FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Opponent Modelling Library for StarCraft AI
Student:	Tomáš Bohuslav
Supervisor:	RNDr. Michal Čertický, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

The aim of the thesis is to implement an open-source opponent modelling library for Artificial Intelligence agents playing strategy game StarCraft Brood War. The library will be written in C++, on top of BWAPI framework, and will provide the following functionality:
1) Keep track of all the known units of the opponent and their locations.
2) Automatically infer additional information about the opponent in the partially-observable scenarios, using expert domain knowledge.
3) Predict what the opponent will have in the near future, based on in-game collected information and game knowledge.
4) Predict which strategy will be opponent playing using machine learning methods, based on previous games.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 24, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Opponent Modelling Library for StarCraft AI

Department of Software Engineering
Supervisor: RNDr. Michal Čertický, Ph.D.

May 15, 2018

Acknowledgements

I would like to thank to my supervisor, RNDr. Michal Čertický, Ph.D., for his patience, valuable advices and his AI research for Starcraft community. Thanks to my friends and family who supported me during difficult times.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 15, 2018 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Bohuslav, Tomáš. *Opponent Modelling Library for StarCraft AI*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se zabývá knihovnou pro modelování protivníka za použití umělé inteligence a strojového učení. Knihovna je navržena pro umělou inteligenci hrající počítačovou hru Starcraft, ovšem nabízená řešení základních problémů při modelování oponenta se dají uplatnit nejen ve hře Starcraft, ale i v celé rovině real-time-strategy her.

Klíčová slova Implementace knihovny, Modelování opponenta, Starcraft, Umělá inteligence, Strojové učení, Neuronová síť

Abstract

This thesis implements a library for opponent modelling problem using artificial inteligence and machine learning methods. The library is designed for AI modules playing computer game Starcraft. However, provided solutions of fundamental opponent modelling problems can be used not only in Starcraft, but in the sphere of all real-time-strategy games.

Keywords Library implementation, Opponent modeling, Starcraft, Artificial intelligence, Machine learning, Neural network

Contents

List of Figures

Introduction

Computer games have large impact on todays entertainment and also on science researches, especially in section of artificial intelligence. Not only players compete against each other. Special tournaments allow to programme own AI bot. Those bots then compete in the tournament instead of humans. Great example is an SSCAIT (Student StarCraft AI Tournament and Ladder) [5]. Generally, AI tournaments have special added value. It is a brilliant way how to improve skills and knowledge of machine learning and artificial intelligence for the programmers.

Undoubtedly, Starcraft is one of the most difficult real-time-strategy games. Players must perform hundreds of actions every single minute of the game and even the best still have room for improvements. Human players are simply not possible to control each single unit separately every second of the game. However, computers can. Computers can give commands to all units every second. Moreover, it can give tasks to its units every frame of the game. They simple dominate over humans with mechanics and unit control but they struggle with decision making and strategy moves. Starcraft is a very complex game with many specific areas of skill and knowledge players or AI agents have to cover in order to be successful. One of the main areas to cover is modelling opponents moves and adapting strategies as a reaction on scouting information. Imagine playing chess without considering opponents moves on the board, nearly impossible.

Implementing an opponent modelling tool for AI modules is a challenge which requires a good knowledge not only about artificial intelligence, but also about the game itself. Opponent modelling in Starcraft is this thesis topic. We will analyse scouting information AI agents acquire, separate key and redundant information, deduce opponents state and predict the state for next few minutes in the game.

Thesis structure

The first chapter summarizes the aim of the thesis. The second chapter focuses on theoretical background about Starcraft AI research, introduces related works and extendable bots, which programmers can use as a starting point for their Starcraft AI research. The third chapter describes implementation details, used technologies and programming solution to the aim of the thesis. It also explains the ideas behind chosen design and class structure. The fourth chapter provides a brief introduction into artificial neural networks and its usage in our implementation. The fifth chapter analysis measured results and evaluates correct model for our problem.

Chapter 1

Aim-of-thesis

The main goal of this thesis is to create a strong tool for AI agents that will help in keeping track of their opponents and provide essential information necessary for future game plan. If we try to summarize main conditions for making an accurate model of an opponent, we basically get the assignments of this thesis.

First of all, we must keep track of all known units and their location. Based on those collected data, we can automatically infer additional information about opponent. It will maximize accuracy of the model and make it closer to real situation. Built on current opponent's model, we can predict game state in near future using machine learning methods. After each game, Starcraft allows to save replay of the game. We can use this feature to study past games and carry some information into next games in order to predict opponents strategy and moves even more accurate.

The aim of the theoretical part is to introduce technologies for writing an AI agent in Starcraft, analyse thesis tasks and current methods used in Starcraft AI programming for opponent modelling.

Practical section will focus on creating an opponent modelling library written in C++ on top of BWAPI framework. We will describe necessary functionalities for saving information about an opponent and the ability to predict game states in next few minutes of the game.

Chapter 2

State of the art

In the context of Starcraft AI development there are various libraries and existing frameworks to support the research or provide tools for creating own AI module. In this chapter are introduced the most essential works, which names became almost a synonym with Starcraft AI.

2.1 Starcraft AI development

2.1.1 BWAPI

According to [6], The Brood War Application Programming Interface (BWAPI) is a free and open sourced framework written in C++ used to interact with computer game Starcrat: Broodwar. BWAPI only reveals the visible parts of the game state to AI modules by default. Information on units that have gone back into the fog of war is denied to the AI. This enables programmers to write competitive non-cheating AIs that must plan and operate under partial information conditions. BWAPI also denies user input by default, ensuring the user cannot take control of game units while the AI is playing. Each unit in the game has a unique Unit object identified by a numeric value. Unit objects are not deleted until the end of the match [7].

Although, C++ is not the only programming language, which supports Starcraft AI development. An alternative for Java developers might be BWMirror API. It allows programmers to treat native BWAPI C/C++ objects as if they were Java objects [8] or JNIBWAPI, which provides a Java interface for the Brood War API (BWAPI), using Java Native Interface (JNI) to communicate over a shared memory bridge [9].

2.1.2 BWTA

Broodwar Terrain Analyzer (BWTA) is an add-on for BWAPI which analyses the current starcraft map and returns the set of expansion locations, regions,

and choke points. Built on BWTA a new fork called BWTA2 was programmed. It offers more functionalities and compatibility with BWAPI 4.

Figure 2.1: BWTA ilustration of Starcraft map divided into regions. Choke points are located between adjacent regions [1].

2.1.3 BWME

Brood War Easy Map (BWME) is a C++ library that analyses Brood War's maps and provides relevant information such as areas, choke points and base locations. It is built on top of the BWAPI library. It first aims at simplifying the development of bots for Brood War, but can be used for any task requiring high level map information. It can be used as a replacement for the BWTA2 add-on, as it performs faster and shows better robustness while providing similar information [2]. On the other hand, it doesn't provide any geometric description (polygon) of the computed areas like BWTA.

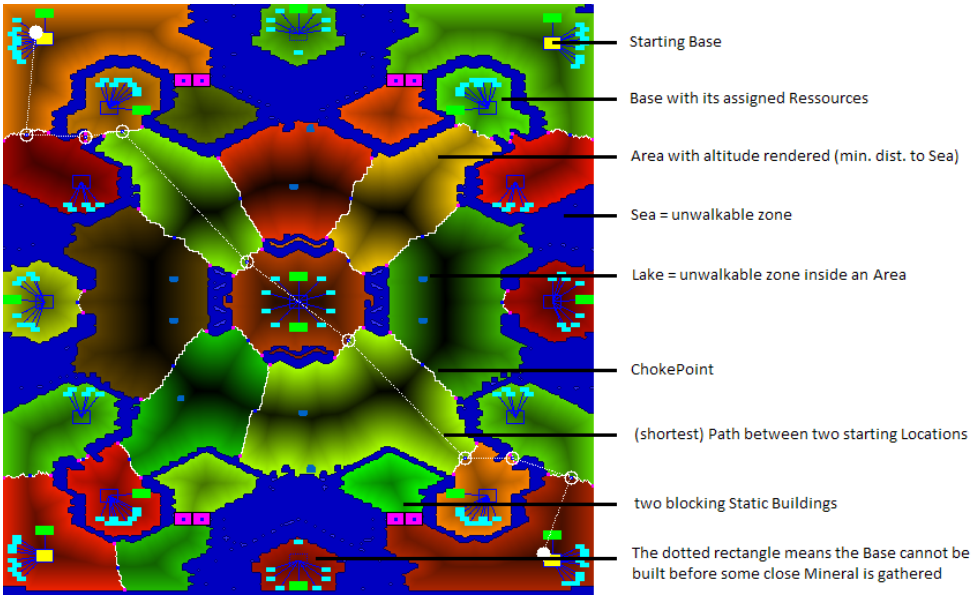

Figure 2.2: BWEM ilustration of regional split [2].

2.2 Extendable AI modules

BWAPI comes with an AIModule virtual class which allows any programmer to implement own Starcraft AI module. It offers 17 virtual methods called when an appropriate event occurs. Adding source code into those methods enables AI module perform actions.

Developing AI module from the ground up is long and very difficult process. Thus, usual way is to extend a source code of already existing bot. Probably the most known Starcraft AI bot is an UAlbertaBot bot written in C++. It has competed in most major StarCraft AI Competitions since 2010 and won the 2013 AIIDE StarCraft AI Competition. It is written by David Churchill, Assistant Professor of Computer Science at Memorial University of Newfoundland and co-organizer of the AIIDE StarCraft AI Competition [10].

Chapter 3

Implementation

This chapter describes library design and its most important implementation details. We will go through library classes, modelling problems with their solutions and introduce external libraries used in the implementation.

3.1 Library design

Library was designed as statically-linked library due to direct compilation into bot final executable file. Starcraft is a game which doesn't require drastic memory usage. However, dynamically-linked library might use some extra CPU time during the game, which could be used by the AI modules for other purposes. Although, library is released with all the source code, because prediction in the library implementation uses neural network. In case programmer would like to change the topology and train his own neural network or extend functionalities of the library, he needs to adjust the source code to satisfy his requirements. Although, library was constructed using Microsoft Visual Studio 2015 available from [11] and it is recommended to use it for external development of the library.

3.1.1 Classes

CPlayerModel is a parent class describing common data for all three races in the games. If an instance of this class is created, it is considered there are no information about opponent available and he chose a random race.

CPlazerZerg, **CPlayerProtoss** and **CPlayerTerran** are child classes of CPlayerModel. Naturally, class represents a player with one of Starcraft races. Each class overrides virtual methods from parent class and adjust them to fit inheritance for the race it is describing.

CNeuralNetworkModel represents model of neural network. It is a parent class which implements training, testing and validating functionalities

for the network. Using its methods, programmers are able to set parameters for the network or print training and testing results into files.

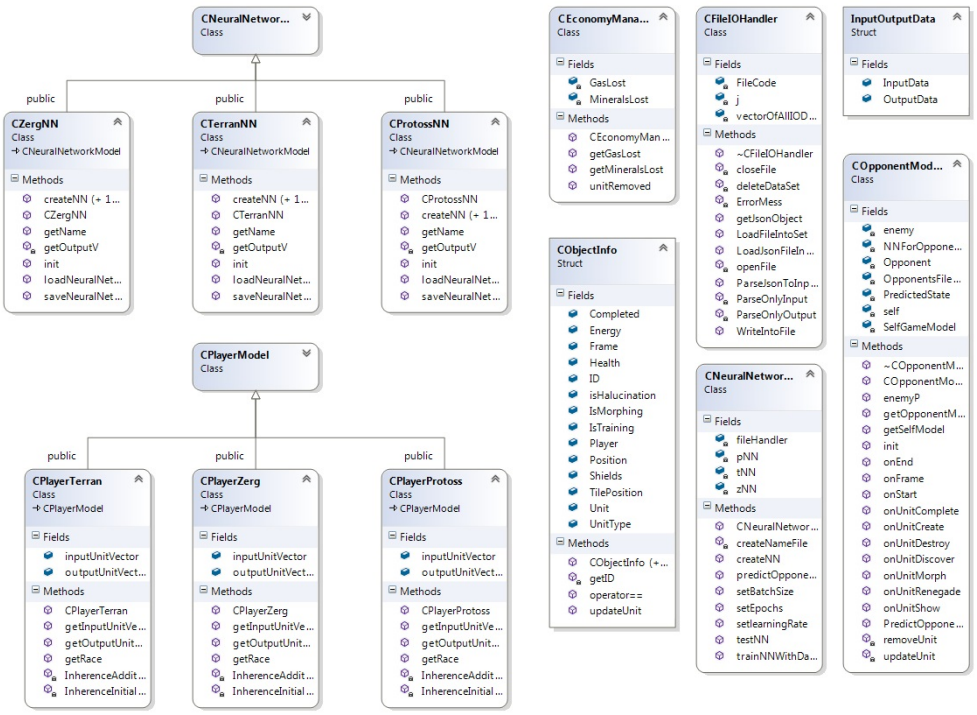

Figure 3.1: Opponent modelling library class diagram.

CNNZerg, **CNNTerran** and **CNNProtoss** are child classes of **CNeuralNetworkModel**. Similarly as CPlayer classes, CNN classes represent neural network model for each Stracraft race. Their inherited methods and instance variables are adjusted to fit the corresponding race. Importance of separated networks for each race is evident as each race has its own unique units and design.

CNeuralNetworkHandler gathers all neural networks created for game prediction of the future game state. The idea behind is the ability to set the parameters and manipulate with models of neural networks using only one object.

CFileIOHandler handles input and output operations with files. It can read and save files in JSON format, which simplifies reading for humans and uses conventions that are familiar to programmers of the C-family of languages [15].

CEconomicManager separates Economics from physical objects (units and buildings). It stores information about amount of minerals and gas opponent have lost during the game.

COpponentModelling connects together all pieces of the library. Through **COpponentModelling** instance, programmers are able to use features library provides.

3.1.2 External libraries

In the implementation are used two external libraries. The first one, `Niels Lohmann's JSON` [12], is used for input and output operations with files using JSON format. The second one, `Tiny_dnn` [13], is used for creating neural networks and learning them the ability to predict opponent's future unit composition. Both of them are introduced and described in this section.

- `Niels Lohmann's JSON` is written for Modern C++ and C++11 as header-only JSON class. It is very intuitive for usage, especially because it works similarly as C++11 STL containers. In fact, it even satisfies the ReversibleContainer requirement [14]. Usage in Opponent Modelling is for saving data into files in JSON format in order to make it easy for humans to read and for machines to parse and generate [15].

- `Tiny_dnn` is a C++14 implementation of deep learning. It is suitable for deep learning on limited computational resource, embedded systems and IoT devices. `Tiny_dnn` is pure C++ portable and header-only implementation which requires no installation of any additional software. It has clear understandable documentation with tutorial guide and usage explanation on various examples. All of the advantages are the reason why it has been chosen for implementing neural network in Opponent Modelling library.

3.2 Unit database

Knowledge of opponent's units and buildings is the very basic information in every RTS computer game. Without proper scouting, players or AI modules are left blind to guess countless probabilities of opponent's tactic, game plan or unit composition. Thus, scouting opponent's units is necessary in order to build own game plan, which naturally counters the opponent's one. Scouting only doesn't provide information about current situation. Build on current situation, we can infer additional units and buildings opponent needed to reach the current game state.

As mentioned in section 2.1.1, BWAPI does not allow AI agents to access any unit covered by the fog of war, even though the unit was previously seen. It follows AI modules must keep their own database of all scouted enemy unit in order to track opponents steps. AI agents have a decent advantage compared to humans. Because every unit has unique numeric ID available through BWAPI while the unit is visible, AI agent can determinate whether

the unit that showed, went back to fog of war and appeared again is still the same unit or not.

Opponent modelling library has three essential pieces for representing the unit database. It is **CObjectInfo** structure, vector **AllUnitsStored** and **UnitsMap**. Our implementation saves every unit seen during game as it's own new object defined as an instance of **CObjectInfo** structure. We copy BWAPI's unit all available data and place it into newly created unique object representing the unit in our model. Whenever there is a request for all previously seen units, library can provide them as **CObjectInfo** instances. Those instances are stored in two containers. The first one is a vector called **AllUnitsStored**. It stores all units and buildings sorted by their ID. The second one is a **UnitsMap**, which stores all units in vectors sorter by their type according to BWAPI::UnitTypes. In the event of request for all units of given type, map returns them in a single vector. It allows programmers to access units directly or get their number by calling a size method on the vector.

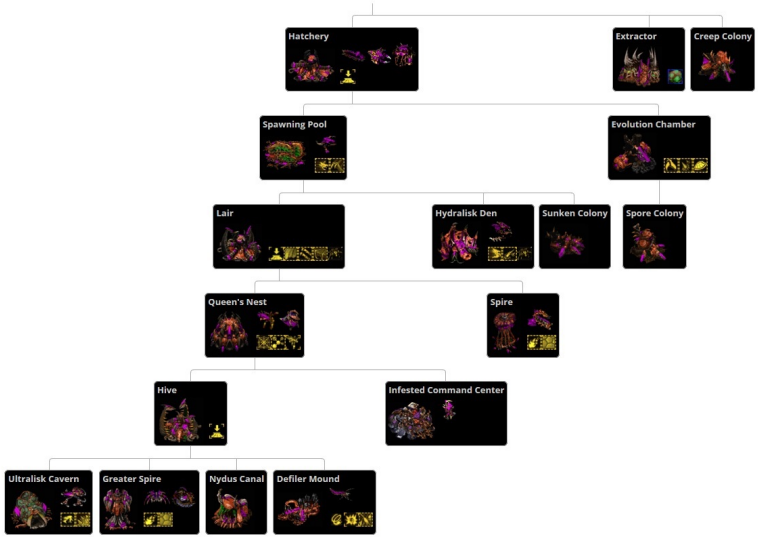

Figure 3.2: Ilustration of zerg technology tree [3].

3.3 Inference

Sometimes we can add units to the database even without seeing them and still be sure they exists. According to each race technology tree [3], we can deduce additional information about opponent's current situation whenever a unit is spotted. The idea behind is, find the unit in technology tree, get all

previous requirements for building this unit, check if they are already in our database and add those, who are not.

For instance, it is fourth minute of the game. AI agent scouted the first enemy unit and it is a Zerg hydralisk. Technology tree describes hydralisk as a unit created for 75 minerals and 25 gas units, after building hydralisk den was built by the player. It means the player must have built at least one hydralisk den, spawning pool, extractor, and overall must have mined at least 375 minerals and 75 gas units. Our library must deduce and record those information, if we want to provide reliable data model.

3.4 Prediction

Prediction in RTS games is the alpha and the omega factor for wining. Our goal is to provide accurate information about opponent in the near future, based on current situation and scouted or inferenced knowledge. To achieve our goal, we use machine learning methods, specifically neural network, to predict numbers of units and building in next three minutes. There are countless opportunities for various combination of the army and building composition, but based on collected information from previous games, we can train the network to recognize situations and provide as accurate prediction as possible. The whole process of training and evaluating is described in 4.2.

Chapter 4

Neural network

Artificial neural network is a computational model of connected nodes (neurons) inspired by the biological neural networks. Neural nets are a means of doing machine learning, in which a computer learns to perform some task by analyzing training examples. To each of its incoming connections, a node will assign a number known as a "weight." When the network is active, the node receives a different number from each of its connections and multiplies it by the associated weight. It then adds the resulting values together, yielding a single number [16].

4.1 Introduction to neural networks

There are several types of neural networks currently being used in machine learning. In this thesis, we cover only basics on feedforward type of network, evaluation of input values and measuring error of output values.

4.1.1 Feedforward neural network

As stated in [4], feedfoward neural networks are artificial neural networks where the connections between units do not form a cycle. The information only travels forward in the network (no loops), first through the input nodes, then through the hidden nodes (if present), and finally through the output nodes. Feedforward neural networks compute a function f on fixed size input x such that $f(x) \approx y$ for training pairs (x, y). Generalized artificial neural network consists of an input layer, some number (possibly zero) of hidden layers, and an output layer. Neurons in each layer are fully connected with neurons in neighbouring layers, but not with neurons in the same layer. Each connection is specified by a numeric value called weight. Whenever an input is proceed into the network, each neuron sums all the inputs from previous layer multiplied by the weigh of connection, use the activation function on the

sum and passes the output to next layer. Process is repeated in next layer until result reaches the output layer.

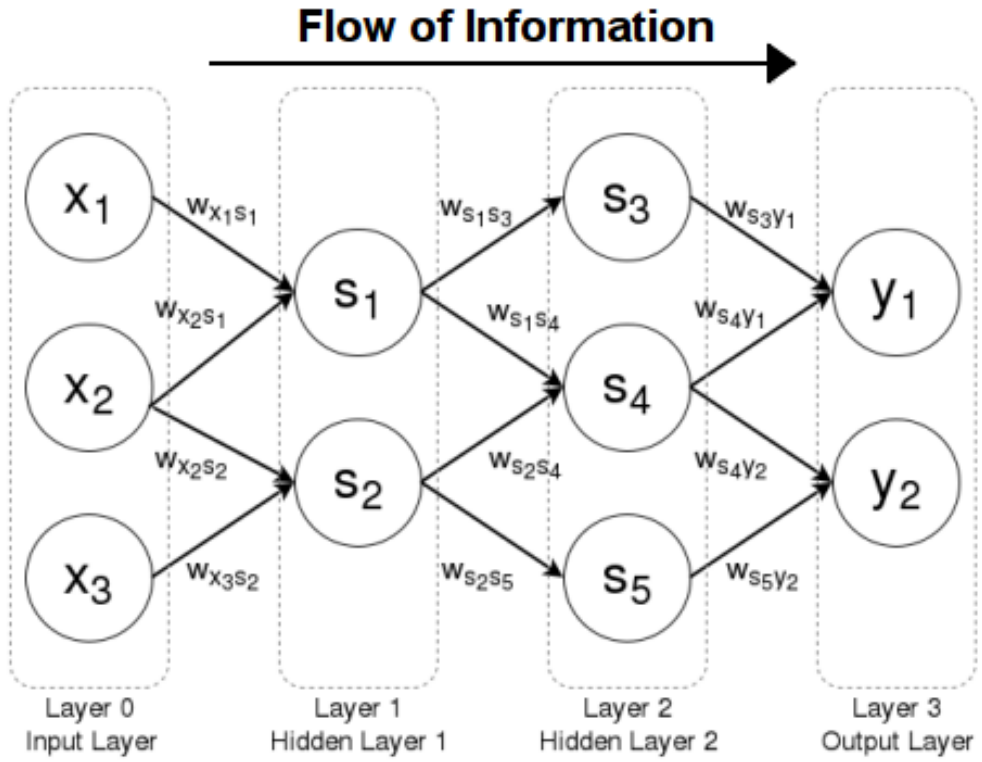

Figure 4.1: A four layer feedforward neural network example [4].

4.1.2 Activation function

Activation function defines the output of neuron. It is a mathematical function giving corresponding output value for each possible input value. Used activation function differ from type of problem neural network is programmed to solve.

4.1.3 Classification and regression

There two types predictive modelling. Classification and regression. Classification predictive modelling is the task of approximating a mapping function *(f)* from input variables x to discrete output variables y. The output variables are often called labels or categories. The mapping function predicts the class or category for a given set of data. It is common for classification models to predict a continuous value as the probability of a given example belonging to

each output class. On the other hand, regression is the task of approximating a mapping function *(f)* from input variables x to a continuous output variables y. The output variables are a real-value, such as an integers or floating point values. These are often quantities, such as amounts and sizes [17].

4.1.4 Loss functions and training

Typically, the learning process requires the definition of an error function that quantifies the difference between the computed output of the network and the true value for a given input. In our implementation of neural network, we use mean squared error function (MSE), defined for a set of N input-output pairs as:

$$E(X) = \frac{1}{N}\sum_{i=1}^{N}(o_i - y_i)^2 = \sum_{i=1}^{N}(g(w \cdot x_i) - y_i)^2 \tag{4.1}$$

where o_i denotes the network output, w represents the weights, g stands for an activation function and y_i denotes the desired output for given x_i input. Training neural network means adjusting the weights of the network in order to minimize the error of loss function. For this purpose, we use gradient descent to correct the parameters, which yields into following delta equation for each iteration:

$$\partial w_{ij}^{k} = -\alpha\frac{\partial E(X)}{\delta w_{ij}^{k}} \tag{4.2}$$

where w_{ij}^{k} is a weight of neuron j in layer k for node i and α is a learning rate of the network. The expansion of the delta rules can be found using backward propagation of errors, because the gradient information flows backwards through the network [4].

4.2 Prediction model

Reaching our goal to predict opponent's game state, we train a feedforward neural network. First, we must clarify what type of problem we are facing and what do we want the output values to represent. If we think of the neural network as a black box, the idea is to give it current game state situation and take back the game state in next three minutes. Consequently, input and output vector values are numbers representing number of units and buildings opponent has. In classification model, output vector represents categories and the mapping function predicts the class or category for a given observation. It follows, prediction model is a regression type where output values are represented as quantities. Moreover, because input and output data have linear dependencies, we can formalize the problem as multidimensional linear

regression. It follows, for linear regression problem, an ideal activation function should have linear relation, so we chose identity function as an activation function in our model.

4.2.1 Parsing replays

Training and test samples for our network are recorded game states during game with interval of three minutes. For training our network properly, we downloaded replays played by professional players from [18] and parsed them into vectors. For this purpose we had to create a special replay pseudo bot. Whenever a new unit is created pseudo bot saves two new game states (input and output game state) and adds current time plus three minutes into its time queue. Every next created unit in those three minutes is added to the output game state. The implemented time queue checks game time every frame and if it matches with time on the front of the queue, it saves the original input state and modified one, which contains all units as the input state plus units created in three minutes. Game states are saved into file using JSON format for easy machine parsing and human reading.

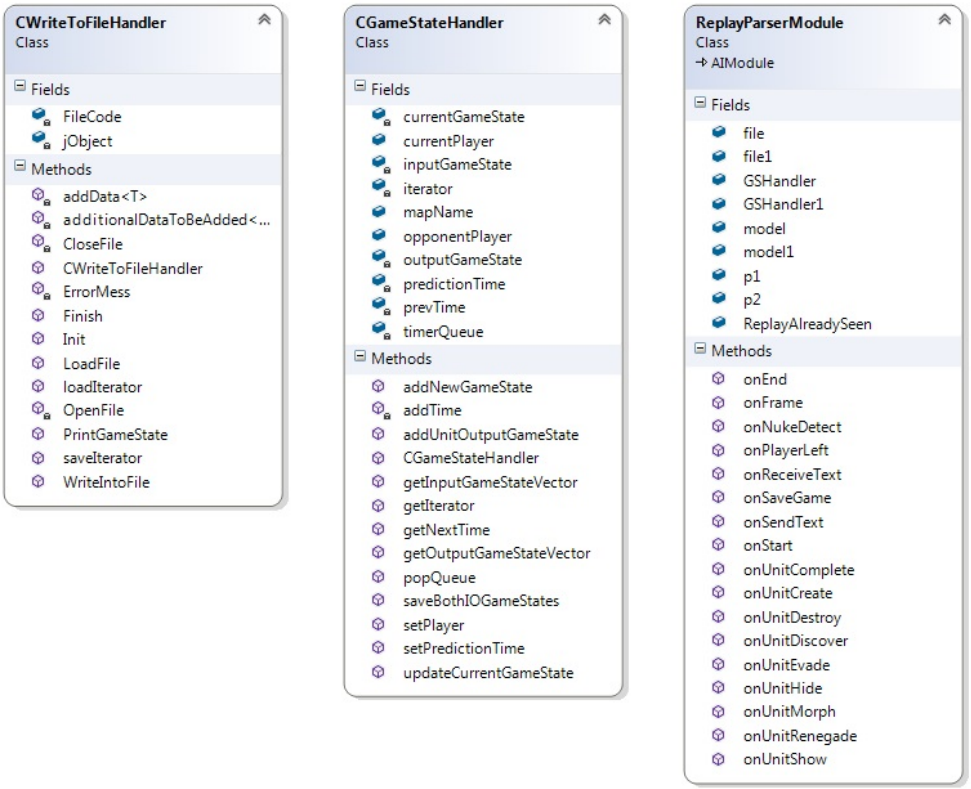

Figure 4.2: Class diagram of a replay pseudo bot

```
{
    "GameInfo": {
        "OpponentsNick": "gosu",
        "PlayersNick": "Drone._.",
        "Race": "Zerg",
        "RaceOpponent": "Terran"
    },
    "Prediction 00000": {
        "Input": {
            "SUPPLY": 10,
            "Zerg_Creep_Colony": 0,
            "Zerg_Defiler": 0,
            "Zerg_Defiler_Mound": 0,
            "Zerg_Devourer": 0,
            "Zerg_Drone": 5,
            "Zerg_Evolution_Chamber": 0,
            "Zerg_Extractor": 0,
            "Zerg_Greater_Spire": 0,
            "Zerg_Guardian": 0,
            "Zerg_Hatchery": 1,
            "Zerg_Hive": 0,
            "Zerg_Hydralisk": 0,
            "Zerg_Hydralisk_Den": 0,
            "Zerg_Lair": 0,
            "Zerg_Lurker": 0,
            "Zerg_Mutalisk": 0,
            "Zerg_Nydus_Canal": 0,
            "Zerg_Overlord": 1,
            "Zerg_Queen": 0,
            "Zerg_Queens_Nest": 0,
            "Zerg_Scourge": 0,
            "Zerg_Spawning_Pool": 0,
            "Zerg_Spire": 0,
            "Zerg_Spore_Colony": 0,
            "Zerg_Sunken_Colony": 0,
            "Zerg_Ultralisk": 0,
            "Zerg_Ultralisk_Cavern": 0,
            "Zerg_Zergling": 0
        },
        "Output": {
            "Zerg_Defiler": 0,
            "Zerg_Devourer": 0,
            "Zerg_Drone": 16,
            "Zerg_Extractor": 0,
            "Zerg_Guardian": 0,
            "Zerg_Hatchery": 3,
            "Zerg_Hive": 0,
            "Zerg_Hydralisk": 0,
            "Zerg_Lair": 0,
            "Zerg_Lurker": 0,
            "Zerg_Mutalisk": 0,
            "Zerg_Queen": 0,
            "Zerg_Scourge": 0,
            "Zerg_Spore_Colony": 0,
            "Zerg_Sunken_Colony": 0,
            "Zerg_Ultralisk": 0,
            "Zerg_Zergling": 0
        }
```


Figure 4.3: Example of parsed data using replay pseudo bot

4.2.2 Network topology and configuration

Choosing correct neural network topology is very difficult problem for AI programmers. Furthermore, one neural network does not cover different unit compositions of each race and naturally, cannot simulate prediction for all three races. Thus, we created a neural network for each race separately and designed them to fit their own race. Each network is a feedforward type, uses MSE as an error function with gradient descent backpropagation. The reason MSE was chosen is it uses quadratic cost, which is a smooth function. It turns out to be easier to figure out how to make small changes in the weights with smooth error function so as to get an improvement in the cost [19].

The remaining criteria we cannot simply chose is network topology. Vari-

ous hidden layers and number of neurons effect predicting accuracy of neural network. Thus, our goal is to find a configuration (number of layers and neurons in each layer), which provides the best results for test sample of data after learning the training samples. According to [20], recommended number of neurons in hidden layers to start with should be less than twice the size of the input layer. Based on this information, we tried 9 different topologies with three hidden layers and 11 topologies with four hidden layers. As input vector sizes are almost identical for all three races, we started with forty neurons and adjusted the numbers by adding or removing 5 - 10 neurons in hidden layers. For each newly created network, we measure its error on testing data set and compared it to previous networks. The measured error (the lower the better) indicates network's success rate. If the measured error starts rising compared to previous configurations, we stop the adding or removing neurons. Using this simple method, we will try various topologies and choose the one with the lowest measured error.

Another key parameter for our network is a learning rate. Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the loss gradient. The learning rate affects how quickly our model can converge to a local minima and achieve the best accuracy. According to [21], we can determine the learning rate by starting training our model with low learning rate and increase it at each iteration and plot it against loss. However, our training set contains shuffled replays of various games, where training data often causes the oscillation of measured error itself. In order to still find a good learning rate for our model, we decided to train the same network configurations with same training and testing data sets, but different learning rates. We started with relatively small learning rate of 0.00001 and multiplied it by 10 every next training round until the loss measured by MSE showed sign of divergence.

Chapter 5

Evaluation of prediction model

Using replay pseudo bot, we were able to parse 4549 replays with game time longer than three minutes. Every game was played one versus one, so it gives us two players to observe per game, resulting into 9098 files with parsed input and output game states. Every log file has different number of game states as it depends on replay length. In order to produce independent results and minimize the chance of overfitting, we split parsed replays randomly into two sets.

5.1 Training

The first data set is called the training set. Training set contains 8070 replay log files and serves for training our neural networks. There are 2362 protoss logs, 2592 terrans logs and 3116 zerg logs. After each batch, we record the MSE of the training sample. The problem of training is equivalent to the problem of minimizing the loss function. It is much easier to optimize the loss function using gradient descent than maximize the number of correct output directly. It follows, our network adjusts its weights based on measured error to perform better.

5.2 Testing

The second data set is called testing set. It has 1028 replay log files containing 237 zerg log files, 334 protoss log files and 457 terran log files. Its purpose is to validate the accuracy of previously trained neural network. We saves sum of MSE for all test samples and compare the results with other configurations. Based on the performance, we choose the best network configuration as a prediction model for our library.

Overall, we trained 153 neural networks with different topologies and configurations. All trained models and results are enclosed on the flash drive

B.

The best network configurations for three hidden layers are shown in table 5.1. All networks with three layers showed the best results using 0.001 learning rate.

Number of neurons in 3 hidden layers	Protoss error	Terran error	Zerg error
50 50 50	1,67111	5,91781	8,70496
30 30 30	1,53838	6,28287	9,49769
40 40 40	1,68253	6,17739	9,16184
40 50 40	1,63387	5,81438	9,03851
30 50 30	1,69244	6,3101	9,45717
25 30 25	1,59959	6,58285	10,2891
30 40 30	1,74851	6,27422	9,09962
35 40 35	1,56946	5,90594	8,78274

Figure 5.1: Measured error for networks with 3 hidden layers. Note: For better representation, errors displayed in the table are divided by 10^6

Same as three hidden layers configurations, four hidden layers configurations performed better with learning rate 0.001. According to our measured data in 5.2, some of four hidden layer topologies performed better than all of the three hidden layer topologies. However, both configurations showed that increasing the number of neurons over forty, results in worse performance.

Number of neurons in 4 hidden layers	Protoss error	Terran error	Zerg error
35 40 40 35	1,62020	6,03487	8,71140
40 45 45 40	1,64600	5,34001	8,70870
45 50 50 45	1,74722	6,06076	8,43229
50 50 50 50	1,71779	5,76442	8,72360
50 55 55 50	1,64371	5,81063	8,64568
30 35 35 30	1,51459	5,91581	10,51610
35 30 30 35	1,58321	6,09193	9,62786
25 30 30 25	1,56461	5,79727	9,41562
30 45 45 30	1,79096	6,23336	9,38979
40 60 60 40	1,61942	6,16456	8,56211
35 50 50 35	1,66032	5,78439	7,93665

Figure 5.2: Measured error for networks with 4 hidden layers. Note: For better representation, errors displayed in the table are divided by 10^6

Besides the error measurement, which might be abstract to imagine, we also stored result files for networks testing procedure. Result files contain all test game states with desired output values and the output values predicted by currently testing network.

```
Protoss_Probe                predicted = 62.3248     desired = 60
Protoss_Zealot               predicted = 13.4253     desired = 14
Protoss_Dragoon              predicted = 27.4681     desired = 28
Protoss_Shuttle              predicted = 1.45659     desired = 0
Protoss_Dark_Templar         predicted = 0.172292    desired = 0
Protoss_High_Templar         predicted = 0.795762    desired = 0
Protoss_Archon               predicted = 0.670085    desired = 0
Protoss_Dark_Archon          predicted = -0.0567189  desired = 0
Protoss_Scout                predicted = 0.0300036   desired = 0
Protoss_Corsair              predicted = -0.198952   desired = 0
Protoss_Arbiter              predicted = 0.175431    desired = 0
Protoss_Carrier              predicted = 0.398055    desired = 0
Protoss_Reaver               predicted = -0.0760468  desired = 0
Protoss_Observer             predicted = 4.50827     desired = 4
Protoss_Nexus                predicted = 4.4515      desired = 3
Protoss_Assimilator          predicted = 1.26902     desired = 3
Protoss_Gateway              predicted = 8.96274     desired = 10
Protoss_Shield_Battery       predicted = -0.0747435  desired = 0
Protoss_Robotics_Facility    predicted = 0.951641    desired = 1
Protoss_Photon_Cannon        predicted = 0.450791    desired = 1
Protoss_Stargate             predicted = 0.226534    desired = 0
```

Figure 5.3: Example of prediction recorded into result file. It shows prediction for protoss player.

5.3 Conclusion

The best fitting trained models according to our measurement are:

- Protoss: 4 hidden layers with neurons 30, 35, 35, 30 and learning rate 0.001.
- Terran: 4 hidden layers with neurons 40, 45, 45, 40 and learning rate 0.001.
- Zerg: 4 hidden layers with neurons 35, 50, 50, 35 and learning rate 0.001.

Even though error measurement of our models is nowhere near zero, the best performing networks does not provides inaccurate predictions. Having a closer look at 5.3, if we round all the decimals, provided prediction is very accurate. The numbers of units like dragoons and zealots can never fit perfectly. Starcraft is a very complex game and various army compositions can have same strength. Even though, the network provides a little different output in terms of units, it can satisfy our goal. However, networks predicting zerg game state have the worst predictions from all three races. Most probably it is caused by the different play style against other races. In our training samples, we focused only on one player and did not consider the opponent's race. In order to improve out model, we would have to create 9 neural networks. One for every possible match up.

Chapter 6

Usage

This chapter describes library usage, pseudo bot usage and testing methods used during implementation. This chapter provides a guide how to use our library and what are its benefits.

6.1 Integration with AI module

In order to integrate the library functionalities into AI module, programmer must include the OpponentModelling header file into his module and create an instance of **COpponentModelling** class. COpponentModelling contains similary named methods as BWAPI AI module class. In order to pass the game information into our library, programmer must call those specified methods using created COpponentModelling object. Those methods must be called in AI module class in virtual functions whose name corresponds with COpponentModelling methods as described in 6.1

```
virtual void onStart() {
    OppModellingObject->onStart(BWAPI::Broodwar->self(), ::Broodwar->enemy());
}
virtual void onFrame() {
    OppModellingObject->onFrame();
}
virtual void onUnitDiscover(BWAPI::Unit unit) {
    OppModellingObject->onUnitDiscover(unit);
}
virtual void onUnitDestroy(BWAPI::Unit unit) {
    OppModellingObject->onUnitDestroy(unit);
}
virtual void onUnitCreate(BWAPI::Unit unit) {
    OppModellingObject->onUnitCreate(unit);
}
virtual void onUnitMorph(BWAPI::Unit unit) {
    OppModellingObject->onUnitMorph(unit);
}
```

Figure 6.1: Example of calling corresponding methods for COpponentModelling object

If this procedure is followed correctly, COpponentModelling object can provide functionalities by calling its methods. For instance, **PredictOpponentGameState** method returns opponent's game state model in next three minutes predicted by implemented neural network. List of all methods provided by our library is described in added doxygen documentation.

6.2 Setting pseudo bot

Pseudo bot for parsing replays operates with four text files and one directory. At first, we describe input and output operations with files and then the configuration of pseudo bot itself.

- **ReplayLogs** is a directory containing all log files parsed by pseudo bot. Parsed data are saved there directly as text files if the location is not specified to a different folder.
- **Iterator.txt** This file contains a single digit number. It is a counter value in order to numerize parsed replays.
- **ReplayLogNames.txt** is a text file containing names of log files parsed by the bot.
- **ReplayOriginalNames.txt** is a text file containing names of all parsed replays. It helps to keep track about already parsed replays.
- **ConfigPseudoBot.h** is a configuration file defining paths to locations of previously mentioned files.

Pseudo bot output JSON file can be modified by adding or removing source code from **addData** method located in **CWriteToFileHandler** class. The method takes two vectors, where the first one describes key values for saved JSON and the second one their corresponding values. The default settings of this method can be seen in 6.2.

```
template <class T>
void CWriteToFileHandler::addData(std::vector<T> &KeyValue, std::vector<T> &insertValue, CGameStateHandler & GSHandler) {

    KeyValue.push_back(RACE);
    KeyValue.push_back(OPPRACE);
    KeyValue.push_back(PLAYER_NICK);
    KeyValue.push_back(OPPONENTS_NICK);

    insertValue.push_back(GSHandler.currentPlayer->getRace().getName());
    insertValue.push_back(GSHandler.opponentPlayer->getRace().getName());
    insertValue.push_back(GSHandler.currentPlayer->getName());
    insertValue.push_back(GSHandler.opponentPlayer->getName());
}
```

Figure 6.2: Default settings of addData function

6.3 Training custom neural network

For creating own neural network, we recommend to start a new project and separate it from AI modules. All programmer needs is a set of training log files and set of validating (test) log files. Creating custom neural network is relatively simple and can be done in few lines of code. **CNeuralNetworkHandler** is a class, which offers various functionalities to adjust the network construction and configuration. At first, a new CNeuralNetworkHandler object must be created. After that, using the object programmer can call function **createNN** with specified parameters for network topology. Then, call functions for setting parameters of the network such as number of epochs, learning rate or batch size and finally calling methods **trainNNWithDataSet** and **testNN** with given paths to log files, will start training the network and testing it using MSE as described in 6.3. Files with outputs and predicted values with measured error are saved in the OpponentModelling project in directory called **NeuralNetworkInfoFiles**.

```
// Create a neural network handler object
CNeuralNetworkHandler MyNeuralNetwork;

// Set number of epochs
MyNeuralNetwork.setEpochs(1);

// Set learning rate of neural networks
MyNeuralNetwork.setlearningRate(0.001);

// Create network consisting of numLayers hidden layers and vector of numbers of neurons in each layer
MyNeuralNetwork.createNN(numLayers, NumOfNeuronsVector);

// Train the network with data set
MyNeuralNetwork.trainNNWithDataSet(FileNamePath, FileWithLogNames);

// Test the network with test set
MyNeuralNetwork.testNN(TestDataSetPath, TestDataSetLogNames);
```

Figure 6.3: Example of creating own neural network

In order to adjust loss function or activation functions, programmer must modify the source code of **CNeuralNetworkModel**, specifically the **createNN** method. Input and output vectors for neural networks are located in **CPlayerModel** child classes. Those are static vectors that can be modified based on programmer needs. Neural network will then automatically create an input and output layer based on the vectors sizes.

6.4 Testing

For testing purposes, we created another project called **TestOpponentModelling**, which implements visual studio unit tests. Unit testing is a level of software testing where individual software components are tested. The purpose is to validate that each unit of the software performs as designed. It

usually has a few inputs and a single output [22]. Implemented unit tests validate components of our library for essential things such as adding new units into the database or removing destroyed units.

```
TEST_METHOD(RemovingUnits)
{
    COpponentModelling OppMod;
    CObjectInfo *UnitTmpArchon[5];

    for (int i = 0; i < 5; i++)
    {
        UnitTmpArchon[i] = new CObjectInfo();
        UnitTmpArchon[i]->ID = i;
        UnitTmpArchon[i]->UnitType = BWAPI::UnitTypes::Protoss_Archon;
        UnitTmpArchon[i]->Health = 10;
        UnitTmpArchon[i]->Shields = 350;
        OppMod.getOpponentModel()->addUnit(UnitTmpArchon[i]);
    }

    Assert::AreEqual((int)OppMod.getOpponentModel()->getMapOfAllUnits()[BWAPI::UnitTypes::Protoss_Archon].size(), 5);
    Assert::AreEqual((int)OppMod.getOpponentModel()->getAllUnits().size(), 5);
    OppMod.getOpponentModel()->removeUnit(1);
    OppMod.getOpponentModel()->removeUnit(2);
    Assert::AreEqual((int)OppMod.getOpponentModel()->getAllUnits().size(), 3);
    Assert::AreEqual((int)OppMod.getOpponentModel()->getMapOfAllUnits()[BWAPI::UnitTypes::Protoss_Archon].size(), 3);

    Assert::AreEqual((int)OppMod.getOpponentModel()->getAllUnits().at(0)->ID, 0);
    Assert::AreEqual((int)OppMod.getOpponentModel()->getAllUnits().at(1)->ID, 3);
    Assert::AreEqual((int)OppMod.getOpponentModel()->getAllUnits().at(2)->ID, 4);

    Assert::AreEqual((int)OppMod.getOpponentModel()->getMapOfAllUnits()[BWAPI::UnitTypes::Protoss_Archon].at(0)->ID, 0);
    Assert::AreEqual((int)OppMod.getOpponentModel()->getMapOfAllUnits()[BWAPI::UnitTypes::Protoss_Archon].at(1)->ID, 3);
    Assert::AreEqual((int)OppMod.getOpponentModel()->getMapOfAllUnits()[BWAPI::UnitTypes::Protoss_Archon].at(2)->ID, 4);

}
```

Figure 6.4: Example of a testing method for removing units

Conclusion

The aim of this thesis was to create a tool allowing AI modules playing computer game Starcraft to model their opponent, automatically infer additional information and predict future situations. Library was successfully implemented and tested with all requirements. Unit database saves all opponent's units with inferred data, built on current data our trained neural network predicts game state in near future and the library design and implementation allows programmers to use replay pseudo bot by themselves to parse replays of their rivals and create and train their own neural network.

As part of future work, there is an idea to extend the neural networks and create 9 networks overall, one for every race match up in the game. Unfortunately, such extension requires not only three times more computational time, but also three times more evaluating time for choosing the best fitting network model in every match up.

Last but not least, I would like to point out that this thesis was based on a computer game, which shows computer games are not just source of fun in today's society, but can also serve as an important model mechanisms to support various parts of science.

Bibliography

[1] Brood War Terrain Analyzer. [online]. Accessed 02.05.2018. Available from: https://code.google.com/archive/p/bwta/

[2] Brood War Easy Map. [online]. Accessed 02.05.2018. Available from: http://bwem.sourceforge.net/

[3] Liquipedia, Technology tech tree. [online], accessed 10.04.2018. Available from: http://liquipedia.net/starcraft/Technology_tree

[4] McGonagle, J. Feedforward neural networks. *Brilliant*, accessed 24.04.2018. Available from: https://brilliant.org/wiki/feedforward-neural-networks/

[5] M. Čertický, D. C. The Current State of StarCraft AI Competitions and Bots. [online], accessed 10.04.2018. Available from: https://sscaitournament.com

[6] The Brood War API BWAPI. [online], accessed 10.04.2018. Available from: https://github.com/bwapi/bwapi

[7] BWAPI::UnitInterface Class Reference. [online], accessed 10.04.2018. Available from: https://bwapi.github.io/class_b_w_a_p_i_1_1_unit_interface.html

[8] BWMirror API - An API for SC:Broodwar AIs. Accessed 02.05.2018. Available from: https://github.com/vjurenka/BWMirror

[9] Java interface for the Brood War API. Accessed 13.04.2018. Available from: https://github.com/JNIBWAPI/JNIBWAPI

[10] Churchill, D. UAlbertaBot. Accessed 26.04.2018. Available from: https://github.com/davechurchill/ualbertabot/

[11] Corporation, M. Microsoft Visual Studio. Accessed 10.05.2018. Available from: https://www.visualstudio.com/

[12] Lohmann, N. JavaScript Object Notation for Modern C++. [online], accessed 27.04.2018. Available from: https://www.json.org/

[13] Implementation of deep learning. [online], accessed 25.04.2018. Available from: https://github.com/tiny-dnn/tiny-dnn

[14] ReversibleContainer. [online], accessed 06.05.2018. Available from: http://en.cppreference.com/w/cpp/concept/ReversibleContainer

[15] JavaScript Object Notation. [online], accessed 30.04.2018. Available from: https://www.json.org/

[16] Hardesty, L. Explained: Neural networks. *MIT News Office*, accessed 25.04.2018. Available from: http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414

[17] Brownlee, J. Difference Between Classification and Regression in Machine Learning. *Machine learning mastery*, accessed 28.04.2018. Available from: https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/

[18] Starcraft brood war replay packs. *Team Liquid*, accessed 02.05.2018. Available from: http://www.teamliquid.net/forum/brood-war/310883-replays

[19] Gupta, T. Deep learning feedforward neural network. *Towards Data Science*, 2017, accessed 19.04.2018. Available from: https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7

[20] Heaton, J. *Introduction to Neural Networks with Java*. Heaton Research, Inc, 2008, ISBN 1-60439-008-5, accessed 18.04.2018.

[21] Zulkifli, H. Understanding Learning Rates and How It Improves Performance in Deep Learning. *Software testing fundalmentals*, 2018, accessed 12.05.2018. Available from: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10

[22] Unit Testing. *Software testing fundalmentals*, 2017, accessed 13.05.2018. Available from: http://softwaretestingfundamentals.com/unit-testing/

Appendix A

Acronyms

BWAPI Brood War Application Programming Interface

RTS Real Time Strategy

AI Artificial Intelligence

STL Standard Template Library

JSON JavaScript Object Notation

IoT Internet of Things

BWME Brood War Easy Map

BWTA Brood War Terrain Analyzer

MSE Mean Squared Error

Appendix B

Contents of enclosed flash disk

- `readme.txt` ..... the file with flash disk contents description
- `T.Bohuslav_thesis.pdf` ..... the thesis text in PDF format
- `projects` ..... the directory of thesis projects
  - `doc` ..... the directory with technical documentation (doxygen)
  - `OpponentModeling` ..... the directory with library implementation and trained models of neural networks
  - `PseudoBotForParsingReplays` . the directory with pseudo bot project
  - `TestOpponentModelling` .... the directory with project for testing the library
  - `doxygen_config_file` ..... the doxygen configuration file
  - `Makefile` ..... the Makefile for compilation