yandell, "Torchview," 2022. [Online]. Available: https://github.com/mert-kurttutan/torchview

[45] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison *et al.* (2019) Pytorch: An imperative style, high-performance deep learning library. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

[46] J. Močkus, "On bayesian methods for seeking the extremum," in *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*, G. I. Marchuk, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 400–404. [Online]. Available: https://doi.org/10.1007/3-540-07165-2_55

[47] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison *et al.* (2019) Pytorch: An imperative style, high-performance deep learning library. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html