TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Exploiting Linguistic Data for Modeling Players' Behaviour in Strategic Board Games

Diploma Thesis

Maria Apostolidou

COMMITTEE
Advisor: Georgios Chalkiadakis, Associate Professor
Member: Stergos Afantenos, Associate Professor (IRIT CNRS, Université Paul Sabatier, Toulouse, France)
Member: Michail G. Lagoudakis, Associate Professor

Chania, August 2020

'I checked it very thoroughly' said the computer, 'and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is.'

–Deep Thought

from The Hitchhiker's Guide to the Galaxy,
written by Douglas Adams

Abstract

Many multi-agent strategic games entail social aspects realized often via natural language exchanges. Unfortunately few attempts have been made to take into account both actions and linguistic information for modeling agents. In this thesis the goal is to leverage both types of information in order to create a model that is capable of emulating players' actions taking into account actions performed by all players in the past as well as their previous linguistic exchanges. Recent advances in neural network architectures and more precisely recurrent models allow one to sequentially update representations of the game state or linguistic data, as well as the sharing of parameters between disparate representations. Thus, in this thesis we produced and employed combined representations for the game state and for the linguistic exchanges, in order to model players' actions and enable the prediction of their moves.

We demonstrate our approach in the "Settlers of Catan" multi-agent strategic game domain. As a first step the raw data was processed to form a Dataset suitable for use in machine learning projects. This step entailed a novel modeling of the way in which information about a game of "Settlers of Catan" is represented. Then linguistic and gameplay information from the created Dataset was exploited by neural networks to predict the players' actions. Architectures of Feed Forward Neural Networks, Recurrent Neural Networks (such as Long Short-term Memory Networks) as well as combined architectures of the two were investigated in the context of this thesis. We note that data collected in the context of the ERC Advanced Grant project STAC was used for this work, as well as the GloVe vectors for word representation.

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αξιοποίηση Γλωσσολογικών Δεδομένων για τη Μοντελοποίηση Στρατηγικής Συμπεριφοράς σε Παίγνια Πολλών Παικτών

Διπλωματική Εργασία

Μαρία Αποστολίδου

ΕΠΙΤΡΟΠΗ
Επιβλέπων: Γεώργιος Χαλκιαδάκης, Αναπληρωτής Καθηγητής
Μέλος: Στέργος Αφαντενός, Αναπληρωτής Καθηγητής (IRIT CNRS, Université Paul Sabatier, Toulouse, France)
Μέλος: Μιχαήλ Γ. Λαγουδάκης, Αναπληρωτής Καθηγητής

Χανιά, Αύγουστος 2020

Περίληψη

Πολλά πολυπρακτορικά επιτραπέζια ή ψηφιακά στρατηγικά παιχνίδια απαιτούν κοινωνικές αλληλεπιδράσεις μεταξύ των παικτών μέσω της συνομιλίας σε φυσική γλώσσα. Δυστυχώς λίγες απόπειρες έχουν γίνει ώστε να ληφθούν υπόψιν τόσο οι ενέργειες των παικτών όσο και η γλωσσική πληροφορία για την μοντελοποίηση πρακτόρων. Σε αυτή την εργασία ο στόχος είναι να εξισορροπήσουμε και τα δύο είδη πληροφορίας με σκοπό να δημιουργήσουμε ένα μοντέλο που να είναι ικανό να μιμηθεί τις ενέργειες των παικτών λαμβάνοντας υπόψιν τις ενέργειες που έκαναν πρωτύτερα οι παίκτες καθώς και τις προηγούμενες γλωσσικές συνομιλίες τους. Η πρόοδος που έχει σημειωθεί πρόσφατα στο χώρο των νευρωνικών δικτύων και ειδικότερα στα αναδρομικά νευρωνικά δίκτυα επιτρέπει την ακολουθιακή ενημέρωση των αναπαραστάσεων των καταστάσεων του παιχνιδιού ή και των γλωσσικών δεδομένων καθώς και την κοινή χρήση παρα-μέτρων μεταξύ διαφορετικών αναπαραστάσεων. Ως εκ τούτου σε αυτή την εργασία παράγαμε και χρησιμοποιήσαμε συνδυαστικές αναπαραστάσεις τόσο των καταστάσεων του παιχνιδιού όσο και των γλωσσικών συνομιλιών, ώστε να μοντελοποιηθούν οι ενέργειες των παικτών και να γίνει δυνατή η πρόβλεψη των κινήσεών τους.

Η προσέγγισή μας εφαρμόστηκε στο στρατηγικό παιχνίδι «Άποικοι του Κατάν». Σαν πρώτο βήμα, τα ανεπεξέργαστα δεδομένα επεξεργάστηκαν για να σχηματιστεί ένα Σύνολο Δεδομένων κατάλληλο για χρήση σε διεργασίες μηχανικής μάθησης. Αυτό συμπεριέλαβε μια πρωτότυπη μοντελλοποίηση του τρόπου με τον οποίο αναπαρίσταται η πληροφορία αναφορικά με το παιχνίδι «Άποικοι του Κατάν». Εν συνεχεία, γλωσσική πληροφορία και πληροφορία αναφορικά με το παιχνίδι από το δημιουργηθέν Σύνολο Δεδομένων αξιοποιήθηκε από νευρωνικά δίκτυα για να προβλεφθούν οι ενέργειες των παικτών. Αρχιτεκτονικές όπως Νευρωνικά Δίκτυα Έμπροσθεν Τροφοδότησης και Αναδρομικά Νευρωνικά Δίκτυα (όπως Δίκτυα Μακροπρόθεσμης Μνήμης) καθώς και συνδυαστικές αρχιτεκτονικές των δύο ερευνήθηκαν στο πλαίσιο αυτής της εργασίας. Για την εργασία αυτή αξιοποιήθηκαν τα δεδομένα που έχουν συλλεχθεί στο πλαίσιο του ERC Advanced Grant project STAC, καθώς και τα GloVe διανύσματα για αναπαράσταση λέξεων.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Games have played a central role in the development of Artificial Intelligence (AI) and Machine Learning (ML) techniques, since the inception of the idea of AI. Games form a very convenient and efficient way for measuring the capacity of AI techniques, since the restricted set of rules that accompany them leads very often (if not always) to very well defined metrics of performance and thus allow straightforward quantitative results that lead to comparisons and operation evaluation of different models and approaches. More precisely, strategic games, which are ubiquitous, provide an ideal testbed to study human strategic decision making and interactions. With that in mind, the aim of this thesis is to study whether the exploitation of natural language as well as previous behavior of players in strategic games can help with understanding the decisions that the players do or do not take during the course of a game.

In order to do so, the present work will focus on the Settlers of Catan (SoC) game, more specifically making use of its online sibling *jSettlers*, which is a faithful replica of the original board game. The added bonus of jSettlers is that conversations between the players is implemented through a chat system, hence there is a corpus of natural language data available that could be of potential use for building and testing different architectures. SoC (and thus jSettlers) is a strategy board game in which players try to settle upon the deserted island of Catan. In order to colonize the island they need to gather and trade resources so that they can build cities and connect them with roads. The game has gained the interest of the public recently and is increasingly examined in research projects. The main reason for the latter is that SoC is an excellent candidate, and in fact a very challenging framework, to test

the performance of algorithms, artificial agents or computational systems because it combines a complex set of game rules, non deterministic elements due to the use of dice and player interaction due to trading between players.

The goal is to create a testbed for studying human behavior in the context of strategic games. The term *human behavior* for the purposes of this work will be defined as "the decisions that the players make at each step (turn) of the game". The model that we look to develop and examine should on the one hand be informed by previous decisions that the players made during the game, on the other hand by natural language data collected from the dialogues of the players during the game and eventually will look to combine these two aspects. The investigation of the different approaches will allow us to have some initial indications as to whether models informed by natural language data could potentially outperform models that are based solely on previous actions data.

To that end, we employed the data collected in the context of the ERC Advanced Grant project STAC [1]. We exploited this data to develop an informative data set of both the players' actions and dialogues throughout the various SoC games. Finally we designed neural network architectures (with feed forward and recurrent components) to model the players' behaviour and discern whether language informed models provide useful insights.

1.2 Related Work

1.2.1 Agent oriented approaches

It should be emphasized from the beginning that to the best of our knowledge no previous efforts have been made in order to study the play between language and prior game behavior of a player. A lot of the work concerning the SoC is focused on developing artificial intelligence (AI) agents to play the game. Thomas [2], along with a Java open source platform of the SoC game, implemented an agent that adopts a strategy structured upon three fundamental pillars. The first has to do with the determination of the available options the agent has, the second is involved in building decisions and the third concerns the negotiation and trading behaviour of the agent. Pfeifer [3] as well built a reinforcement learning agent with the use of hand-coded high-level heuristics and low-level model trees.

SoC has often been used as a test-bed for Monte Carlo Tree Search (MCTS) methods. Szita et al. [4] implemented an agent that selects actions based on MCTS methods, but with the aid of a priori domain knowledge and hand-coded evaluations. A similar approach was adopted by the agent

of Panousis [5]. However both of these implementations have excluded some actions, mainly those of negotiating and trading, along with other elements of the game for the sake of simplicity

Due to the complexity of the Settlers of Catan, only a few implementations take into account the full game rules. One that does so is the MCTS agent of Karamalegkos [6]. Trading, being a vital feature of SoC, is included in this implementation as well as other complex elements of the game (e.g. development cards) that are overseen and omitted in most other works.

Finally Cuayáhuitl et al. [7] and Xenou et al. [8] incorporated deep reinforcement learning to improve upon the trading decisions taken by an agent. The former uses a fully-connected multi-layer neural network to decide upon a trading action. The later exploited the advantages of RNNs and specifically LSTMs along with the concept of Q-decomposition [9] to estimate the Q-functions of possible values. The training in this case is done online, during game play and the action with the maximum estimated Q-function is chosen by the agent as the trading action. For reasons of simplicity both works constrained the action set to 72 specific trading actions out of the numerous possible trading actions that are available to the agents. All other actions are decided in the same way as in [2]. The fact that there was improvement in the performance demonstrates how crucial trading is in SoC and provides evidence to argue that deep reinforcement learning is a promising approach for training agents in strategic environments.

1.2.2 Human-oriented approaches

All of the aforementioned approaches are agent-oriented, i.e. they concern the case where AI agents compete against other agents and do not take into consideration aspects that involve human players' behaviour and interaction. Work focused on the human players is mostly concerned with the language task since players interact via dialogue in order to negotiate about trading.

Within the scope of the ERC Advanced Grant project Strategic Conversation (STAC), the online version of the SoC developed by Thomas in [2] was used to collect a corpus of strategic conversation. Unlike most models of conversation, that are governed by a strong notion of cooperation because interlocutors try to achieve the common goal of effective communication, in strategic conversation their motives don't align as each is trying to achieve a personal goal. SoC is a multi-player game with considerable players interaction via dialogue and trading, hence it makes a very good example to study this type of conversational model. For this aim the dialogue between players during games of SoC was collected as chat history along with all the game history which details all of the extra-linguistic events (e.g., dice rolls, card

plays etc) from the game. The chat references of the corpus were annotated as described in [1].

The central goal of the STAC project, in general, is to understand the **linguistic** strategies adopted by interlocutors to achieve their conversational goals, especially when these goals are opposed. This is pursued mainly by studying the discourse structure of multi-party dialogues, as done by Afantenos et al. in [10].

1.3 Thesis Contribution

Although the STAC project has a particularly linguistic nuance, the corpus is so rich and the game history so detailed that information to replay a whole game is stored in it. Despite that fact, there is a profound scarcity of work in that direction in the literature.

The current thesis attempts for the first time to combine information of both linguistic references and actions of the players during game play in a machine learning scheme aiming to model the human agents' behaviour. In particular, the goal pursued here is to model the players' actions by predicting the next action in a game of SoC taking into account previous actions by all players in the past as well as their linguistic exchange. The first and very cumbersome step to achieving that aim was the modelling of the game so as to represent the concept of players' actions, given the format of the data and the game. We addressed this by denoting *gamestates* that illustrate the game course and *predicting action labels*, detailed in Section 4.1.

Following that, we exploited the information stored in the STAC corpus concerning the actions of the players so that it could be effectively transformed into a data set suitable for training, as described in Sections 4.2 - 4.5. Although this process was long and arduous it did result in a formalization of the game and an accompanying code base that can be used again in the future to transform new additional SoC data collected from the jSettlers platform into appropriate datasets to be used in other machine learning projects. In Appendix C we provide the API to this code.

The next step is that of designing the architecture. The challenge that we had to overcome here was that the game data is mixed and hence their processing is not a straightforward process. Indeed, that is also the reason why neural networks were chosen as the modeling method, as they alleviate some of these data problems through autonomous learning. The model that was tested in this thesis employs various components and combinations of feed forward neural networks and LSTM components to represent the input and make the label predictions.

At this point, we acknowledge that our research is restricted by a substantial limitation: a lack of huge amounts of SoC data in order to study such a complex system as SoC and produce conclusive results about human behaviour. That said, we wish to emphasize that the main scope of this thesis was to examine whether the linguistic data and game data could be for the first time combined in a conducive way, and not to necessarily create a perfect predictor of players' actions. Apart from that, the API and problem modeling that we developed here can be used to easily transform future SoC game data into a pertinent data set and, although the time line exceeds the purpose of this thesis, new data can be collected from building an online community in jSettlers.

1.4 Thesis Outline

The remainder of this thesis is structured as follows: Chapter 2 outlines the game rules of Settlers of Catan and describes the jSettlers framework, using which the data was collected; Chapter 3 provides a theoretical background of the basic concepts and machine learning methods used in this project; Chapter 4 elaborates on how the data of the STAC project was processed to create a dataset appropriate for machine learning; in Chapter 5 the NN architecture that was implemented in this thesis is described; the results are presented and examined in Chapter 6; final conclusions and remarks are included in Chapter 7. Some additional details concerning the jSettlers framework can be found in Appendix A. Appendix B contains a complete list of all the network configurations tested in this thesis and their resulting scores. In Appendix C the documentation of the API for the SoC DataSet creation is included.

Chapter 2

Settlers of Catan

Settlers of Catan is a multi-player strategy board game played between 2 to 4 players. The objective of the game is to become the first player to obtain 10 Victory Points (VP). This can be achieved by building roads, settlements, and cities. The players can also gain VP through buying Development Cards, building the longest road, or amassing the largest army. In order to build something or buy a Development Card, the players have to spend resources. These resources are obtained from their settlements and cities as well as through maritime trading and/or trading with other players.

This chapter explains the rules of the game and describes the jSettlers framework.

2.1 Rules of the game

2.1.1 General objective

Catan is a small island formed by 19 hexagons of land and surrounded by sea. There are 6 different types of land on the island of Catan, 5 of which produce different types of resources and one producing no resource, being a desert. The fertile types of land are The Forest, that produces lumber, The Hills, producing brick, The Mountains that give ore, The Fields producing grain and The Pasture that produces wool. The purpose of the game is for one player to colonise the island of Catan. This can happen by collecting Victory Points, which are awarded when building a road, building a settlement or upgrading one to a city and some other conditions discussed in the following.

Players start the game with two settlements and two roads each. They can spend their resources to build further roads and new settlements or upgrade their settlements to cities. The necessary resources are accumulated by

building settlements and cities on the appropriate resource-producing land types and resources can be also exchanged between the players (by trading) or with the stash using the special port tiles (maritime trade).

2.1.2 Game board

The board consists of three different types of tiles; in particular 19 terrain hexagonal tiles, 18 sea hexagonal tiles and 18 numbered tiles. The terrain tiles are further categorised in six (6) distinct types with different properties (summarized in Table 2.1) while the sea tiles are categorised in two (2) distinct types (see Table 2.2). The numbered tiles ($\in [2, 12]$) represent all possible dice outcomes. With the exception of 2 and 12, which are the least probable outcomes to get with two dices, there are two tile copies for each of the other numbers.

To set up the board the land tiles are shuffled and placed randomly in columns of 3-4-5-4-3 tiles, forming a large hexagonal island (see Fig. 2.1). The sea tiles are placed all around, surrounding the island. Lastly, the numbered tiles are placed on top of the land tiles, skipping the desert tile. In this way every land tile is associated with a dice result number. When the dice are rolled the land tile or tiles that corresponds to the dice outcome are activated, i.e. they produce resources for the settlements that are built adjacent to them.

LAND TILES

Type	**Property**
Forest	Produces lumber (wood)
Hill	Produces brick (clay)
Mountain	Produces ore (iron)
Field	Produces grain (wheat)
Pasture	Produces wool
Desert	Home of the robber Does not produce any resources

Table 2.1: Description of the land tiles in SoC. Every settlement can pay off 3 different resource types to its owner, as nodes are adjacent to three hexagons. Exception are nodes neighbouring to the sea and the desert tile. Players who build settlements adjacent to these tiles will receive resources only from the other two adjacent, fertile tiles.

Figure 2.1: SoC game board.

SEA TILES

Type	Property
Plain sea	None
Harbour	Miscellaneous port (3:1) : three units of any type resources can be exchanged here for 1 desired resource unit, as long as all three are of the same type.
	Non miscellaneous port (2:1) : two units of one specified type of resource can be exchanged here for 1 desired resource unit. Dedicated to one type of resource.

Table 2.2: Description of the sea tiles in SoC. Players that build settlements adjacent to a sea tile will receive resources only from the other two fertile land tiles when the dice results correspond to their number. However, if the sea tile hosts a harbour, they will have the extra benefit of trading resources with a more profitable, fixed ratio (at least better than trading with the bank).

2.1.3 Resources

Players gain resources by having settlements or cities built on the vertices of resource producing terrain tiles. After the dice are rolled, the terrain tiles associated with the numbered tiles corresponding to the sum of the dice numbers produce the respective resources. All players gain the appropriate resources corresponding to their settlements and cities in every round. Settlements provide the players with one (1) resource unit per each of their adjacent fertile terrain tiles, while cities provide two (2) resource units. For every round, the player who rolled the dice can make further actions that include trading, constructing and developing, using the special Development Cards. After receiving resources the player who rolled the dice may trade resources and build as much as they want until they are done.

2.1.4 Actions

Trading

The player who rolled the dice can trade their resources during their round with the bank/stash, via their ports or by negotiating with other players through. More specifically, 4 trading options are available to the players:

a) Trading with other players

   The player who rolled the dice can negotiate with the other players to exchange resources at a rate they discuss and decide between them. Trading transactions must always include resources to be exchanged by both parties, i.e. giving away resources for free is not permitted. Other than that, any type of arrangement between players is allowed.

   Important note: players may only trade with the player whose turn it is, i.e. the other players may not trade among themselves.

b) Sea trading via a miscellaneous harbour

   If a player has built a settlement adjacent to a sea tile with a harbour, they can exchange goods with the stash with a fixed rate of 3:1. All 3 resources offered by the player must be of the same type, i.e. 3 woods or 3 ores can be exchange for 1 desired resource unit but not 2 ores and 1 wood.

c) Sea trading via a non-miscellaneous harbour

   If a player has built a settlement adjacent to a sea tile with this particular type of harbour, they can exchange goods with the stash with an

even better fixed rate of 2:1. These ports however are dedicated to one specific type of resource (that is denoted on the tile), meaning that in a wood harbour the player can exchange 2 woods for 1 desired resource unit but nothing else.

d) Trading with the bank

The players can always trade with a ratio of 4:1 with the stash, even if they have not built in port location. The 4 resource units offered by the player must be again of the same type. Negotiating with other players, however, is advised as it can yield a more profitable exchange ratio.

Constructing

During their round players can construct new roads and settlements or upgrade their existing settlements to cities. Each of the aforementioned construction actions will cost the players specific resources, while at the same time there are certain rules that the actions must abide by. The cost and rules are described in detail below. As a general rule, the maximum number of constructions that the players can have is dictated by the number of pieces they have in their inventory (a total of 5 settlements, 4 cities and 15 roads per player).

a) Road construction rules (Cost: 1x Clay + 1x Wood)

- New roads can be constructed on the edges of the terrain tiles, each edge supporting one and only one road.
- New roads must be adjacent to another road, settlement or city belonging to the same player.
- The player that has constructed the longest (with at least 5 pieces) continuous (there are no interrupting settlements or cities owned by other players) line of roads gains the special card The Longest Road, that grants the owner with two (2) Victory Points.

b) Settlement construction rules (Cost: 1x Clay + 1x Wood + 1x Wheat + 1x Wool)

- New settlements can be constructed on the vertices of terrain tiles only if the neighbouring vertices are unoccupied (by settlements or cities).

- New settlements must be adjacent to at least one of the player's roads.
- Settlements award one (1) Victory Point to the player.

c) City upgrade rules (Cost: 3x Iron + 2x Wheat)

- New cities can be constructed only as upgrades to existing settlements.
- After the upgrade, the settlement becomes available to the player to use again.
- Cities award two (2) Victory Points to the player.

Special Cases

a) Activating the Robber

- When a player rolls seven (7) on the dice, the Robber is activated and no player receives any resources.
- Players with more than seven (7) resource cards must select half of them, rounded down, and return them to the bank.
- The player that rolled the seven then moves the Robber to any other terrain hexagon (including returning the Robber to the desert).
- The player steals one (1) random resource card from an opponent player who has a settlement or city adjacent to the terrain hexagon that the Robber was moved to.
- If there are more than one player's settlements or cities adjacent to the terrain hexagon that the Robber was moved to, the player who moved the Robber chooses which player to steal from.
- The terrain hexagon that the Robber is occupying will stop producing resources for the duration of time that the Robber is occupying it.
- After that the player continues with his turn (to build roads, settlements, trade etc)

b) Playing Development Cards

- A player can play (reveal) only one (1) of their Development Cards during their round.

- Development Cards cannot be played during the same round they were bought (except for a Victory Point card).
- A Development Card can be played at any point during a player's turn, even before rolling the dice, as long as it has been bought at a previous game round.
- In total there are 25 Development Cards in the deck, of which 14 are knight cards, 6 are progress cards and 5 are Victory Point cards.

  i) Knight cards
  - When a player plays a Knight card the Robber is activated. The players do not discard half of their resources but the one that played the Knight card moves the Robber to a new hexagon location and steals a resource from a player that has built a settlement adjacent to this hexagon.
  - The player that has played (revealed) the most Knight cards (and more than three (3) in total) receives the special card *Largest Army*, which is worth two (2) Victory Points.
  - In the case that another player exceeds the number of Knights of the largest amassed army of the game until that point, he takes the special card from its previous proprietor, along with the two (2) Victory Points that accompany it.

  ii) Progress cards
  - A player can play (reveal) one of their progress cards, which instructions should be followed. The particular card is then removed from the game.
  - The *Road Building* card allows a player to immediately place two (2) free roads on the board (according to the normal building rules).
  - The *Year of Plenty* or *Discovery* card allows a player to immediately take any two (2) resource cards from the stash (and these cards can be used immediately for building in the same round).
  - The *Monopoly* card allows the player to name one (1) resource and all the other players must give the player all of their resource cards of that specific type (opponent players that have no such resource cards do not give anything to the player).

- In total there are 2 road building cards, 2 discovery cards and 2 monopoly cards in the deck.

iii) Victory Point cards

- Victory Point cards are hidden from the rest of the players and are only revealed by a player during their round when they have accumulated in total ten (10) Victory Points, making them the winner of the game.

2.2 Game Phases

2.2.1 Game Setup

At the start of the game each player has five (5) settlements, four (4) cities and fifteen (15) road pieces in their inventory. The first player is randomly selected and then the other players follow in a clockwise order. During the setup phase, following the play order, each player in turn puts a settlement and a road piece on the board. The procedure is then repeated in reverse order. Right after the placement of their second settlement, each player receives from the bank/stash their initial resources, i.e. for every adjacent hexagon to their second settlement they receive one (1) resource unit accordingly. When all players have placed their second settlement the game can begin and the player who placed last their second settlement rolls the dice to begin the first game turn.

2.2.2 Turn Overview

During a turn:

- the player rolls the dice for resource production
- the player may trade resource cards with other players and/or use maritime trading, build roads and settlements, upgrade settlements to cities and/or buy Development Cards
- the player may play one Development Card at any time during their turn, even before rolling the dice

2.2.3 Game Over

The game will end when one of the players has gathered ten (10) Victory Points and they announce it during their turn.

2.3 The jSettlers framework

Many software platforms are available for one to play Settlers of Catan. One of the most notable is that of Robert S. Thomas [2] which is an open source, Java version of the game that is the basis of many SoC servers online. This framework includes also various heuristic-based AI players.

The data used in this thesis comes from the jSettlers framework, hence it is important to present it to some extent.

The interface

Upon entering a game, the player is presented with the jSettlers interface, consisting of four (4) player regions (for the four different players), the game board region and the messages region. Initially, all player regions are empty

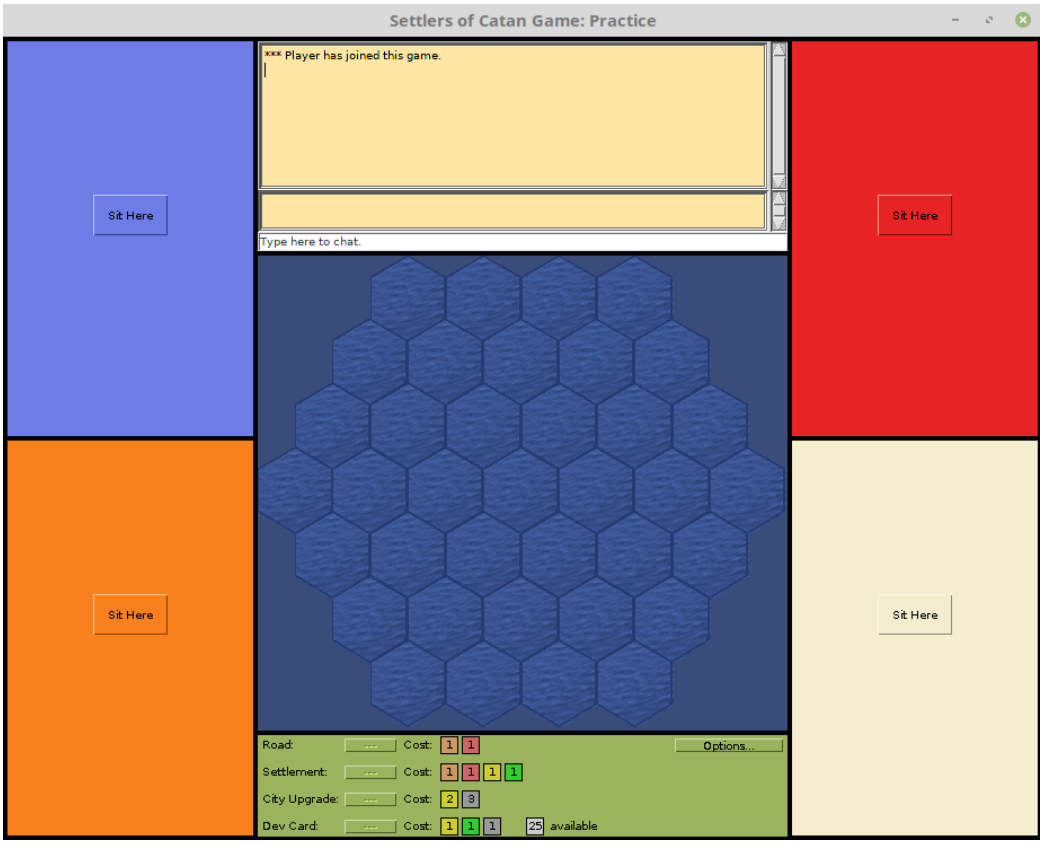


Figure 2.2: The jSettlers interface. The view of the interface's regions a player initially sees before the game is ready to begin. The player chooses where to sit and when everyone is seated the game can start.

and the game board hexagons are all sea tiles (see Fig. 2.2).

After the players are seated the game board is generated at random (see Appendix A) and the game starts. A player is selected at random to start placing their initial settlement and road. When the initial set up of 2 settlements and 2 roads is completed as dictated by the SoC rules (see Section 2.2.1), the aforementioned player starts the first turn of the game by rolling the dice. When the game will have progressed for a few turns the jSettlers interface will look something like Fig. 2.3.

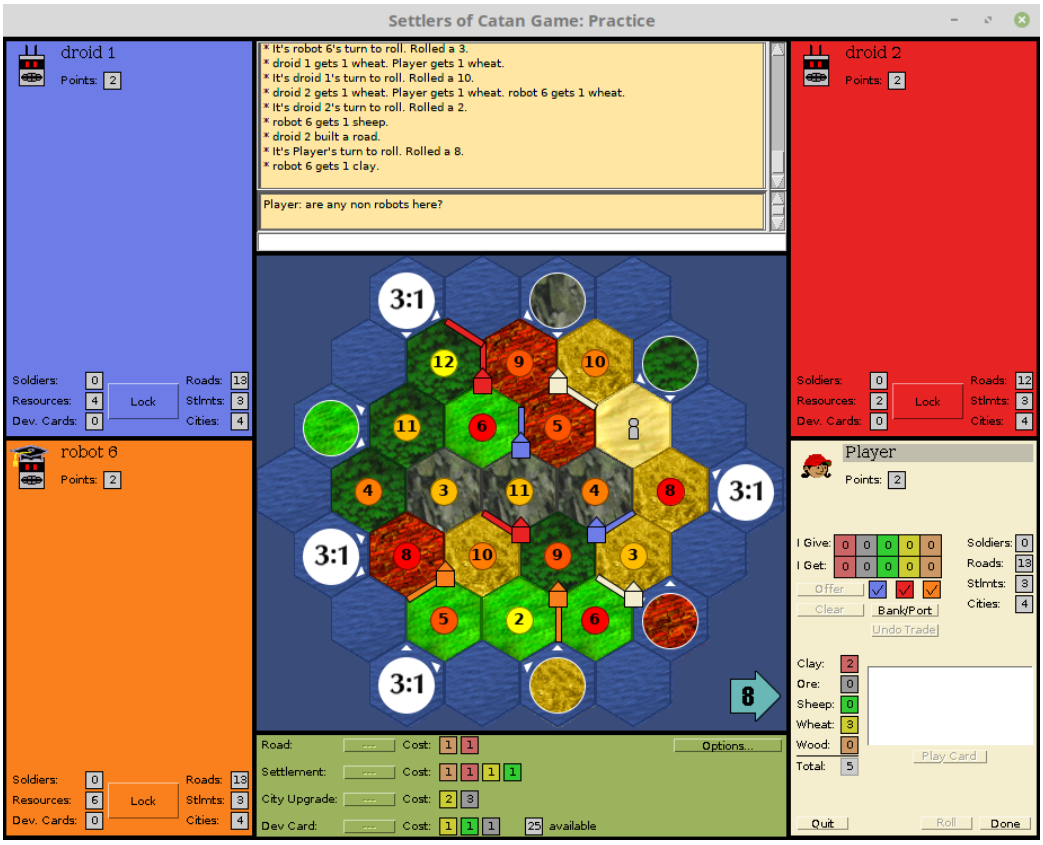


Figure 2.3: View of the jSettlers interface during gameplay.

On the left and right of the game board the four colored regions of the players contain information about their resources, development cards, knights etc. and the trading tools, i.e. a set of buttons and numeric boxes that allow the player to initiate and control the maritime trading and trading with the other players.

On the bottom part of the screen, below the game board, is a green box with information and buttons for constructing roads and settlements, upgrading to cities and buying Development Cards. This information includes

the costs for each building action, the cost to buy a Development Card, the number of available Development Cards in the deck, as well as buttons that provide information on the game statistics and the game settings.

Above the game board, the messages region is split into three parts. The small one at the bottom, in white color, is the chat input window, prompting the players to type a message. Above that, in light yellow, all the chat history from the beginning of the game is displayed. At the very top there is a light yellow box displaying the sever messages from the beginning of the game. The server messages describe the actions of the players, the dice results, the allocation of resources, prompts to the players when they need to take specific actions (e.g. move the Robber, discard cards in case a 7 was rolled, monopolize a resource when they have played the corresponding Development Card etc) and announcement to the players for the progression of the game (e.g. whose turn it is to roll the dice, place a settlement etc); in essence, a game history.

The hexagonal coordinates system

One of the main contributions in Thomas's PhD dissertation [2] was the game board coordinate system. All the locations on the board are encoded using a hexadecimal number, so that the symmetry of the hexagonal board is deployed. The coordinates are such that it's easy to compute a tile, node, or edge's neighbours by adding and subtracting. The coordinates of hexagonal tiles, nodes and edges of the board are shown in Fig. 2.4.

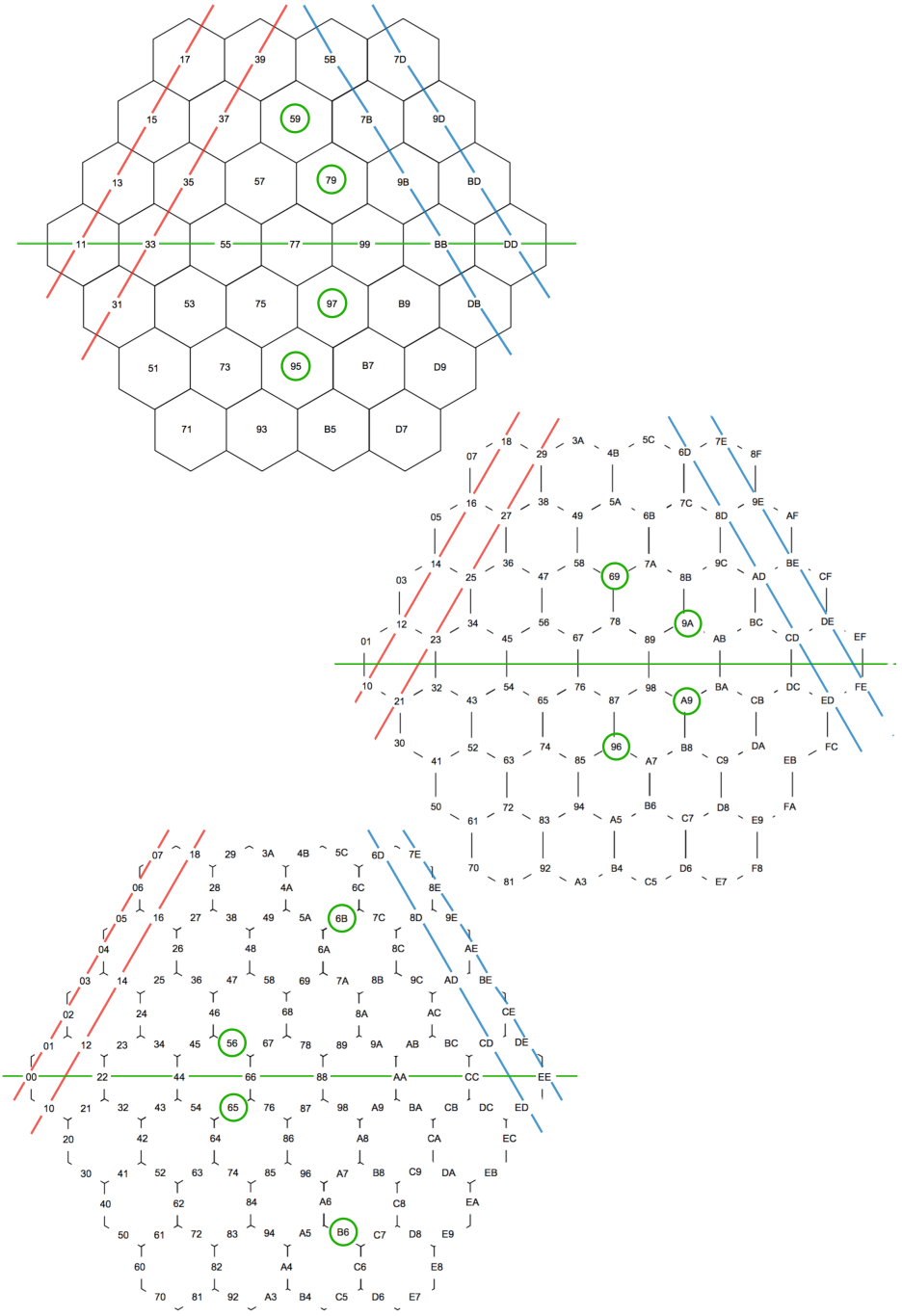


Figure 2.4: The coordinate system of jSettlers for tiles, nodes and edges of the board. Symmetrical points on the board, with respect to the central green line, have mirrored hexadecimal coordinates.

Other game options of jSettlers

The most recent versions of the jSettlers feature some additional game options, e.g. specified game board settings, trading options, Victory Points necessary to win a game etc (see Fig. 2.5). Also extra features have been added in accordance with the game boards *extension* versions. These versions allow more than four players to play the game and include a bigger game board, additional building options (e.g. towers, ships) and other trading caveats (e.g. pirates).

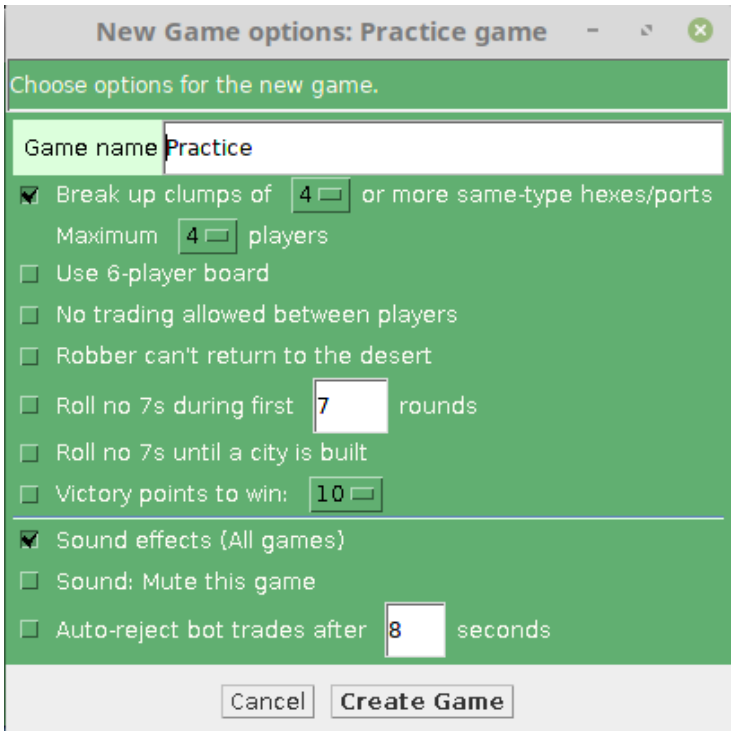


Figure 2.5: Game options menu in jSettlers

Chapter 3

Theoretical Background

This chapter provides an overview of the fundamental workings and the mathematical foundation of neural networks and other machine learning techniques and methods used in this thesis. This chapter follows the approach presented in [11].

3.1 Overview and Applications

With the advance of modern technology two fundamental factors have changed the way information is handled; data is stored easily and computers have become faster. These two premises, i.e. the access to large data sets and high computational power, have rekindled the interest in Machine Learning, a field that grew out of Artificial Intelligence in the 1950s but is now more relevant than ever. Machine Learning pertains a subset of algorithms that learn from data or experience and build mathematical models in order to make predictions or decisions without being explicitly programmed to perform the task. Such methods that were up until recently too demanding in data and hardware resources, with these obstacles removed, are now in the spotlight and thriving.

Artificial Neural Networks (NNs) are computational models of machine learning that have gained immense popularity recently, both within the scope of machine learning research as well as industrial applications, due to many breakthrough results that they exhibit in performing difficult tasks. Inspired by the way the human brain processes information, NNs have the ability to discover intricate relationships and patterns in data, without prior knowledge of the domain of the task at hand.

The most straightforward form of a neural network is the multi-layer perceptron (MLP), i.e. a fully connected feed-forward neural network. The main

advantage MLPs offer is that they include non-linear elements, hence they can be used to approximate complex functions [12]. They are used with increasing frequency to solve classification problems and predictions problems, with successful applications spanning over a vast variety of scientific fields, including Economics [13, 14] , Medicine and Biology [15], Environmental sciences [16] and many others. Additionally, the exploitation of NNs has been proven very fruitful for tasks of signal processing [17], speech recognition [18] and many linguistic applications, namely syntactic parsing [19], language modeling [20], machine translation [21], sentiment classification [22] etc.

Another prolific class of neural networks is that of convolutional neural networks (CNNs). These networks exploit the hierarchical patterns of data by assembling more complex patterns from smaller and simpler patterns. Applications of CNNs are mainly found in image related tasks and computer vision [23–25] but go as far as music recommender systems [26], seizure prediction systems [27] and speech recognition [28].

CNNs excel at extracting local patterns in the data, a task very useful in Natural Language Processing (NLP). As a matter of fact, they are capable of extracting meaningful local patterns that are sensitive to word order, regardless of where they appear in the input and for that reason have been used in various language related tasks including sentence classification [29], text categorization [30, 31], sentiment classification [32], semantic role labeling [33], question answering [34] and others.

However, in the field of NLP, the type of NNs that has caused the most excitement, and for very good reasons, is that of recurrent neural networks (RNNs) and in particular Long Short-Term Memory neural networks (LSTMs). These models are specialized for dealing with sequential data, which is exactly the case in arguably every language task. On top of that, they allow abandoning the Markov assumption that was prevalent in NLP for decades and in that view they have revolutionized the field and established new ways of handling language related problems. LSTMs have produced state of the art results in many NLP tasks and are actively becoming the prominent approach, replacing the long dominating methods in the field. A plethora of work in NLP using LSTMs includes language modelling [35–38], sequence tagging [39, 40], machine translation [41–43], response and dialogue generation [44–46], text summarization [47, 48], sentence simplification [49], question answering [50] and many others.

3.2 Feed Forward Neural Networks

3.2.1 From the Perceptron to Neural Networks

The origins of neural computational models date back to 1958, when Frank Rosenblatt invented the perceptron algorithm [51]. The inspiration was drawn by the way the brain makes use of the various neural cells that are accordingly activated and combined in networks in order to execute complicated actions.

Indeed, a neural cell alone is just an electrically excitable cell with a seemingly simple operation. A neural cell consists of three main parts as seen in Fig. 3.2. The cell body that contains the cell's nucleus is called *Soma* and is the part of the neuron that receives information. The *Dendrites* are thin filaments that carry information from other neurons to the soma and are the "input" part of the cell. The *Axon* is a long projection that carries information from the soma and sends it off to other cells. This is the "output" part of the cell. If a neuron receives a large number of inputs from

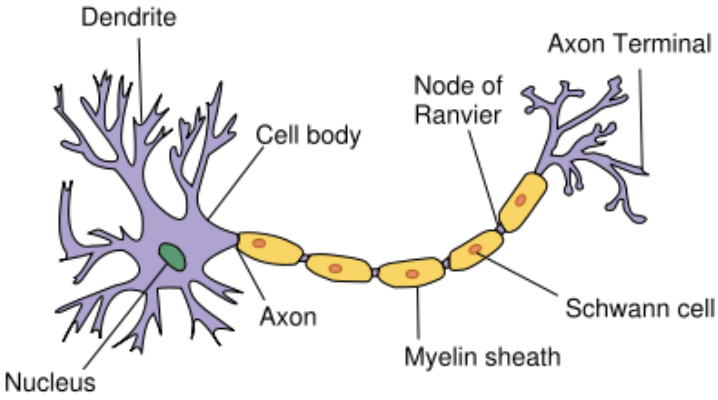


Figure 3.1: Diagram of a Neuron (source: "Anatomy and Physiology" by the US National Cancer Institute's Surveillance, Epidemiology and End Results (SEER) Program)

other neurons, these signals add up until they exceed a particular threshold. Once this threshold is exceeded, the neuron is triggered to send an impulse along its axon which is called an *Action potential*. Such a mechanism does not seem to hold a lot of computational power; yet by combining the action of many neural cells the brain and central nervous system are capable of executing tremendously complicated operations.

Hence the perceptron, as a metaphor of a neural cell, is a small independent computational unit. It functions as a simple binary classifier and

computes a simple threshold function that maps its input $\mathbf{x}$ (a real-valued vector) to an output value $f(\mathbf{x})$ (a single binary value):

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

where $\mathbf{w}$ is a vector of real-valued weights and $\mathbf{w} \cdot \mathbf{x}$ is the dot product $\sum_{i=1}^{m} w_i x_i$, where m is the number of inputs to the perceptron, and b is the bias.

Rosenblatt's ambition was that by combining many perceptrons in networks called *Multi Layer Perceptrons (MLPs)* a much more powerful classifier would emerge. This was shown to be true, however, the linear nature of the perceptron set limitations to the computational potential of MLPs. On top of that, computers at the time lacked sufficient power to process useful neural networks. Therefore the research on MLPs stagnated for more than a decade.

In the 1980's the interest in computational networks resurged as the development of metal–oxide–semiconductor (MOS) very-large-scale integration (VLSI), in the form of complementary MOS (CMOS) technology, enabled the development of practical artificial neural networks. Additionally, the linearity obstacle was overcome with the inclusion of a non-linear function in the model.

A single neuron, as the one depicted in Fig. 3.2, can be seen as a computational unit with scalar inputs and outputs. Each input is associated to a weight. The function of the neuron is to multiply each input by its weight, sum all of the weighted inputs, apply a nonlinear function to the result and send it to its output.

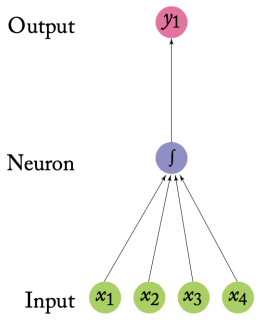


Figure 3.2: A single neuron with four inputs. (source: [11])

A network, like the one depicted in Fig. 3.3, is formed when multiple

neurons are connected to each other. Such networks can function as strong computational devices that are able to approximate a wide range of mathematical functions, provided that the weights, number of neurons and the nonlinear activation function are configured properly.

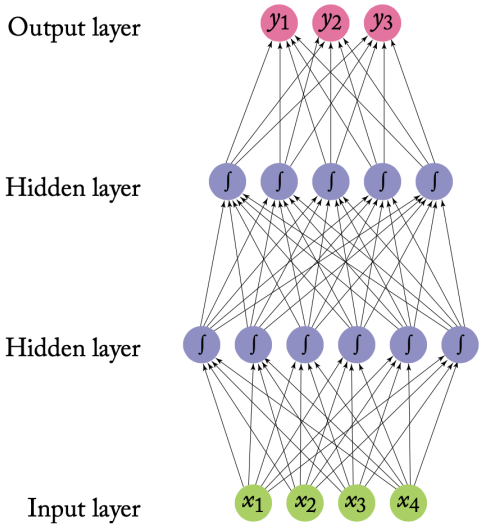


Figure 3.3: Feed-forward neural network with two hidden layers. The sigmoid shape inside the neurons in the middle layers represent a nonlinear function (e.g. the logistic function $1/(1+e^{-x})$) that is applied to the neuron's value before passing it to the output. (source: [11])

The neurons in a network are arranged in layers, reflecting the flow of information. The bottom and top layers are called *input* and *output* layer respectively. All the intermediate layers are called *hidden* layers. A *fully connected* or *affine* layer is one where all neurons are connected to every neuron of the next layer.

In essence, a feed-forward network is simply a stack of linear models separated by nonlinear functions. The values of each row of neurons in the network can be thought of as a vector. Using the same mathematical notation as in chapter 4 of [11] a fully connected layer implements a vector-matrix multiplication, $\mathbf{h} = \mathbf{xW}$ where the weight of the connection from the ith neuron in the input row to the jth neuron in the output row is $\mathbf{W}_{[i,j]}$. The values of $\mathbf{h}$ are then transformed by a nonlinear function g that is applied to each value before being passed on as input to the next layer. Ignoring the bias terms, the whole computation from input to output can be written as: $(g(\mathbf{xW^1})\mathbf{W^2})$ where $\mathbf{W^1}$ are the weights of the first layer and $\mathbf{W^2}$ are the weights of the second one. If we now consider the network in Fig. 3.3

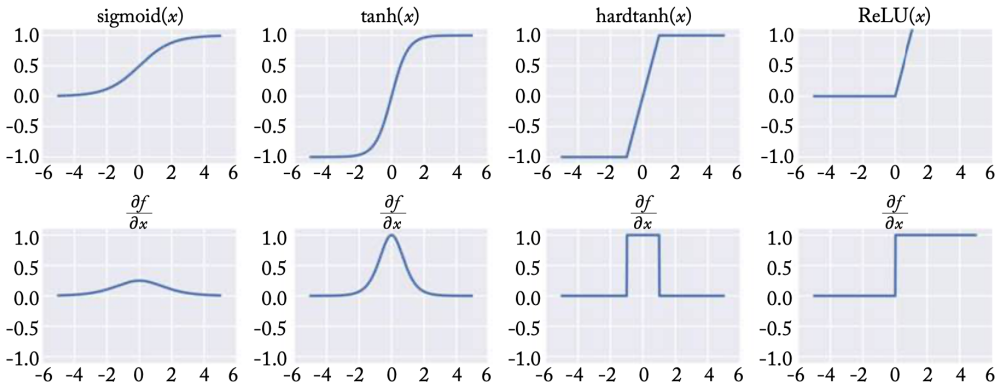


Figure 3.4: Common activation functions of neural networks (top) and their derivatives (bottom) (source: [11]).

including the bias terms that are implied we can write the equivalent equation of the network as:

$$NN(\mathbf{x}) = (g^2(g^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1})\mathbf{W^2} + \mathbf{b^2}))\mathbf{W^3} \tag{3.2}$$

When dealing with deeper networks, it becomes more clear if we write the network equation using intermediary variables. The equation of the network depicted in Fig. 3.3 is equivalently written as:

$$\begin{aligned} NN(\mathbf{x}) &= \mathbf{y} \\ \mathbf{h^1} &= g^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1}) \\ \mathbf{h^2} &= g^2(\mathbf{h^1}\mathbf{W^2} + \mathbf{b^2}) \\ \mathbf{y} &= \mathbf{h^2}\mathbf{W^3} \end{aligned} \tag{3.3}$$

The network's ability to represent complex functions is highly dependent on the nonlinear activation function. In the absence of a nonlinear activation function only linear transformations of the input can be represented. As mentioned in [11] "there is currently no good theory as to which nonlinearity to apply in which conditions, and choosing the correct nonlinearity for a given task is for the most part an empirical question" adding that "as a rule of thumb, both ReLU and tanh units work well, and significantly outperform the sigmoid." In Fig. 3.4 we summarize some commonly used non linear activation functions.

3.2.2 Training Neural Networks

The problem of training neural networks is fundamentally an optimization problem. As in any supervised learning algorithm, the input consists of a *training set* of n training examples $x_{1:n} = x_1, x_2, ..., x_n$ along with a set of corresponding labels $y_{1:n} = y_1, y_2, ..., y_n$. The objective is for the training algorithm to find a function $f()$ that maps the inputs to the correct labels. In other words a function that will make predictions $\hat{\mathbf{y}} = f(\mathbf{x})$ such that $\hat{\mathbf{y}}$ will approximate $\mathbf{y}$. To quantify the deviation of the predicted labels $\hat{\mathbf{y}}$ from the true labels $\mathbf{y}$ we use a *loss function* $L(\hat{\mathbf{y}}, \mathbf{y})$ that assigns a numerical score to the network's output. The loss function is bounded from below and the minimum is obtained when the prediction of the network is correct.

The learned function $f()$ is defined by its parameters, i.e. the matrix $\mathbf{W}$ and the biases vector $\mathbf{b}$. In mathematical notation we can write this as $f(\mathbf{x}; \Theta)$. The loss function can hence be expressed with respect to the parameters Θ as:

$$\mathcal{L}(\Theta) = \frac{1}{n} \sum_{i=1}^{n} L(f(\mathbf{x_i}; \Theta), \mathbf{y_i}) \tag{3.4}$$

where L is the per-instance loss function and $\mathcal{L}$ is the mean value of the loss over all the samples. The optimization objective is to minimize the loss over these parameters Θ):

$$\hat{\Theta} = \arg\min_{\Theta} \mathcal{L}(\Theta) = \arg\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} L(f(\mathbf{x}_i; \Theta), \mathbf{y}_i) \tag{3.5}$$

In this view, training the model is equivalent to solving the optimization problem of Eq. 3.5. The solution to this optimization problem can be found using gradient-based methods, that repeatedly compute an estimate of the loss $\mathcal{L}$ over the training set and then compute the gradients of the parameters Θ with respect to the loss estimate, in order to move the parameters in the opposite directions of the gradient. Among the most commonly used optimization algorithms are the *stochastic gradient descent* (SGD) and the *Adam* optimization algorithm [11]. The various optimization methods differ in how they estimate the error and how they update the parameters Θ.

As we mentioned earlier, neural networks can be expressed as differentiable parameterized functions and as such they are trained using gradient-based optimization methods. The non-linearity of the activation function however introduces a predicament that impedes the gradient calculation. Therefore for neural networks the *backpropagation* algorithm is used. Basically the backpropagation algorithm uses the chain rule, while caching intermediary results to efficiently calculate the gradients [52, 53].

3.2.3 Loss Metrics

It would be wise at this point to get into a little bit more detail regarding the loss functions used in neural networks and other machine learning training algorithms. In theory, the loss can be an arbitrary function mapping two vectors to a scalar, but for the purposes of optimization, functions for which the gradient can be easily computed are preferred. The most commonly used in neural networks are the Hinge loss (binary or multi class), the log loss (a "softer" variation of the Hinge loss) and the binary or categorical cross-entropy loss [11].

The binary cross-entropy loss, also referred to as *logistic loss* is used in binary classification with conditional probability outputs. We assume a set of two target classes labeled 0 and 1, with a correct label $y \in \{0, 1\}$. The classifier's output $\hat{y}$ is transformed using the sigmoid (also called the logistic) function $\sigma(x) = 1/(1 + e^{-x})$ to the range $[0, 1]$, and is interpreted as the conditional probability $\hat{y} = \sigma(\hat{y}) = P(y = 1|\mathbf{x})$. The prediction rule is:

$$prediction = \begin{cases} 0 & \hat{y} < 0.5 \\ 1 & \hat{y} \geq 0.5 \end{cases} \tag{3.6}$$

The network is trained to maximize the log conditional probability $logP(y = 1|\mathbf{x})$ for each training example $(\mathbf{x}, y)$. The logistic loss is defined as:

$$L_{logistic}(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \tag{3.7}$$

The logistic loss is useful the aim is for a model to produces class conditional probability for a binary classification problem. When using the logistic loss, it is assumed that the output layer is transformed using the sigmoid function [11].

When a probabilistic interpretation of the scores is desired, the categorical cross-entropy loss (also referred to as *negative log likelihood*) is used. Let $\mathbf{y} = \mathbf{y}_{[1]}, ..., \mathbf{y}_{[n]}$ be a vector representing the true multinomial distribution over the labels $1, ..., n$, and let $\hat{\mathbf{y}} = \hat{\mathbf{y}}_{[1]}, ..., \hat{\mathbf{y}}_{[n]}$ be the linear classifier's output, which was transformed by the softmax function and represents the class membership conditional distribution $\hat{\mathbf{y}}_{[i]} = P(y = i|\mathbf{x})$. The categorical cross entropy loss measures the dissimilarity between the true label distribution $\mathbf{y}$ and the predicted label distribution $\hat{\mathbf{y}}$, and is defined as cross entropy:

$$L_{cross-entropy}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i \mathbf{y}_{[i]} \log(\hat{\mathbf{y}}_{[i]}) \tag{3.8}$$

The cross-entropy loss produces a multi-class classifier that not only predicts the one-best class label but also outputs a distribution over the possible

labels. When using the cross-entropy loss, it is assumed that the classifier's output is transformed using the softmax transformation [11].

3.2.4 Evaluating the Performance

One of the key concepts of training a model is that we seek for a function $f()$ that can generalize well to unseen examples, i.e. it can make correct predictions even for examples it has not specifically seen during the training process. Therefore it becomes evident that the accuracy of model must be assessed over a set of examples that have been introduced to the model during training. There exist several approaches, according to the way that one separates the examples that will be seen during a training process, i.e. the *training set* of examples and those that will be examined only to evaluate the accuracy of the learned model, i.e. the *test set* or *held-out set* of examples. Some of the most known methods is the leave-one out cross-validation and the k-fold cross validation. A more efficient solution in terms of computation time is to split the training set into two subsets (e.g. 80% to 20%) and then train the model on the larger subset and test its accuracy on the smaller subset.

3.2.5 The Bias vs Variance Problem

It has been established by this point that the objective of the training process in machine learning is to minimize the loss. This approach however is not always the best as it presents caveats, i.e. it may result in *overfitting* of the training data. Overfitting is encountered when the training error is small but the error of unseen examples is large. This means that the algorithm mimics the specific data it was given during training and does not learn the general pattern of the data, therefore lacks generalization. This is also known as a *high-variance* problem. To avoid this, although it may sound counter-intuitive, it is actually advisable to allow for some error in the training process in order to achieve better generalization.

Techniques to avoid overfitting in machine learning involve some pre-processing of the data e.g. to extract outliers from the dataset, normalize numerical values etc. Another very efficient technique is using a *regularization* term, especially when working with multi-layer networks that involve a lot of parameters, thus making them prone to overfitting. The regularization term R can be added to the optimization objective in order to control the complexity of the parameter value. Eq. 3.5 then becomes:

$$\begin{aligned}\hat{\Theta} &= \arg\min_{\Theta} \mathcal{L}(\Theta) + \lambda R(\Theta) \\ &= \arg\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} L(f(\mathbf{x}_i; \Theta), \mathbf{y}_i) + \lambda R(\Theta)\end{aligned} \tag{3.9}$$

The regularization term considers the parameter values, and scores their complexity. The aim is to find parameter values that provide both low loss scores and are low in terms of complexity. A hyperparameter λ is used to control the amount of regularization. For ensuring low complexity, the regularizers R measure the norms of the parameter matrices, and favor the learning process toward solutions that have low norms. Common choices for R are the L_2 norm, the L_1 norm, and the elastic-net. Additionally, in neural networks another effective way to avoid overfitting is the use of *dropout training*. The dropout method prevents the network from relying on specific weights by randomly dropping (setting to 0) some neurons in the network during training.

On the other end of the spectrum, there may be the case that the error is large when assessing both the training examples and the validation ones. This is the case of *high bias* or *underfitting* of the data. To put it in an illustrative way, it is as if the model is biased towards an original assumption and, even though the data that it sees during the training process seems to disagree with that, it remains stubbornly fixed in that original idea. Fixing a high-bias problem involves adding more complexity to the model, i.e. a bigger/deeper neural network, more parameters or more features in the modeling of the problem we are trying to solve.

Table 3.1 presents a list of common corrective steps one can try when troubleshooting a machine learning problem with high bias or variance and Fig. 3.5 provides a visual example of the bias vs variance problem.

High Bias	**High Variance**
add polynomial features/ try a more complex neural network	get more training examples
get additional features	try a smaller set of features
decrease the regularization hyperparameter λ	increase the regularization hyperparameter λ

Table 3.1: Troubleshooting bias or variance problems.

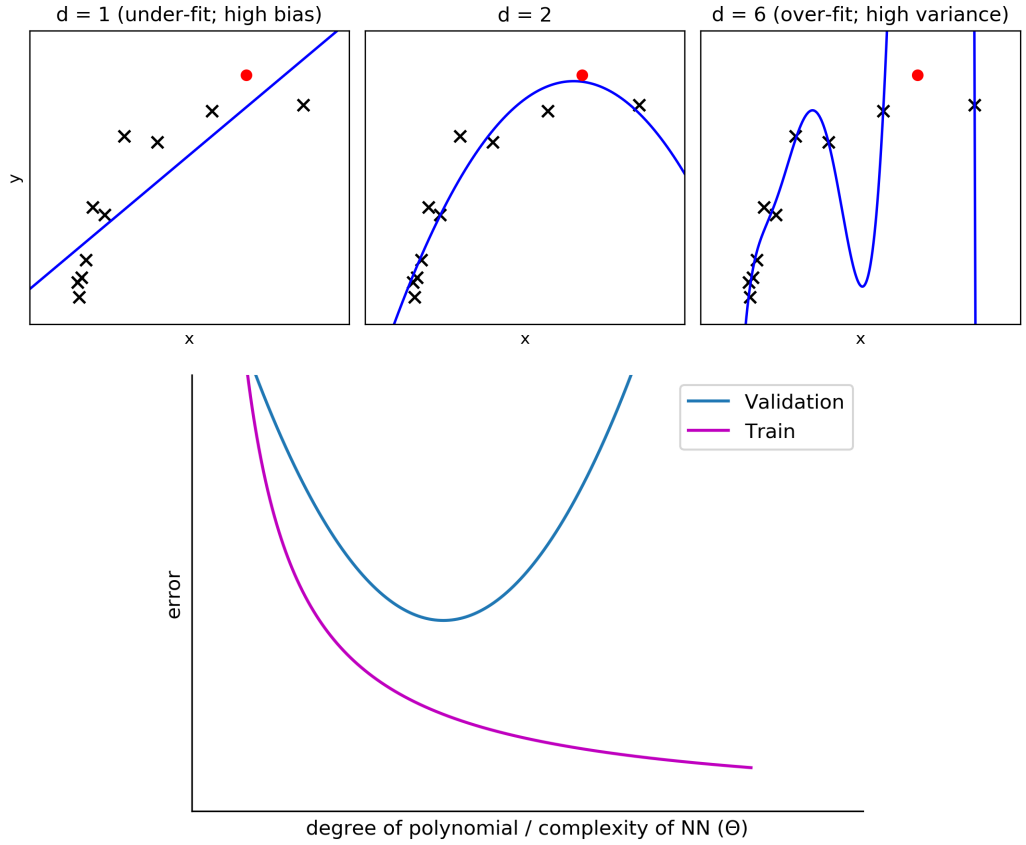


Figure 3.5: The Bias vs Variance problem. On the leftmost example we see that when we try to fit the data with a straight line (linear classifier) both the training error and the validation error are high. That is because the complexity of a straight line model is not sufficient to represent the problem we are trying to solve and the data is underfitted. On the rightmost example, although the training error approaches zero, the model is far from optimal. This model lacks generalization as the data has been overfitted and for that reason the validation error in that case is again high. When faced with an unseen example (red dot) both these models produce significant error. The optimal model is the one in the middle, when we see that the training error has decreased and validation error does not start to increase.

3.3 Input encoding for Neural Networks

When dealing with a scheme that concerns numerical data (e.g. prices and square feet of apartments when trying to predict the value of properties with a linear regression model or height, weight and hormonal levels when trying to predict the likelihood that a patient develops diabetes with a binary classification model) handling the input is fairly simple; there just needs to be some pre-processing of the input data to remove outliers and perhaps a normalization of the inputs to $[0, 1]$ in order to avoid vanishing gradient problems.

But when dealing with an NLP problem the input is textual data, that means that we might be dealing with words, letters or part-of-speech tags. All these instances fall into the more general class of categorical data. So the question that naturally arises is how do we go from textual data or categorical data to numerical representations that neural networks and other machine learning methods can process?

3.3.1 One-hot encoding of categorical features

One way to numerically represent categorical inputs is to assign a unique dimension for each possible feature. To illustrate this, let us borrow an example from chapter 8 of [11]. "If we need to represent as input a word from a vocabulary of 40,000 items, $\mathbf{x}$ will be a 40,000 dimensional vector, where dimension number 23,257 for instance corresponds to the word *dog* and dimension number 12,725 corresponds to the word *cat*. This is called *one-hot encoding* as the resulting feature vector $\mathbf{x}$ for a word will have a single dimension with value 1 and all other dimensions will have a value of 0." In this way each feature has its own dimension. Consequently the dimensionality of the one-hot input vector is the same as the number of distinct features. That raises the first caveat of working with one-hot encoded data; namely, for high cardinality variables (i.e. variables with many unique categories) the dimensionality of the transformed vector may be overbearing.

However, the main disadvantage of one-hot encoding is that it lacks in generalization properties. If some features provide similar clues, it is worthwhile to provide a feature representation that is able to maintain these similarities. To demonstrate this let us consider a frivolous yet illustrative example. Suppose there is a groceries store that sells apples, oranges and, oddly enough, octopi(!) and we want to model the customers' buying preferences to better accommodate their shopping needs. Using one-hot encoding we denote each selling item to one dimension and represent our input $\mathbf{x}$ with a tree-dimensional feature vector as shown in Fig. 3.6.

	x		
Apple	1	0	0
Orange	0	1	0
Octopus	0	0	1

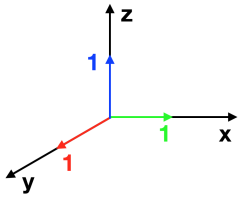


Figure 3.6: Example of one-hot vector encoding

The features here are completely independent from one another. The feature "item is apple" is as dissimilar to "item is orange" as it is to "item is octopus". Indeed, the cosine similarity of these vectors is 0, seeing that the vector are orthogonal. Semantically though we could argue that apples are closer to oranges, both being fruits, than they are to octopi. With the one-hot encoding this connection was lost, but if instead we had encoded the data with two dimensions we could have assigned the features to values that conserve this property. Even by arbitrarily encoding them as demonstrated in Fig. 3.7 we can ensure that apples will be closer to oranges in the vector space than they would be to octopi.

	x	
Apple	1/√2	1/√2
Orange	1	0
Octopus	-1	0

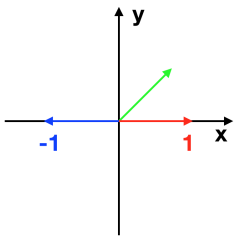


Figure 3.7: Example of dense vector encoding

3.3.2 Dense encodings (Feature Embeddings)

In the previous example we moved from sparse inputs of representing each feature as a unique dimension, to representing the input more densly in a space of lower dimensions. That is indeed the basic idea behind dense vector representations, where the core features are *embedded* into a d dimensional space, i.e. they are represented as a d dimensional vector. The dimension d is usually much smaller than the number of features. For the case of the previous example from [11] "each item in a vocabulary of 40,000 items (previously encoded as 40,000-dimensional one-hot vectors) can be represented as 100

or 200 dimensional vectors". These vector representations called *embeddings*, are treated as parameters of the network and can be trained like any other parameter of the function f.

Apart from reducing the dimensionality of the input, the dense representations offer another great benefit, which is the generalization power. Dense vectors provide a representation that is able to capture similarities in features that provide similar clues. When enough training data is available, the feature embeddings can be treated as any other model parameter i.e. they can be initialized to random values and then get updated during the training process to be converted to "good" vectors. If not enough training data is available "good vectors" must be somehow provided to the model. Indeed, there are a handful of algorithms available in the literature that provide us with "good" vectors, called *pre-trained word embeddings*.

In the sections below we briefly illustrate the workings of the algorithms that are used to extract the vectors of the two most widely used pre-trained embeddings packages in the literature; Word2Vec and Glove. Embedding vectors created using these algorithms have many advantages compared to earlier algorithms, such as latent semantic analysis and other matrix factorization variant methods.

Word2Vec

The widely popular Word2Vec algorithm was developed by Tomáš Mikolov and colleagues over a series of papers [54–58]. Mikolov et al. managed to reduce the computational complexity of learning word representations and hence made it possible to learn high dimensional word vectors on large amounts of data.

Unsupervised approaches, like Word2Vec, are based on the key concept that embedding vectors of similar words should have similar vectors. In principle, word similarity is hard to define and is strongly dependent to the specific task at hand. The current approaches, however, derive from the distributional hypothesis, which (contrary to the Chomsky tradition) claims that contextual information alone constitutes a viable representation of linguistic items. In other words, words are similar if they appear in similar contexts, with context being defined as neighboring words (Fig. 3.8).

The Word2Vec model is shown to perform well in many language tasks, including answering analogy questions, i.e. questions of the form *«a is to b as c is to ?»*. As Mikolov et al. mention "Somewhat surprisingly, it was found that similarity of word representations goes beyond simple syntactic regularities. Using a word offset technique where simple algebraic operations are performed on the word vectors, it was shown for example that vector("King")

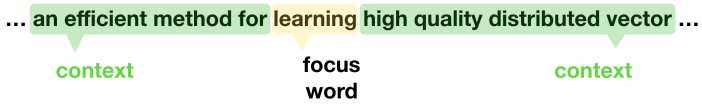


Figure 3.8: Example of the word context and focus word as used in Word2Vec

– vector("Man") + vector("Woman") results in a vector that is closest to the vector representation of the word Queen."

Glove

Another popular algorithm for word embeddings is GloVe. The name stands for *global vectors for word representation*, since the global corpus statistics are captured by the model. Glove is an unsupervised learning algorithm for obtaining word embeddings by aggregating the word co-occurrence matrix from a corpus. The resulting embeddings show interesting linear substructures of the words in vector space, as showcased in Fig. 3.9.

The work in GloVe is based on the argument of Pennington et al. [59] that the relationship of words can be examined by studying the ratio of their co-occurrence probabilities with various probe words, k. An example of this is illustrated in Fig. 3.10.

This argument suggests that word vector learning should be concerned with ratios of co-occurrence probabilities rather than the probabilities themselves. Therefore, the training objective of GloVe is to learn word vectors such that their dot product equals the logarithm of the words' probability of co-occurrence. Given that the logarithm of a ratio equals the difference of logarithms, this training objective associates the logarithm of ratios of co-occurrence probabilities with vector differences in the word vector space. Hence the information that these ratios of co-occurrence probabilities contain is also transferred to the vector differences.

The Glove model was trained over many corpora to produce word embeddings, including a Wikipedia and Gigaword corpus of 6 billion tokens, a Common Crawl web data corpus of 42 billion tokens, another Common Crawl corpus of 840 billion tokens and a Twitter corpus of 27 billion tokens[1].

The GloVe word vectors, similarly to word2vec, perform very well on word analogy tasks. Compared to word2vec, for the same amount of training time GloVe shows improved accuracy on the word analogy task (Fig. 3.11). Apart from that, the Glove model produces state of the art results in other tasks

[1]The GloVe pre-trained word vectors are available to download at https://nlp.stanford.edu/projects/glove/

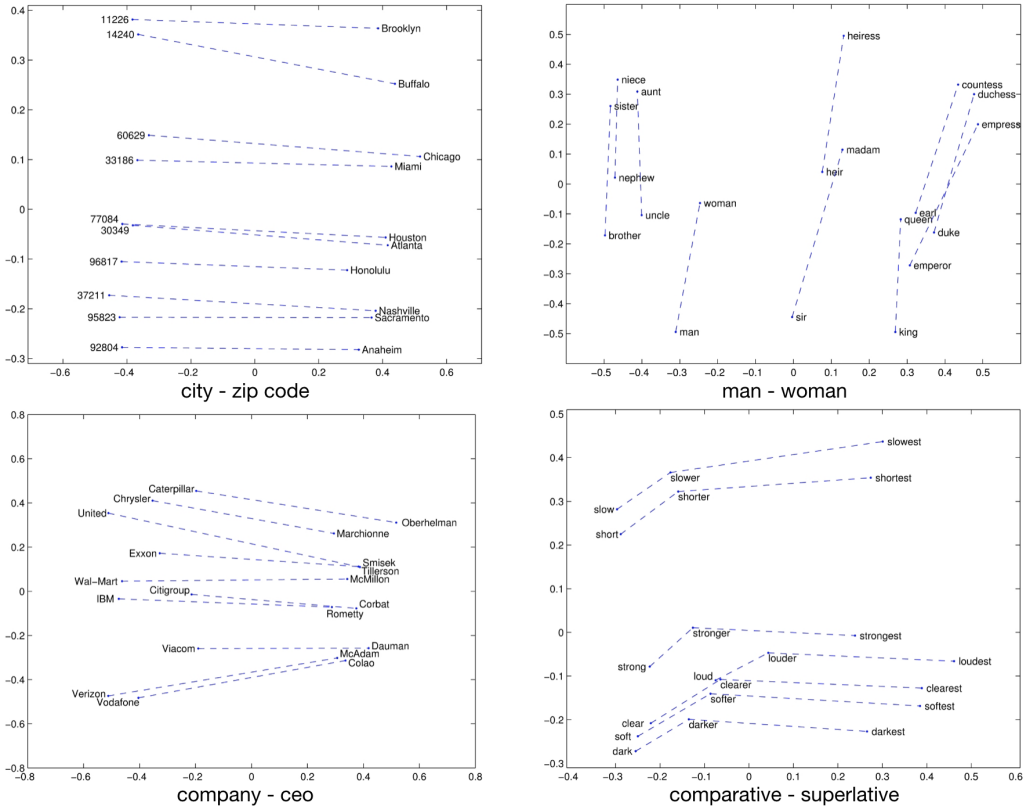


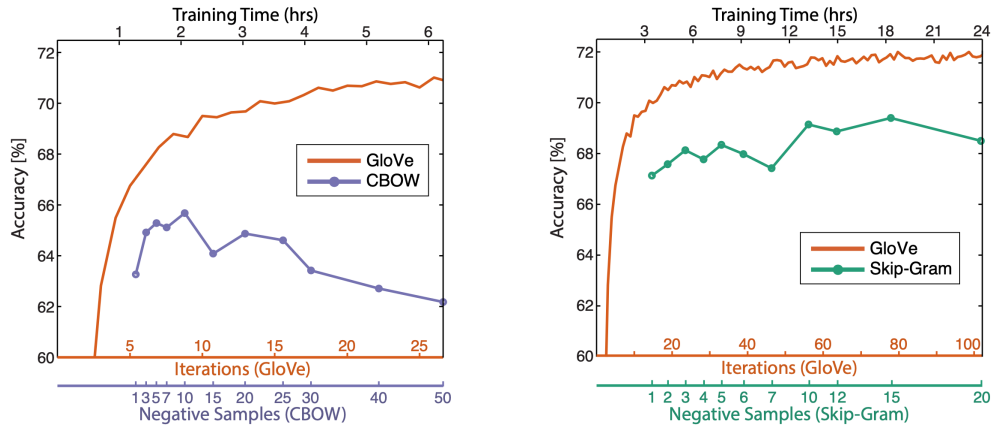
Figure 3.9: Glove vectors visualizations: The underlying concept that distinguishes man from woman, i.e. sex or gender, may be equivalently specified by various other word pairs, such as king and queen or brother and sister. To state this observation mathematically, we might expect that the vector differences man - woman, king - queen, and brother - sister might all be roughly equal. This property and other interesting patterns can be observed in the above set of visualizations. (source: https://nlp.stanford.edu/projects/glove/)

such as word similarity and named entity recognition [2].

[2]Named-entity recognition (NER) is a subtask of information extraction that seeks to locate and classify named entity mentioned in unstructured text into pre-defined categories such as person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc.

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Figure 3.10: Co-occurrence probabilities for target words *ice* and *steam* with selected context words from a 6 billion token corpus. Only in the ratio does noise from non-discriminative words like *water* and *fashion* cancel out, so that large values (much greater than 1) correlate well with properties specific to *ice*, and small values (much less than 1) correlate well with properties specific of *steam* (source: [59])



(a) GloVe vs CBOW (b) GloVe vs Skip-Gram

Figure 3.11: Comparison of Glove to Word2Vec: Overall accuracy on the word analogy task as a function of training time, which is governed by the number of iterations for GloVe and by the number of negative samples for CBOW (a) and skip-gram (b) (source [59])

3.4 Recurent Neural Networks

In the previous sections, we covered the topics of supervised learning and described feed-forward neural networks. Apart from the feed-forward networks, there exist other architectures, e.g. convolutional-and-pooling architectures (CNNs), and recurrent neural networks (RNNs) that are more specialized for dealing with language data. We will maintain our focus around the RNNs that were the networks investigated in this thesis.

RNNs are neural architectures designed to capture patterns and regularities in sequences. The main advantage that they offer is that they can look

at "infinite windows" around a focus word and pinpoint informative sequential patterns in those windows, thus allowing the modeling of non-markovian dependencies.

The RNN architectures are primarily used as feature extractors, meaning that they are not used as a standalone component, but rather produce a vector (or a sequence of vectors) that are given as input to other parts of the network that will eventually make the predictions. The network is trained end-to-end, i.e. the predicting component and the recurrent component are trained jointly, and the recurrent component of the network will capture those elements of the input that are useful for the prediction task.

Thus RNNs allow representing arbitrarily sized sequential inputs in fixed-size vectors, while paying attention to the structured properties of the inputs [60]. In particular, those with gated architectures, such as the LSTM and the Gated Recurrent Unit (GRU), perform very well at capturing the statistical patterns of sequential inputs.

To describe the RNN architecture in mathematical terms, we follow chapter 14 of [11]. Let us use $\mathbf{x_{i:j}}$ to denote the sequence of vectors $\mathbf{x_i}, ..., \mathbf{x_j}$. The RNN is a function that takes as input an arbitrary length ordered sequence of n d_{in}-dimensional vectors $\mathbf{x_{1:n}} = \mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n}$, ($\mathbf{x_i} \in \mathbb{R}^{d_{in}}$) and returns a single d_{out} dimensional vector $\mathbf{y_n} \in \mathbb{R}^{d_{out}}$:

$$\begin{aligned} \mathbf{y_n} &= RNN(\mathbf{x_{1:n}}) \\ \mathbf{x_i} \in \mathbb{R}^{d_{in}}, \mathbf{y_n} &\in \mathbb{R}^{d_{out}} \end{aligned} \tag{3.10}$$

We can further define a function RNN* that will be returning the sequence:

$$\begin{aligned} \mathbf{y_{1:n}} &= RNN^*(\mathbf{x_{1:n}}) \\ \mathbf{y_i} &= RNN(\mathbf{x_{1:i}}) \\ \mathbf{x_i} \in \mathbb{R}^{d_{in}}, \mathbf{y_i} &\in \mathbb{R}^{d_{out}} \end{aligned} \tag{3.11}$$

The output vector $\mathbf{y_n}$ is then used for further prediction and we also see how the RNN function provides a framework for conditioning on the entire history $\mathbf{x_1}, ..., \mathbf{x_i}$ without resorting to the Markov assumption.

The RNN can also be defined recursively, using a function R that takes a state vector $\mathbf{s_{i-1}}$ and a vector $\mathbf{x_i}$ as inputs and returns a new state vector $\mathbf{s_i}$. The state vector $\mathbf{s_i}$ is then mapped to an output vector $\mathbf{y_i}$ (usually the state vector is identical to the output and only in rare cases a deterministic function $O()$ is used to do the mapping). The base of the recursion is an initial state vector, $\mathbf{s_0}$, which is also an input to the RNN.

$$
\begin{aligned}
RNN^*(\mathbf{x_{1:n}}; \mathbf{s_0}) &= \mathbf{y_{1:n}} \\
\mathbf{y_i} &= \mathbf{s_i} \quad (rarely \quad \mathbf{y_i} = O(\mathbf{s_i}) \ ) \\
\mathbf{s_i} &= R(\mathbf{s_{i-1}}, \mathbf{x_i}) \\
\mathbf{x_i} \in \mathbb{R}^{d_{in}}, \mathbf{y_i} \in \mathbb{R}^{d_{out}}, \mathbf{s_i} &\in \mathbb{R}^{d_out} \ (rarely \quad \mathbf{s_i} \in \mathbb{R}^{f(d_{out})})
\end{aligned}
\tag{3.12}
$$

The fact that function R remains the same means that the parameters θ are shared across all time steps. The computation of the RNN changes only through the state vector $\mathbf{s_i}$. Graphically, the RNN has been traditionally presented as in Fig. 3.12.

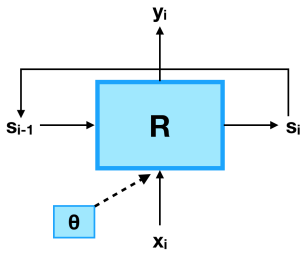


Figure 3.12: Graphical representation of an RNN (recursive). The parameters θ are shared across all time steps.

The illustration in Fig. 3.12 adheres to the recursive definition and is theoreticaly valid for arbitrarily long sequences. However in practise all input sequences, no matter how long, are finite sequences. Hence one can unroll the recursion and end up with an illustration like the one in Fig. 3.13. Intuitively, the process of unrolling an RNN can be thought of as a chronophotography of the network in motion (see Fig. 3.14).

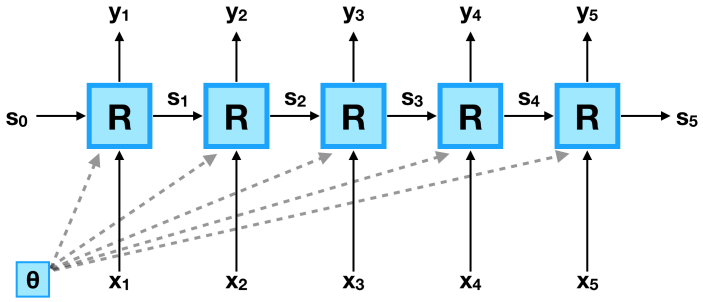


Figure 3.13: Graphical representation of an RNN (unrolled). The parameters θ are shared across all time steps.

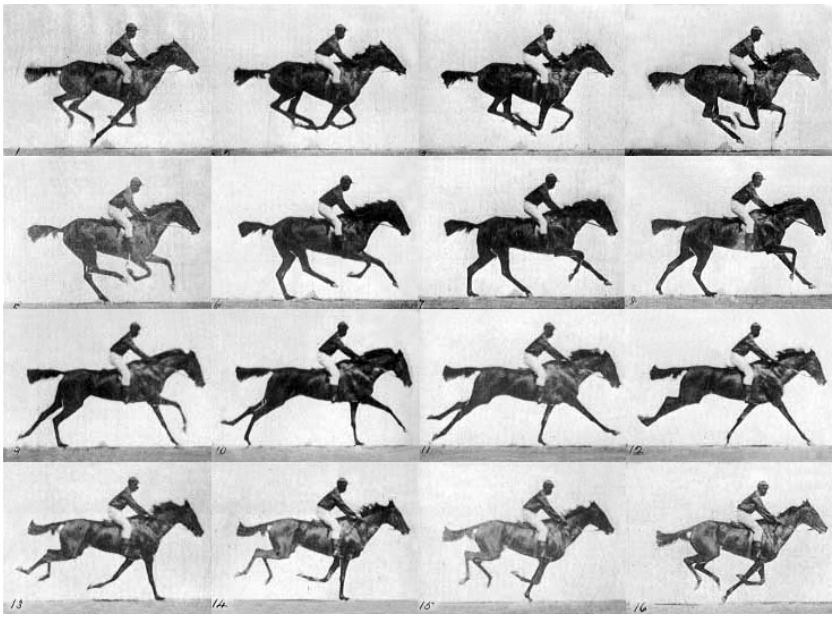


Figure 3.14: Chronophotography as a metaphor for RNN unrolling: Chronophotography is a photographic technique from the Victorian era, which captures multiple phases of movements. The unrolled RNN schematic can be thought of as chronophotography which captures the phases of a sequence encoding instead of a movement. (source: The Horse in Motion, Eadweard Muybridge (1878))

The function R is the factor that differentiates the various RNN architectures, e.g. an LSTM network from a GRU network. Different instantiations of the R function will result in different network structures, that will exhibit different properties. However, the same abstract interface represents any kind of RNN.

Finally we note that $\mathbf{s_n}$ and $\mathbf{y_n}$ are in fact *encoding* the entire input sequence. This can be easily demonstrated by expanding the recursion in Eq. 3.12. For example, assuming $i = 4$ we get:

$$\begin{aligned}
\mathbf{s_4} &= R(\mathbf{s_3}, \mathbf{x_4}) \\
&= R(\overbrace{R(\mathbf{s_2}, \mathbf{x_3})}^{\mathbf{s_3}}, \mathbf{x_4}) \\
&= R(R(\overbrace{R(\mathbf{s_1}, \mathbf{x_2})}^{\mathbf{s_2}}, \mathbf{x_3}), \mathbf{x_4}) \\
&= R(R(R(\overbrace{R(\mathbf{s_0}, \mathbf{x_1})}^{\mathbf{s_1}}, \mathbf{x_2}), \mathbf{x_3}), \mathbf{x_4})
\end{aligned} \tag{3.13}$$

Thus training the network is equivalent to setting the parameters of R in a way that the states convey useful information for the prediction task that follows.

3.4.1 Long Short-Term Memory networks

The Long Short-Term Memory (LSTM) architecture [61] was designed to solve the vanishing gradients problem, and is the first to introduce the gating mechanism. The LSTM architecture, inspired by the function of human memory, splits the state vector $\mathbf{s_i}$ into two halves, the first one functioning as "memory cells" and the other functioning as working memory. The memory cells are designed to preserve the memory and the error gradients across time. They are controlled through *differentiable gating components*, i.e. smooth mathematical functions that simulate logical gates. For every input state encountered during the training process, a gate is used to decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forgotten. Mathematically, the LSTM architecture is defined as:

$$
\begin{aligned}
\mathbf{s_j} = R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) &= [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} &= \mathbf{f} \odot \mathbf{c_{j-1}} + \mathbf{i} \odot \mathbf{z} \\
\mathbf{h_j} &= \mathbf{o} \odot \tanh(\mathbf{c_j}) \\
\mathbf{i} &= \sigma(\mathbf{x_j}\mathbf{W^{xi}} + \mathbf{h_{j-1}}\mathbf{W^{hi}}) \\
\mathbf{f} &= \sigma(\mathbf{x_j}\mathbf{W^{xf}} + \mathbf{h_{j-1}}\mathbf{W^{hf}}) \\
\mathbf{o} &= \sigma(\mathbf{x_j}\mathbf{W^{xo}} + \mathbf{h_{j-1}}\mathbf{W^{ho}}) \\
\mathbf{z} &= \tanh(\mathbf{x_j}\mathbf{W^{xz}} + \mathbf{h_{j-1}}\mathbf{W^{hz}})
\end{aligned}
$$

$$
\mathbf{y_j} = \mathbf{O}_{LSTM}(\mathbf{s_j}) = \mathbf{h_j}
$$

$$
\mathbf{s_j} \in \mathbb{R}^{2d_h}, \mathbf{x_i} \in \mathbb{R}^{d_x}, \mathbf{c_j}, \mathbf{h_j}, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{z} \in \mathbb{R}^{d_h}, \mathbf{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \mathbf{W^{ho}} \in \mathbb{R}^{d_h \times d_h} \tag{3.14}
$$

The state at time j is composed of two vectors, $\mathbf{c_j}$ which is the memory component and $\mathbf{h_j}$ that is the hidden state component. There are three gates, $\mathbf{i}$, $\mathbf{f}$, and $\mathbf{o}$, controlling for **i**nput, **f**orget, and **o**utput. The computation of those gates is done by linearly combining the current input $\mathbf{x_j}$ and the previous state $\mathbf{h_{j-1}}$, and then passing it through a sigmoid activation function. The $\mathbf{z}$ vector is a potential candidate for update. It is computed by linearly combining $\mathbf{x_j}$ and $\mathbf{h_{j-1}}$, and then passing it through a tanh activation function. The forget gate controls how much of the previous memory to keep $(\mathbf{f} \odot \mathbf{c_{j-1}})$[3], and the input gate controls how much of the proposed update to keep $(\mathbf{i} \odot \mathbf{z})$ when updating the memory $\mathbf{c_j}$. The value of $\mathbf{h_j}$ is computed by passing the memory content $\mathbf{c_j}$ through a tanh activation function and then through the output gate. The output $\mathbf{y_j}$ is the same as $\mathbf{h_j}$. The gating mechanisms allow for gradients related to the memory part $\mathbf{c_j}$ to stay high across very long time ranges, giving the network the ability to "remember".

[3]The symbol $\odot$ is the Hadamard product, i.e the elementwise multiplication

Chapter 4

Data Set Creation

In every machine learning project the first and often very tedious task is to find or create a data set on which to work on. Regarding the task at hand, the STAC corpus provides a dataset that holds information collected from players participating in SoC games on the jSettlers platform. This dataset incorporates annotated chat conversations that took place between players during gameplay along with all the extra-linguistic events, concerning game outcomes and players' actions.

The annotated corpus of the chat data has been extensively used in research that has yielded interesting results. The data regarding the game actions, on the other hand, has been neglected so far, despite the fact that merging this information is very often mentioned as a future step to the research. One plausible reason that hampers the investigation of a combined chat and game action setup, may be attributed simply to inconvenience. In other words, the problem may lie in the fact that, in contrast to the annotated STAC dialogue corpus, that is explicit to use and user-friendly, no effort has been made to develop and organise the game information to an equivalently easy to use dataset.

This chapter elaborates on how the game log files from the STAC corpus were processed to create a clear and informative data set, suitable for training in machine learning projects.

4.1 Problem modeling

In this thesis the endeavor was to develop an architecture that would emulate players' behaviour, in the sense that it would predict actions in a SoC game, taking into account the progress of the game, specifically all the game actions and chat discussions that have taken place during gameplay.

Given the complexity of the game, neural networks were considered as the most appropriate approach for this prediction task. In particular a combined architecture, that would concatenate a representation of the state in the game (board layout, actions, etc.) and a representation of the chat dialogues between players to produce a prediction of the next move in a SoC game (see Fig. 4.1).

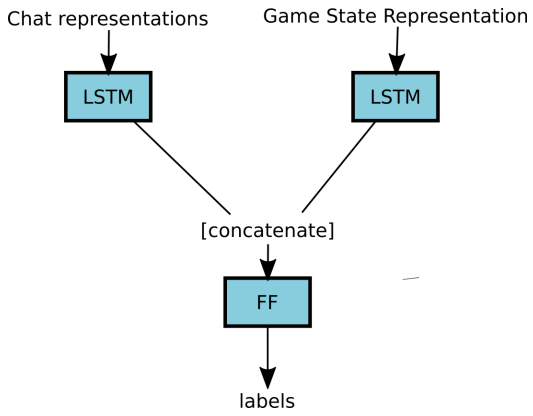


Figure 4.1: Schematic of the basic neural network architecture for combining the chat dialogue data and the gameplay data to predict the next move in a SoC game.

In the context of SoC, neural networks have been employed for Deep Reinforcement Learning (DRL) in [7, 8]. In both these implementations the task was to improve the negotiation skills of a jSettlers agent, examining different network architectures. The DRL agents' objective is to learn to choose the most profitable action from a manually restricted set of 72 trading actions, given as input the state in a SoC game. The input to the neural network is therefore given as a description of the game board, namely the type of resource each hexagon tile on the board produces, the location where settlements,cities and roads have been placed by the agent and his opponents, information about the robber's position and information the agent privately holds about his resources.

Following the same notion that the state in a SoC game can be represented by the description of the board of the game, the features in this thesis were selected to depict the board and the players' states. In the features that represent the gamestate the information is divided in three main categories: turn, board and players. More precisely, the *turn* feature signifies how far the game has progressed, the *board* feature set holds information for the board layout, e.g. the initial setup of the hexagonal tiles, the position of the robber on the board etc, and the *players* feature set represents the relevant

information for each of the 4 players that can participate in a SoC game. Each player has a corresponding playerstate, containing their identification information i.e. number id and nickname, their constructions information i.e. roads, settlements and cities coordinates on the board and their development cards information.

The caveat here is that initially all players place two settlements and two roads on the board but as the game progresses the number of pieces they place on the board is not strictly defined by the game rules. That means that one player may choose a "building strategy" and focus on placing many pieces on the board while another one may remain on his initial two constructions and instead focus on buying development cards in order to reach the 10 victory points needed to win the game. In order to restrict the playerstate vector to a fixed length its size is predefined by the max number of pieces the player has in his inventory and the total number of development cards forming the deck. Naturally it is a rare case that some player will manage to place all of his building block on the board or buy all of the development cards of the deck but this sets an upper bound for the playerstate representation. The representation of the gamestate features can be seen schematically in Fig. 4.2

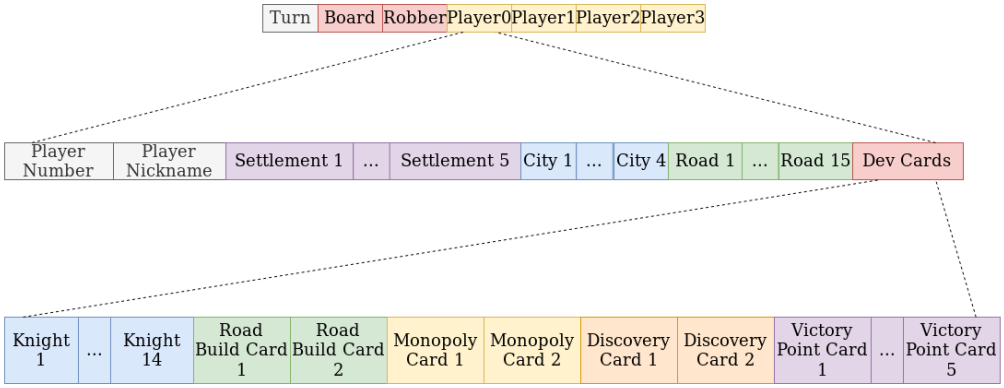


Figure 4.2: Gamestate features - representation of the board and players' state in the game. In the first layer we see the main feature sets, distinguished by color (gray for turn, blue for board, yellow for players). Each player is represented by his playerstate feature set shown in the levels below.

As far as the chat between the players is concerned its representation holds the discourse/sentence emitted along with information of the turn of the game during which this sentence was written in the chat dialogue box and the nickname of the player that wrote it.

A main challenge regarding the modelling of the game becomes obvious from the following quote taken from the SoC game rules: "after receiving

resources the player who rolled the dice may trade resources and build as much as they want until they are done". In essence, during each turn of the game the player who rolled the dice has the freedom to perform as many actions as they please without restrictions explicitly imposed by the game rules. For example, the player during their round may perform multiple times the same action (e.g. build roads) and/or perform many different types of actions (e.g. build a road and buy a development card etc). In practice, the number of such actions is rather limited, due to a plethora of conditions stemming from the rules that need to be satisfied.

Since a fixed length of input features and output labels are required for the NNs that will model the game, some restrictions needed to be set for the creation of the labels in order to address the first part of the challenge, i.e. multiple actions of the same type. Specifically, the restriction employed here is that labels show if some action was or was not performed but not how many times it was performed. For example if a player builds 2 roads the ground truth label for the prediction task is an indication that *roads were built during this round* and the number of roads is ignored.

Furthermore, and in order to address the second part of the challenge, i.e. multiple types of action, the problem was modeled as a *multi-label* classification task instead of the more common *multi-class* classification task. The notion of labels[1] or tags is preferred instead of classes to make the distinction that labels are not mutually exclusive in contrast to the definition of classes, demanding mutual exclusiveness. A primary example of multi-label classification type problems is that of text categorisation, where each text may belong to several predefined topics simultaneously e.g. a newspaper article may need to simultaneously be assigned to labels/tags indicating *foreign affairs*, *economy* and *2019*. In a similar fashion for the case of a SoC game, a player during his round may not just build roads per se, but may perform further different actions as well. In such cases the output is not a one-hot vector that indicates to which class an example belongs to, rather a vector of ones and zeros that indicates which labels hold true for that given example.

The features and prediction labels described above are retrieved from the game log files of the STAC corpus after some processing in the steps shown in Fig. 4.3 and described in the following sections of the chapter. After the log file processing, 3 data tables are assigned to each game containing the gamestate features, the chat discourses and the prediction labels respectively.

[1]In order to avoid confusion the reader should pay some attention to the different definitions of the term *label* in the literature, used both to define the ground truth outcome of a prediction task as well as to provide an alternative to the definition of the term *class*.

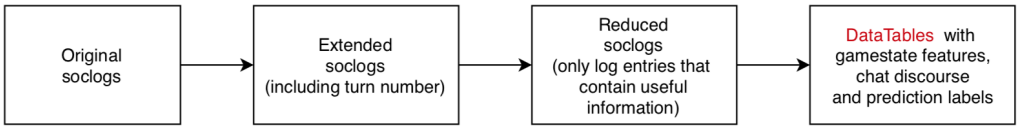


Figure 4.3: SoC game log file processing pipeline schematic. Each log file undergoes the different stages shown above, resulting in the final three tables that contain the features and labels of the prediction task.

4.2 Original soclog files

The data used in this thesis is part of the STAC corpus, a corpus of strategic chat conversations manually annotated with negotiation-related information, dialogue acts and discourse structures. This corpus was collected from the jSettlers framework and consists of 60 log files. These log files contain the information of the Server messages, Chat history (as described in section [ref section jSettlers of chapter SOC (2)]) as well as various log entries from different classes of the jSettlers code. These log files, with the extension .soclog (from now on refered as soclog files) allow the replay of an entire game. Of the 60 soclog files, 45 are segmented into Elementary Discourse Units (EDUs) and are annotated and available online[2].

In this thesis there is no need for the annotated version and the original soclog files are used to process and develop the dataset used for this work. Of the 60 original soclog files, 2 files (pilot18 and pilot19) were ignored due to errors that forced the players to reset the board at some point during the game. Notice that all of our data come from version 1 of the jSettlers code. In later versions of the game changes have been made to this code. These modifications will need to be taken into account in case the data processing code developed in this work needs to be used for expanding the existing dataset in the future, using soclog files of these later version of the game. These attention points are listed throughout the following sections.

Below is an example of a log entry taken from pilot01 soclog file.

```
2012:06:06:20:12:10:230:+0100:GAME-TEXT-MESSAGE:[game=3|
player=inca|speaking-queue=[]|clay=0|ore=1|sheep=1|
wheat=0|wood=0|unknown=0|knights=0|roads=[99,152,82]|
settlements=[116,152]|cities=[]|dev-cards=1|
text=not me, sorry]
```

The various fields are distinguished by : and | delimiters. The translation

[2]https://www.irit.fr/STAC/corpus.html

of this log entry is detailed in Table 4.1.

Field	Description
2012:06:06:20:12:10:230:+0100	Timestamp
GAME-TEXT-MESSAGE	type of message
game=3	code number of the game
player=inca	nickname of the player that took action
speaking-queue=[]	speaking queue of the chat server is empty
clay=0|ore=1|sheep=1|wheat=0| wood=0|unknown=0	resources of the player "inca"
knights=0	number of knights player "inca" has played
roads=[99,152,82]	hex encoded coordinates of the roads that player "inca" has built
settlements=[116,152]	hex encoded coordinates of the settlements that player "inca" has built
cities=[]	player "inca" has not upgraded settlements to cities
dev-cards=1	player "inca" possesses one development card
text=not me, sorry	player "inca" writes the text message "not me, sorry" to the chat box

Table 4.1: Explanation of a message from a raw soclog file

4.3 Extended soclog files

The first step of the processing is to clear the timestamp and instead segment the entries according to game turns. The notion adopted for this task was to name the initial setup phase of the game when players place the first two roads and settlements as *turn 0*. From then onward turns are counted whenever a player rolls the dice.

In the jSettlers code there are some signals, that are used to coordinate the running of different classes of the code. These signals are saved in the log entries with a message of *SOCGameState* type. An example of such a message is shown below.

```
2011:10:10:16:26:20:741:+0100:SOCGameState:game=pilot01|
state=10
```

These signals are encoded by an integer number $\in [0, 1000]$ that is saved in the log entry and a signal name that is used through the jSettlers code. A listing of these signals, together with their explanations according to the v.1 of the jSettlers code can be seen in Table 4.2. In future versions of the game the signal list has been expanded as new game options have been added.

Using these code signals the soclog files are extended to include the information of the turn number as discussed earlier. The timestamp is replaced by the appropriate turn number that is incremented every time a game state=15 signal is encountered in the log entries.

4.4 Reduced soclog files

In this step of the file processing the soclog files are reduced in size as specific entries are selected. The selected entries are those of message types that contain information that can be useful and the rest of the entries that carry redundant or insufficient information are ignored. After this selection process the useful messages are parsed to extract the necessary information and form the concise representation of the game state (see Fig. 4.2), the labels and chat messages. The message types of the selected entries are listed and explained below.

SOCSitDown

```
0:SOCSitDown:game=pilot01|nickname=rennoc1|playerNumber=0|
robotFlag=false
```

This message appears in the soclogs when the players hit the *Sit Down* button and hold the information needed to collect players' nicknames and their corresponding number ids. Since some message refer to the players with their ids and others (including the chat references) refer to them only by their nickname, a connection/corelation of the two was considered necessary. The players' nickname are initialised as 'dummy' and when/if someone sits he changes the nickname.

The nickname information can also be found at the *SOCJoinGame* messages, however these also include the nicknames of people that join the game just to watch, like for example *Markus* in pilot01, who hosts the game and does not play but talks to the chat to help the players with the game rules, their moves and the interface of jSettlers.

Code	Name	Explanation
1	READY	game is ready to begin
5	START1A	players place their 1st settlement
6	START1B	players place their 1st road
10	START2A	players place their 2nd settlement
11	START2B	players place their 2nd road
15	PLAY	start a normal turn, time to roll the dice or play a card
20	PLAY1	done rolling / moving the robber on rolled 7 case
30	PLACING ROAD	player is placing a road piece on the board
31	PLACING SETM	player is placing a settlement piece on the board
32	PLACING CITY	player is placing a city piece on the board
33	PLACING ROBBER	the robber is being moved on a new land hexagon
50	WAITING FOR DISCARDS	waiting for players to discard resources when 7 was rolled
51	WAITING FOR CHOICE	waiting for a player to choose a rival from which to steal a card after he rolled 7 or played a Knight card. (renamed since v.2)
52	WAITING FOR DISCOVERY	after discovery card, waiting for the player to choose 2 resources
53	WAITING FOR MONOPOLY	after monopoly card, waiting for the player to choose a resource
1000	OVER	someone won or all players have left the game

Table 4.2: SoC state signals explanation. Described here are the most important state signals, integer code number, signal name in jSettlers code and brief explanation of the meaning of this signal. (updated in later versions of the game)

SOCBoardLayout

```
0:SOCBoardLayout:game=pilot01|hexLayout={ 9 6 67 6 6 2 5 1
66 8 2 3 1 2 6 6 5 3 4 1 4 11 36 5 4 0 5 6 6 4 3 3 97 21
6 12 6 }| numberLayout={ -1 -1 -1 -1 -1 8 9 6 -1 -1 2 4 3
7 -1 -1 5 1 8 2 5 -1 -1 7 6 -1 1 -1 -1 3 0 4 -1 -1 -1 -1
-1}|robberHex=0x97
```

The *hexLayout* field of this message holds the information of the numerical representation of the hexagonal board tiles that make up the board. The encoded numbers that run from 0 to 5 are used to represent the assorted land tiles while the encoded numbers that run from 6 and higher are used to represent the different sea tile types, as described in Table 4.3. In a similar fashion the *numberLayout* field holds the information of the dice tiles that correspond to each hexagonal tile on the board. For the land tiles the encoding number represents the appropriate dice result as described in Table 4.4. The sea and desert tiles, that are not associated with a dice tile, are encoded with -1. Finally, the *robberHex* field holds a hexadecimal number that refers to the coordinate of the robbers location on the board, initially being the desert tile.

It is useful to mention that in future version of jSettlers the code for sea tile encoding number (6) and the desert tile encoding number (0) have been interchanged. An example of a board tile encoding from pilot01 can be seen in Fig. 4.4.

Number	**Tile Description**
6	Water
0	Desert
1	Clay
2	Ore
3	Sheep
4	Wheat
5	Wood
7	Miscellaneous port (3:1) facing direction 1
8	Miscellaneous port (3:1) facing direction 2
9	Miscellaneous port (3:1) facing direction 3
10	Miscellaneous port (3:1) facing direction 4
11	Miscellaneous port (3:1) facing direction 5
12	Miscellaneous port (3:1) facing direction 6
16+	Non-miscellaneous ports (2:1) of various resources and directions

Table 4.3: The jSettlers encoding of the board tiles. The *hexLayout* field of the *SOCBoardLayout* messages uses these numbers to describe the different land and sea tiles that make up the game board. Attention: in later versions of the game the water and desert encoding numbers have been interchanged.

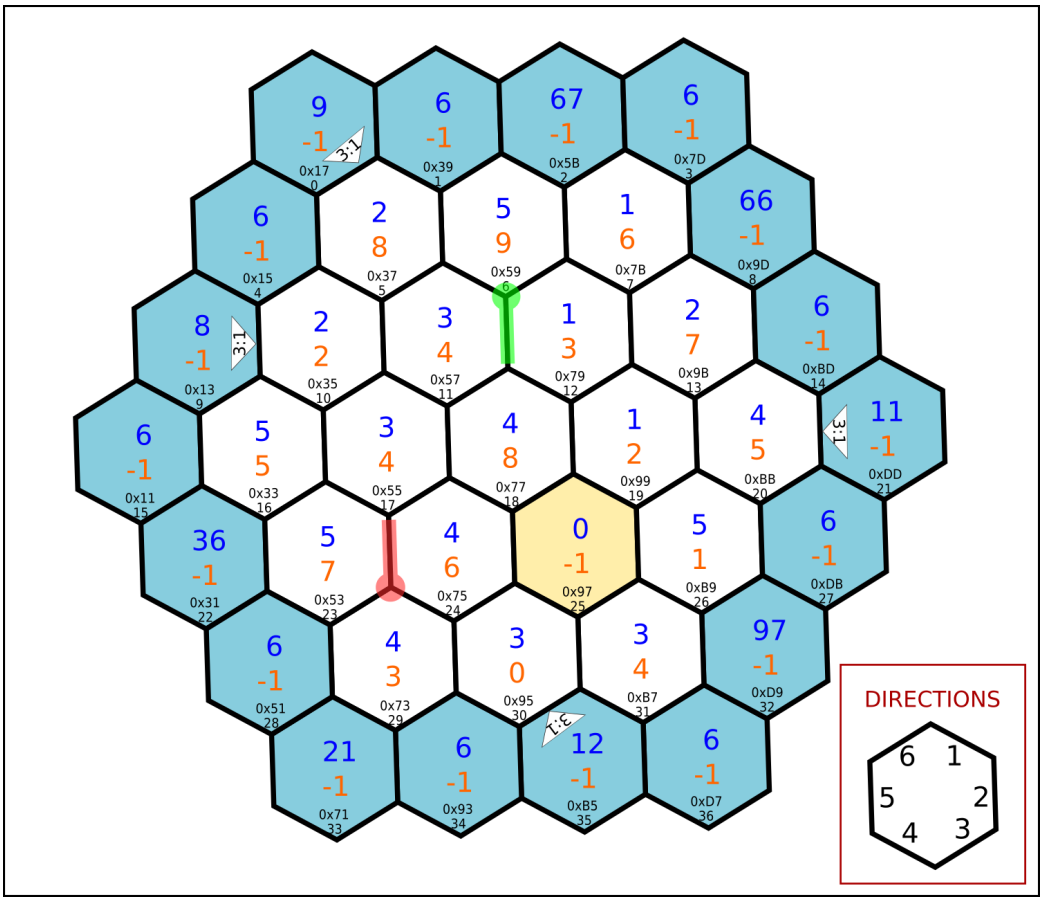


Figure 4.4: The jSettlers numbered encoding of the game board. The different types of land and sea tiles are identified by a *hexLayout* number (marked in blue) that indicates the resource that is produced or the tile's properties. The dice results that will activate each tile are encoded by the *numLayout* numbers (marked in orange). Also depicted are the placements of the first two roads and settlements for *rennoc1* (in green) and *Dave* (in red). Attention: in later version of the game the water and desert encoding numbers have been interchanged.

SOCMoveRobber

```
8:SOCMoveRobber:game=pilot01|playerNumber=1|coord=55
```

This type of message is registered in the soclogs whenever a player moves the robber and is used to update the information concerning the robber's position. The value of the *coord* field is the hexadecimal number of the coordinate on the board.

Number	Dice Outcome
0	2
1	3
2	4
3	5
4	6
5	8
6	9
7	10
8	11
9	12
-1	no dice result association

Table 4.4: The jSettlers encoding of the dice tiles. As mentioned in Chapter 2, two dice are used in SOC, giving an outcome in $[2, 12]$. There are two dice tiles to represent each possible outcome twice with the exception of 2 and 12, i.e. the least possible outcomes, that have only one copy and 7 that triggers a special case. In jSettlers the dice outcomes are encoded in the *numberLayout* field of the *SOCBoardLayout* message by an integer number in $[0, 9]$. The sea and desert tiles that are not associated with a dice tile have a value of -1.

SOCPutPiece

```
8:SOCPutPiece:game=pilot01|playerNumber=1|pieceType=0|
coord=aa
```

This message is used to update the constructions of a player. The message appears in the soclogs when the player of number id *playerNumber* places a piece on the game board. The values of the *pieceType* field are listed in Table 4.5 The value of the *coord* field is again the hexadecimal number of the coordinate on the board.

Number	Piece
0	Road
1	Settlement
2	City

Table 4.5: Encoding of piece types in jSettlers. In later version of jSettlers this list has been expanded to include ships, fortresses and villages.

SOCDevCard

```
6:SOCDevCard:game=pilot01|playerNum=3|actionType=0|cardType=0
```

This message is used to acquire information of development cards usage. Similar information can be found in *SOCSetPlayedDevCard* messages, but since they appear multiple times in the log with false value and they do not present the information as concisely, they were discarded. Explanations of *actionType* and *cardType* are provided in Tables 4.6 and 4.7.

Action	Type
player bought a development card	0
player played a development card	1

Table 4.6: Action types for development cards in jSettlers.

Development Card	Type
unknown	0
road building	1
discovery (year of plenty)	2
monopoly	3
Victory Point (capital/ governor's house)	4
Victory Point (library/ market)	5
Victory Point (university)	6
Victory Point (temple)	7
Victory Point (tower)	8
knight	9

Table 4.7: Development card types encoding in the JSettlers SOCDevCard class. Attention: in later versions of the game the unknown development card corresponds to card type 9 and the knight card corresponds to card type 0.

SOCPlayerElement

```
10:SOCPlayerElement:game=pilot01|playerNum=3|actionType=101|
elementType=4|value=1
```

These types of entries appear in the soclogs every time there is a change in the *elements* of a player, i.e. resources or construction blocks. The number of

the *value* field denotes how many elements were involved in a specific action of those described in Table 4.9. The codes of the various *elementTypes* are described in Table 4.8.

The *SOCPlayerElement* messages were used to construct a dataset that contained information about the players' resources as well. However it was thought that the information of the players' resources would give an unfair knowledge advantage to the prediction system, as some hidden exchanges (e.g. stealing cards from another player) are recorded in the soclog, despite the fact that this information is not available to all the players participating in the game.[3] Hence the dataset with the resources information was not used in training, but is available upon request for future projects.

Number	Element type
1	clay
2	ore
3	sheep
4	wheat
5	wood
6	unknown
10	road
11	settlement
12	city

Table 4.8: Element types encoding in jSettlers. Attention : *SOCPutPiece* and *SOCElementType* refer to the construction blocks with different code numbers.

SOCMakeOffer

```
12:SOCMakeOffer:game=pilot01|offer=game=pilot01|from=1|
to=true,false,false,false|give=clay=0|ore=1|sheep=0|wheat=0|
wood=0|unknown=0|get=clay=1|ore=0|sheep=0|wheat=0|wood=0|
unknown=0
```

This type of message is used to detect trading actions for the labels.

[3]When player x steal a resource from player y the Server informs the other players that "x stole a resource from y" but only x and y know which specific type of resource was stolen. Similarly, the resources gained after the dice have been rolled are announced to everyone and the players can be sure about their rivals' resources, but after the 7 is rolled they can only guess which of their half resources they have secretly discarded.

Action	Code	Description
SET	100	The element described by the *elementType* has a number of instances that is given in the *value* field.
GAIN	101	The element described by the *elementType* has gained as many instances as the *value* field dictates.
LOOSE	102	The element described by the *elementType* has been reduced by as many instances as the *value* field dictates.

Table 4.9: Action types for elements in jSettlers.

The *SOCMakeOffer* messages appear in the soclogs when a player makes a trading offer to another player or players. It describes the players to whom the offer is addressed, the resources that are being offered to them and the resources that are asked in exchange.

SOCAcceptOffer

```
32:SOCAcceptOffer:game=pilot01|accepting=1|offering=3
```

This type of message is used to detect trading actions for the labels as well. The *SOCAcceptOffer* is registered in the soclogs when a transaction of resources takes place between players and states the numbers of resource units that are being exchanged. This refers only to trading between players and does not include resource exchanges with the bank or from a port.

SOCGameTextMsg

```
2011:10:10:16:24:43:002:+0100:SOCGameTextMsg:game=pilot01|
nickname=Server|text=Dave built a settlement.
```

```
2011:10:10:16:33:29:517:+0100:SOCGameTextMsg:game=pilot01|
nickname=Dave|text=does anyone have any wood they would be
willing to trade for?
```

This type of message is listed in the soclogs every time the server or a player produces a text message and is predominantly used to collect the chat references of the players in the chat. The messages produced by people that are connected but do not participate in the game (e.g. administrator,

server host, etc) are saved along with the players' chat utterances as they are essential for the dialogue (see also SOCSitDown).

Another very potent candidate to extract this type of information could be the *GAME-TEXT-MESSAGE* type of message (see example at Section 4.2 and explanation at Table 4.1). Not only does it include the chat utterances of the players but also offers a very precise summary of the players' state. However these messages appear only when a player decides to speak in chat, meaning that silent players are never registered and would remain unnoticed.

```
2011:10:10:17:03:49:050:+0100:SOCGameTextMsg:game=pilot01|
nickname=Server|text=Tomm traded 4 sheep for 1 wood from
the bank.
```

```
2011:10:10:17:34:12:199:+0100:SOCGameTextMsg:game=pilot01|
nickname=Server|text=Tomm traded 2 sheep for 1 ore from a
port.
```

The text messages produced by the Server to inform the players about the developments in the game are distinguished from the players' chat by the nickname field. Hence the messages registered with the *Server* nickname are for the main part ignored with a few exceptions. In some cases where the game's code does not produce unambiguous log entries, the log entries contain insufficient information and/or are written multiple times with different values within the same game turn (e.g. there is some periodical checking of a condition, resulting in multiple false values until some action is detected) the server's announcements can be employed as a reliable source of information.

Such actions during the creation of this dataset were encountered when dealing with the bank and port trading transactions. The equivalent *SOCBankTrade* messages do not contain sufficient information as they register even the failed attempts to exchange goods with a bank or port (e.g. when a player tries to exchange resource with the bank but with the wrong ratio). Therefore the only reliable indication of a successful trading transaction with the bank or a port is the server's message.

4.5 Final Dataset

The end result of the soclogs parsing is the formation of a comprehensive dataset via the summarization of the information into datatables. Each of the

Index	Turn	boardHexLayout	boardNumLayou	Robber	player0num	player0nickname	pl0setm1	pl0setm2	pl0setm3	pl0setm4	pl0setm5	pl0city1	pl0city2	pl0city3	pl0city4	pl0road1	pl0road2	pl0road3
0	0	[9, 6, …	[-1, -1…	0x97	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
1	1	[9, 6, …	[-1, -1…	0x97	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
2	2	[9, 6, …	[-1, -1…	0x97	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
3	3	[9, 6, …	[-1, -1…	0x97	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
4	4	[9, 6, …	[-1, -1…	0x97	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
5	5	[9, 6, …	[-1, -1…	0x97	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
6	6	[9, 6, …	[-1, -1…	0x97	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
7	7	[9, 6, …	[-1, -1…	0x97	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
8	8	[9, 6, …	[-1, -1…	0x55	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
9	9	[9, 6, …	[-1, -1…	0x99	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
10	10	[9, 6, …	[-1, -1…	0xbb	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
11	11	[9, 6, …	[-1, -1…	0xbb	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
12	12	[9, 6, …	[-1, -1…	0xbb	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
13	13	[9, 6, …	[-1, -1…	0xbb	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	None
14	14	[9, 6, …	[-1, -1…	0xbb	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	0x58
15	15	[9, 6, …	[-1, -1…	0x57	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	0x58
16	16	[9, 6, …	[-1, -1…	0x79	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	0x58
17	17	[9, 6, …	[-1, -1…	0x79	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	0x58
18	18	[9, 6, …	[-1, -1…	0x57	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	0x58
19	19	[9, 6, …	[-1, -1…	0xbb	0	rennoc1	0x69	0x89	None	None	None	None	None	None	None	0x68	0x78	0x58

Figure 4.5: An example of the first fields from the Gamestates Datatable extracted from pilot01. The red annotations point out some interesting information that can be obtained from a first glance of the datatable. Starting the game all players build their first roads and settlements. In the subsequent rounds building is not that likely to occur because the necessary resources are not gathered yet. The first thing that is built after the initial pieces is a road, as it is needed to meet the construction rules. Also the positioning of the robber reveals some interesting tactics of the players, as the robber is repeatedly placed in the same or nearby positions to target a specific settlement with valuable resources or a strong opponent, the players may use tit-for-tat tactics and steal from the one that stole from them in previous rounds, or avoid stealing from a player with whom they have started collaborating to trade goods etc.

58 game is represented in this dataset by 3 tables, containing the gamestate features, the chat discourses and the prediction labels respectively.

The gamestate features, as it was described in Section 4.1 (see Fig. 4.2), in a way act as a photograph of the game board at each turn of the game. A description of the table that holds the gamestate feature values is found in Table 4.10 below.

The number of rows of this table are as many as the number of game rounds of the game, with each one of the rows being dedicated to the corresponding game turn. The number columns of these table are equal to the number of features, i.e. 208 values that depict the state of the board and the players. An example of this datatable from pilot01 can be seen at Fig. 4.5.

The table that contains the chat history of a game encompasses the information of the game turn and the emitters nickname. The structure of this table is described in Table 4.11 more thoroughly.

In this table the number of rows is not equal to the number of game rounds but to the number of chat turns instead. This means that each row contains one chat turn, i.e. one phrase that a player wrote to the chat window

Index	Turn	mitter_nicknam	text
0	0	Tomm	'ello
1	0	Tomm	Just got a connection reset
2	0	Markus	Hm, shouldn't happen (and hasn't before). Let me know if you have problems.
3	0	Tomm	Got a SocketException error appear in this chat line
4	0	Tomm	I guess we'll see if it happens again
5	0	Markus	Yes, fingers crossed.
6	0	Tomm	I take it this is the right game to be in?
7	0	Markus	Yes, I don't know where the other two are.
8	0	Markus	It's been very last minute ...
9	0	Tomm	Excellent! Yeah, well it always feels last minute when you are getting participants, doesn't it?
10	0	Markus	Hehe, yes. Perhaps they're still in a lecture.
11	0	Tomm	So, does this generate a random Catan board?
12	0	Markus	Yes.
13	0	Tomm	I'm still rather impressed at a Hex grid in Java to be honest... Java layout always feels like a nig
14	0	Tomm	*nightmare
15	0	Markus	Indeed; I had the pleasure editing the source code of this application.
16	0	Tomm	Ah. Fun I take it? Still, it looks very nicely laid out to be honest.
17	0	Markus	Well, it's an open source projet, so it's OK. But I had to dig deeper than I wanted.
18	0	Tomm	Ahh! I see. One more to go, I guess.
19	0	Markus	Yes, it's supposed to be three of you.
20	0	Markus	Hm, how are you two for time? Can we wait a bit longer?
21	0	Tomm	I'm good for a little bit longer.
22	0	rennoc1	I don't mind waiting for a while.
23	0	Markus	Great. I guess if person number 3 doesn't show up in 5 or 10 minutes it'll be you only you two anywa
24	0	Markus	y.
25	0	Dave	hi
26	0	Markus	Ah, great. We can start.
27	0	Markus	Sombody has to click START GAME.

Figure 4.6: An example of the first rows from the Chats Datatable extracted from pilot01. It is common for players to engage in conversation unrelated to the SoC as this game is very often seen as a means of socializing.

and then hit the enter, and many chat turns took place during a single game turn. Following that, the turn field of this table is not unique to each row and values of the turn column indeed are not all different. An example of this datatable from pilot01 can be seen at Fig. 4.6.

One of the main reasons that attract people to SoC is the socializing aspect of the game. Factors like the effort to coordinate your actions with your fellow players, the negotiations to exchange goods and the long duration of the games lead to the chat being used for small talk and very often irrelevant conversations start taking place. Also the language used by the players very often contains misspellings (intended or not), abbreviations (e.g. players write *4u* instead of *for you*), syntactic deviations and errors, etc. All these elements add complexity to the handling of the language data but are an omnipresent aspect of dealing with language produced by human speakers.

Last but not least, the table of the prediction labels contains the actions that were performed in each game turn. The number of rows of this table is equal to the number of game turns that took place in the SoC game and the values that it holds are boolean, with the *True* value indicating that the corresponding action was performed in that specific round. The actions examined in this dataset are described in Table 4.12 below.

The values of the labels are initialized at the beginning of a game turn

Index	Turn	playedDevCard	builtRoad	builtSettlement	upgradedCity	boughtDevCar	madeOffer	radedWithPlaye	tradedWithBank	tradedWithPort	no_action
0	0	False	False	False	False	False	False	False	False	False	False
1	1	False	False	False	False	False	False	False	False	False	True
2	2	False	False	False	False	False	False	False	False	False	True
3	3	False	False	False	False	False	False	False	False	False	True
4	4	False	False	False	False	False	False	False	False	False	True
5	5	False	False	False	False	False	False	False	False	False	True
6	6	False	False	False	False	True	True	True	False	False	False
7	7	False	False	False	False	False	False	False	False	False	True
8	8	False	True	False	False	True	False	False	False	False	False
9	9	True	False	False	False	False	False	False	False	False	False
10	10	False	False	False	False	False	False	False	False	False	True
11	11	False	False	False	False	False	False	False	False	False	True
12	12	False	False	False	False	False	True	True	False	False	False
13	13	False	True	False	False	False	False	False	False	False	False

Figure 4.7: An example of the first fields from the Labels Datatable extracted from pilot01. The red annotations point to some interesting conclusions. During the first rounds of a game the players do not have enough resources to actually take action, hence the *no_ action label* is very common. It also demonstrates the multi label classification task, described in the beginning of this chapter, as many true values appear in a single row (in one round multiple labels become activated).

with false everywhere. If by the end of the round no action has been made then the *no action* label is set to true. The only exception to this is the *turn 0*, i.e. the setup phase of the game, for which all labels have a *False* value. An example of this datatable from pilot01 can be seen at Fig. 4.7.

The labels represent all types of actions the players have at their disposal. Actions like *move the robber* are not actually useful for predictions because they depend on luck, i.e. the player rolled a 7, and are therefor excluded from this dataset. However there is a strategic aspect in choosing the position of the robber as players tend to prefer specific tiles, etc (see also Fig. 4.5) and this could be an interesting topic for future work.

The information about the trading actions reveals potentially useful trading patterns and could act as fruitful tool for negotiation agents. Following the example that is set in this thesis and expanding it, future work could incorporate more information about the trading transactions (e.g. what type and amount of a resource was exchanged, between which players etc) to model more specifically the players' trading strategies, their aptitude for collaboration with others and the formation of coalitions in the game, or their revengeful tit-for-tat approaches.

Overall in this dataset of 58 games there are 4211 game turns and 11729 chat turns. Assuming that all players stick to one, identifying nickname and do not change it when they re-enter the server, 92 different players have participated in these games.The majority of them only play a single game.

There are however a few players that participate in multiple games, with the player nicknamed *inca* holding the maximum participation score of 11 games. The players' participation in games is displayed in Fig. 4.8.

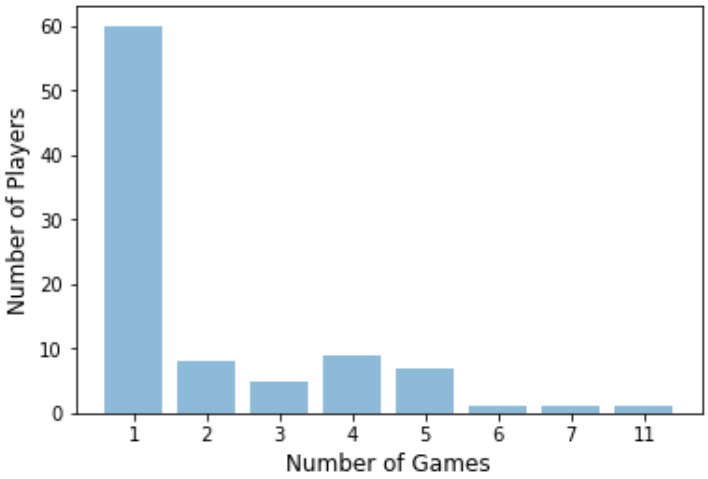


Figure 4.8: Player Participation in the Dataset. The histogram displays a distribution of the number of unique players over the total number of games they have participated in.

As far as the labels are concerned, in 4211 game turns the most common one is the *no action*, with the *road building* being the second most common action of the players (see Fig. 4.9). This is indeed a logical outcome if one accounts for the rules of the game that practically force the players to build roads before they can expand their territory. The next most frequent label is that of *buying development cards*. Hence it seems reasonable to argue that the players follow two main strategies; one aiming at building constructions on the board and another focused on buying and playing development cards.

As a final thought on the distribution of labels it's worth noting that although the no action label appears to be dominant in the dataset at first glace, in fact it only covers 44% of the game turns. Therefore it should be safe to regard that the dataset does not have data asymmetries.

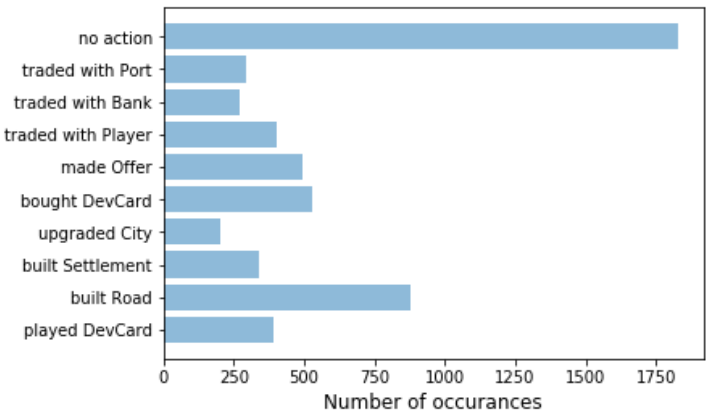


Figure 4.9: Label occurances in the Dataset.

Field	Type	Description
Turn	int	the number of the game round
BoardHexLayout	list	the tile type of the board in jSettlers numerical encoding, viewed from top left to bottom right corner
BoardNumLayout	list	the dice results of the board in jSettlers numerical encoding, viewed from top left to bottom right corner
Robber	int (hex)	the hexadecimal coordinate of the robber's position
player0num	int	the numerical id of the player sitting at position 0 (equals to 0)
player0nickname	str	the player's nickname, default value is "dummy" if no one plays in this position
pl0setm1 ... pl0setm5 pl0city1 ... pl0city4 pl0road1 ... pl0road15	int (hex)	the hexadecimal coordinates of the player's pieces on the board, default value is None if the structure has not been built
pl0knight1 ... pl0knight14 pl0roadbuilding1 pl0roadbuilding2 pl0monopoly1 pl0monopoly2 pl0discovery1 pl0discovery2 pl0vp1 ... pl0vp5	bool	Development Cards section, True for the development cards the players has played
player1num ... pl1vp5	int (hex)/ bool	Features of player sitting at position 1
player2num ... pl2vp5	int (hex)/ bool	Features of player sitting at position 2
player3num ... pl3vp5	int (hex)/ bool	Features of player sitting at position 3

Table 4.10: Gamestates Datatable: Description of the values each column of this table holds.

Field	Type	Description
Turn	int	the number of the game round when this chat utterance took place
emitter_nickname	str	the nickname of the player who wrote in the chat window
text	str	the message written on the chat in that particular chat turn

Table 4.11: The Chats Datatable: Description of the values that each column of this table holds.

Label	Type	Description
Turn	int	Number of the game round
Played Development Card	bool	The player who rolled the dice played a development card that he had bought in a previous round
Built Road	bool	The player who rolled the dice placed a road piece on the board
Built Settlement	bool	The player who rolled the dice placed a settlement piece on the board
Upgraded to City	bool	The player who rolled the dice upgraded an existing settlement of theirs to a city
Bought Development Card	bool	The player who rolled the dice bought a development card
Made Offer	bool	Some player suggested a trading offer to the player who rolled the dice
Traded with another Player	bool	The player who rolled the dice exchanged resources with another player
Traded with bank	bool	The player who rolled the dice successfully exchanged resources with the bank
Traded with port	bool	The player who rolled the dice successfully exchanged resources via a harbour that he has access to
No Action	bool	The player only rolled the dice

Table 4.12: Labels Datatable: Description of the values that each column of this table holds. Notice that it is possible that some player might suggest a trading offer that gets rejected, hence enabling the *Made Offer* but not the *Traded with another Player* label. That way the trading preferences of players can be rudimentary depicted, even with this information.

Chapter 5

Neural Network architecture

Here we explain how the data was preprocessed and how the parameters of the neural networks were configured. For the purposes of this thesis we used Keras, a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. Keras is particularly convenient in our case since it both allows for easy and fast prototyping (through user friendliness, modularity, and extensibility) and also supports convolutional networks and recurrent networks, as well as combinations of the two, the later of which is the fundamental concept of this work. In this chapter we also justify the network configuration that we employed for modeling the problem of predicting the actions taken in SoC game rounds.

5.1 One Hot Vector Encoding

The coordinate system to describe the different board positions in the jSettlers code and the SoC dataset consists in fact of categorical values. They are a representation of position that helps the jSettlers calculations but in terms of a neural network these numbers have equivalent meaning to actually providing literal written board direction like 'on the far west end of the game board' or 'on the most left land tile'. As mentioned in Chapter 3, the most straightforward approach to handling categorical values is to convert them to one hot encoding vectors.

Not all the coordinates of jSettlers are usefull though and we save space by including only the coordinates that are actually accessible to the players to build upon. For example the upper left corner has coordinates that can never appear in any SoCc dataset as nothing can happen on a sea tile corner. The players' constructions can be only placed on nodes or edges to which the None value is also added for the case that a given piece it has not been place

in any coordinate, i.e. has not been built. These coordinates sum up to 79 categories.

Similarly the coordinates on which the robber can be placed are also restricted, as the robber in this version of the game cannot be placed on a sea tile. The land coordinates are only used to place the robber and these sum up to 19 land tiles only.

The board can be fully described using only 9 tile categories, 6 land types and 3 extra categories to denote the plain sea, non-miscellaneous harbors and miscellaneous harbors

5.2 Text Data Preprocessing

In order to run Keras models in parallel the inputs need to have the same number of samples. In order to have equal number of chat samples and gamestate samples, the chats dataset is grouped by the Turn column and the chats that took place during a gameturn are concatenated together. However, concatenating the chat instances that took place during a gameturn is not enough, since there are occasions where during a gameturn everyone was silent, hence there is no chat data to be concatenated. For gameturns where noone spoke an empty instance is added in the data.

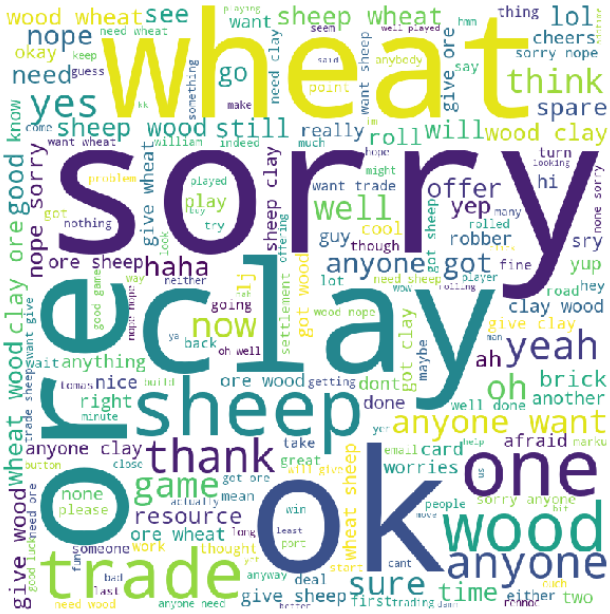


Figure 5.1: Wordcloud of all the chat data in the given dataset.

Then the Keras tokenizer is used to create the vocabulary over the text

data. The Keras tokenizer converts all text to lower case and removes punctuation from the corpus so that it can be used in conjunction with pre-trained embeddings. The text data is then padded with zeros so that all sequences have the same length, i.e. the length of the longest sequence in the corpus and the embedding vectors of the words of the vocabulary are loaded to an embedding matrix. The Keras Embeddings layer will then convert words to embedding vectors during the training process.

In this thesis we used the Glove pre-trained embeddings on the *Wikipedia 2014 + Gigaword 5* package which was retrieved from 6 billion tokens and holds vectors for a vocabulary of 400.000 words.[1]. Specifically, we used 100-dimensional word vectors and the embedding matrix was created over the chat corpus that has a vocabulary of 3368 words and the longest text sequence in the chats is 771 words long.

The final step of the data pre-processing is its partition to train and test sets. The 4211 samples are split to a 80% train - 20% test ratio. The *sklearn* method *train_test_split* that was used to do that also shuffles the data before splitting it. In the end we have a train set of 3368 samples and a test set of 843 samples.

5.3 Network Configuration

5.3.1 Multi Label Learning

As mentioned in Chapter 4, the problem of predicting the actions taken at each round of a SoC game falls into the category of *multi label classification* problems. Multi label classification (also referred to as multi output classification) is a variant of the classification problem where multiple labels may be assigned to each instance. Multi-label classification is, to put it more precisely, a generalization of multi class classification. That stems from the fact that multi class classification is the single-label problem of categorizing instances into precisely one of more than two classes. In the multi-label classification however there is no constraint on how many of the classes the instance can be assigned to. In mathematical terms, multi-label classification is the problem of finding a model that maps inputs $\mathbf{x}$ to binary vectors $\mathbf{y}$ (assigning a value of 0 or 1 for each element (label) in $\mathbf{y}$).

Approaches to handling multi label classification problems include problem transformation schemes, the most straightforward of which is *binary relevance*. In binary relevance each label is treated as a separate binary classification and the initial multi-label problem is converted to single-label

[1]Visit `https://nlp.stanford.edu/projects/glove` for more information

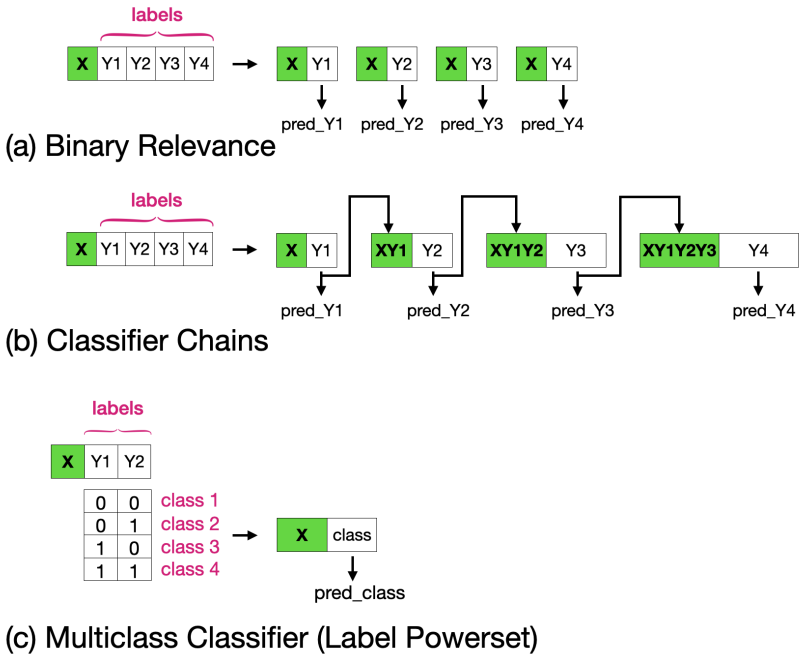


Figure 5.2: Transformation Methods for Multi-label Classification problems

individual problems. Binary relevance's simplicity comes at the cost of not taking into account labels' correlations as each label is treated independently of all others.

A way to incorporate label dependencies is to use *classifier chains*. Here, once again, each label is treated as a separate binary classification but the first classifier is trained just on the input data and then each consecutive classifier is trained on the input plus all the previous classifiers in the chain. In this way label correlation is preserved but the caveat is that in absence of label correlation classifier chains may perform worse than binary relevance and the order of the classifiers is crucial as a misclassification in an early stage of the chain propagates that error to the following classifiers.

Another way of transforming the multi label problem is to convert it into a multiclass classification. The label powerset transformation creates one binary classifier for every label combination of the training set. This method can give high accuracy, especially when the number of labels is small. However the big disadvantage is that the number of classes increases exponentially as the labels increase and this added complexity to the problem ends up lowering the accuracy.

Ensemble methods can also be used along with the problem transformation. In this case multiple learning algorithms are used in order to obtain

better performance than could be obtained from any of the constituent learning algorithms alone. For example, according to binary relevance one classifier can be used for each label but the predictions can be combined by an ensemble method, usually a voting scheme.

Working in opposite direction, multi label problems can be addressed by adapted algorithms. In that context rather that transforming the problem into different subsets of problems an existing algorithm is adapted to directly perform multi-label classification. Multi label version of known algorithms include MLkNN and decison trees adaptations such as 'Clare'. With regards to neural networks, BP-MLL is a multi-label adaptation of the back propagation algorithm, that works on minimizing the ranking loss of labels. However, it has been demonstrated that BP-MLL ranking loss minimization can be replaced by the commonly used cross entropy error function and simple NN models equipped with advanced techniques such as ReLU activation functions, dropout regularization and gradient descent algorithms like AdaGrad perform better, rendering the algorithm obsolete [62].

The most common approach, and the one taken in this thesis, to handing multi-label learning with neural networks is to adapt the commonplace multi-label classification network by using a sigmoid activation on the last layer and binary cross entropy as loss metrics. To explain this, let us consider multi-class classification with neural networks. In this case the employed neural network has the same number of output nodes as the number of the individual classes. In this way each output node corresponds to a specific class and outputs a score for that class. The scores of the last layer are then passed through a softmax layer. The softmax layer converts the scores into probability values. Finally, data is classified into the class that has the highest probability value.

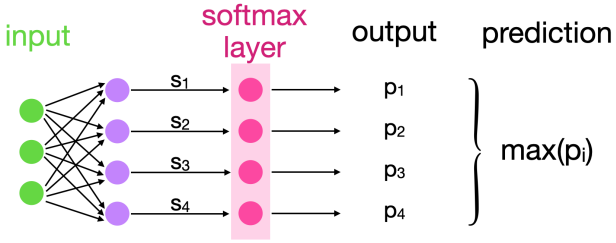


Figure 5.3: Multi-class Classification with Neural Networks. The last layer in the network is a softmax activation layer that transforms the output to a probability distribution over the possible classes. The input example is classified to the most likely class, i.e. the one that scored the highest probability.

The loss metric used for multi-class classification is categorical crossen-

tropy and the target vector is a one-hot encoded vector. For example if the data belongs to the second class (out of a total of 4 possible classes) the target vector would be [0 1 0 0]. The accuracy of the network is calculated by categorical accuracy which in Keras models is calculated as follows:

```
def categorical_accuracy(y_true, y_pred):
    return K.cast(K.equal(K.argmax(y_true, axis=-1),
        K.argmax(y_pred, axis=-1)),
        K.floatx())
```

In this way a predicted label that does not match the true label is considered an misclassification. Only correctly classified examples, ones where the predicted label exactly matches the true one, are taken into account.

The only difference with multi-label classification is that a data sample can belong to multiple classes. Hence the final score for each class should be independent of each other. That prevents us from applying a softmax activation in the final layers, because softmax converts the score into probabilities taking other scores into consideration. By using the sigmoid activation function on the final layer instead, the score of the final node is converted to a number between 0 and 1 independent of what the other scores are. If the score for some class is more than 0.5, the data is classified into that class. That allows for multiple classes to have a score of more than 0.5 independently. Thus the data could be classified into multiple classes.

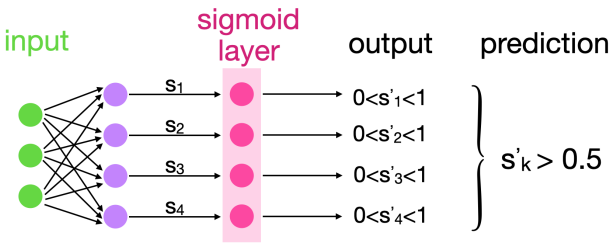


Figure 5.4: Multi-label Classification with Neural Networks. The last layer in the network is a sigmoid activation layer that transforms each output independently to a score number between [0,1]. The input example is classified to those labels that score higher than 0.5.

Since we are using a sigmoid activation function, we have to pair it with binary crossentropy loss. The target vector in this case is binary but there can be multiple 1s in it. For example, if the data belongs to class 2 and 4, our target vector would be as following [0 1 0 1]. The accuracy of the network is calculated in Keras by binary accuracy given as:

```
def binary_accuracy(y_true, y_pred):
    return K.mean(K.equal(y_true, K.round(y_pred)),
        axis=-1)
```

This way of calculating the accuracy gives a great benefit to this way of modeling the problem. That is because, if out of 4 labels the classifier for a given example has predicted two of them correctly but the other two wrong it would be unfair to consider this example misclassified as a whole. In this way however each label is considered separately as far as the accuracy is concerned. For example, if y_true is [1 1 0 0] and the predicted output of the network y_pred is [0.98 1 0 0.6] then the binary accuracy is 3/4 or 0.75; the misclassification of the last label does not annul the other three correctly predicted labels.

5.3.2 Gamestates only Architecture

Up to this point it has become clear that the last layer of the network needs to be a layer with sigmoid activation function and size equal to the number of predicting classes and the loss metric of the training needs to be binary accuracy. In this case the classes represent a set of 10 different actions a player can take, hence the last, predicting layer in all network architectures is configured as a 10 node sigmoid layer.

As for the hidden layers before that, two options are examined in this thesis. The first and most straight forward consists of a feed-forward network, the second uses an lstm layer as a feature extractor, given that gamestate information, much like speech, is sequence dependent due to the rules of SoC.

The particularity of the gamestates data is that after being converted to one hot vectors,it consists of sparse positive inputs. On top of that the default initialization of Keras layers (Glorot normal [2]) takes into account the fan in and fan out of the layer. The input of the gamestates has a large fan in, resulting in initial weights that are close to zero. Therefore, despite its popularity the Relu activation function is not a good choice as this kind of data used with a Relu activation function leads to a network where all weights get a value very close to zero and never get updated during training.

To overcome this caveat the default Keras initialization was changed to the Random normal and the biases were initialized to zero. As for the activation function all hidden layers use a sigmoid activation function. The tanh activation was also tested but found to perform sub-optimally. Other parameters examined during training include the size of the network, batch size,

[2]For more information on Keras initializers visit `https://keras.io/api/layers/initializers/`

epochs, learning rate of the Adam optimization algorithm (see Appendix B).

The architecture that incorporates an LSTM layer for feature extraction over the gamestate data runs and learns faster that the feed forward network. Again various parameters were tested for this model, including the Adam learning rate, the number of units in the LSTM layer, batch size, training epochs and dropout layers (see Appendix B).

5.3.3 Chat only Architecture

The model that uses information only from the chat data employs an embedding layer and an lstm layer for feature extraction. The input to this network is embedding vectors hence there is no problem with sparse input data. The default tanh activation of the Keras LSTM layer works properly in this case.The last layer, responsible for the final predictions is again a sigmoid layer with 10 nodes. Parameters accounted for during training include number of units, batch sizes and training epochs (see Appendix B).

5.3.4 Combined Architecture

In this architecture the two types of information are combined. One branch of the network works over gamestate input data, another over chat input data. The intermediate ouputs are concatenated and fed together as input to a joined predictor branch. The network is then trained jointly over the gamestate and chat input data.

For the gamestate branch of the network both the feed forward and the lstm approach were examined. The parameters of each branch were chosen taking into account observations and outcome during the training of the independent architectures and different learning rates, batch sizes and network sizes where tested (see Appendix B).

Chapter 6

Results

In this Chapter we present a summary of the results that we obtained from implementing the different network architectures discussed in detail in Chapter 5. We further discuss important points that originate from the performance of the different architectures[1].

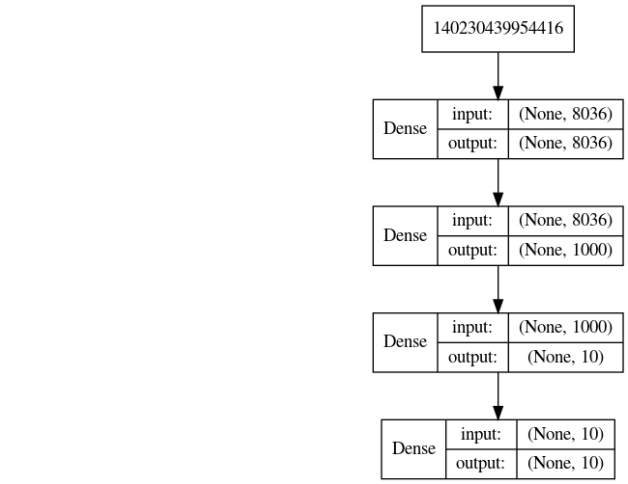
6.1 Performance

To recapitulate on the different architectures presented in the previous chapter, we note that these fall into three distinct categories, i.e. models concerned only with the Gamestates data, models concerned only with the Chat data and finally models that make use of the combination of Gamestates and Chat data. The results presented here constitute a subset of different experiments ran during the course of this work, and correspond to models that have achieved the highest accuracy scores. For a complete listing of all the network configurations tested in this thesis see Appendix B.

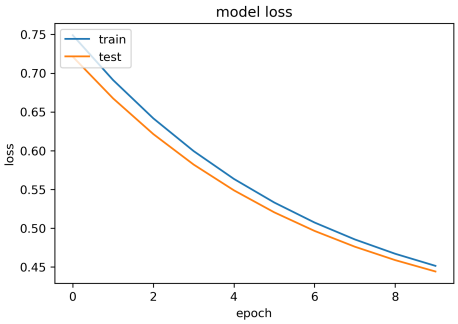
First, results from training only on the Gamestates using a feed-forward network are summarised in Fig. 6.1. The highest accuracy score was achieved for a model of two hidden layers of 1000 and 10 node respectively. All the layers of the network used a sigmoid activation function and the weights were initialized according to the RandomNormal distribution with a standard deviation of 10. The bias parameters were initialized to zero. The model was trained using the Adam optimizer with the default learning rate of 0.001 and achieved an accuracy score of 0.8535 on the 10th epoch. The training was performed on batches of 100 samples over 10 epochs. The EarlyStopping Keras callback was set to stop the training process if the loss of the model

[1]The complete code for this thesis is available at `https://github.com/apostolidoum/modeling-behaviour-of-SoC-players`

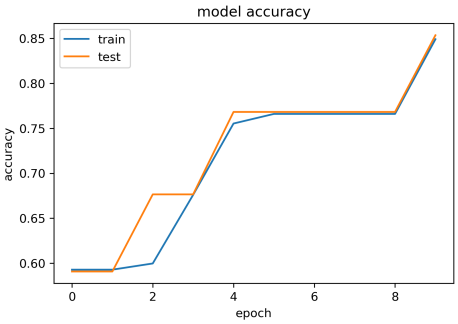
did not improve for 5 consecutive epochs.



(a) model summary



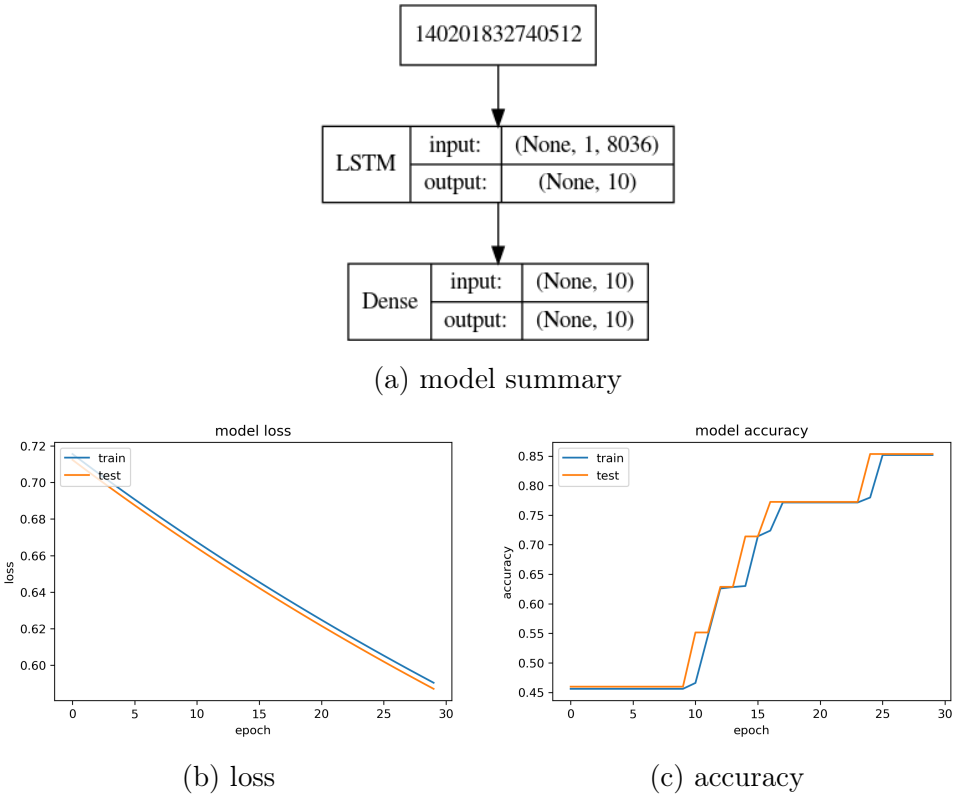
(b) loss



(c) accuracy

Figure 6.1: Results from Gamestates model with Feed Forward Network. The maximum accuracy achieved in the test set was 0.8535.

Fig. 6.2 shows the results from training only on the Gamestates using an LSTM feature extractor. The LSTM contained 10 memory blocks and a sigmoid activation function. The weights of the network were initialized according to the RandomNormal distribution with a standard deviation of 10. The bias parameters were initialized to zero. The model was trained using the Adam optimizer with a learning rate of 0.0001 and achieved an accuracy score of 0.8535 on the 25th epoch. The training was performed on batches of 100 samples over 30 epochs. The EarlyStopping Keras callback was set to stop the training process if the loss of the model did not improve more that 0.01 for 10 consecutive epochs.



(a) model summary

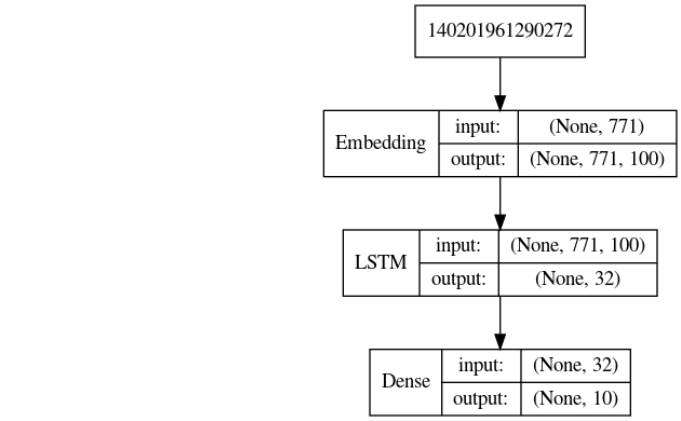
(b) loss

(c) accuracy

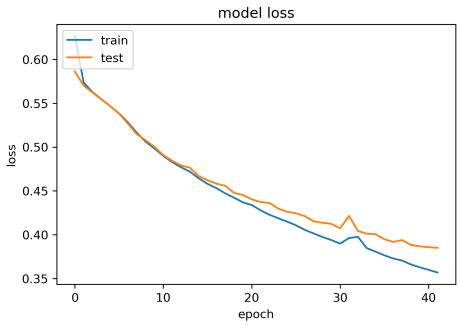
Figure 6.2: Results from Gamestates model with LSTM Network. The maximum accuracy achieved in the test set was 0.8535.

For the model that uses the Chat data as input an LSTM layer of 32 memory blocks was used after the Embedding layer that converts the text input to embedding vectors. The LSTM layer used the default tanh activation function. The model was trained using the Adam optimizer with a learning rate of 0.001 and achieved an accuracy score of 0.8791 on the 16th epoch. The training was performed on batches of 100 samples over 50 epochs. The EarlyStopping Keras callback was set to stop the training process if the loss of the model did not improve more that 0.01 for 5 consecutive epochs. The results are summarized in Fig. 6.3.

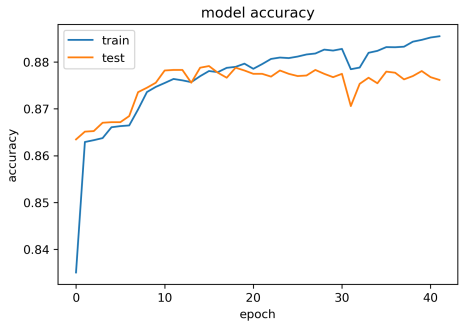
The configuration for the combined Gamestates and Chat architecture was partially setup considering hints from the performance of the individual network architectures. Fig. 6.4 shows the results of the model that combines the result of a feed forward network over the Gamestates with the result of an LSTM component over the Chat data. The two branches were configured in the same way as the best performing aforementioned individual networks.



(a) model summary



(b) loss

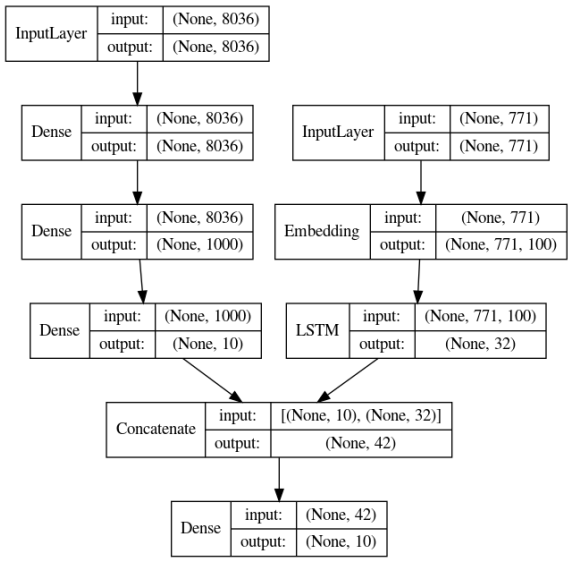


(c) accuracy

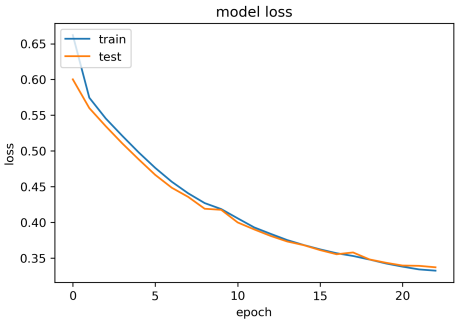
Figure 6.3: Results from Chats model with LSTM Network. The maximum accuracy achieved in the test set was 0.8791.

The highest achieved accuracy performed by this model was 0.8798 on the 20th epoch. The model was trained using the Adam optimizer with a learning rate of 0.001 on batches of 100 samples.

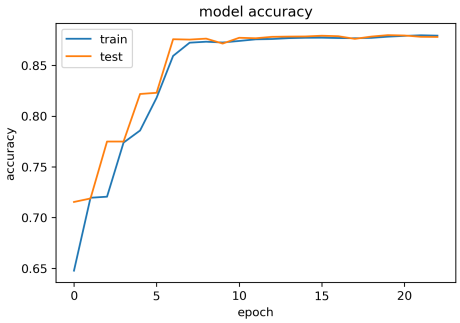
Fig. 6.5 shows the results of the model that combines the result of an LSTM component over the Gamestates with the result of an LSTM component over Chat data. The two branches were configured in the same way as the best performing aforementioned individual networks. The highest achieved accuracy performed by this model was 0.88 on the 6th epoch. The model was trained using the Adam optimizer with a learning rate of 0.001 on batches of 10 samples.



(a) model summary



(b) loss

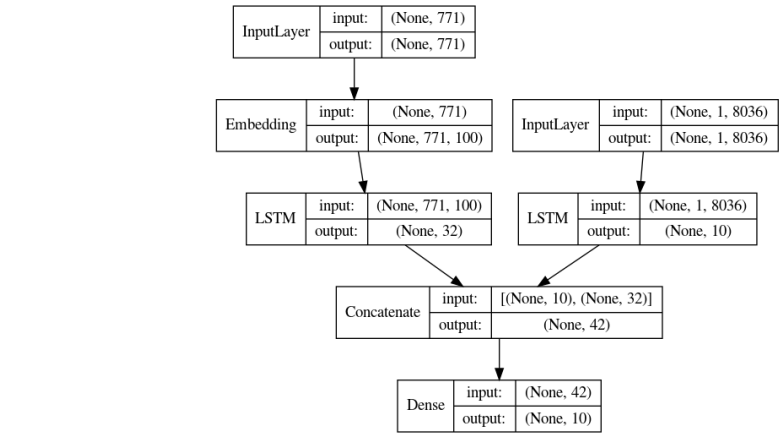


(c) accuracy

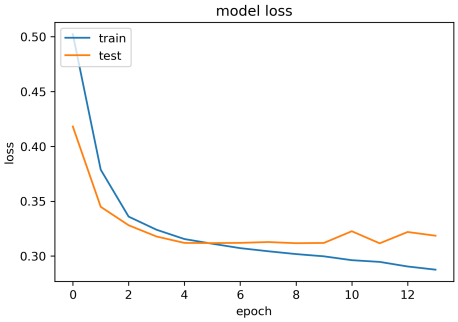
Figure 6.4: Results from Gamestates and Chats model with FF-LSTM Network. The maximum accuracy achieved in the test set was 0.8798.

6.2 Further discussion

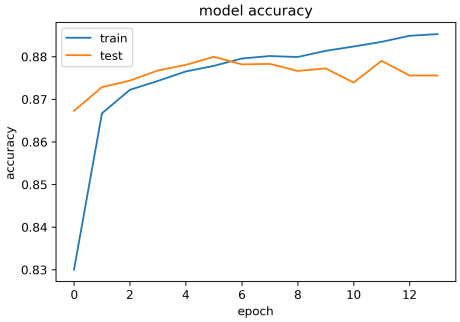
The main scope of this thesis was to examine how different models perform and whether combination architectures would show improved performance. Apart from implementing and testing these five different models we also examined the impact of various other parameters to the performance of the models. A general rule of thumb that derived from the experimentation is that a batch size of 100 samples provides good results in most occasions for this dataset. Also it is often a good idea to lower the default learning rate of the Adam optimization algorithm from 10^{-3} to 10^{-4} (in some occasions



(a) model summary



(b) loss



(c) accuracy

Figure 6.5: Results from Gamestates and Chats model with LSTM-LSTM Network. The maximum accuracy achieved in the validation set was 0.88.

even 10^{-5}) in order to allow the network to explore more and avoid getting stuck in local minima.

In general the use of LSTM networks speeds up the learning process and these networks seem to provide a slight advantage over the feed forward networks in terms of accuracy. The chat data produces a lot more stable results. This is due to the use of embedding vectors that alleviate the problem of sparse inputs. The gamestate inputs could be improved in a similar manner. One way could be to encode the information in some other, less sparse way by different design. Another, more interesting approach would be to use the existing embedding algorithms to produce embedding vectors for the gamestates inputs, especially if more data becomes available. As was explained in Chapter 5 the gamestates is categorical data and the embedding approach for encoding categorical variables shows promising results [63].

Finally, we find that the combination of gamestate and chat information is advantageous and improves the predictive performance of the models. The standalone chat model can approach the performance of the combination architectures if allowed to run over many epochs, but the combination architectures get there much faster. Hence the additional gamestate information helps speed up the learning process. Also the validation accuracy results could very likely see an extra improvement, even with the existing data and models, if cross validation methods would be used.

Chapter 7

Conclusions

In this thesis we formulated a dataset from SoC gameplay and discourse between human players and explored five different architectures to try and emulate the players' behaviour by trying to predict the action they are going to take. We managed to achieve an accuracy score of 0.88 for the architecture that uses LSTM feature extractors for the gamestates and the chat inputs and provides the combined information as a joint input to a feed-forward network for prediction.

Apart from implementing and testing these five different models we also examined the impact of various other parameters to the performance of the models. The presented results are promising and suggest that with further tuning of the networks' parameters, accuracy of the models could improve even further. Some of the things that could be done in the future to improve the performance could be to examine other architectures (e.g. GRU layers, bi-LSTMs), different embedding vectors (e.g. Word2Vec pre-trained embeddings, other dimensions of Glove embeddings or even custom trained word embeddings over the chat data), different network configurations (size of networks, optimization with stochastic gradient descent, cross validation etc). Additionally the fundamental issue of insufficient data could be addressed in future research. Given more resources additional data could be collected by building an online jSettlers community where people can play and chat more and now that there is an API to convert the game log files to a comprehensive dataset, this data can easily be utilized for this or other machine learning schemes.

The anticipated shortcoming regarding the scarcity of available data was mentioned in the introduction of the thesis. Admittedly, this work relates to studying human behaviour and decision making (an already difficult and cumbersome research area) in the presence of imposed complex gamerules. Normally a huge amount of data is needed in order to study such behaviors.

Nonetheless, the results reported are encouraging and show that research even with such data limitations is not futile.

The difficulty of finding corpora to work with NLP in such a restricted and task specific project of modeling human behaviour in strategic games is expected. First and foremost, corpora are hard to find because they are hard to produce, since they require a lot of human involvement (for annotations, corrections etc) and this is a laborious and time consuming task. Secondly, there exist some datasets that are accessible online but usually appertain to very general purposes and are not in the context of strategic games. It was therefore compelling that the already available STAC corpus should be utilized, despite its limited size of data.

As a final note, we point that another endeavor of this thesis was to rekindle research in the SoC project and incorporate the STAC corpus in order to use for the first time the actions of the players during gameplay and the linguistic information jointly and to examine whether the combination of these two aspects of the game was constructive. The results confirm that this was a step in the right direction and suggest that continued research could be indeed fruitful.

In future work, the knowledge mined in this thesis from the corpus of human players could be used in conjunction with adversarial networks. Also, with a larger amount of available data, specific type of players or winning strategies of players could be studied and modeled. Such a larger pool of available training data can pave the way to new research approaches with potentially interesting results. As an example, it might be worth investigating the use of *entity embeddings* (as described in [63]) for encoding the gamestates inputs. Another approach could be the use of the existing word2vec or GloVe algorithms to train embedding vectors that would encode the categorical variables of the SoC board coordinates to dense embedding vectors, reducing the sparsity of the one-hot encoded inputs and hopefully capturing interesting information about the locations on the board.

Bibliography

[1] N. Asher, J. Hunter, M. Morey, B. Farah, and S. Afantenos, "Discourse structure and dialogue acts in multiparty dialogue: the STAC corpus," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*. Portorož, Slovenia: European Language Resources Association (ELRA), May 2016, pp. 2721–2727. [Online]. Available: https://www.aclweb.org/anthology/L16-1432

[2] R. S. Thomas, "Real-time Decision Making for Adversarial Environments Using a Plan-based Heuristic," PhD Thesis, Northwestern University, Evanston, IL, USA, 2003.

[3] M. Pfeiffer, "Reinforcement learning of strategies for Settlers of Catan," May 2019.

[4] I. Szita, G. Chaslot, and P. Spronck, "Monte-Carlo Tree Search in Settlers of Catan," in *Advances in Computer Games*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, H. J. van den Herik, and P. Spronck, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6048, pp. 21–32. [Online]. Available: http://link.springer.com/10.1007/978-3-642-12993-3_3

[5] Konstantinos Panagiotis Panousis, "Real-time Planning and Learning in the "Settlers of Catan" Strategy Game," Diploma Thesis, Technical University of Crete, 2014.

[6] Emmanouil Karamalegkos, "Monte Carlo Tree Search in the "Settlers of Catan" Strategy Game," Diploma Thesis, Technical University of Crete, 2016.

[7] H. Cuayáhuitl, S. Keizer, and O. Lemon, "Strategic Dialogue Management via Deep Reinforcement Learning," *arXiv:1511.08099*

[cs], Nov. 2015, arXiv: 1511.08099. [Online]. Available: http://arxiv.org/abs/1511.08099

[8] 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, vol. 11450, pp. 233–248. [Online]. Available: http://link.springer.com/10.1007/978-3-030-14174-5_16

[9] S. Russell and A. L. Zimdars, "Q-Decomposition for Reinforcement Learning Agents," p. 8.

[10] S. Afantenos, E. Kow, N. Asher, and J. Perret, "Discourse parsing for multi-party chat dialogues," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, 2015, pp. 928–937. [Online]. Available: http://aclweb.org/anthology/D15-1109

[11] Y. Goldberg, "Neural network methods for natural language processing," *Synthesis Lectures on Human Language Technologies*, vol. 10, no. 1, pp. 1–309, 2017. [Online]. Available: https://doi.org/10.2200/S00762ED1V01Y201703HLT037

[12] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, Jan. 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/089360809190009T

[13] K. Y. Tam and M. Y. Kiang, "Managerial Applications of Neural Networks: The Case of Bank Failure Predictions," *Management Science*, vol. 38, no. 7, pp. 926–947, Jul. 1992. [Online]. Available: https://pubsonline.informs.org/doi/abs/10.1287/mnsc.38.7.926

[14] T. Kimoto, K. Asakawa, M. Yoda, and M. Takeoka, "Stock market prediction system with modular neural networks," in *1990 IJCNN International Joint Conference on Neural Networks*, Jun. 1990, pp. 1–6 vol.1.

[15] J. Khan, J. S. Wei, M. Ringnér, L. H. Saal, M. Ladanyi, F. Westermann, F. Berthold, M. Schwab, C. R. Antonescu, C. Peterson, and P. S. Meltzer, "Classification and diagnostic prediction of cancers using gene expression profiling and artificial neural networks," *Nature Medicine*, vol. 7, no. 6, p. 673, Jun. 2001. [Online]. Available: https://www.nature.com/articles/nm0601_673

[16] H. R. Maier and G. C. Dandy, "Neural networks for the prediction and forecasting of water resources variables: a review of modelling issues and applications," *Environmental Modelling & Software*, vol. 15, no. 1, pp. 101–124, Jan. 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364815299000079

[17] A. Lapedes and R. Farber, "Nonlinear signal processing using neural networks: Prediction and system modelling," Jun. 1987. [Online]. Available: https://www.osti.gov/biblio/5470451

[18] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[19] D. Chen and C. Manning, "A Fast and Accurate Dependency Parser using Neural Networks," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 740–750. [Online]. Available: http://aclweb.org/anthology/D14-1082

[20] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A Neural Probabilistic Language Model," p. 19.

[21] A. de Gispert, G. Iglesias, and B. Byrne, "Fast and Accurate Preordering for SMT using Neural Networks," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Denver, Colorado: Association for Computational Linguistics, 2015, pp. 1012–1017. [Online]. Available: http://aclweb.org/anthology/N15-1105

[22] M. Iyyer, V. Manjunatha, J. Boyd-Graber, and H. Daumé III, "Deep Unordered Composition Rivals Syntactic Methods for Text Classification," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, 2015, pp. 1681–1691. [Online]. Available: http://aclweb.org/anthology/P15-1162

[23] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with con-

volutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015, pp. 1–9.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3098997.3065386

[25] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv:1409.1556 [cs]*, Sep. 2014, arXiv: 1409.1556. [Online]. Available: http://arxiv.org/abs/1409.1556

[26] A. van den Oord, "Deep content-based music recommendation," p. 9.

[27] N. D. Truong, A. D. Nguyen, L. Kuhlmann, M. R. Bonyadi, J. Yang, S. Ippolito, and O. Kavehei, "Convolutional neural networks for seizure prediction using intracranial and scalp electroencephalogram," *Neural Networks*, vol. 105, pp. 104–111, Sep. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608018301485

[28] T. N. Sainath, A. Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep convolutional neural networks for LVCSR," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 8614–8618.

[29] Y. Kim, "Convolutional Neural Networks for Sentence Classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1746–1751. [Online]. Available: http://aclweb.org/anthology/D14-1181

[30] R. Johnson and T. Zhang, "Effective Use of Word Order for Text Categorization with Convolutional Neural Networks," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Denver, Colorado: Association for Computational Linguistics, 2015, pp. 103–112. [Online]. Available: http://aclweb.org/anthology/N15-1011

[31] P. Wang, J. Xu, B. Xu, C. Liu, H. Zhang, F. Wang, and H. Hao, "Semantic Clustering and Convolutional Neural Network for Short Text Categorization," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*

(Volume 2: Short Papers). Beijing, China: Association for Computational Linguistics, 2015, pp. 352–357. [Online]. Available: http://aclweb.org/anthology/P15-2058

[32] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A Convolutional Neural Network for Modelling Sentences," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Baltimore, Maryland: Association for Computational Linguistics, 2014, pp. 655–665. [Online]. Available: http://aclweb.org/anthology/P14-1062

[33] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural Language Processing (Almost) from Scratch," *NATURAL LANGUAGE PROCESSING*, p. 45.

[34] L. Dong, F. Wei, M. Zhou, and K. Xu, "Question Answering over Freebase with Multi-Column Convolutional Neural Networks," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, 2015, pp. 260–269. [Online]. Available: http://aclweb.org/anthology/P15-1026

[35] M. Auli and J. Gao, "Decoder Integration and Expected BLEU Training for Recurrent Neural Network Language Models," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Baltimore, Maryland: Association for Computational Linguistics, 2014, pp. 136–142. [Online]. Available: http://aclweb.org/anthology/P14-2023

[36] H. Adel, N. T. Vu, and T. Schultz, "Combination of Recurrent Neural Networks and Factored Language Models for Code-Switching Language Modeling," p. 6.

[37] T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur, "Recurrent Neural Network Based Language Model," p. 4.

[38] T. Mikolov, S. Kombrink, L. Burget, J. Černocký, and S. Khudanpur, "Extensions of recurrent neural network language model," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2011, pp. 5528–5531.

[39] W. Xu, M. Auli, and S. Clark, "CCG Supertagging with a Recurrent Neural Network," in *Proceedings of the 53rd Annual*

Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers). Beijing, China: Association for Computational Linguistics, 2015, pp. 250–255. [Online]. Available: http://aclweb.org/anthology/P15-2041

[40] O. Irsoy and C. Cardie, "Opinion Mining with Deep Recurrent Neural Networks," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 720–728. [Online]. Available: http://aclweb.org/anthology/D14-1080

[41] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1724–1734. [Online]. Available: http://aclweb.org/anthology/D14-1179

[42] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," p. 9.

[43] M. Sundermeyer, T. Alkhouli, J. Wuebker, and H. Ney, "Translation Modeling with Bidirectional Recurrent Neural Networks," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 14–25. [Online]. Available: http://aclweb.org/anthology/D14-1003

[44] A. Sordoni, M. Galley, M. Auli, C. Brockett, Y. Ji, M. Mitchell, J.-Y. Nie, J. Gao, and B. Dolan, "A Neural Network Approach to Context-Sensitive Generation of Conversational Responses," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Denver, Colorado: Association for Computational Linguistics, 2015, pp. 196–205. [Online]. Available: http://aclweb.org/anthology/N15-1020

[45] A. Kannan, P. Young, V. Ramavajjala, K. Kurach, S. Ravi, T. Kaufmann, A. Tomkins, B. Miklos, G. Corrado, L. Lukacs, and M. Ganea, "Smart Reply: Automated Response Suggestion for Email," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*. San Francisco,

California, USA: ACM Press, 2016, pp. 955–964. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2939672.2939801

[46] J. Li, W. Monroe, A. Ritter, D. Jurafsky, M. Galley, and J. Gao, "Deep Reinforcement Learning for Dialogue Generation," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, 2016, pp. 1192–1202. [Online]. Available: http://aclweb.org/anthology/D16-1127

[47] S. Narayan, S. B. Cohen, and M. Lapata, "Ranking Sentences for Extractive Summarization with Reinforcement Learning," *arXiv:1802.08636 [cs]*, Feb. 2018, arXiv: 1802.08636. [Online]. Available: http://arxiv.org/abs/1802.08636

[48] S. Chopra, M. Auli, and A. M. Rush, "Abstractive Sentence Summarization with Attentive Recurrent Neural Networks," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, 2016, pp. 93–98. [Online]. Available: http://aclweb.org/anthology/N16-1012

[49] X. Zhang and M. Lapata, "Sentence Simplification with Deep Reinforcement Learning," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, 2017, pp. 584–594. [Online]. Available: http://aclweb.org/anthology/D17-1062

[50] M. Iyyer, J. Boyd-Graber, L. Claudino, R. Socher, and H. Daumé III, "A Neural Network for Factoid Question Answering over Paragraphs," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 633–644. [Online]. Available: http://aclweb.org/anthology/D14-1070

[51] F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton Project Para*, ser. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957. [Online]. Available: https://books.google.gr/books?id=P_XGPgAACAAJ

[52] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[53] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[54] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[55] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., 2013, p. 3111–3119.

[56] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Atlanta, Georgia: Association for Computational Linguistics, Jun. 2013, pp. 746–751. [Online]. Available: https://www.aclweb.org/anthology/N13-1090

[57] X. Rong, "word2vec parameter learning explained," *arXiv preprint arXiv:1411.2738*, 2014.

[58] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *CoRR*, vol. abs/1402.3722, 2014. [Online]. Available: http://arxiv.org/abs/1402.3722

[59] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[60] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1

[61] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[62] J. Nam, J. Kim, E. Loza Mencía, I. Gurevych, and J. Fürnkranz, "Large-scale multi-label text classification — revisiting neural networks," in *Machine Learning and Knowledge Discovery in Databases*, T. Calders, F. Esposito, E. Hüllermeier, and R. Meo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 437–452.

[63] C. Guo and F. Berkhahn, "Entity embeddings of categorical variables," *CoRR*, vol. abs/1604.06737, 2016. [Online]. Available: http://arxiv.org/abs/1604.06737

Appendices

Appendix A

Game Board Generation in jSettlers

The process for setting up the board is schematically outlined in Fig. A.1. After an initial shuffling of the terrain tiles, the central board column of five tiles is set, followed by setting the four tiles columns on the right and left side of the central column. Finally, the right and left three tiles columns are set alongside the four tiles columns. After the terrain is completed, the port and sea tiles are set alternating around the outermost edges of the island's hexagon.

The game board is completed after placing one numbered tile on each of the terrain tiles. The process starts from a terrain tile at one of the six corners of the outline of the island's hexagon. Further numbered tiles are then placed in a counter-clock spiral towards the central terrain tile. The desert terrain tile is skipped in this process. The numbered tiles are placed

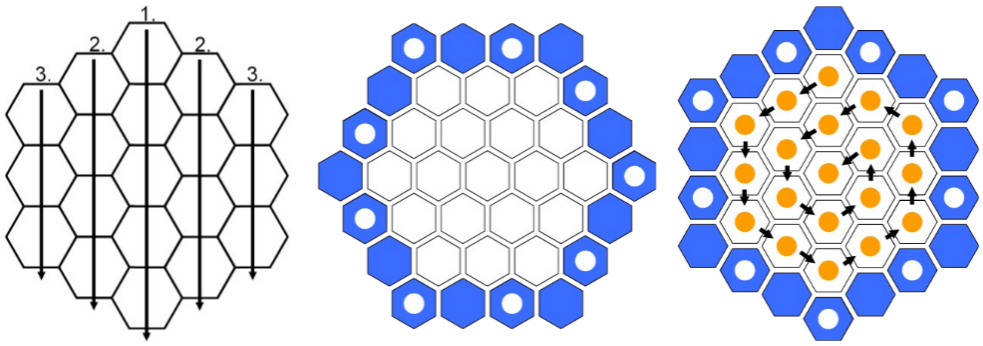


Figure A.1: Schematic representation of the board generation process in jSettlers.

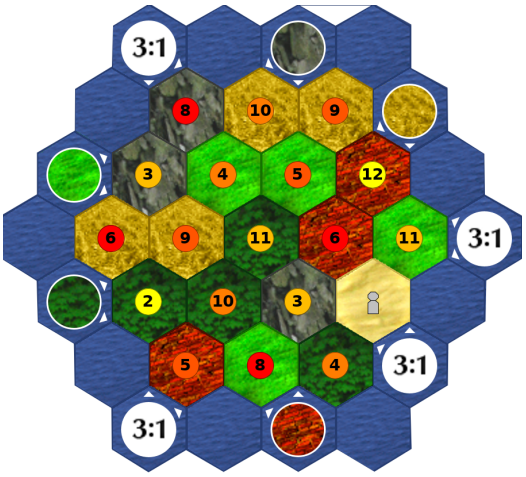


Figure A.2: Example of the final board layout, after the setting up process in jSettlers.

on the terrain tiles following the pattern:
$5 \rightarrow 2 \rightarrow 6 \rightarrow 3 \rightarrow 8 \rightarrow 10 \rightarrow 9 \rightarrow 12 \rightarrow 11 \rightarrow 4 \rightarrow 8 \rightarrow 10 \rightarrow 9 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow 11$
completing the setup of the game board.

The end result is a number layout that is pseudorandom, as the selection of the initial corner and the location of the dessert tile incorporate some non-deterministic elements in an otherwise deterministic sequence of numbers.

Appendix B

Network Configurations

Below we provide a listing of all the network configurations tested in this thesis. The results presented in this thesis were obtained by running the code on an Intel(R) Core(TM) i7-8550U CPU, running Linux Mint 19 (Kernel: Linux 4.15.0-43-generic, Architecture: x86-64).

The $RandomNormal()$ Keras initializer as well as the $train\_test\_split()$ method of sklearn, that by default shuffles the data before splitting it, result to different outcomes every time they are computed. Also included in this appendix is a complete list of the packages used in this thesis. Running the architectures with different package configurations may result in different results.

Results that show the model test loss surpassing the train loss (e.g. Chats network architecture in Fig. 6.3) suggest a tendency towards overfitting. This can be addressed by collecting additional data. Finally, additional runs of the presented architectures and configurations would ensure safer results.

The result plots presented in Chapter 6 correspond to the highlighted runs of the network configurations summarized in Figs B.1-B.5. For completeness, we also include in Section B.1 accuracy and loss plots for all the runs of the combined FF-LSTM architecture (with network configurations summarized in Fig. B.4) and in Section B.2 accuracy and loss plots for all the runs of the combined LSTM-LSTM architecture (with network configurations summarized in Fig. B.5).

Furthermore, we also include in Section B.3 result plots that were obtained using the highlighted architectures on a different machine, which display worse performance.

Gamestates FF

layers	activations	initialization	loss	accuracy	optimizer	learning rate	batch size	Epochs (max)	highest accuracy achieved	on epoch
[8036] [1000] [100] [10]	[relu] [relu] [relu] [sigmoid]	Golrot (keras default)	binary crossentropy	binary accuracy	Adam	0.001	10	20	0.8662	1
[8036] [1000] [100] [10]	[relu] [relu] [relu] [sigmoid]	Golrot (keras default)	binary crossentropy	binary accuracy	Adam	0.001	100	10	0.8662	1
[8036] [100] [10]	[relu] [relu] [sigmoid]	Golrot (keras default)	binary crossentropy	binary accuracy	Adam	0.001	100	10	0.8662	1
[8036] [10] [10]	[relu] [relu] [sigmoid]	Golrot (keras default)	binary crossentropy	binary accuracy	Adam	0.001	100	10	0.8662	1
[8036] [1000] [10] [10]	[relu][relu][relu][sigmoid]	Golrot (keras default)	binary crossentropy	binary accuracy	Adam	0.1	100	10	0.8662	1
[8036] [1000] [10] [10]	[relu][relu][relu][sigmoid]	Golrot (keras default)	binary crossentropy	binary accuracy	Adam	1E-06	100	10	0.8662	1
[8036][1000][10][10]	[relu][relu][relu][sigmoid]	kernel: RandomNormal(stdev=0.01) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	10	0.8662	1
[8036][1000][10][10]	[relu][relu][relu][sigmoid]	kernel: RandomNormal(stdev=0.5) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	10	0.8662	1
[8036][1000][10][10]	[relu][relu][relu][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	10	0.8662	1
[8036][1000][10][10]	[sigmoid][sigmoid][sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	10	0.8535	10
[8036][1000][10][10]	[sigmoid][sigmoid][sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	20	0.8662	6
[8036][1000][10][10]	[sigmoid][sigmoid][sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	15	0.8662	9
[8036][1000][10][10]	[sigmoid][sigmoid][sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	15	0.8662	8
[8036][100][10]	[sigmoid][sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	15	0.6288	1
[8036][1000][10][10]	[tanh][tanh][tanh][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	15	0.8662	4
[8036][1000][10][10]	[sigmoid][sigmoid][sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	15	0.8535	3

Figure B.1: Network Configurations for Gamestates model with Feed Forward Network

Gamestates LSTM

layers	activations	initialization	loss	accuracy	optimizer	learning rate	batch size	Epochs (max)	highest accuracy achieved	on epoch
[LSTM100][10]	[tanh][sigmoi]	Golrot (keras default)	binary crossentropy	binary accuracy	Adam	0.001	10	10	0.8662	1
[LSTM100][10]	[tanh][sigmoi]	kernel: RandomNormal(stdev=0.5) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	10	10	0.8662	1
[LSTM10][10]	[sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	10	10	0.8662	1
[LSTM10][10]	[sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	1E-06	10	10	0.642	1
[LSTM10][10]	[sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.0001	10	10	0.8662	4
[LSTM10][Dropout0.2][10]	[sigmoid][-][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.0001	10	10	0.8662	3
[LSTM10][Dropout0.2][10]	[sigmoid][-][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	1E-05	10	10	0.7777	5
[LSTM10][Dropout0.2][10]	[sigmoid][-][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.0001	100	15	0.5266	2
[LSTM10][10]	[sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.0001	100	15	0.6887	12
[LSTM10][10]	[sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.0001	100	20	0.6274	15
[LSTM10][10]	[sigmoid][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.0001	100	30	0.8535	25

Figure B.2: Network Configurations for Gamestates model with LSTM network

Chats LSTM

layers	activations	initialization	loss	accuracy	optimizer	learning rate	batch size	Epochs (max)	highest accuracy achieved	on epoch
[Embed][LSTM32][10]	[-][tanh][sigmoid]	keras default	binary crossentropy	binary accuracy	Adam	0.001	10	10	0.8788	6
[Embed][LSTM32][10]	[-][tanh][sigmoid]	keras default	binary crossentropy	binary accuracy	Adam	0.001	10	10	0.879	6
[Embed][LSTM32][10]	[-][tanh][sigmoid]	keras default	binary crossentropy	binary accuracy	Adam	0.001	100	10	0.8771	10
[Embed][LSTM32][10]	[-][tanh][sigmoid]	keras default	binary crossentropy	binary accuracy	Adam	0.001	100	25	0.8796	25
[Embed][LSTM32][10]	[-][tanh][sigmoid]	keras default	binary crossentropy	binary accuracy	Adam	0.001	100	50	0.8791	16

Figure B.3: Network Configurations for Chats model with LSTM Network

Combined FF-LSTM

layers	activations	initialization	loss	accuracy	optimizer	learning rate	batch size	Epochs (max)	highest accuracy achieved	on epoch
[[8036][1000][10]+[Embed][LSTM32]][10]	[[sigmoid][sigmoid][sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	25	0.8762	7
[[8036][1000][10]+[Embed][LSTM32]][10]	[[sigmoid][sigmoid][sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	25	0.8798	20
[[8036][1000][10]+[Embed][LSTM32]][10]	[[sigmoid][sigmoid][sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	50	0.8782	16
[[8036][1000][10]+[Embed][LSTM32]][10]	[[sigmoid][sigmoid][sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	50	0.8797	12
[[8036][1000][10]+[Embed][LSTM32]][10]	[[sigmoid][sigmoid][sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	50	10	0.8786	4

Figure B.4: Network Configurations for Gamestates and Chats Network with FF-LSTM

Combined LSTM-LSTM

layers	activations	initialization	loss	accuracy	optimizer	learning rate	batch size	Epochs (max)	highest accuracy achieved	on epoch
[[LSTM10]+[Embed][LSTM32]][10]	[[sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	50	0.8794	22
[[LSTM10]+[Embed][LSTM32]][10]	[[sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	10	50	0.8788	5
[[LSTM10]+[Embed][LSTM32]][10]	[[sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	100	2	0.8278	2
[[LSTM10]+[Embed][LSTM32]][10]	[[sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.001	10	50	0.88	6
[[LSTM10]+[Embed][LSTM32]][10]	[[sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.0001	100	50	0.8757	50
[[LSTM10]+[Embed][LSTM32]][10]	[[sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.0001	100	80	0.8764	78
[[LSTM10]+[Embed][LSTM32]][10]	[[sigmoid]+[-][tanh]][sigmoid]	kernel: RandomNormal(stdev=10) bias:Zeros()	binary crossentropy	binary accuracy	Adam	0.0001	10	80	0.8785	35

Figure B.5: Network Configurations for Gamestates and Chats Network with LSTM-LSTM

The results presented in the thesis were obtained using the following versions of conda packages :

```
# packages in environment at
# Name                    Version                   Build  Channel
_libgcc_mutex             0.1                         main   anaconda
absl-py                   0.6.1                     pypi_0   pypi
alabaster                 0.7.12                    py36_0
asn1crypto                0.24.0                    py36_0
astor                     0.7.1                     pypi_0   pypi
astroid                   2.1.0                     py36_0
babel                     2.6.0                     py36_0
backcall                  0.1.0                     py36_0
blas                      1.0                         mkl
bleach                    3.0.2                     py36_0
ca-certificates           2020.4.5.1            hecc5488_0   conda-forge
cairo                     1.14.12               h8948797_3   anaconda
certifi                   2020.4.5.1         py36h9f0ad1d_0   conda-forge
cffi                      1.11.5             py36he75722e_1
chardet                   3.0.4                     py36_1
cloudpickle               0.6.1                     py36_0
cryptography              2.4.2              py36h1ba5d50_0
cycler                    0.10.0                    py36_0
dbus                      1.13.2                h714fa37_1
decorator                 4.3.0                     py36_0
docutils                  0.14                      py36_0
entrypoints               0.2.3                     py36_2
expat                     2.2.6                 he6710b0_0
fontconfig                2.13.0                h9420a91_0
freetype                  2.9.1                 h8a8886c_1
fribidi                   1.0.5                 h7b6447c_0   anaconda
gast                      0.2.2                     pypi_0   pypi
glib                      2.56.2                hd408876_0
gmp                       6.1.2                 h6c8ec71_1
graphite2                 1.3.13                h23475e2_0   anaconda
graphviz                  2.40.1                h21bd128_2   anaconda
grpcio                    1.17.1                    pypi_0   pypi
gst-plugins-base          1.14.0                hbbd80ab_1
gstreamer                 1.14.0                hb453b48_1
h5py                      2.9.0                     pypi_0   pypi
harfbuzz                  1.8.8                 hffaf4a1_0   anaconda
icu                       58.2                  h9c2bf20_1
idna                      2.8                       py36_0
imagesize                 1.1.0                     py36_0
intel-openmp              2019.1                      144
ipykernel                 5.1.0              py36h39e3cac_0
ipython                   7.2.0              py36h39e3cac_0
ipython_genutils          0.2.0                     py36_0
isort                     4.3.4                     py36_0
jedi                      0.13.2                    py36_0
jeepney                   0.4                       py36_0
jinja2                    2.10                      py36_0
jpeg                      9b                    h024ee3a_2
jsonschema                2.6.0                     py36_0
jupyter_client            5.2.4                     py36_0
jupyter_core              4.4.0                     py36_0
keras                     2.2.4                     pypi_0   pypi
keras-applications        1.0.6                     pypi_0   pypi
keras-preprocessing       1.0.5                     pypi_0   pypi
keyring                   17.1.1                    py36_0
kiwisolver                1.0.1              py36hf484d3e_0
lazy-object-proxy         1.3.1              py36h14c3975_2
```

```
libedit                   3.1.20170329         h6b74fdf_2
libffi                    3.2.1                hd88cf55_4
libgcc-ng                 8.2.0                hdf63c60_1
libgfortran-ng            7.3.0                hdf63c60_0
libpng                    1.6.36               hbc83047_0
libsodium                 1.0.16               h1bed415_0
libstdcxx-ng              8.2.0                hdf63c60_1
libtiff                   4.1.0                h2733197_0    anaconda
libuuid                   1.0.3                h1bed415_2
libxcb                    1.13                 h1bed415_1
libxml2                   2.9.8                h26e45fe_1
markdown                  3.0.1                    pypi_0    pypi
markupsafe                1.1.0              py36h7b6447c_0
matplotlib                3.0.2              py36h5429711_0
matplotlib-base           3.1.1              py36hfd891ef_0    conda-forge
mccabe                    0.6.1                    py36_1
mistune                   0.8.4              py36h7b6447c_0
mkl                       2019.1                     144
mkl_fft                   1.0.10             py36ha843d7b_0
mkl_random                1.0.2              py36hd81dba3_0
music21                   5.5.0                    pypi_0    pypi
nbconvert                 5.3.1                    py36_0
nbformat                  4.4.0                    py36_0
ncurses                   6.1                  he6710b0_1
notebook                  5.7.4                    py36_0
numpy                     1.15.4             py36h7e9f1db_0
numpy-base                1.15.4             py36hde5b4d6_0
numpydoc                  0.8.0                    py36_0
olefile                   0.46                       py_0    conda-forge
openssl                   1.1.1g               h516909a_0    conda-forge
packaging                 18.0                     py36_0
pandas                    0.24.1             py36he6710b0_0
pandoc                    2.2.3.2                       0
pandocfilters             1.4.2                    py36_1
pango                     1.42.4               h049681c_0    anaconda
parso                     0.3.1                    py36_0
pcre                      8.42                 h439df22_0
pexpect                   4.6.0                    py36_0
pickleshare               0.7.5                    py36_0
pillow                    7.0.0              py36hb39fc2d_0
pip                       18.1                     py36_0
pixman                    0.38.0               h7b6447c_0    anaconda
prometheus_client         0.5.0                    py36_0
prompt_toolkit            2.0.7                    py36_0
protobuf                  3.6.1                    pypi_0    pypi
psutil                    5.4.8              py36h7b6447c_0
ptyprocess                0.6.0                    py36_0
pycodestyle               2.4.0                    py36_0
pycparser                 2.19                     py36_0
pydot                     1.4.1                    py36_0    anaconda
pyflakes                  2.0.0                    py36_0
pygments                  2.3.1                    py36_0
pylint                    2.2.2                    py36_0
pyopenssl                 18.0.0                   py36_0
pyparsing                 2.3.0                    py36_0
pyqt                      5.9.2              py36h05f1152_2
pysocks                   1.6.8                    py36_0
python                    3.6.8                h0371630_0
python-dateutil           2.7.5                    py36_0
python_abi                3.6                    1_cp36m    conda-forge
pytz                      2018.7                   py36_0
pyyaml                    3.13                     pypi_0    pypi
```

```
pyzmq                     17.1.2          py36h14c3975_0
qt                        5.9.7               h5867ecd_1
qtawesome                 0.5.3                   py36_0
qtconsole                 4.4.3                   py36_0
qtpy                      1.5.2                   py36_0
readline                  7.0                 h7b6447c_5
requests                  2.21.0                  py36_0
rope                      0.11.0                  py36_0
scikit-learn              0.20.2          py36hd81dba3_0
scipy                     1.1.0           py36h7c811a0_2
secretstorage             3.1.0                   py36_0
send2trash                1.5.0                   py36_0
setuptools                40.6.3                  py36_0
sip                       4.19.8          py36hf484d3e_0
six                       1.12.0                  py36_0
snowballstemmer           1.2.1                   py36_0
sphinx                    1.8.2                   py36_0
sphinxcontrib             1.0                     py36_1
sphinxcontrib-websupport  1.1.0                   py36_1
spyder                    3.3.2                   py36_0
spyder-kernels            0.3.0                   py36_0
sqlite                    3.26.0              h7b6447c_0
tensorboard               1.12.2                  pypi_0    pypi
tensorflow                1.12.0                  pypi_0    pypi
termcolor                 1.1.0                   pypi_0    pypi
terminado                 0.8.1                   py36_1
testpath                  0.4.2                   py36_0
tk                        8.6.10              hed695b0_0    conda-forge
tornado                   5.1.1           py36h7b6447c_0
traitlets                 4.3.2                   py36_0
typed-ast                 1.1.0           py36h14c3975_0
urllib3                   1.24.1                  py36_0
wcwidth                   0.1.7                   py36_0
webencodings              0.5.1                   py36_1
werkzeug                  0.14.1                  pypi_0    pypi
wheel                     0.32.3                  py36_0
wordcloud                 1.6.0           py36h516909a_0    conda-forge
wrapt                     1.10.11         py36h14c3975_2
wurlitzer                 1.0.2                   py36_0
xz                        5.2.4               h14c3975_4
zeromq                    4.2.5               hf484d3e_1
zlib                      1.2.11              h7b6447c_3
zstd                      1.3.7               h0b5b093_0    anaconda
```

B.1 Combined Architecture FF-LSTM

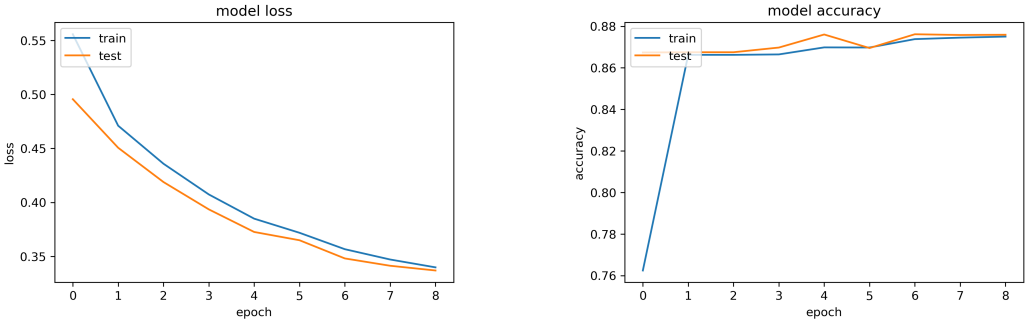


Figure B.6: Result plots for run 1 of Combined FF-LSTM architecture.

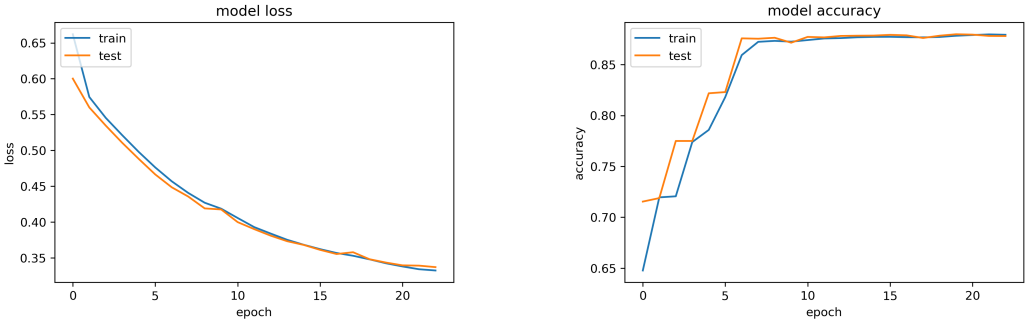


Figure B.7: Result plots for run 2 of Combined FF-LSTM architecture.

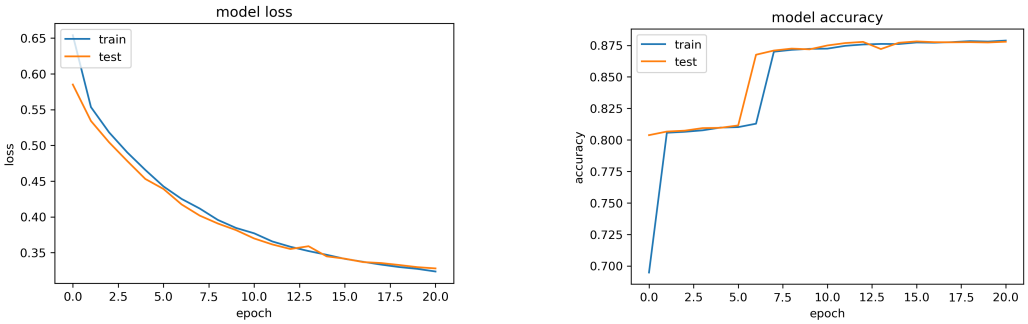


Figure B.8: Result plots for run 3 of Combined FF-LSTM architecture.

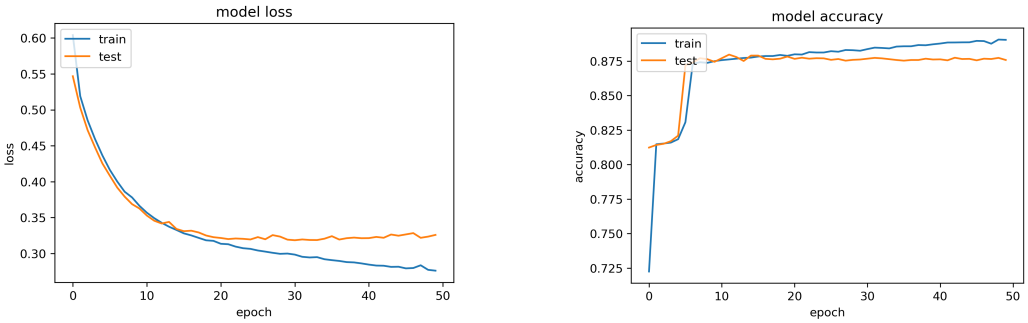


Figure B.9: Result plots for run 4 of Combined FF-LSTM architecture.

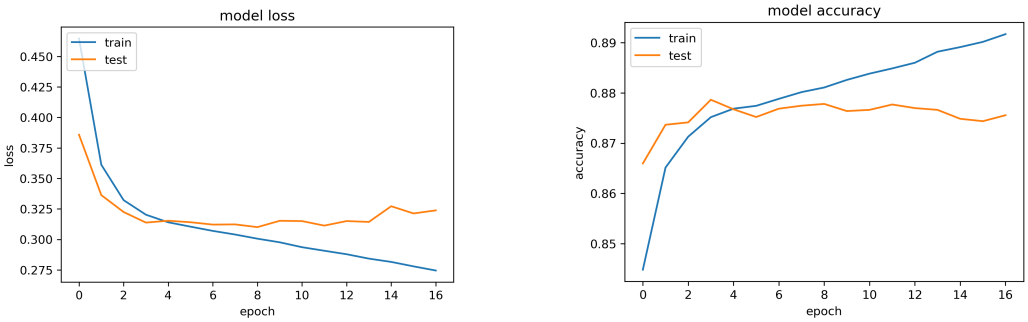


Figure B.10: Result plots for run 5 of Combined FF-LSTM architecture.

B.2 Combined Architecture LSTM-LSTM

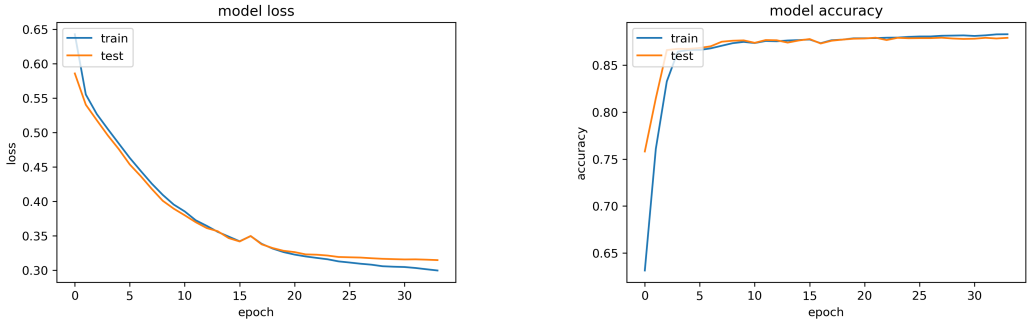


Figure B.11: Result plots for run 1 of Combined LSTM-LSTM architecture.

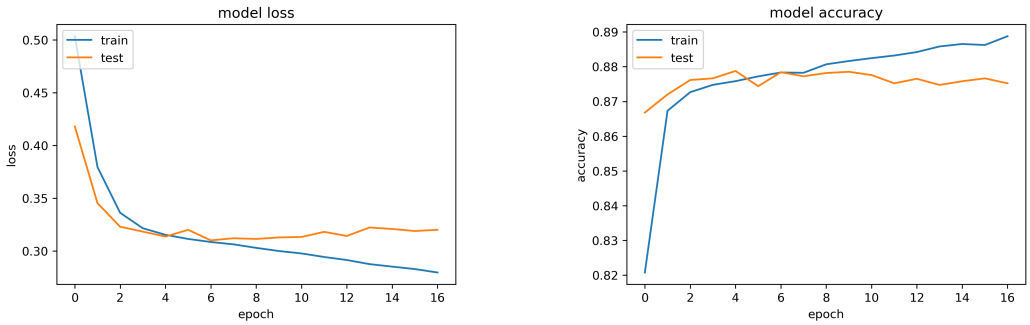


Figure B.12: Result plots for run 2 of Combined LSTM-LSTM architecture.

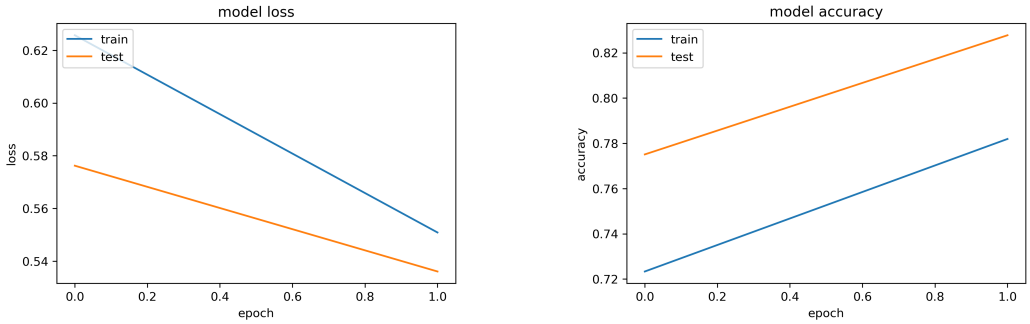


Figure B.13: Result plots for run 3 of Combined LSTM-LSTM architecture.

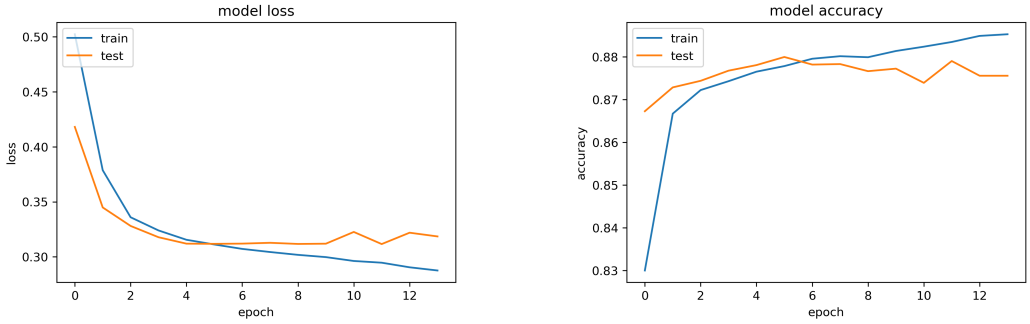


Figure B.14: Result plots for run 4 of Combined LSTM-LSTM architecture.

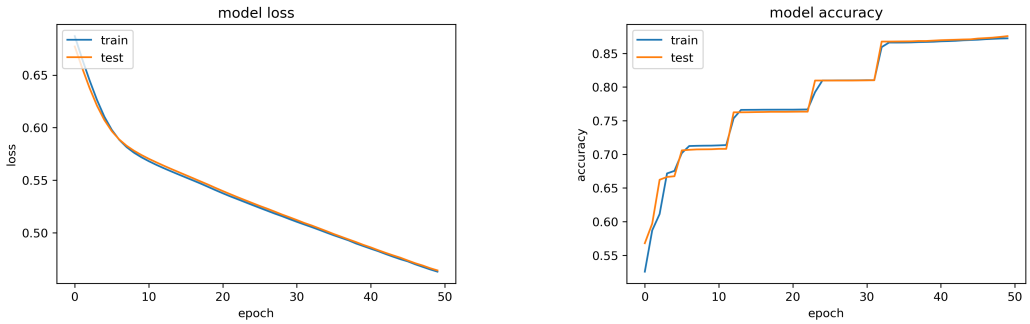


Figure B.15: Result plots for run 5 of Combined LSTM-LSTM architecture.

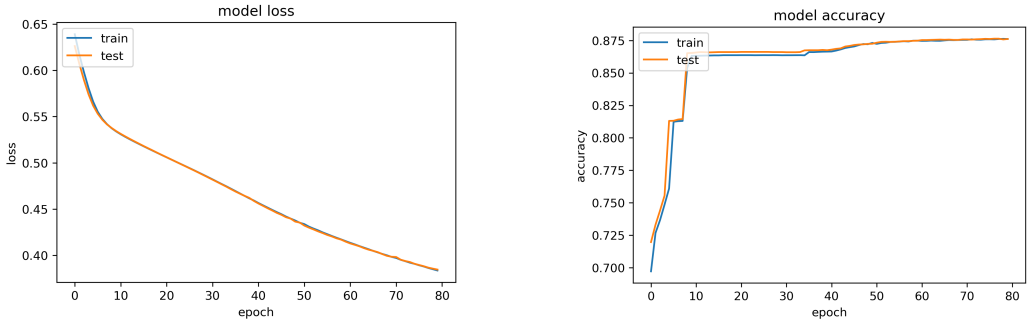


Figure B.16: Result plots for run 6 of Combined LSTM-LSTM architecture.

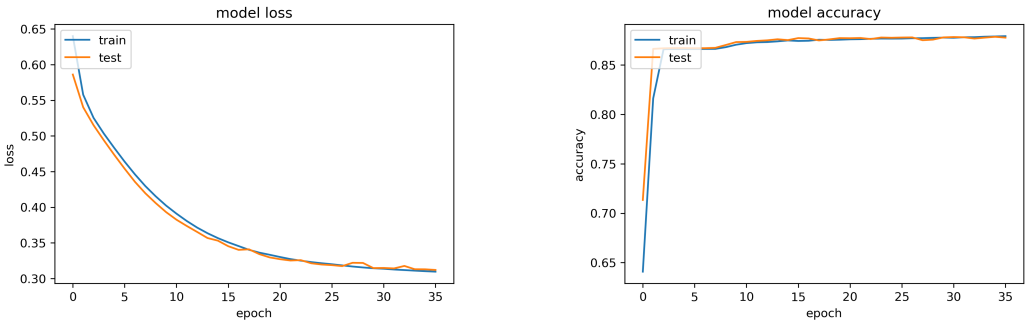


Figure B.17: Result plots for run 7 of Combined LSTM-LSTM architecture.

B.3 Result plots from running on a different platform

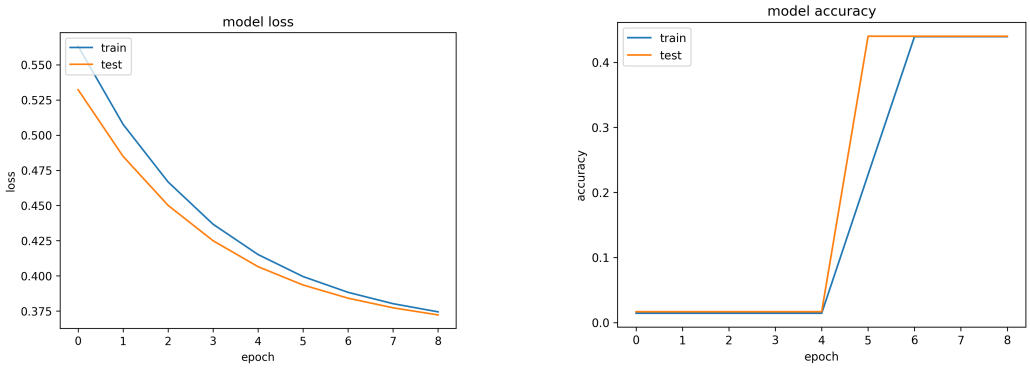


Figure B.18: Result plots from Gamestates model with FF network.

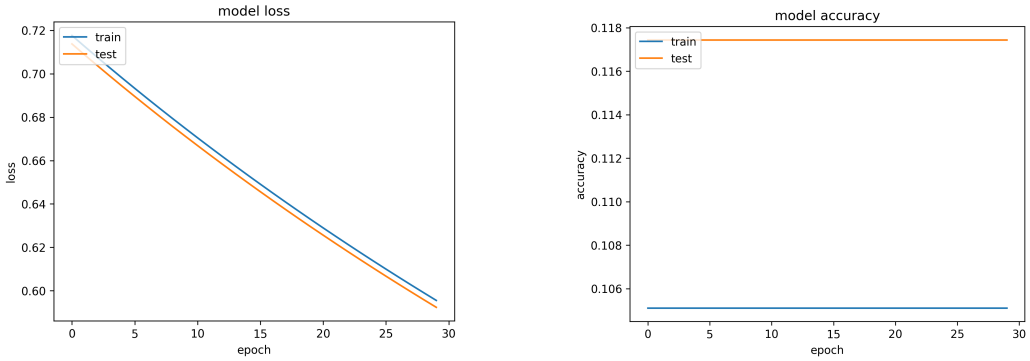


Figure B.19: Result plots from Gamestates model with LSTM network.

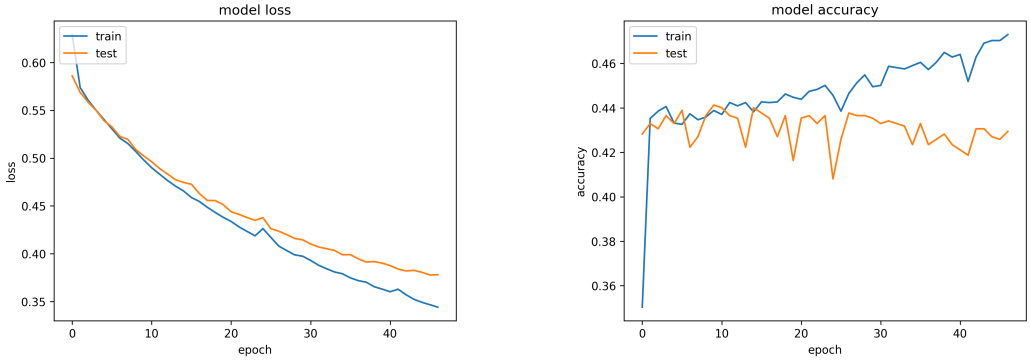


Figure B.20: Result plots from Chats model with LSTM network.

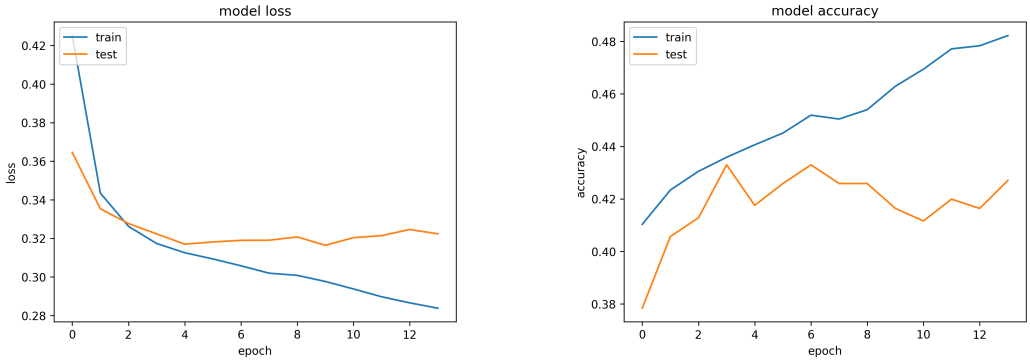


Figure B.21: Result plots from Combined model with FF-LSTM network.

Appendix C

Code Documentation

jSettlers Dataset API

Maria Apostolidou

Feb 10, 2020

CONTENTS:

CHAPTER

ONE

JSETTLERS DATASET API

Documentation of all the scripts, classes and methods used to convert .soclog files to a jSettlers dataset.

1.1 turn.py

Inserting the turn number to soclogs

Write new soclog files where the first field is the turn number. Initial set up of the game is considered turn 0 When it is time for a new turn the SOCGameState sends a message that the game state is state=15 state=15 (PLAY, start a normal turn, time to roll or play card) (for more information about states, their codes and meanings see SOCGame.java)

turn.**add_turns**(*inputfile*, *outputfile*)

Adds the turn number to the data

Reads an original soclog file and inserts the turn number at the beggining of each log. The new, extended soclog is saved as a new soclog file

Parameters

- **inputfile** (*file*) – The original soclog file to be modified
- **outputfile** (*file*) – The destination where the new soclogfile will be written

1.2 reduced_logs.py

Collecting the useful data from the soclogs

Selects the useful messages from the soclogs to extract the informations needed for the state features. SOCGameTextMsg, GAME-TEXT-MESSAGE, SOCGameState

reduced_logs.**read_soclog**(*soclogfile*)

Reads a soclog and separates basic columns

Read a soclog and separates columns by ':' The first 8 columns are the timestamp, when the message was sent The 9th column is the MessageType (for details about MessageTypes see /messages of jSettlers) The 10th column is the Message

Parameters **soclogfile** (*file*) – The soclog .soclog file

Returns **soclog** – the soclog in dataframe form

Return type dataframe

reduced_logs.**reduce_log**(*soclog*)
Creates a dataframe that contains only the useful parts of the logs

Creates a dataframe that contains only rows of MessageType SOCGameTextMsg | GAME-TEXT-MESSAGE | SOCGameState. Deletes the timestamp from the row. Saves to a csv using the '|' as delimeter

Parameters **soclog** (*dataframe*) – The dataframe of a soclog (as returned from read_soclog)

Returns **data** – A dataframe with the useful data (rows) collected from the soclog

Return type dataframe

reduced_logs.**write_to_csv**(*data*, *filename*, *delimiter*)
Writes a dataframe to a csv file

Parameters

- **data** (*dataframe*) – The dataframe to be written to the csv file
- **filename** (*file*) – The destination file
- **delimeter** (*str*) – The separator two put between fields in the csv (e.g. '|', ':')

1.3 collectfeatures.py

Creating the game state and chat features and labels

Makes a feature vector for every game turn that shows the game state #GameStates = #game turns for each log file Makes the labels Write all the chat data to a file

collectfeatures.**devCard**(*df*)
method to call for a SOCDevCard message

This method is called when a message of type SOCDevCard is found in the log. Used to update the development cards of a player

Parameters **df** (*pandas dataframe*) – The row of the game dataframe that holds SOCDevCard message

Returns

- **playernum** (*player number*)
- **actionType** (*bought or played*)
- **cardType** (*type of development card*)

collectfeatures.**get_board**(*message*)
Gets the board layout from the data

Fills in the board layout part of the game state feature vector. The hexLayout refers to the resources each hexagon on the board offers. The numLayout refers to dice number corresponding to each hexagon. Initial placement of the robber (in the dessert hexagon) is also found here. This information is retrieved from the message of type SOCBoardLayout, during the initial set up phase of the game (turn 0).

Parameters **message** (*str*) – A SOCBoardLayout message

Returns

- **hexLayout** (*int list*) – The resources on the board. (See jSettlers boardlayout class to map these numbers to the coresponding resource types.)
- **numLayout** (*int list*) – The dice number on each board hexagon (See also jSettlers board-Layout)

- **robberHex** (*hexadecimanl coordinate*) – Initial position of the robber (on the dessert hexagon)

collectfeatures.**get_buildings0**(*df*)
Returns the positions where the players placed 1st road and settlement.

During the initial setup phase all players place a settlement and a road on the board.

Parameters **df** (*pandas dataframe*) – The rows of the game dataframe that hold SOCPutPiece messages at turn 0

Returns **buildings** – players' settlements and roads at turn 0

Return type dictionary

collectfeatures.**get_chat**(*df*)
method to get a chat message of a SOCGameTextMsg

Parameters **df** (*pandas dataframe*) – The row of the game dataframe that holds a SOCGameTextMsg message

Returns

- **nickname** (*str*) – The nickname of the emitter
- **chat_msg** (*str*) – The chat message emitted

collectfeatures.**get_chats0**(*df*)
method to save the chats of the initial setup phase to chatsDF

Parameters **df** (*pandas dataframe*) – The rows of the game dataframe that hold SOCGameTextMsg messages at turn 0

collectfeatures.**get_int_value**(*message*, *base*)
Converts a field from a str message to a number

From a string of the form "varname=val" returns only the value in hex

Parameters

- **message** (*str*) –
- **base** (*str, hex or dec*) –

Returns **val**

Return type int (hexadecimal)

collectfeatures.**get_int_values**(*layoutmessage*)
Converts a layout from a str message to a list of numbers

From a string of the form "infoname= { ...int values... }" returns only the int values in a list of integers

Parameters **layoutmessage** (*str*) – A str describing the hex or num Layout (e.g. hexLayout= { 50 6 ... 6})

Returns **layout** – The mapped list of integers

Return type int list

collectfeatures.**get_players**(*df*)
Finds players' nicknames and number ids

From the SOCSitDown messages of the soclogs, finds the nicknames and player ids of the people participating in the game

Parameters **df** (*pandas dataframe*) – The rows of the game dataframe that hold SOCSitDown messages at turn 0

Returns **nicks** – players' nicknames and numbers, key:playerid(0,1,2,3) value:nickname

Return type dictionary

collectfeatures.**get_state**(*df*, *prev_state*)
Creates a game state for a specific turn

For the logs of a game turn calls methods to extract the infromation from each MessageType.

Parameters

- **df** (*pandas dataframe*) – The soclogs in dataframe form, as returned from read_soclog() for a given turn
- **prev_state** (*game state at the previous turn*) –

Returns

- **game_state** (*GameState feature vector*) – The game state feature vector of the game turn
- **labels** (*Labels*) – The prediction labels for this game turn

collectfeatures.**initial_setup_state**(*df*)
Creates the game state for the initial setup phase of the game

Creates the game state for the setup phase of the game, i.e. turn 0. During this phase each player places on board his first 2 settlements and 2 roads. Also returns all chat messages during this phase to save to chatsDF

Parameters **df** (*pandas dataframe*) – The soclogs in dataframe form, as returned from read_soclog() for tun 0

Returns

- **turn0_game_state** (*GameState feature vector*) – The game state feature vector from the initial set-up phase
- **chats0** (*a pandas df*) – The chat messages during the setup phase

collectfeatures.**moveRobber**(*df*)
method to call for a SOCMoveRobber message

This method is called when a message of type SOCMoveRobber is found in the log. Updates the position of the robber on the board.

Parameters **df** (*pandas dataframe*) – The row of the game dataframe that holds SOCPutPiece message

Returns **coord** – Hexadecimal coordinate on the board

Return type hex int

collectfeatures.**playerElement**(*df*)
method to call for a SOCPlayerElement message

This method is called when a message of type SOCPlayerElement is found in the log. Used to update the resources of a player

Parameters **df** (*pandas dataframe*) – The row of the game dataframe that holds SOCPlayerElement message

Returns

- **playernum** (*player number*)

- **actionType** (*Set, Gain or Loose a resource*)
- **elementType** (*Type of the resource*)
- **val** (*int*)

collectfeatures.**putPiece**(*df*)
method to call for a SOCPutPiece message

This method is called when a message of type SOCPutPiece is found in the log. Updates the state of a player.

Parameters **df** (*pandas dataframe*) – The row of the game dataframe that holds SOCPutPiece messages

Returns **building** – Holds the player id, the type of piece he placed on the board and the coordinates where he placed it

Return type dictionary

collectfeatures.**read_soclog**(*soclog*)
Reads a soclog file as a panda dataframe

Reads a soclog file from the reduced soclogfiles. These are the files that contain only the useful rows of data from the original log files. Produces a pandas dataframe with columns Turn, MessageType and Message.

Parameters **soclog** (*filename*) – The .soclog file from the reduced soclogs (see /reduced)

Returns A pandas dataframe with the Turn, MessageType and Message columns

Return type dataframe

1.4 DevCard.py

class DevCard.**DevCard**(*cardType*)
A class used to present a development card

In the Settlers of Catan there are 25 development cards: 14 knight cards 2 Road Building cards 2 Monopoly cards 2 Discovery cards 5 Victory Point cards

cardType
codes for the types of Dev Cards as in jSettlers

Type int

bought
Dev card bought flag

Type boolean

played
Dev card played flag

Type boolean

1.5 GameState.py

class GameState.**GameState**(*turn*, *hexLayout*, *numLayout*, *robberHex*, *player0state*, *player1state*, *player2state*, *player3state*)

A class used to present a game state

A game state feature vector describes the game state at a specific turn. A collection of (#game turns) features describes a whole game

turn
the gameturn

Type int

robber
the hex coordinates of the robber's position

Type int

player0
playerstate of the player sitting at position 0 (player id = 0)

Type Playerstate

player1
playerstate of the player sitting at position 1 (player id = 1)

Type Playerstate

player2
playerstate of the player sitting at position 2 (player id = 2)

Type Playerstate

player3
playerstate of the player sitting at position 3 (player id = 3)

Type Playerstate

place_robber(*coord*)
set robber position on board

print_GameState()
prints the values of a gamestate

write_to_DF()
returns the game state in list form to write to gamestateDF

1.6 Labels.py

class Labels.**Labels**(*turn*)

A class used to present the labels of a game turn

labels that are true mean that at the given turn the player did the action indicated by the label. if no action label is turned into True a no action label is "activated"

turn
the gameturn

Type int

played_dev_card

Type boolean

built_road

Type boolean

built_setm

Type boolean

upgraded_city

Type boolean

bought_dev_card

Type boolean

made_offer
player suggested a trading offer to another player

Type boolean

traded_with_player

Type boolean

traded_with_bank

Type boolean

traded_with_port

Type boolean

no_action

Type boolean

check_no_action()
Check if no action was made during a game turn

if no action label has been turned into true during a game turn turn the no action label true. Call this function just before you are ready to finalize and save the labels

print_labels()
Print the labels' values

update_buildings(*pieceType*)
Update the labels when the player builts a road, city or settlement

Check what was built by the player when he put a piece on the board and update the labels of the turn

Parameters **pieceType** (*str*) – road, setm or city

write_to_DF()
return labels to list form to write a row at labelsDF

1.7 Piece.py

class Piece.**Piece**(*type*, *location*)
A class used to present a piece of the game

A class that represents the settlements, roads and cities that a player can build on the board. Pieces use the same labels as the jSettlers code (PieceTypes).

type
0 for road, 1 for settlement, 2 for city, as in jSettlers

Type int

location
the hexadecimal number indicated the board coordinate

Type int

1.8 PlayerState.py

class PlayerState.**PlayerState**(*number*, *nickname='dummy'*)
A class used to present a player state

A game state feature vector describes a players state at a specific point during the game. A seperate dataset including the resources information (true nubmer of resources each player holds at each game turn) was collected, but was not used in training because it does not distinguish between resource information known to all players and resource information that is secret (stealing cards, discarding cards etc.)

number
the player's number id

Type int

nickname
the player's nickname

Type str

clay
the player's resource units of clay

Type int

wood
the player's resource units of wood

Type int

ore
the player's resource units of ore

Type int

wheat
the player's resource units of wheat

Type int

sheep
the player's resource units of sheep

Type int

settlements
the coordinates of the player's settlements

Type list

cities
the coordinates of the player's cities

Type list

roads
the coordinates of the player's roads

Type list

dev_cards
the development cards the player has bought and/or played

Type list

already_exists(*pieceType*, *coord*)
check if a pieceType has already been place on these coords

Parameters

- **piceType** (*int*) – 0 for road, 1 for settlement and 2 for city
- **coord** (*int*) – the hex coordinate on the board

Returns True if successful, False otherwise.

Return type bool

bought_devCard(*cardType*)
Updates development card list when player has bought a new card

Called from card_action method when there is a SOCDevCard message in the log of ActionType = 0

Parameters **cardType** (*int*) – DevCard types as in jSettlers (see DevCard)

built_road(*location*)
updates list of players roads

Parameters **location** (*int*) – the hex coordinate location on board

built_settlement(*location*)
updates list of players settlements

Parameters **location** (*int*) – the hex coordinate location on board

card_action(*actionType*, *cardType*)
Called when there is a SOCDevCard message in the log

Parameters

- **actionType** (*{'BOUGHT', 'PLAYED'}*) – player bought of played a devcard, other actions ignored
- **cardType** (*int*) – DevCard types as in jSettlers (see DevCard)

Returns bought, played, ignore (other action type of unknown card type, i.e. 9)

Return type Str

change_in_resources(*actionType*, *elementType*, *value*)
called when there has been a change in the player's resources

get_piece_at_location(*pieceType*, *loc*)
Return the piece of piecetype at the given location

Similar to already_exists, but returns the item rather than true/false

Parameters

- **pieceType** (*int*) – 0 for road, 1 for settlement, 2 for city
- **location** (*int*) – hex coordinate location on board

Returns the piece built in that location

Return type *Piece*

new_build(*type*, *coord*)
player built something, disambiguation

if road, setm or city are successfully built returns a str with the type that was built (to change the labels value accordingly)

Parameters

- **type** (*int*) – 0 for road, 1 for settlement, 2 for city
- **coord** (*int*) – the coordinate on the board

Returns road, setm of city

Return type Str

played_cards(*cardType*)
Number of cards the player has played

Returns a number that show how many cards of cardType the player has played

Return type int

played_devCard(*cardType*)
Updates development card list when player has played a dev card

Called form card_action when there is a SOCDevCard message in the log of ActionType = 1 Notice that victory point cards are played immediately when bought

Parameters **cardType** (*int*) – DevCard types as in jSettlers (see DevCard)

print_playerState()
print the attributes of a playerState

to_list()
convert a playerstate to a list of 56 features

Returns The playerstate list

Return type list

upgraded_city(*location*)
updates list of players cities and settlements

From the player's settlements list deletes the settlement built in that location and appends the city in the list of cities

Parameters **location** (*int*) – the hex coordinate location on board

PYTHON MODULE INDEX

INDEX

R

S

T

U

W