BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

COMBAT MANAGEMENT IN STARCRAFT II GAME BY MEANS OF ARTIFICIAL INTELLIGENCE

ŘÍZENÍ BOJE VE HŘE STARCRAFT II POMOCÍ UMĚLÉ INTELIGENCE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **KAREL KRAJÍČEK**
AUTOR PRÁCE

SUPERVISOR **doc. RNDr., Ph.D. PAVEL SMRŽ**
VEDOUCÍ PRÁCE

BRNO 2018

Abstract

This thesis focuses on the use of Artificial Intelligence and design of working module in Real-Time Strategy (RTS) game, StarCraft II. The proposed solution uses Neural Network and Q-learning for combat management. For implementation, the StarCraft 2 Learning Environment has been used as a means of communication between the designed system and the game. Evaluation of the system is based on its ability to make progress over time.

Abstrakt

Táto práca sa zaoberá využitím umelej inteligencie a návrh funkčného modulu pre strategickú hru StarCraft II. Riešenie využíva neurónové siete a Q-learning pre boj. Pre implementáciu systému a jej prepojenie s hrou StarCraft používam StarCraft 2 Learning Environment. Vyhodnotenie systému je založené na jej schopnosti vykonať pokrok.

Keywords

artificial intelligence, machine learning, Real-Time Strategy games, StarCraft II, SC2LE, PySC2, Q-learning, artificial neural network

Klíčová slova

Umělá inteligence, strojové učení, strategické hry v reálném čase, StarCraft II, SC2LE, PySC2, Q-learning, umělá neuronová síť

Reference

KRAJÍČEK, Karel. *Combat Management in StarCraft II Game by means of Artificial Intelligence*. Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. RNDr. Pavel Smrž Ph.D.

Combat Management in StarCraft II Game by Means of Artificial Intelligence

Declaration

I hereby declare that this bachelor's thesis was prepared as an original author's work under the supervision of doc. RNDr., Ph.D. Pavel Smrž. All the relevant information sources which were used during the preparation of this thesis are properly cited and included in the list of references.

……………………
Karel Krajíček
17.5.2018

Acknowledgements

I would like to thank my, Professor Pavel Smrž, for his guidance during the creation of this work.

Extended abstract

Hlavná úloha tejto práce je design a implementácia systému ktorý používa strojové učenie pre RTS hru StarCraft II. Hlavná časť RTS na ktorú sa sústredím v tejto práci je použitie umelej inteligencie je boj a integrácia do celkovej štruktúry tohoto systému.

Navrhnuté riešenie je vytvorené v programovacom jayzku Python. Hlavný dôvod je ten aby som mohol využiť StarCraft 2 Learning Environment (SC2LE) krorý používa modul PySC2, ktorý je vytvorený pre využitie AI pre StarCraft II.

Techniky strojového učenia ktoré som použil pre tento projekt, sú Q-learning a neurónové siete. Aplikácia Neurónovej sieťe ktorú som použil pre tento modul je MLPClassifier z Pythonovskej knižnice scikit-learn.

Behom projektu som vykonal niekolko experimentov použitím rôznych nastavení Q-learningu a neurónovej siete ako aj odlišné nastavenia interpretacie stavu prostredia a "odmien".

StarCraft II má 3 odlišné rasy Terran, Zerg, a Protoss ktoré majú úplne odlišné jednotky, budovy a používajú velmi odlišné stratégie a preto som pridal všetky 3 rasy ale iba rasa Terran je plne implementovaná. To znamená že strojové učenie pre boj je implementované iba pre rasu Tearran.

Okrem strojového učenia som vytvoril celý modul ktorý organizuje vytváranie základní, trénovanie jednotiek a posielanie jednotiek odhalovať hernú mapu.

Okrem originálneho plánu som aj implementoval použitie strojového učenia pre ténovanie jednotiek, ale táto časť nebola poriadne otestovaná.

Systém ukazuje pokrok jako je ukázané v experimentech a je schopný poraziť defaultnáho bota v StarCrafte vytvořeného Blizzardom.

Aby som toho dosiahol, musel som sa naučiť jako pracovať s SC2LE a naučiť sa používať Neurónové siete a Q-learning a jako iuch implementovať do modulu.

Z experimentov som vyvodil záver že Neurónová sieť je ovela efektívnejšia jako Q-learning pre bojové scenária.
Moje riešenie nie je konečné, a je tu toho dosť čo sa dá vylepšiť, například implementácia ostatních aspektov hry pre strojové učenie. Časť ktorú určite mžem vylepšiť je plná implementáciapre rasy Zerg a Protoss. Téma umelej inteligencie pre RTS ponúka velkú škálu možností pre vývoj.

Contents

1 Introduction

The main objective of this thesis is the design and implementation of a system that uses Machine Learning techniques to play the Real-Time Strategy game StarCraft II.
The main part of Real-Time Strategy that I focus in this thesis is the use of artificial intelligence for combat scenarios and integration of this use into the whole structure of the system.

The proposed solution is designed in the programming language Python. The reason why I decided to use this language was so that I could use StarCraft 2 Learning Environment (SC2LE) [5] [10] that uses module PySC2 [24] that is perfectly suited for AI research in StarCraft II.

Machine learning techniques that I chose for this project are Q-learning and Neural Networks. The Neural Network I used for this Project is MLPClassifier from the Python library scikit-learn [27].

Throughout this project, I made several experiments using different settings of Q-learning and Neural Networks as well as different settings of "states" and "rewards". These experiments are shown later in this thesis.

StarCraft II has 3 distinct races, Terran, Zerg, and Protoss, that have completely different types of units, buildings and uses very different types of strategies. Because of that, I implemented all 3 races, but only race Terran is fully developed. That means that the Machine learning in combat is implemented only for Terran and the type of strategy Terran units uses.

Besides Machine learning, I have developed an entire module that organizes building bases, training units and scouting of environment.

Beyond the original plan, I have also implemented the use of Machine Learning for the training of combat units, but this part was not so well tested.

1.1 Motivation and Goals

In the last decade, AI has become very popular and plays an important role in our daily lives. From something as simple as adding suggestions and classification of pictures on the internet to their use in air travel and voice recognition.
I want to contribute to the development of the field of Artificial Intelligence and for that, I have used the environment of a Real-Time Strategy game StarCraft II that contributes with many interesting challenges and opportunities for the development of AI. [5]
My thesis is a response to the Call for AI Research in RTS Games from Michael Buro [8] who is

introducing many problems and challenges that RTS game genre brings into the AI research and need for the improvement of the game bots.
Working on this thesis has also helped me understand the use and principles of various machine learning techniques, such as Artificial Neural Networks and Q-learning.

Goals:

Learn how to use Machine Learning techniques in Real-Time Strategy games and representation of game states and encoding them for learning algorithms.
-Learn about Q-Learning
-Learn about Artificial Neural Networks
-Make a research about the use of Machine Learning in RTS games

Review the rules, conditions, and limitations of the environment defined within the Starcraft II interface.
-Asses CommandCenter [11]
-Asses PySC2 [10]

Using existing implementations create a basic module for building bases, creation of units and their use for combat and scouting.
-Find tutorial or example, that could help me build my own module [11] [35]
-Divide the system into sub-functions, where each sub-function will be using different aspect of the game

Design and implement system that will be able to learn from the game and suggest the best strategies in order to maximize the chance of winning.

Test the created system and compare it to other systems or human players.
-Use testing to find the best optimization of reinforcement learning technique

2 Environment

In this chapter, I will be explaining about the environment that is being used for my system, i.e. RTS game StarCraft II and the libraries that I will need to use, in order to make my system run on StarCraft II.

2.1 Real-Time Strategy Games

Real-Time Strategy or in short RTS is a subgenre of strategy games and is widely played in the modern game world. [1]

The main concept of RTS games is that they are played in real-time, meaning the state of the game is constantly changing without waiting for an action of the player, unlike turn-based strategy games where the state of the game is waiting for the player's response.

Later in this thesis, we will find out that this definition is not so well founded because the learning environment that is used for this project is dividing the game into steps and returning an action for each step, which is rather closer to turn-based strategy games than RTS.

The important part of the RTS is how players use different tactics and strategies by making a set of moves/actions in order to achieve a goal or to improve the strategy status of the player.

The player uses units under their control to achieve goals. Most of the time resources such as food, gold or minerals are required to create the units, and the resources must be obtained during the game, which eventually becomes a goal by itself.

RTS games started to become popular in the 90's but Sega Genesis' Herzog Zwei from 1989 can be considered as the first RTS game [2]. They are interesting for AI research as described in Call for AI Research in RTS Games from Michael Buro [8], mainly for all the problems that agents have to encounter, in order to win the game like real-time planning, predicting of opponent moves, space and time evaluation, resource management, and co-operation.

Turn-based strategy (TBS) is a game genre where players take turns, meaning, they are waiting for a move of other players, based on the rules of the game. [3]

First-person is a game genre where players control only one unit, and the view of the player is centered around the view of that unit. Advanced 3D graphics is usually required for FP games.

RTS	**TBS**	**First-person**
moves are happening simultaneously, in real-time	players are making moves in order, waiting for each other	moves are happening simultaneously, in real-time
time for making a decision about the move is really short, sometimes it's only a few milliseconds	players have limited time for making move as well, but it's much longer, in case of Chess, it can be several minutes or hours.	time for making a decision about the move is really short, sometimes it's only a few milliseconds
game tree size is theoretically infinite.	game tree size is calculated by all the possible move, the players can make.	game tree size is theoretically infinite.
most RTS games are non-deterministic, because of randomness set in games, for example, range of damage	most are deterministic, like Chess, but there are non-deterministic games like Ludo, which is caused by a throw of the dice	some games have randomness in them, which can cause non-determinism, but otherwise, they are deterministic
strategy - control of multiple units	strategy - control of multiple units	control of only one unit, the first person shooter himself
limitation of view - fog of war hides parts of the map, where are no units of the player, therefore, player must scout the map, to gather intelligence about environment and enemy	in case of board games like chess no limitation of view, the player can see everything that's happening on the board, but in case of computer games like Heroes of Might and Magic or Total War series, there is fog of war	limitation of view - player sees the only point of view of his character

Table 1: Comparison of Real-Time Strategy against Turn-based strategy and First-person

There is a case of interesting transition from RTS to First-person in game series Warcraft, where all Warcraft games, until Warcraft III Frozen Throne are RTS based, but with exception bonus campaign, The Founding of Durotar which have more elements of First-person than RTS and then its followed by Creation of World of Warcraft, famous MMORPG , which fits of all our criteria of First-person.

No matter how complicated RTS games are, they will always allow the great complexity of strategy and tactics to arise, due to their nature, as is described in Table 1. If we want to successfully manage this complex environment, we need to divide this complex problem into smaller, manageable sub-problems. This solution is greatly described in work of Professor David Churchill, Heuristic Search Techniques for Real-Time Strategy games [4] who takes the main idea of categories Strategy, Tactics, and Reactive Control, from literatures of AI research [21] and military command [22]. In his thesis, Professor David Churchill describes the division of this problem based on the time scale and the level of abstraction and divided it into categories Strategy, Tactics, and Reactive Control as shown in Figure 2.

The strategy is the highest level of a decision-making process, which dictates what happens on a global scale and which part of the game, military or economic one, the player should focus on. Tactics decide what happens on the smaller scale, like which type of building to build or managing local battles.
Reactive Control then works with individual units and executes actions of the game. These categories imitate a military command hierarchy, in terms of chain of command as well as information processing.

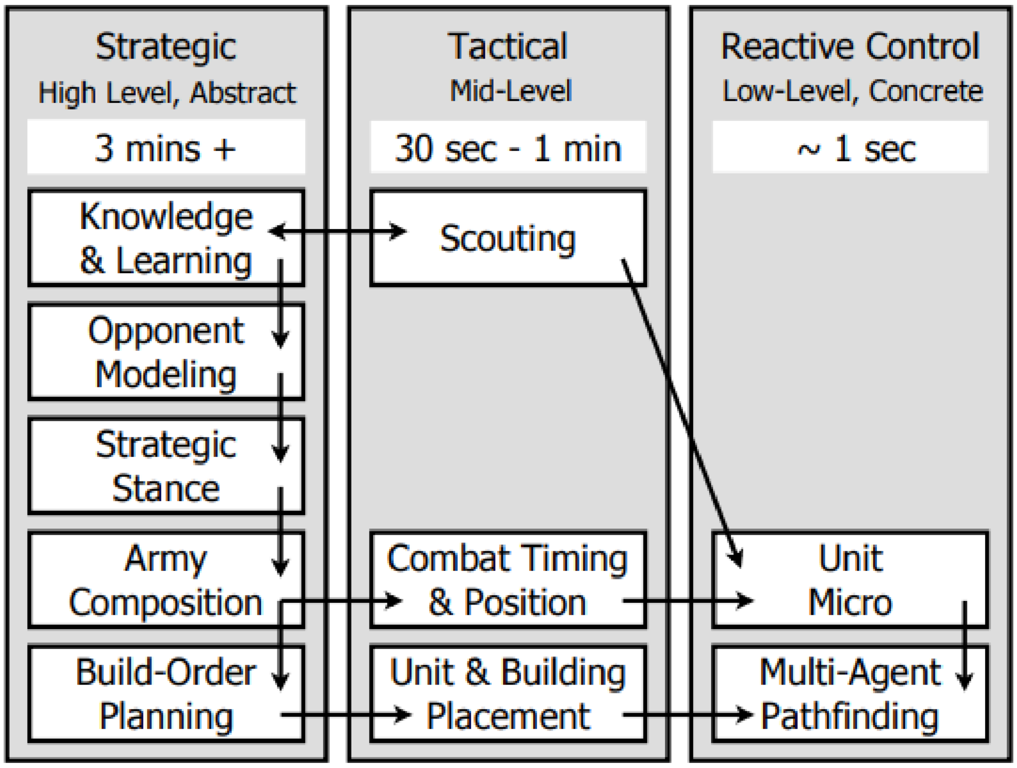


Figure 2: Division of sub-problems in RTS, copied from [4] .

2.2 StarCraft II

StarCraft II is a successful science fiction RTS game created by Chris Metzen and James Phinney and owned by Blizzard Entertainment, and released in July 2010. [5] [6]

The StarCraft franchise has been popular with players competing in tournaments for more than 20 years.

StarCraft also makes an ideal environment for AI research due to its rich, multi-layered gameplay. The StarCraft games have been used by Artificial Intelligence and Machine Learning researchers, who annually compete in the Artificial Intelligence and Interactive Digital Entertainment (AIIDE) StarCraft AI Competition. [7]

Researchers are also intrigued by the large pool of keen players that daily compete online. The game also has other qualities that appeal to researchers, such as the large pool of avid players that compete online every day. This confirms that there is a considerable amount of replay data to learn from.

The main aim is to beat the opponent and in order to achieve that, the player must also attain sub-goals like build a base (buildings and units), manage resources (Minerals, Vespene Gas, and Supplies), build an army (combat units), and defeat the enemy.

Supplies are the special type of resources, where the amount of supplies is produced by building like Command Center and Supply Depot, but the produces amount just sets the supply capacity and every generated unit uses part of this capacity. When the supply capacity is fully used, it is not possible to produce more units until more supply capacity is created, or some supplies are being set free by losing some unit.

The player controls their base and army from a third person perspective. StarCraft has three different races—Terran, Protoss, and Zerg, which all have distinct units and strategies.

One game/match can last from a few minutes to hours to finish. Players must manage their strategy carefully because actions taken early in the game can have lasting consequences for the rest of the match. The map is only partially observed hence, programmed agents must use a long-term memory to succeed.

As in all RTS games, StarCraft II also have fog of war, and because of that players must send their units to scout to react to their opponents' strategy. [9]

Another great challenge the StarCraft II presents is huge action space that it provides, with more than 500 basic actions that can be taken [5] [10] [23], (523 in this version [23]) with many of them requiring a position on the screen or minimap as an additional argument. Assuming a minimap size of 64x64 and screen size 84x84 there are roughly hundreds of millions of possible actions. [5]

2.3 StarCraft II Learning Environment

At first, I wanted to use CommandCenter [11], program written in C++ using BWAPI and Blizzard's StarCraft II AI API, written by David Churchill Professor of Computer Science at Memorial University of Newfoundland. After several months of efforts to make CommandCenter work, Professor David Churchill explained to me that CommandCenter is not developed enough for using machine learning techniques, and advised me to use PySC2 [24] instead, which was created for this very purpose. Therefore, I'm using PySC2 for this thesis.

StarCraft 2 Learning Environment or for short, SC2LE exposes game StarCraft II as a research environment. [10] SC2LE provides tools for agent that are identical to the way human players play the game. That means, it's not like a typical bot that can use micromanagement and let every unit control its behavior, but the agent must select one unit or group of units that are visible on the screen and use one action per step as human players would. The agent does not see parts of the map that are covered in fog of war. This approach in result makes agents using this environment weaker, but also it makes them perfect tool for developing artificial intelligence since we want to develop AI that works in the same environment as we humans, and they have to follow same rules as we humans do.

SC2LE consists of three modules: Linux StarCraft II binary, StarCraft II API, and PySC2. The StarCraft II API [25] is an interface that provides full external control of StarCraft II. The API can be used to start a game, get observations and take actions. This API works on Windows and Mac OS. It directly does not support Linux, but instead, it provides a limited headless build for Linux.

This API then builds PySC2, which is a Python open source environment that wraps the StarCraft II API to enable the interaction between StarCraft II and an Agent.
PySC2 defines the action space and provides all the observation for an Agent.

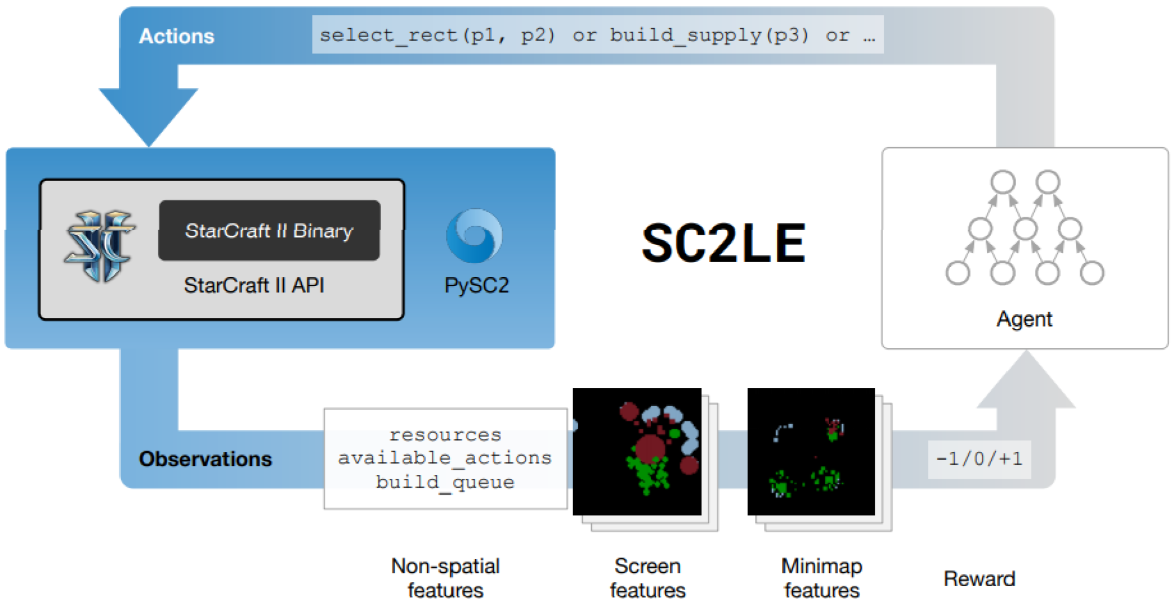


Figure 3: SC2LE, and its components connected with agent, copied from [10].

The core part of PySC2 is class base_agent from which the created agent inherits important methods from whose most important one is method step, where the agent defines everything it will do in one step of the game.

PySC2 gives us two different reward structures. [9] [10] One informing us about the absolute result of the game 1 for victory, 0 for a tie, and −1 for loss. Another is the Blizzard score, which is the sum of the values of killed and created units and buildings and all other factors, but most importantly, it is correlated with final result of the game, for example, more enemy units are killed, the higher chance for victory. While human players can see this score at the end of the game SC2LE provides the option to access the Blizzard score during the game, so the change in the score can be used as a reward for reinforcement learning. The Blizzard score is not zero-sum because it's player-centric

The important part of observations is sets of feature layers with resolution of N x M pixels, with default settings being 64 x 64 pixels for minimap and 84 x 84 pixels for the screen. Each layer provides us with important information about the state of a game. Graphical display of the layers can be seen in Figure 4.

The minimap is a representation of the entire map, and it contains 7 different layers.
Height Map that shows the terrain levels, Visibility layer shows which parts of the map are hidden by the fog of war, parts of the map have been explored but are not visible anymore or parts of the map are visible at the current moment. Creep layer shows land that is covered in zerg creep, Camera layer displays the part of minimap which can currently be seen on the screen.
Player ID layer displays the units with the IDs of the player that owns them.
Player Relative layer shows whether units on the minimap are 1 - belongs to the player, 2 - friendly, 3 - neutral or 4 - enemy. Layer Selected displays which units are selected at the moment.

The screen is a representation of what the player can currently see on the main screen of the game, and it contains 13 different layers.

Height Map has the same purpose as Height Map for minimap that shows the terrain levels, but only for what is currently visible on the screen. Visibility layer same as in minimap shows which parts of the map are hidden by the fog of war and Creep layer and layer Power displays parts of the map that have Protoss power generated by Protoss buildings of the player.
Player ID layer is the same as for minimap, layer Player Relative, and layer Unit Type that displays the IDs of the individual units. Layer Selected shows selected units, layer Hit Points gives information about the hit points/health of individual units and layer Energy shows the amount of energy of the units. Layer Shields shows how much shields the Protoss unit has, layer Unit Density shows how many units are being present in one specific pixel and layer Unit Density AA with show you the same, but its anti-aliased.

Besides screen and minimap, the PySC2 provides additional important nonspatial information, like the amount of Minerals and Vespene Gas, amount of used supplies, supply capacity, list of actions currently available, which is mostly dependent on selected unit and information about selected units.

The big difference when it comes to the rendering of the screen between PySC2 and the screen which human players see is that [10] , the view for human players is in a 3D perspective with high resolution and it's viewed from an angle different to that of PySC2 that renders screen by using top-down orthographic projection. This makes all units to be shown as the same size no matter where they are in view, meaning that every pixel in a feature layer corresponds to the position and size of the rendered unit. Because of that, human players see more in the back of the screen and less in the front, with the consequence for bad representation of actions in replays made by human players.

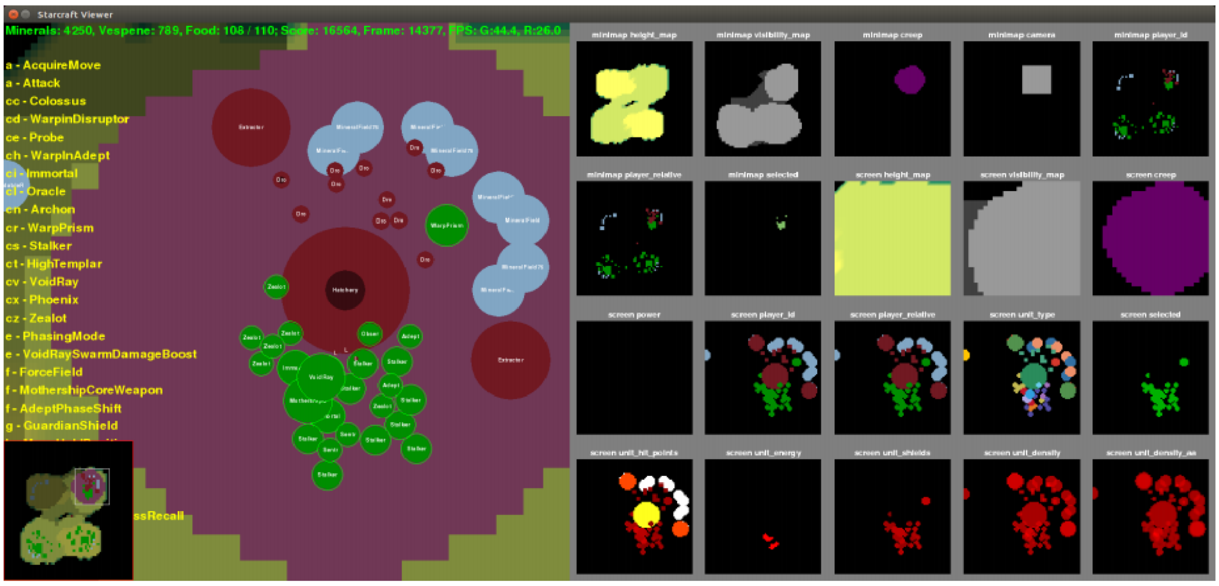


Figure 4: Layers of minimap and screen being displayed, copied from [10]. On the right we can see the list of possible actions in current moment.

The basic moves taken by human players are mostly a combination of actions. For example, in order to build a building on the map, the player must first select the option “Build” or “Advanced Build”, select the desired building, maneuver the cursor to a point on the screen where the building is to be built, and finally click on the selected position. Instead of taking 3 different actions to make this one

move, PySC2 gives us an atomic compound action in a format: `actions.FunctionCall(command_build, [_BUILD_COMMANDCENTER, [ x , y ]])`
PySC2 provides estimated 300 action-function identifiers with 13 possible types of arguments. Not all the actions are available in every state of a game. For instance, the build building command is only available if a worker like SCV is selected.

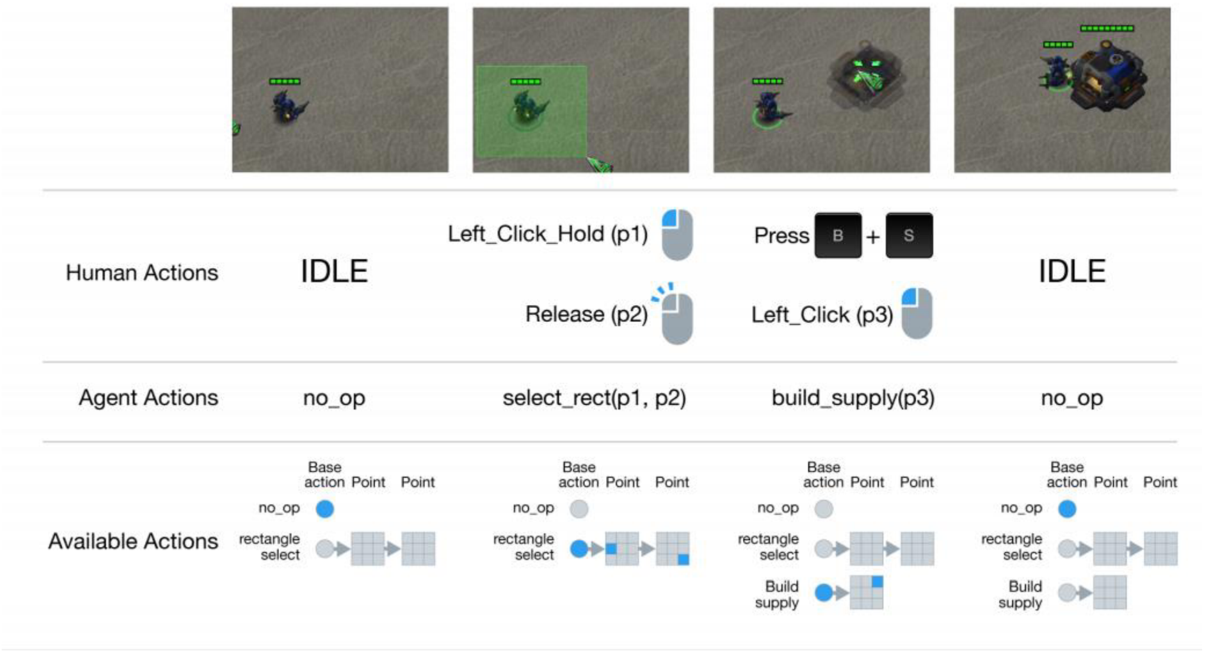


Figure 5: Comparison between human players actions and the PySC2 actions., copied from [10].

3 Machine Learning

Machine Learning is the ability of computers to create the desired function only from input data and feedback of its performance, which are commonly referred to as “punishment” and “reward”. It might use statistical techniques and improve its performance over time. All that without being specifically programed to do the desired task.

Machine learning is being used for solving problems where programming specific algorithms are not feasible or efficient. [12]

Markov Decision process is a framework that helps make decisions in a stochastic environment, an environment where outcomes are partly random and partly under the control of an agent.
The goal of the process is to find a policy, which provides the agent with all optimal actions in each state of the environment.

The process does not require to know past states for making a decision, it only requires present state and reward.

Markov Decision process is more efficient than simple planning because the process will find optimal action even if some unexpected change, change the plan, but simple planning keeps following the plan after finding the best strategy.

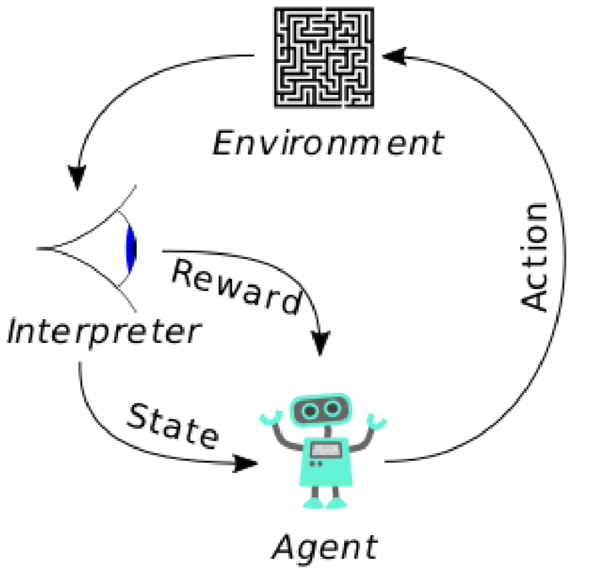


Figure 6: Relationship between Agent, Interpreter and Environment in Reinforcement learning. Picture taken from [13] .

3.1 Q-Learning

Q-Learning is a model-free reinforcement learning technique, which allows agents to learn how to act optimally in controlled Markovian domains. [14]

Model-free means that Q-learning has no idea about the rules of the environment it's applied to, meaning it does not have a model of the environment.

Q-learning creates a table which contains immediate rewards for every action that was taken, in every state of the environment the agent happened to be in.

Let's have some definitions:
S is set of states
s is state $s \in S$
A is set of actions
a is action $a \in A$
R is set of possible rewards
r is reward $r \in R$
Q is Q-learning table where

$Q : S \times A \rightarrow R$
Q(s,a) is record in Q-learning table in state s with action a
α is learning rate, $0 < \alpha \leq 1$
γ is reward decay, $0 \leq \gamma \leq 1$
If γ is lower than 1, it gives the learning process the ability to value the steps(ones that achieved the same reward sooner) more.
ε is threshold , $0 \leq \varepsilon \leq 1$
Threshold will tell us, what the probability of the agent acting based on Q-learning and 1- ε is the probability of the agent choosing action randomly. Threshold is not needed for Q-learning, but it can help us to gather as much data as we need.
max(Q , s) is a function that finds an action with the highest reward for the state s, in the Q-learning table Q

So every time the agent takes an action and enters a new state, the Q-learning table gets updated. Update of record in Q-learning table is calculated as: [15]
$Q (s_t , a_t) = (1 - \alpha) * Q (s_t , a_t) + \alpha * (r_t + \gamma * Q (s_{t+1} , \max(Q , s_{t+1}))))$
but I also encountered equations like: [16]
$Q (s_t , a_t) = Q (s_t , a_t) + \alpha * (r_t + \gamma * Q (s_{t+1} , \max(Q , s_{t+1}))) - \ Q (s_t , a_t))$
which is basically the same equation.
Exception is, when state s_t is final, in that case the update is just calculated as:
$Q (s_t , a_t) = r_t$

By following these principles, after many steps the Q-learning table gets sensible values through all the states that's been encountered, from the states that receive rewards, all the way to initial state. [14] Example on the picture Figure 7.
Down side of this is, that the agent has to go through the same environment many times, until the Q-learning table gets updated all, and that amount is proportional (correlated) to the size of the table.
-Another problem is when space is not exactly Markov domain

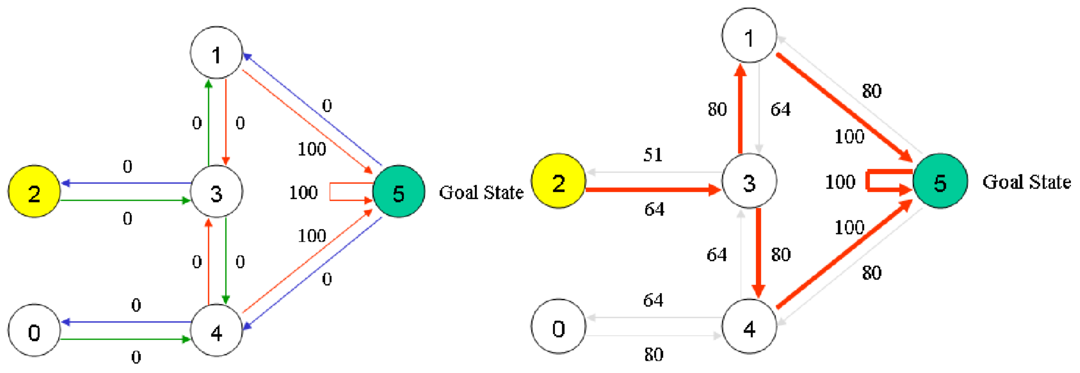


Figure 7: States of Q-learning. Left - Found goal state, Right - Found path between initial state and goal state. Picture taken from [17] .

Q-learning eventually finds an optimal set of steps for any finite Markov decision process. [18]

3.2 Neural Networks

The artificial neural network is an information processing system which contains a substantial amount of interconnected processing neurons which is inspired by the way biological nervous systems work. The neural network learns from input information and expected the result to optimize its process. [19]

Learning in biological neural networks involves adaptations to the synaptic connections that exist between the neurons. An artificial neural network works the same way. It is a setup for a particular task, like data classification or pattern recognition. [20]

Artificial neural networks come across as a recent phenomenon, however, they were invented before the rise of computers.

“Many important advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few researchers. These pioneers were able to develop convincing technology which surpassed the limitations identified by Minsky and Papert. Minsky and Papert published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most without further analysis. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding. The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pits. But the technology available at that time did not allow them to do too much” [20]

Neural networks have a great ability to find patterns and trends in a large amount of data. The finding of these patterns can be too difficult for humans, therefore, a trained network can become an expert in the field it’s trained in.
Neural networks can be used for various purposes, such as:
Adaptive learning, Self-Organisation, Real-Time Operation, Fault Tolerance via Redundant Information Coding. [20]

A huge advantage of neural networks is their contrast to classical algorithmic programs. Usually, when we need a computer to do some work, we need to program every step and every instruction of what needs to be done. Therefore, we need to know how to do the work, we want from program to do, and then we can program it. But neural networks can deal with problems, we don’t know solutions to. For example, it’s really hard to find an algorithm that would be able to distinguish faces of different people, but neural networks seem to be good at this job. [20]

The function of a neuron in the artificial neural network is based on a biological neuron. One neuron has multiple inputs, but only single output.

Let's have some definitions:

X is set of inputs into neuron
x is single input $x \in X$
W is set of weights of neuron
w is single weight $w \in W$
b is bias
Activation function: Sigmoid function $\sigma(x) = 1 / (1 + e^{-x})$
Purpose of sigmoid function is to take any real number between $-\infty$ and $+\infty$ and turn it into a number between 0 and 1 that is positively correlated to the input of the sigmoid function.

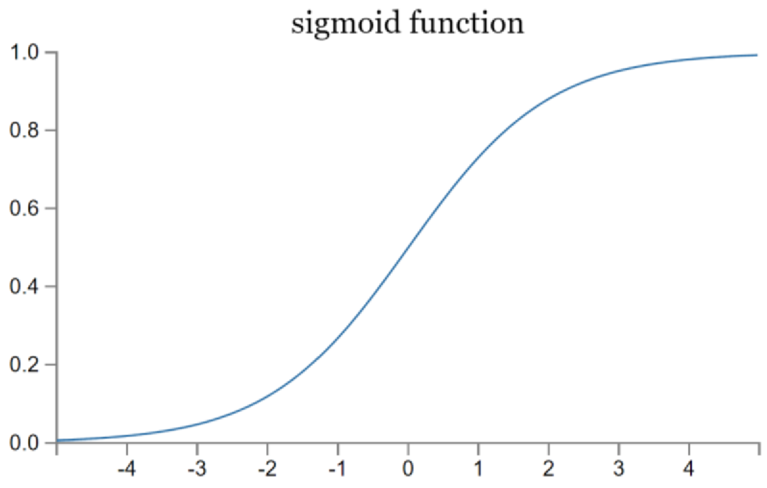


Figure 8: sigmoid function, copied from [26]

Other activation functions are tanh(x) or
if x>0 then 1 else 0
There are also cases where sigmoid function is not needed or it is replaced by other functions that are not limited by 0 and 1. [27] [20] [26]
Principle of computation of neuron is simple. It's just application of following equation:
$\sigma (\sum_n (x_n * w_n) + b) = \text{output}$

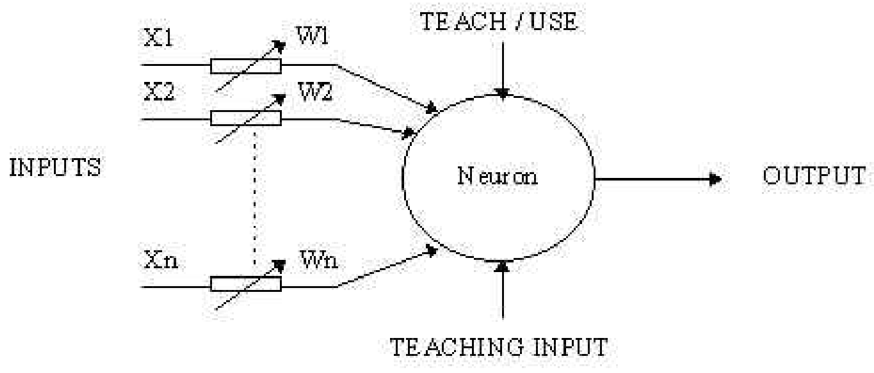


Figure 9. An MCP neuron, copied from [20]

One layer of a neural network is just multiple neurons that share the inputs in the following way: [27] A single input of the layer xi is also input xi of every neuron Nj in the layer. The output of the layer is set in the way that a single output of each neuron Nj is the output oj of the layer. The number of inputs and outputs a layer has are not necessarily the same, in most cases, when the layers are not inner/hidden, they are usually very different. Hence, the whole neural network is just connected layers from inputs to outputs, where the number of outputs of each layer must be the same as the number of inputs of the following layer.

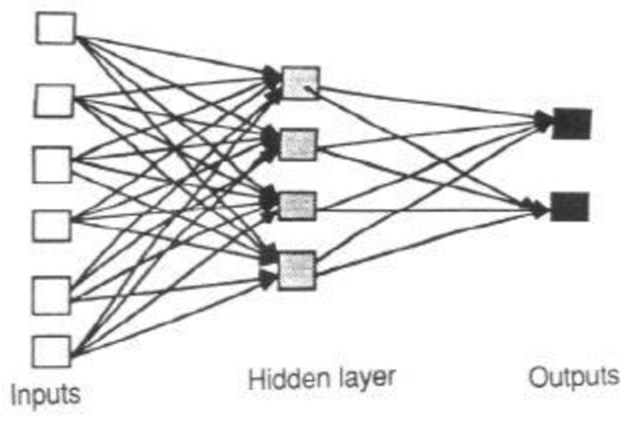


Figure 10. A neural network layers, copied from [20]

3.3 Backpropagation

The function of a learning algorithm is to map a set of inputs to outputs of the neural network the way, the neural network produces desired results. The true miracle of neural networks is the way they learn new information.

The Backpropagation algorithm enables the information to flow backward through the neural network with more than one layer, in order to compute the gradient in a simple and computationally inexpensive way.

The term Backpropagation does not refer to the entire learning algorithm for neural networks, instead, it is only the method for computing the gradient. This gradient is then being used by the stochastic gradient descent for neural networks to learn. [31]

The gradient descent optimization algorithm modifies the weights and biases of neurons.

Here is an oversimplification of how Backpropagation works: [28] [29] [30]
Let's divide function of a neuron by defining z:

$x_{(L)} = \sigma (z)$
$z = \sum_m (x_{(L-1)m} * w_m) + b$
L is the index of specific layer
n is the index of neuron
m is the index of input into neuron

First, we start with the last layer of the neural network, desired/correct outputs and actual outputs.
Cost =sum of all outputs (real output - desired output)^2
$C = \sum_n (x_{(L)(n)} - y_{(n)})^2$
$x_{(L)(n)} = \sigma (z_{(L)(n)})$
$z_{(L)(n)} = \sum_m (x_{(L-1)(m)} * w_{(L)(n)(m)}) + b_{(L)(n)}$

Derivative of weights:
$\partial C / \partial w_{(L)(n)(m)} = \partial z_{(L)(n)}/\partial w_{(L)(n)(m)} * \partial x_{(L)(n)}/\partial z_{(L)(n)} * \partial C/\partial x_{(L)(n)}$
$\partial z_{(L)(n)} / \partial w_{(L)(n)(m)} = x_{(L-1)(m)}$
$\partial x_{(L)(n)} / \partial z_{(L)(n)} = \sigma'(z_{(L)(n)})$
$\partial C / \partial x_{(L)(n)} = 2*(x_{(L)(n)} - y_{(n)})$
$\partial C / \partial w_{(L)(n)(m)} = x_{(L-1)(m)} * \sigma'(z_{(L)(n)}) * 2*(x_{(L)(n)} - y_{(n)})$

Derivative of biases:
$\partial C / \partial b_{(L)(n)} = \partial z_{(L)(n)}/\partial b_{(L)(n)} * \partial x_{(L)(n)}/\partial z_{(L)(n)} * \partial C/\partial x_{(L)(n)})$
$\partial C / \partial b_{(L)(n)} = \sigma'(z_{(L)(n)}) * 2*(x_{(L)(n)} - y_{(n)})$

Derivative of previous layer:
$\partial C / \partial x_{(L-1)(m)} = \sum_n \partial z_{(L)(n)}/\partial x_{(L-1)(m)} * \partial x_{(L)(n)}/\partial z_{(L)(n} * \partial C/\partial x_{(L)(n)}$
$\partial C / \partial x_{(L-1)(m)} = \sum_n w_{(L)(n)(m)} * \sigma'(z_{(L)(n)}) * 2 * (x_{(L)(n)} - y_{(n)})$

Then we use the calculated derivatives of weights and biases for altering of weights and biases in specific neurons and we use a derivative of the previous layer for propagating this change into a layer before the current layer and repeat the whole process.

3.5 Neural network implemented as Q-Learning

Neural networks are good at things like classification, but for playing games, it is much better to use Q-learning, because Q-learning is designed for an agent controlling its actions in an environment.
But neural networks are designed just for answering questions.

Also, another problem is Q-learning itself. Size of Q-learning table is expanding by an incredible rate, as the game is getting more complex, due to the size of the Q-learning table being equal to a number of possible states times number of possible actions, and it gets to the point where Q-learning is not practical anymore. Not only because of memory reasons of storing such massive table, but also because the Q-learning process will be lost with such great amount of states, and it will not be able to find the right path to states with rewards, because every new match will introduce new states that the agent did not encounter before.

One way how to go around it is to limit states to what we consider important for the agent to know and also limit its actions by creating actions where every action is composed of multiple elementary steps, in which the step is just a basic action of the agent in the game. In my system, I also implemented this option but it didn't turn out to be effective.

Therefore, if I want to use a neural network for an agent to move in an environment, it would be a good idea to combine Q-learning and neural networks.

One way I can do this is:

1.) If the neural network is not set, then execute a random action.
2.) Generate a random number and test if that number is smaller than threshold ε , and if it is, then execute a random action.
3.) For every action in the set of all possible actions:
    - Push the state of the environment together with the selected action as an input into the neural network.
    - Compare result from the neural network with the highest probability for the highest reward (or some alteration of this), and if it's higher, then set this result as highest.
4.) Execute an action with a higher result.
5.) Observe reward after executing the action, and train the neural network using the reward following way:
    $Q (s_t , a_t) = \alpha * (r_t + \gamma * Q (s_{t+1} , a_{t+1})))$
    The index of time/step t, in this case, it is not the current step, but all the steps that have already happened in the current game.

This process is based procedure from following source [32].

Problem with this design is Catastrophic interference. Catastrophic interference is when a neural network forgets what it had previously learned after learning new information. To avoid this problem, I will use experience replay. [32]

Let's create a buffer where all the important information, state of the environment, executed actions and rewards will be saved for a large number of steps.

Then the following process can be used:

1.) Test if state is final and if it is, then set variable steps_in_game to 0.
2.) If neural network is not set, then execute a random action.
3.) Generate a random number and test if that number is bigger than threshold ε and if it is, then execute a random action.
4.) For every action in the set of all possible actions:
    - Push the state of the environment together with the selected action as an input into the neural network.
    - Compare result from the neural network with the highest probability for the highest reward (or some alteration of this), and if it's higher, then set this result as highest.
5.) Execute an action with a higher result.
6.) Observe reward after executing the action, and alter the latest data in buffer:
    $Q (s_t , a_t) = \alpha * (r_t + \gamma * Q (s_{t+1} , a_{t+1})))$, the amount of data to alter is defined by the variable steps_in_game.
7.) Store the data, state of environment, executed action and reward from current stap in it.
8.) Test if buffer is full, and if it is, push all the stored data in buffer into neural network. Then empty the buffer.
9.) Increment the variable steps_in_game by 1.

4 Design and Implementation of the Realized System

My system is based and inspired by the PySC2 tutorial [35] made by Steven Brown. I used his work to learn how to control StarCraft II Learning Environment.
His bots are the basis of my system.

Since I'm using pysc2 for my system, I can't use micromanagement for every unit that is under my control but I can control only one selected unit in one step or a selected group of units that can only do one specific command together. I would have to take the same approach as a human player would. Design of my system is similar to what was defined in chapter 2.1 Real-Time Strategy where we talked about division of game into strategy, tactics and reactive control but the difference is I can't use micromanagement for every unit as mentioned above. The main strategy of the game is managed by Strategy Manager which is defined directly in function Step.

From this function, I'm calling other functions-Build Base, Build Army, Attack Base and Scout Base. Since I can do only one action per step, I designed the system to be a bit similar to the operating system and Strategy Manager is handling functions under it, like processes. Meaning that the Strategy Manager calls one function for example Build Base, and when this function is in the process, the function Build Base decides what actions will be made, like build some specific building on specific position by a specific unit. But finishing such action takes time, and before it's finished, other steps can be made. Therefore, if function Build Base is engaged and can't do any other actions before everything is finished, then the function will be waiting, and other functions, like Build Army or Scout Base, can be used. The exact same principle is being used inside functions Build Base and Build Army.

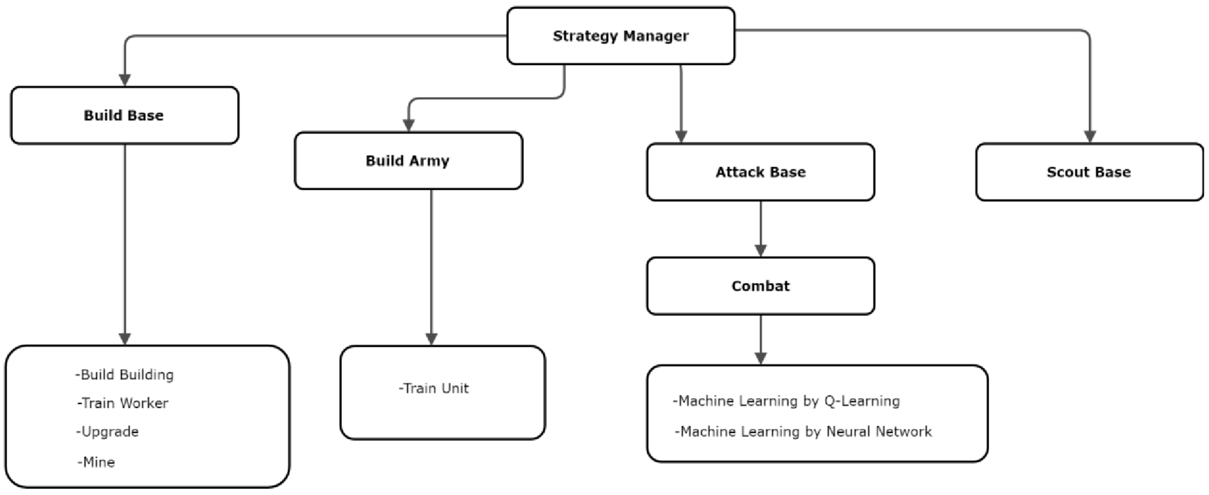


Figure 11. Chain of Command

The core of my system is class `MyAgent(base_agent.BaseAgent)` that inherits all functionality of base_agent of the PySC2.

Before the game begins, I have to locate all the strategic points. In a classical match, where players have to first build a base, gather resources and then build an army, the strategic points are Mineral Mines together with Vespene Geysers. In the function second_init, I have located these strategic points. I take the advantage of the fact that in most StarCraft maps the Mineral Mines together with the Vespene Geysers are in shape of semicircles. In my opinion, the best strategic point for such shape would be near the center of the imaginary circle, but a little farther from Mineral Mines. It's also the place where Command Center, Nexus, and Hive are located at the beginning of the game.

I calculated the position using the following method: First, I take the list of all positions of Mineral Mines and Vespene Geysers and calculate the median point. Next, I calculate the most distant point in the list from the median point, which will be the endpoints of the semicircle and then, I calculate the point most distant from this point, and I get another endpoint of the semicircle. After that, I just calculate point between the two endpoints. Let's call this point "the center". Lastly, I calculate the strategic point as (2 * the center) - the median point

All these parts are calculated at the beginning of every game, in the function called `second_init(self, obs)`.
In this function is doing all the necessary initialization, that needs to be done at the beginning of every game.
It's called second_init because there is already a function called `__init__(self)` and purpose of this function is the initialization of the class MyAgent and also initialization of class Q-learning in earlier versions of the system or Neural Network in later versions of the system.

The first function that's being called in every step of the game is function `step(self, obs)`. It's the main function of the agent and it's being called directly from the PySC2.
The PySC2 gives the function step argument obs, which is really important global structure, that contains all the observation of the environment. The information that is being contained in this structure is described in the chapter 2.3 StarCraft II Learning Environment.

Strategy Manager as mentioned above is programmed in this function. First, it updates the status of the strategic points, where is located my base, where are located enemy units, and if my base is under attack. I detect if there are some functions that are already in process, and if there are none, then the Strategy Manager will choose function base of the state of the strategic points.

Then there are functions that help Strategy Manager evaluate where to progress next:
Function `nearest_place(self,position)` finds the nearest strategic point to the requested point, in most cases it is a point that represents the position of the screen.
Function `nearest_empty_place(self,position)` finds the nearest strategic point which is completely empty or unexplored.

Function `nearest_free_place(self,position)` finds the nearest strategic point where our base is not completed.

Function `scout_base(self, position, obs)` sends some worker or a low-level unit to some unexplored strategic point.

Function `attack_base(self, position, obs)` attacks selected position if we have sufficient army, if not then calls function `build_army(obs)`. This function also serves as a defense of our base.

4.1 Combat and Reinforcement Learning Implementation

Function `combat(self, position, obs)` serves for controlling an army during battle. This function uses Neural Network for making decisions about the action of the army. In earlier versions of this system, it was using Q-Learning instead of Neural Network.

For Q-learning, I defined the state as a list of attributes of the environment that I considered to be crucial for the decision-making process of Q-learning. It is important that the list will contain a minimal amount of information, because with every new information, the size of the Q-learning table will grow exponentially, and it is crucially important to keep the table in a reasonable size for reasons mentioned in chapter 3.1 Q-learning.

Therefore, the information that I picked for the list is active type of unit, boolean informing us if first special ability is free for use, boolean informing us if second special ability is free for use and integer that represents difference in power between me and my enemy, that is calculated by function `log_for_power_of_enemy(obs)`. The reason why I picked to represent only two special abilities by boolean, is because majority of Terran units use maximally 2 special abilities and the only unit, that use more abilities is Ghost, so for this purpose, I divided the use of this unit into two different types, where first type use as a first ability Steady Targeting and as a second EMP Round, and second type uses as first ability Cloak and as a second Tac Nuke Strike. Another unit that uses more special abilities is unit Raven, but PySC2 doesn't offer use of abilities Intervence Matrix and Anti Armor Missile, so I just dismissed them.

The actions that I chose for Q-learning are: Attack weak, Attack close, Retreat, First ability, Second ability, Group change.

With this design, the size of the Q-learning table will be 14 x 2 x 2 x 5 x 6 = 1680 cells.

Action Attack weak tell the whole army to attack the enemy unit that has the lowest health and also highest value. I calculated it as hit points / (cost of unit in minerals + cost of unit in gas) and then I choose the unit with the lowest ratio as the target.

Action “Attack close” tells the whole army to attack the enemy unit that is closest our units.

Action “Retreat” commands the army to run away from enemy units. The point they are set to go to is calculated as (2 * position of the closest enemy unit) - the center of enemy units.

Action “First ability” commands the active type of unit / active group of units to use their first ability. If the ability requires coordinates of some target, then it will use one of these calculated coordinates:

weakest enemy unit with the lowest ratio of = hit points / (cost of unit in minerals + cost of unit in gas), weakest ally unit with lowest ratio of = hit points / (cost of unit in minerals + cost of unit in gas), strongest enemy unit with the highest ratio of = cost of unit in minerals + cost of unit in gas and the closest enemy unit.

Action “Second ability” commands the active group of units to use their second ability with the same rules as the first ability.

Action “Group change” changes the active group of units into a different group. By group, I mean the type of units from all the selected units that can use special abilities at the moment.

The reason why I chose this representation of state and this list of actions for Q-learning was my inspiration from the work of Stefan Wender and Ian Watson, Applying Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game StarCraft: Broodwar [33]. In their work, they compare many types of Q-learning techniques, but the main difference between their work and mine is that their agent controls only one unit, while mine controls the whole army of units of different types.
I would also like to mention another work that I used as a research for combat scenarios, but at the end, I didn't use much of it, and that is Fast Heuristic Search for RTS Game Combat Scenarios from David Churchill, Abdallah Saffidine and Michael Buro [34] .

As a reward, during testing I was trying more systems, I was trying immediate reward, when Reinforcement Learning receive reward immediately when an enemy unit dies, or punishment when own unit dies, calculated as:
```
reward=obs.observation['score_cumulative'][5]-self.last_reward-
punishment,
```
where punishment is a calculated score of own units that were lost and value of each unit is calculated as a sum of its resources needed for its creation.
`Self.last_reward` contains `obs.observation['score_cumulative'][5]` from last step.
Besides immediate reward, I was also testing final reward, where the reward was received only at the end of the game, and it was calculated as:
```
reward = obs.reward * 1000000
if obs.reward > 0 :
      reward -= self.reward_losts_sum
else :
      Reward +=
obs.observation['score_cumulative'][5]+obs.observation['score_cumula
tive'][6]
```

For Q-learning itself, I used an already created class QLearningTable, that was created by Steven Brown [35] who used some code from Morvan Zhou [16].

For a Neural Network, I defined state as a list that contains Visibility minimap, Camera minimap, Player Relative minimap, all with resolution of 16x16, Player Relative screen, Unit Type screen, Hit Points screen, Unit Density screen, again all with resolution of 16x16, type of active group of units, boolean informing us if first special ability is free for use, and boolean informing us if second special ability is free for use.

The actions for Neural Network are exactly the same as for Q-learning.

The output of neural network is one integer that can have 7 different values, numbers from 0 to 6.
Output 0 means that the reward in that one step was less than -300
Output 1 means that the reward in that one step was between -100 and -300
Output 2 means that the reward in that one step was between 0 and -100
Output 3 means that the reward in that one step was 0
(there were lots of outputs like this, mostly where nothing happened)
Output 1 means that the reward in that one step was between 0 and 100
Output 1 means that the reward in that one step was between 100 and 300
Output 6 means that the reward in that one step was more than 300
Again reward is calculated as
`obs.observation['score_cumulative'][5]-self.last_reward-punishment`
Same as for Q-learning. Final reward is just `reward = obs.reward * 100000`

For Reinforcement Learning by Neural Network, I created a class NeuralNetwork that is just rewritten class QLearningTable. In this class, I applied the technique described in chapter 3.2Neural Networks Implemented as Q-Learning.
For the Neural Network itself, I used MLPClassifier from library sklearn [27]. I learned how to use MLPClassifier thanks to the website https://www.python-course.eu/neural_networks_with_scikit.php [36].

The class NeuralNetwork contains functions:

`__init__`
`(self, actions, learning_rate=0.1, reward_decay=0.9, e_greedy=1.0)`

The initialization of NeuralNetwork, where I create MLPClassifier with parameters `( hidden_layer_sizes = ( 2050, 1000, 500), activation = 'logistic', max_iter = 150, alpha = 1e-4, solver = 'sgd', verbose = 10, tol = 1e-4, random_state = 1, learning_rate_init = lr )`
During testing, I was trying different settings of parameters, and trying to find out which setting works more efficiently.

change_policy(self, e_greedy), a function that just changes the threshold value

choose_action(self, observation,used_units), function that tests every possible action and then picks the action with the best score. All according to the procedure that was described in the chapter 3.2Neural Networks Implemented as Q-Learning . The score on base of the action is chosen is calculated as:

```
score=(7*predictions[predic][6])+(5*predictions[predic][5])+(3*predi
ctions[predic][4])-predictions[predic][3]-
(2*predictions[predic][2])-(3*predictions[predic][1])-
(4*predictions[predic][0])
```

where `predictions = self.mlp.predict_proba(testset)`

This part of code means, that it will go through all possible actions with the same state push them into function predict_proba and it will return the probability for every possible output (number from 0 to 6 as is mentioned above) and it will use these probabilities for calculating the score. (the equation above)

Next function is choose_unit(self, unit).

Well, this function is not part of the planned design. The original plan was to use Machine Learning only for combat, but at the last moment, I decided to use a ready created Reinforcement Learning system for making decisions about what units to train. I decided to do that because everything was already created for combat, so it didn't take much time to implement it.

This function loads one random state from a buffer, creates a copy of this state, and in this copy, it will alter data by adding 1 to the value in the state that is informing us about the number of units of a specific type, and that specific type is the argument of this function. Then it will push that one random state and his altered copy into self.mlp.predict_proba(testset), then it calculates the score the same way

```
(7*predictions[0][6])+(5*predictions[0][5])+(3*predictions[0][4])-
predictions[0][3]-(2*predictions[0][2])-(3*predictions[0][1])-
(4*predictions[0][0])
```

for both original state and altered state, and then as an output of this function it will return their difference. This number then will be used as a decisive factor, for making a decision whether to train this unit or not.

The last function of NeuralNetwork class is function learn(self, s, a, r, s_, used_units). This function takes state, action, and reward and does everything with them that was described in chapter 3.2 Neural Networks Implemented as Q-Learning. Meaning, it will store the state, action, and reward into the buffer and if the buffer is full, it will feed it into Neural Network using the function self.mlp.partial_fit(self.learnset, final_learnlabels)

4.2 Other Functions

The function `lower_resolution(map)` lowers the resolution of the minimap or screen to picture of 16x16 pixels.

Function `number_of_selected_units(self,obs,type_of_selected_unit)` returns number of units of specific type that are selected.

Function `log_for_power_of_enemy(self,obs):`returns ratio of how much space on minimap is taken by own units compared to how much space is taken by the enemy units.

Function `ability_free_for_use(self,obs,ability,type_of_selected_unit)` returns boolean that informs us if it is possible to use the requested ability.

Function `find_center_of_battle(self,obs)` finds an enemy unit that is closest to our units, or at least close enough.

Function `build_army(self, obs)` decides which units to create. Neural Network is being used for this decision. This function calls function `choose_unit(self, unit)` for every possible unit and which unit returns the highest ratio, that unit is being created. Of course, only if we have enough resources and the needed requirements for the training.

Function `build_base(self, position, level, obs)`decides which buildings to build and if it is needed to train workers. While working on this part of program, I found out that it is a huge problem for PySC2 to find out how many workers are working in mines, that’s why I’m only looking on how many workers are present on the screen. It's not the ideal solution, but it’s efficient.

Function `selected_unit(self, unit_id, obs)` informs me if the requested type of unit is selected.

Function `build_farm(self, position, obs)` commands worker to build building command center on some near strategic point.

Function `go_mine(self, type_of_mine, obs):` sends worker to mine some specific type of resource.

Function `train_unit(self,unit_id,obs,how_many=0)` commands some building to train train specific type of unit.

Function `train_workers(self,obs)` trains workers, SCV, Drone, Probe. This function is distinct from function `train_unit` because it also should make sure that the workers are directly being send to mine minerals.

Function `train_workers(self,obs)` trains workers, SCV, Drone, Probe. This function is distinct from function `train_unit` because it also should make sure that the workers are directly being send to mine minerals.

Function `build_building(self, type_of_building, obs)` commands worker to build specific type of building. It also improves existing buildings and researches improvements.
General problem with making buildings, is that PySC2 doesn't give back information, if selected point for placing building is free. It can try to estimate, if it might be available, which is the purpose of function `free_land(self, point_x, point_y, size, structure)`, But in the end the only way is just by trying. The reason why i decided to program this system they way, that it handless functions like processes, so it can work on other functions, while worker is being send to construct building.

Functions `how_many_units(self, unit_type_t, obs)` and `how_many_buildings(self, unit_type_t, obs)` Try to estimate how many units or buildings of specific type are present on the screen which help me following progress of development of the base. I chose this function instead of variables that would count them, because destruction of some buildings will be hard to follow.

`return_one_point_of_building(self, unit_type_t, structure)`

Gives mi list of estimated center points of buildings present on the screen. Creation of this function was needed because some unit that were too close to the building were selected instead and I caused problems, like loop.

`def same_screen_position(self, position)` tests if current position of the screen is also desired position, and this function is needed, because action
`actions.FunctionCall(_MOVE_CAMERA, [[position[1],position[0]],])` is returning the view on coordinates little bit different than what I can get by
`self.position_of_screen=(int(screen_y.mean()),int(screen_x.mean()))`
where `_MOVE_CAMERA = actions.FUNCTIONS.move_camera.id`

5 Experimental Evaluation and Discussion

In the experiments, I was testing implemented system with various implementations of reinforcement learning against default programmed bot made by Blizzard that is already part of the StarCraft II. The part that is mostly tested is combat management because that is where the machine learning was implemented. I created several testing maps by using StarCraft II Editor that contained only combat units and no resources, so the agent can focus only on combat and not other parts of the game, like development of the base.

There I was testing only combat of terran units against terran units.
Point of those experiments was to find out if the agent will be improving its strategy over time, which will prove that the reinforcement learning had taken place.

Each test had taken place for around 1000 games, and recorded final score at the end of each game. System was being tested for the score of killing enemy units and the value of its own units that been lost.

Unfortunately, in many cases we cannot compare the tests to each other, because sometimes the testing maps had been changed and some modifications had been added, but what we can compare, is how much the score had been improved over time from the first game, till the last game within one test.

TEST 1

In first test I'm testing the combat with the use of Q-learning as a main RL technique.
The settings of Q-learning process are:

```
Q-learning (self, actions, learning_rate=0.01, reward_decay=0.9,
e_greedy=0.9)
```

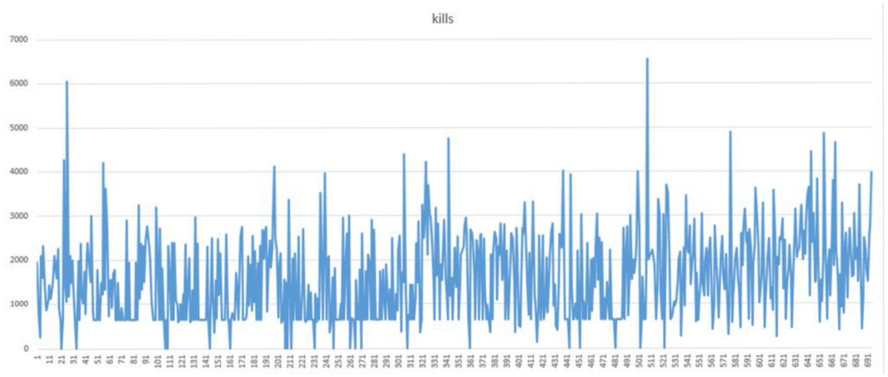
Immediate reward for every step is set as:

```
reward = obs.observation['score_cumulative'][5] - self.last_reward -
punishment
```

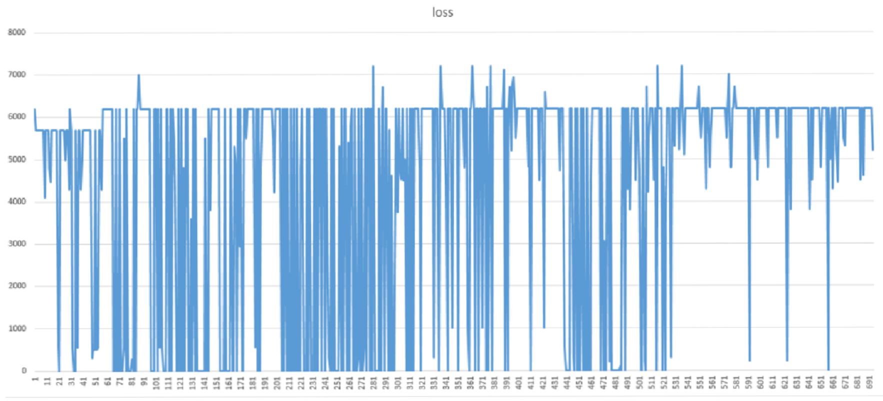
And reward at the end of the game is set as:

```
reward = 1000 * obs.reward
```

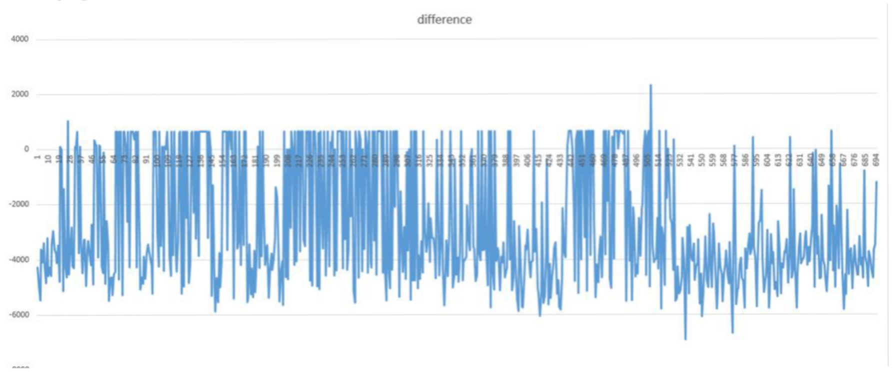
This graph shows the Blizzard score of the killed enemy units in every game. Score is usually calculated as a sum of resources that were needed of creation the specific unit.
X axis represents the number of played games and Y axis represents the score.



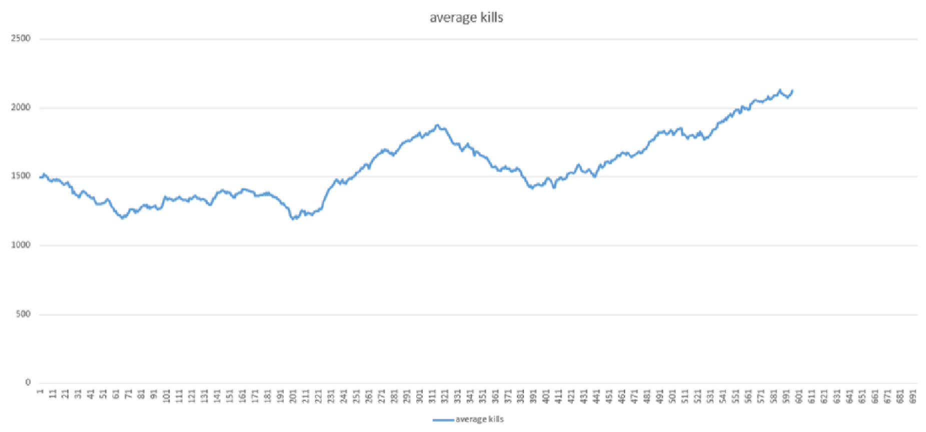
This graph shows the score as the sum of value of lost own units in every game.



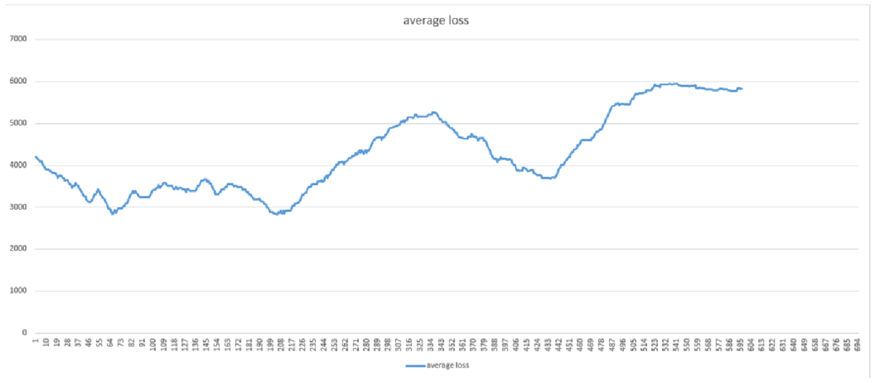
This graph shows the difference between score of the killed enemy units and loss of own units in every game.



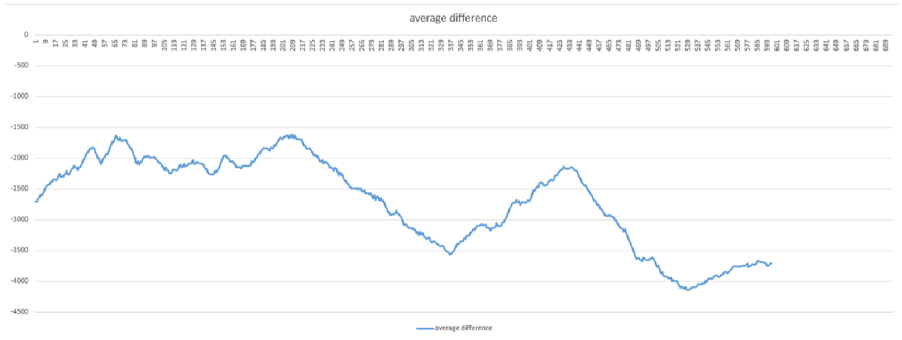
This graph shows the score of the killed enemy units in every game but its averaged for 100 games. This graph is important, because the original graph is not transparent, because incredible variation of results in games.



This graph shows the score of the lost value of friendly units in every game but its averaged for 100 games.



This graph shows the difference between score of the killed enemy units and loss of friendly units in every game but its averaged for 100 games.



In this test we can see that over time the kills score increases, but also the loss increases, and when we sum the kills score and the loss score together, we can see that our score is decreasing over time,which is the exact opposite of what we hoped to achieve. Close to the end of testing, the most frequent actions that were chosen where actions Attack Weak and Second Ability.

TEST 2

In this test I'm testing the combat with the use of Q-learning as a main RL technique.
The settings of Q-learning process are:

```
Q-learning (self, actions, learning_rate=0.05, reward_decay=0.9,
e_greedy=0.9)
```

Immediate reward for every step is set as:

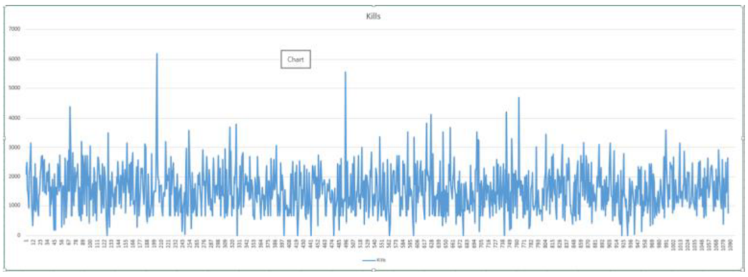
```
reward = obs.observation['score_cumulative'][5] - self.last_reward -
punishment
```

No reward at the end of the game is being set.

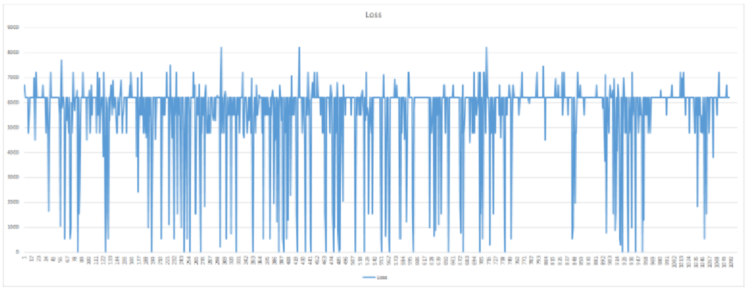
With each game progresses, i set the variable self.epsilon to decrease with every other game, to achieve slow progress over the all test.

```
greedy_incline=(self.runs_of_game-(self.runs_of_game%10))/10
self.qlearn.change_policy(greedy_incline/100)
```

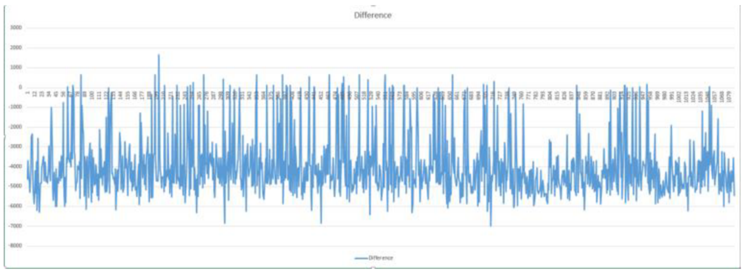
This graph shows the Blizzard score of the killed enemy units in every game.
X axis represents the number of played games and Y axis represents the score.



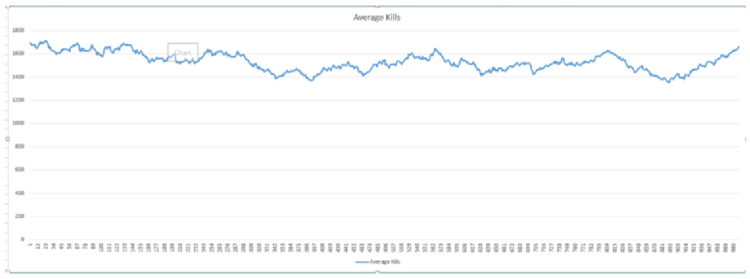
This graph shows the score as the sum of value of lost own units in every game.



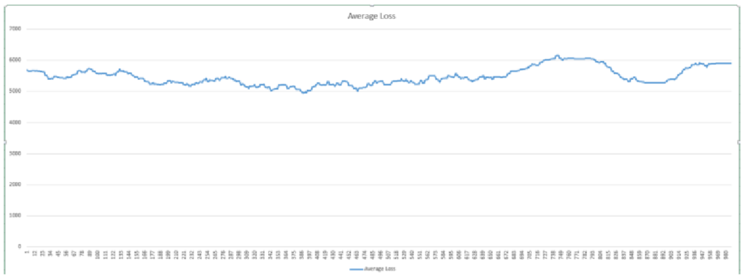
This graph shows the difference between score of the killed enemy units and loss of own units in every game.



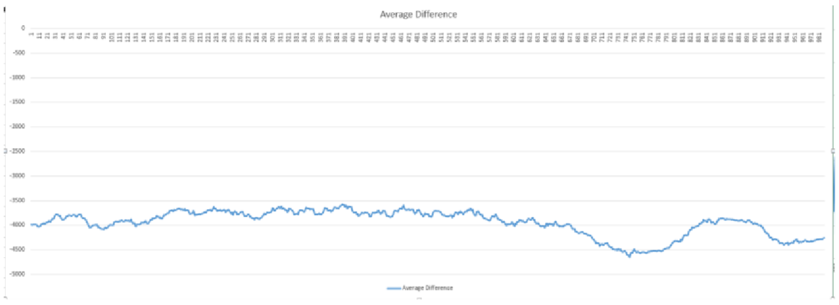
This graph shows the score of the killed enemy units in every game but its averaged for 100 games. This graph is important, because the original graph is not transparent, because incredible variation of results in games.



This graph shows the score of the lost value of friendly units in every game but its averaged for 100 games.



This graph shows the difference between score of the killed enemy units and loss of friendly units in every game but its averaged for 100 games.



In this test we can see that none of the test progresses, which shows that the learning process had no real effect for the results of the game. This is proof that my implementation of Q-learning was not successful. The main reason why this happened is the fact that the states i use as a representation of the environment are not enough for making decision of what action to use.

TEST 3

In this test I'm testing the combat with the use of Neural Network applied as a Q-learning as a main RL technique.

The settings of Neural Network are:

```
Neural Network (self, actions, learning_rate=0.1, reward_decay=0.9,
e_greedy=1.0):
```

```
MLPClassifier(hidden_layer_sizes=(600, 1),activation='logistic',
max_iter=150, alpha=1e-4,solver='sgd', verbose=10, tol=1e-4,
random_state=1,learning_rate_init=.1)
```

activation='logistic' sets the Activation function to Sigmoid function $\sigma (x) = 1 / (1 + e^{-x})$
Before conducting this test, I forgot to set this variable, and it had bad consequences, meaning that the Neural Network didn't see difference between different actions for the output.

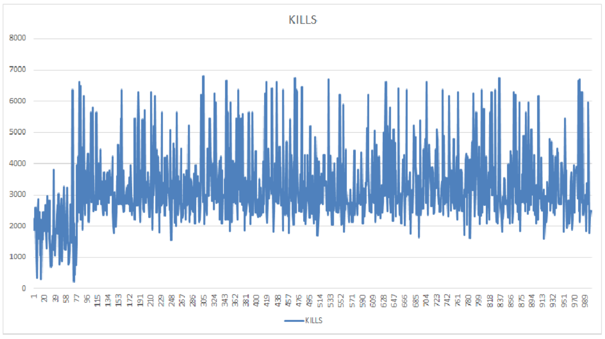
Size of the buffer is being set to 10000 states of environment.

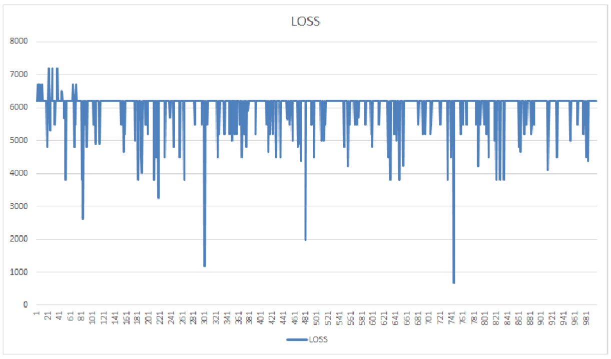
The output of neural network was calculated from reward by following way:

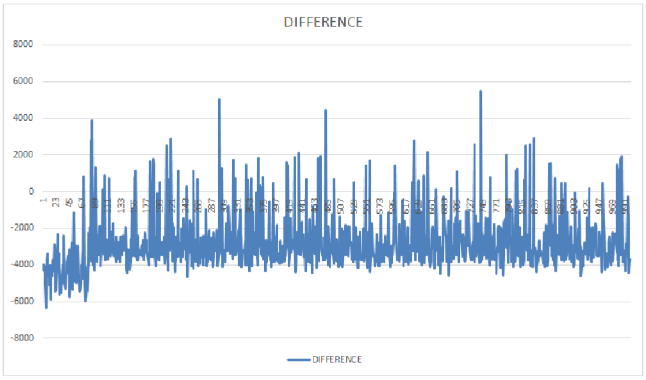
```
        if r>200:
                self.learnlabels.append(6)
        elif r>100:
                self.learnlabels.append(5)
        elif r>0:
                self.learnlabels.append(4)
        elif r==0:
                self.learnlabels.append(3)
        elif r>-100:
                self.learnlabels.append(2)
        elif r>-200:
                self.learnlabels.append(1)
        else:
                self.learnlabels.append(0)
```

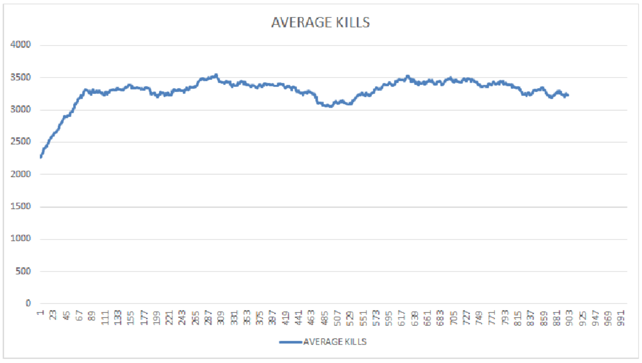
The state of environment for Neural Network are being set as: a list that contains Visibility minimap, Camera minimap, Player Relative minimap, all with resolution of 16x16, Player Relative screen, Unit Type screen, Hit Points screen, Unit Density screen, again all with resolution of 16x16, type of active group of units, boolean informing us if first special ability is free for use, but boolean informing us if second special ability is free for use is missing in this test and following tests.

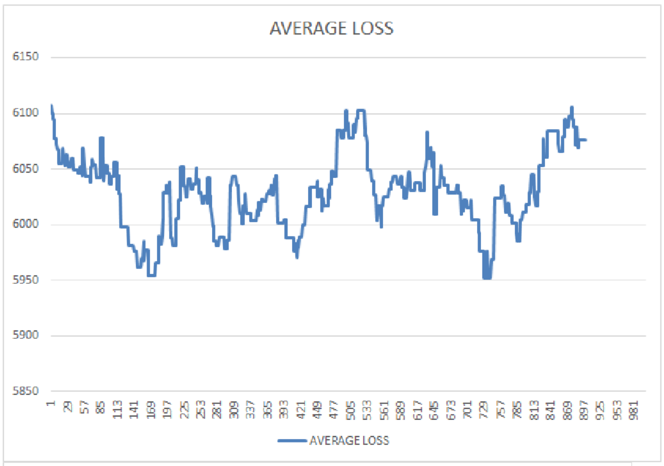
X axis represents the number of played games and Y axis represents the score.

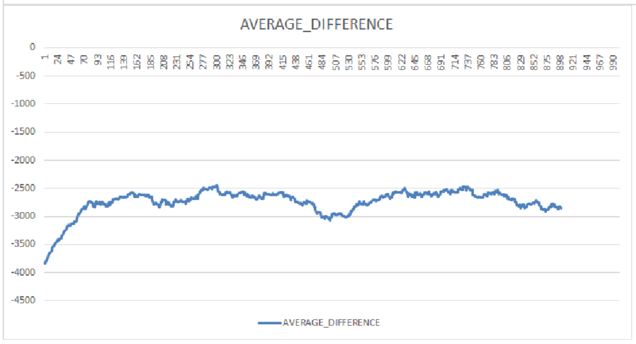



LOSS
8000
7000
6000
5000
4000
3000
2000
1000
0
LOSS


DIFFERENCE
8000
6000
4000
2000
0
-2000
-4000
-6000
-8000
DIFFERENCE


AVERAGE KILLS
4000
3500
3000
2500
2000
1500
1000
500
0
AVERAGE KILLS





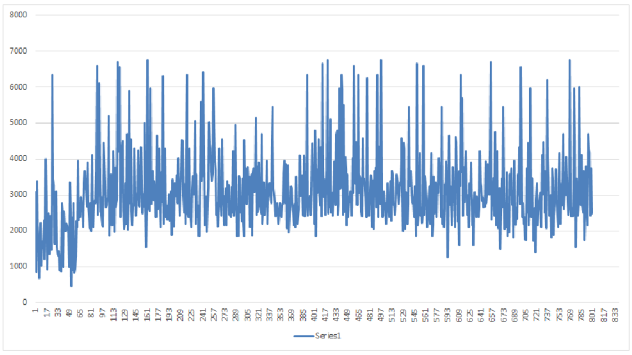
In this test we can see that the learning process had taken place, as shows the progress of kills at the beginning of the graph, which is the time when program gathered enough data and the buffer was filled and the Neural Network was activated. Close to the end of testing, the most frequent action that was chosen, was action attack the weakest unit.

TEST 4

In this test I'm testing the combat with the use of Neural Network applied as a Q-learning as a main RL technique.
The settings of Neural Network are:

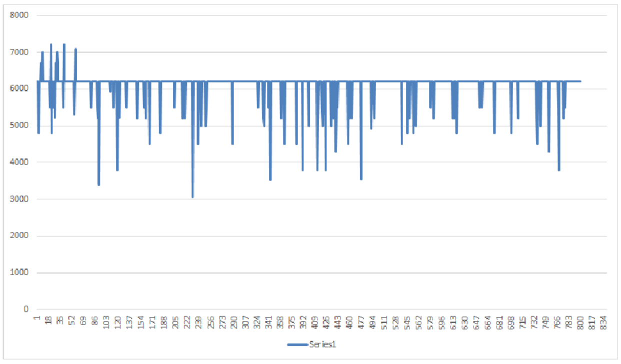
```
Neural Network (self, actions, learning_rate=0.1, reward_decay=0.9,
e_greedy=1.0):
MLPClassifier(hidden_layer_sizes=(600, 3),activation='logistic',
max_iter=150, alpha=1e-4,solver='sgd', verbose=10, tol=1e-4,
random_state=1,learning_rate_init=.1)
```

Kills
This graph shows the Blizzard score of the killed enemy units in every game.
X axis represents the number of played games and Y axis represents the score.



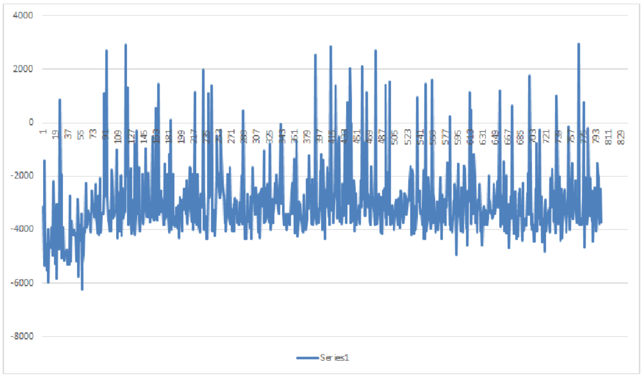
Loss

This graph shows the score as the sum of value of lost own units in every game.



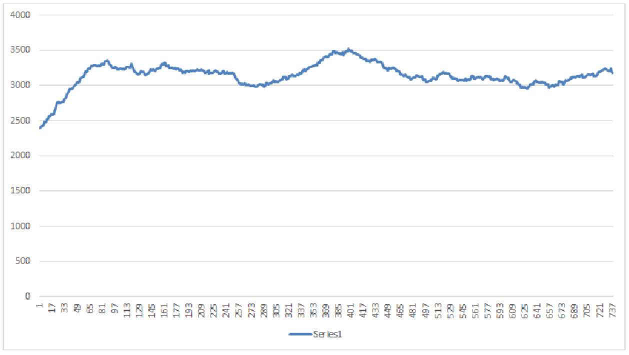
Difference

This graph shows the difference between score of the killed enemy units and loss of own units in every game.



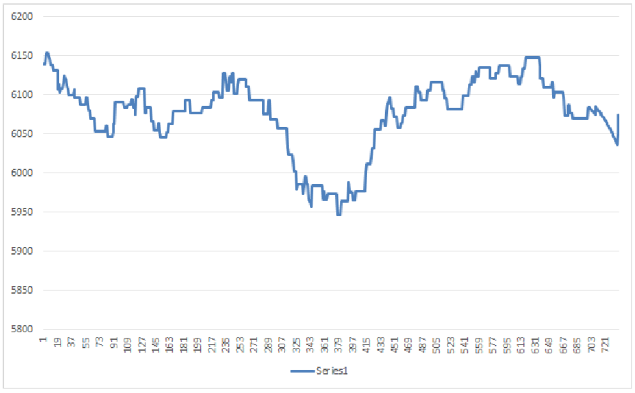
Average Kills

This graph shows the score of the killed enemy units in every game but its averaged for 100 games. This graph is important, because the original graph is not transparent, because incredible variation of results in games.



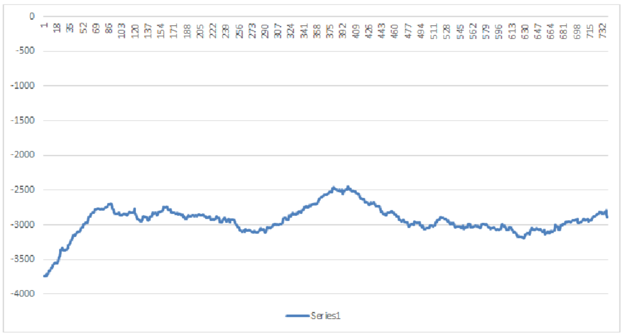
Average Loss

This graph shows the score of the lost value of friendly units in every game but its averaged for 100 games.



Average Difference

This graph shows the difference between score of the killed enemy units and loss of friendly units in every game but its averaged for 100 games.



In this test we can see that the learning process had taken place, as shows the progress of kills at the beginning of the graph, which is the time when program gathered enough data and the buffer was filled and the Neural Network was activated. It's the same as Test 3.Close to the end of testing, the most frequent action that was chosen, was action attack the weakest unit.

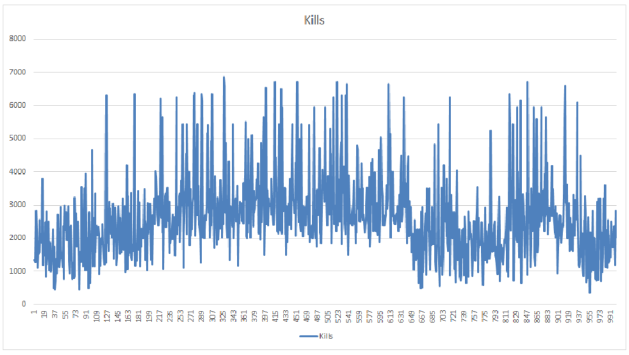
Test 5

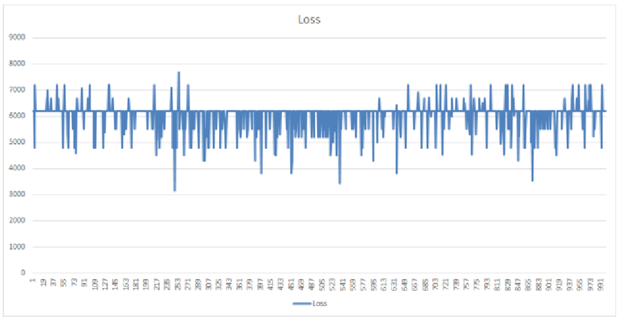
In this test I'm testing the combat with the use of Neural Network applied as a Q-learning as a main RL technique.
The settings of Neural Network are:

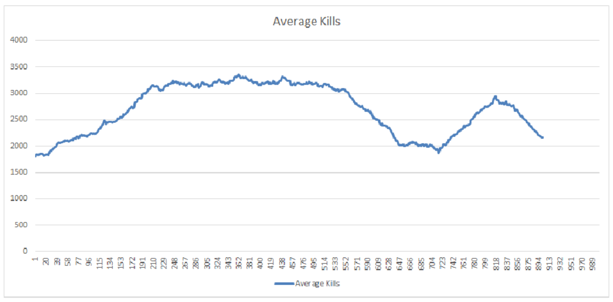
```
Neural Network (self, actions, learning_rate=0.1, reward_decay=0.9,
e_greedy=1.0):
MLPClassifier(hidden_layer_sizes=(600, 3),activation='logistic',
max_iter=150, alpha=1e-4,solver='sgd', verbose=10, tol=1e-4,
random_state=1,learning_rate_init=.1)
```

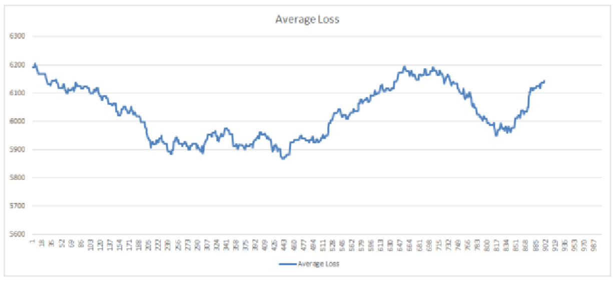
With each game progresses, i set the variable self.epsilon to decrease with every other game, to achieve slow progress over the all test.

```
r_decayed=self.gamma*r
greedy_incline=float(self.runs_of_game/400.0)
        if greedy_incline>1.0:
            greedy_incline=1.0
        self.qlearn.change_policy(greedy_incline)
```









In this test we can see that the learning process had taken place, as shows the progress of kills and decrease of loss for first 200 games, which is the time when `greedy_incline` got near the value of 0.5. Near the game 600 we can see decline in performance and then again grow. I'm not sure what caused that, Maybe because neural network did not get data with enough variance of actions. Close to the end of testing, the most frequent action that was chosen, was action attack the weakest unit.

Test 6

In this test I'm testing the combat with the use of Neural Network applied as a Q-learning as a main RL technique.
The settings of Neural Network are:

```
Neural Network (self, actions, learning_rate=0.1, reward_decay=0.9, e_greedy=1.0):
MLPClassifier(hidden_layer_sizes=(2050,1000,500),activation='logistic', max_iter=150, alpha=1e-4,solver='sgd', verbose=10, tol=1e-4, random_state=1,learning_rate_init=.1)
```

The output of neural network was calculated from reward by following way:

```
        if r>300:
                self.learnlabels.append(6)
        elif r>100:
                self.learnlabels.append(5)
        elif r>0:
                self.learnlabels.append(4)
        elif r==0:
                self.learnlabels.append(3)
        elif r>-100:
                self.learnlabels.append(2)
        elif r>-300:
                self.learnlabels.append(1)
        else:
                self.learnlabels.append(0)
```

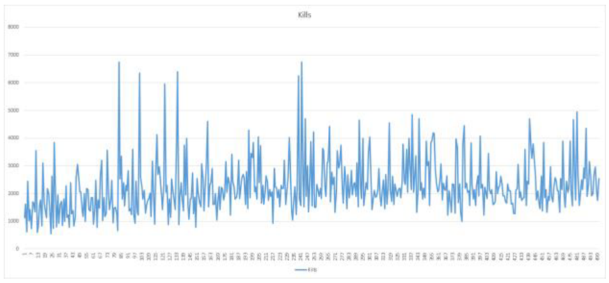
Size of the buffer is being set to 7000 states of environment.

With each game progresses, i set the variable self.epsilon to decrease with every other game, to achieve slow progress over the all test.

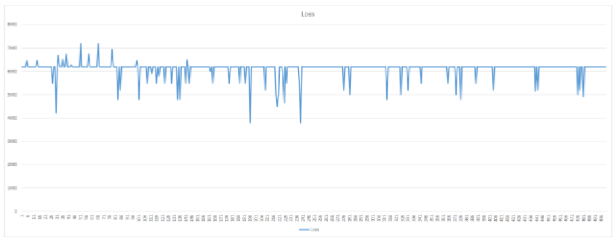
```
r_decayed=self.gamma*r
greedy_incline=float(self.runs_of_game/250.0)
        if greedy_incline>1.0:
            greedy_incline=1.0
        self.qlearn.change_policy(greedy_incline)
```

The state of environment for Neural Network are being set as: a list that contains Visibility minimap, Camera minimap, Player Relative minimap, all with resolution of 16x16, Player Relative screen, Unit Type screen, Hit Points screen, Unit Density screen, again all with resolution of 16x16, type of active group of units, boolean informing us if first special ability is free for use, and boolean informing us if second special ability is free for use had been added for this test and all following tests.

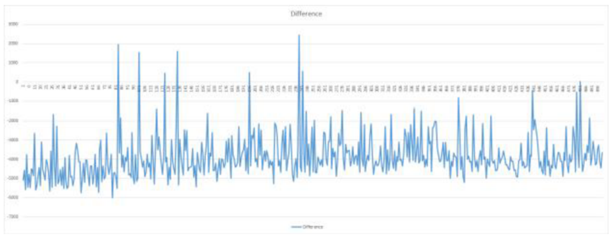
This graph shows the Blizzard score of the killed enemy units in every game.
X axis represents the number of played games and Y axis represents the score.



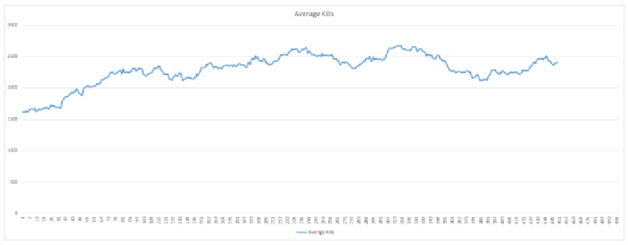
This graph shows the score as the sum of value of lost own units in every game.



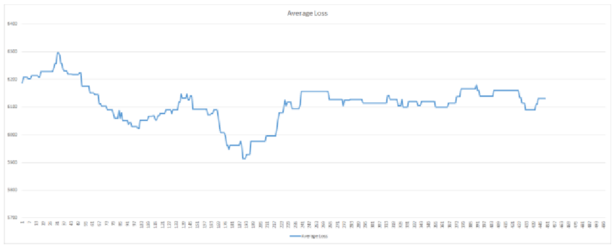
This graph shows the difference between score of the killed enemy units and loss of own units in every game.



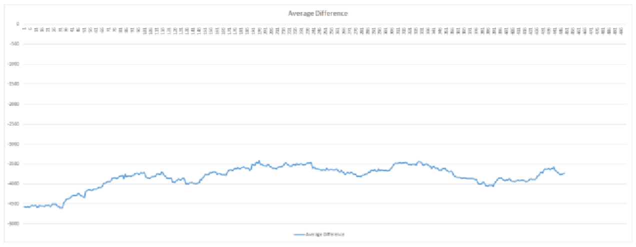
This graph shows the score of the killed enemy units in every game but its averaged for 100 games.



This graph shows the score of the lost value of friendly units in every game but its averaged for 100 games.



This graph shows the difference between score of the killed enemy units and loss of friendly units in every game but its averaged for 100 games.



In this test we can see that the learning process had taken place, as shows the progress of kills and slow decrease of loss for first 150 games. Close to the end of testing, the most frequent action that was chosen, was action Attack Weak.

Test 7

In this test I'm testing the the overall performance on a classic test map Simple64 against single enemy with the use of Neural Network applied as a Q-learning as a main RL technique.
Since the whole game is much longer that just combat, for each test i conducted only 50 games.
Average game lasted around 20 minutes and average combat lasted around 3 minutes of game time.
Race that is selected for my agent is Terran, but the enemy have randomly generated race each game.

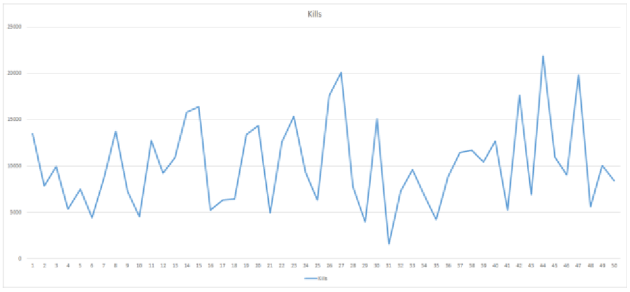
The settings of Neural Network are:

```
Neural Network (self, actions, learning_rate=0.1, reward_decay=0.7,
e_greedy=1.0):
MLPClassifier(hidden_layer_sizes=(2050, 3),activation='logistic',
max_iter=150, alpha=1e-4,solver='sgd', verbose=10, tol=1e-4,
random_state=1,learning_rate_init=.1)
```

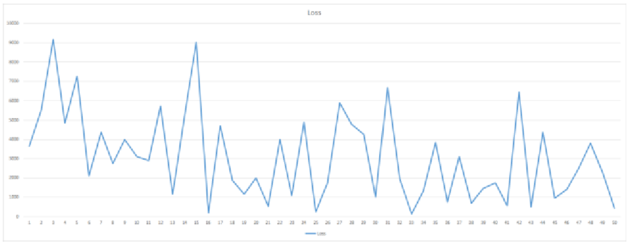
With each game progresses, i set the variable self.epsilon to decrease with every other game, to achieve slow progress over the all test.

```
r_decayed=self.gamma*r
greedy_incline=float(self.runs_of_game/40.0)
        if greedy_incline>1.0:
            greedy_incline=1.0
        self.qlearn.change_policy(greedy_incline)
```

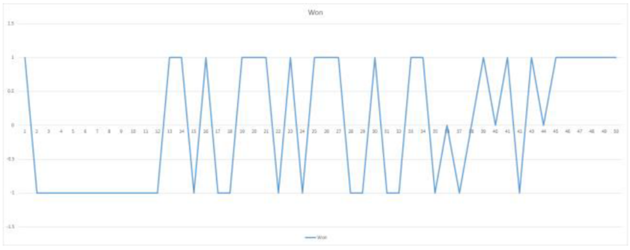
This graph shows the Blizzard score of the killed enemy units in every game.
X axis represents the number of played games and Y axis represents the score.



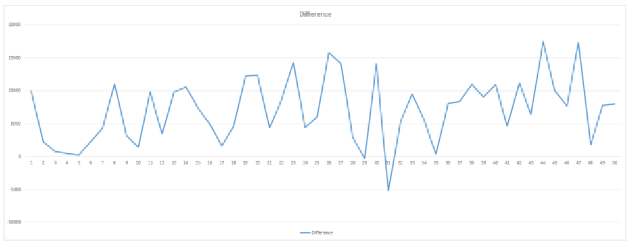
This graph shows the score as the sum of value of lost own units in every game.



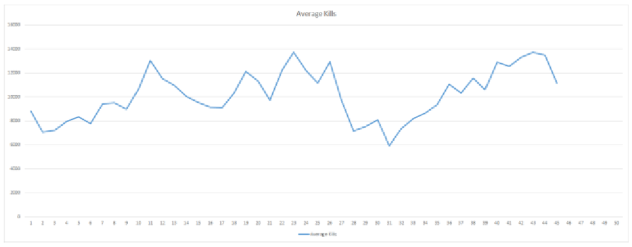
This graph shows the final score of the game 1=win 0=tie -1=loss.



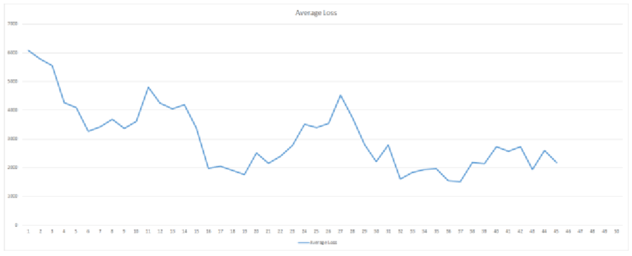
This graph shows the difference between score of the killed enemy units and loss of own units in every game.



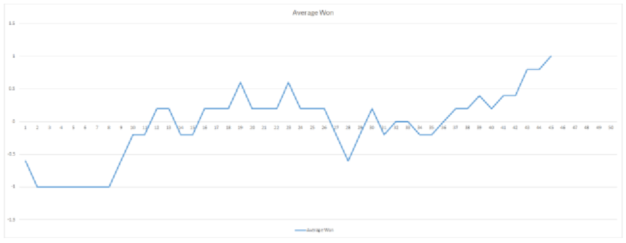
This graph shows the score of the killed enemy units in every game but its averaged for 5 games.



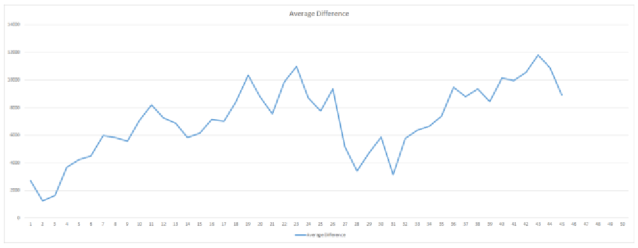
This graph shows the score of the lost value of friendly units in every game but its averaged for 5 games.



This graph shows the final score of the game 1=win 0=tie -1=loss.



This graph shows the difference between score of the killed enemy units and loss of friendly units in every game but its averaged for 5 games.



In this test we can see that the learning process had taken place, as shows the progress of final wins and decrease of loss. This shows that the Agent tried to defeat the enemy faster as it learned more. Close to the end of testing, the most frequent action that was chosen, was action Attack Weak.

Test 8

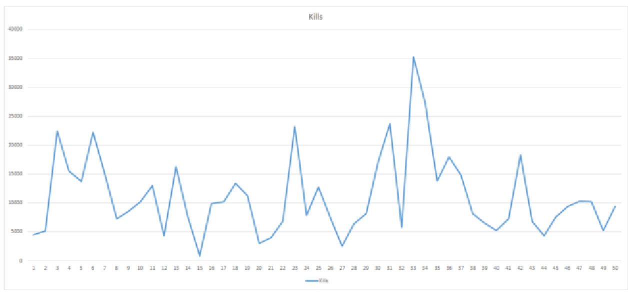
In this test I'm testing the the overall performance on a classic test map Simple64 against single enemy with the use of Neural Network applied as a Q-learning as a main RL technique.
Race that is selected for my agent is Terran, but the enemy have randomly generated race each game.

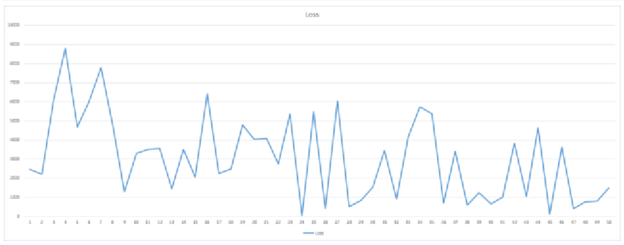
The settings of Neural Network are:

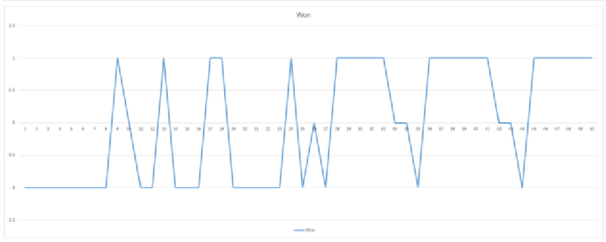
```
Neural Network (self, actions, learning_rate=0.1, reward_decay=0.7,
e_greedy=1.0):
MLPClassifier(hidden_layer_sizes=(2050,1000,500),activation='logisti
c', max_iter=150, alpha=1e-4,solver='sgd', verbose=10, tol=1e-4,
random_state=1,learning_rate_init=.1)
```

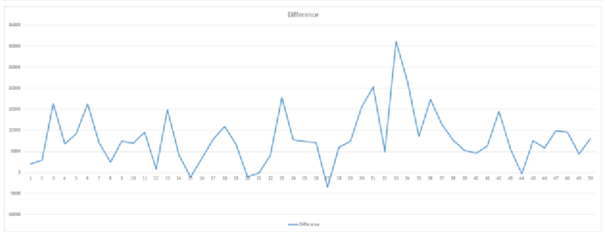
With each game progresses, i set the variable self.epsilon to decrease with every other game, to achieve slow progress over the all test.

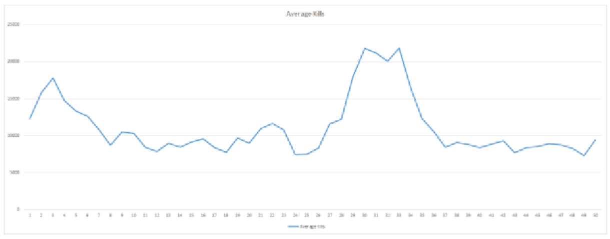
```
r_decayed=self.gamma*r
greedy_incline=float(self.runs_of_game/40.0)
        if greedy_incline>1.0:
            greedy_incline=1.0
        self.qlearn.change_policy(greedy_incline)
```

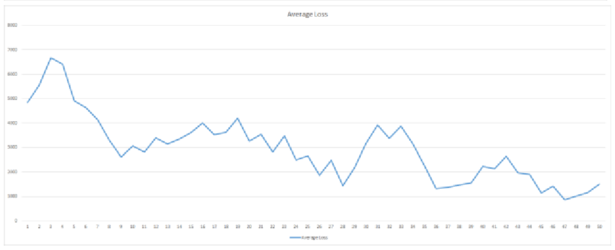


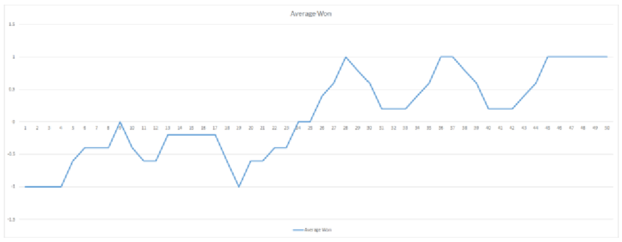


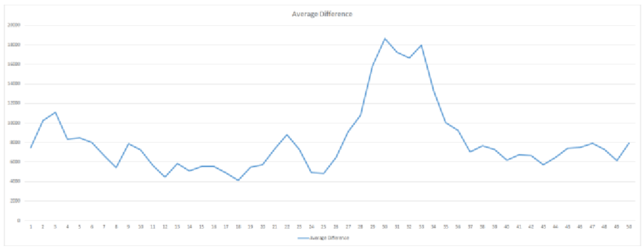












In this test, we can see that the learning process had taken place, as shows the progress of final wins and decrease of loss. This shows that the Agent tried to defeat the enemy faster as it learned more. Close to the end of testing, the most frequent action that was chosen, was action Attack Weak.

The conclusion that we can make out of all the tests that took place is that the number of neurons and the number of layers are not as important, because in the end the Neural Network just chooses to use the action "attack the weakest unit", which is disappointing for me, because I had hoped to see some variation of the use of special abilities. Nevertheless, the learning process had taken place for the use of Neural Network, which is a success, and it proves that the representation of state of the game was sufficient enough to make progress over time.

6 Summary, Conclusions and Future Directions

The goal of this work was to design, implement and evaluate system for the game StarCraft II. System shows progress over time as seen from the experiments and is able to defeat the default bot in StarCraft II created by Blizzard.

In order to achieve this, I had to learn how to use StarCraft 2 Learning Environment for the creation of the system as well as to learn to use Neural Network and Q-learning to incorporate them into the module.

From the experiments, I concluded that the use of Neural Network is more efficient than that of Q-learning for combat scenarios.

The part of the project that I decided not to implement was the use of replays -the record of games made by professional players. The main reason why i didn't use the replays is because, the actions that players take, do not correlate with my implemented actions and it would be hard to detect which parts of the game are happening. Instead, I implemented the learning process by reinforcement, meaning, the agent has to play the game by itself in order to learn.

My solution is not final and in the future there is a lot to improve, perhaps in the application of Machine learning on other aspects of the game. Part that I definitely can improve in future is full development for races Zerg and Protoss. The topic of AI for RTS has a lot to offer and so there are a lot of advancements to be made.

References

[1] GERYK, B. In: *Bruce Geryk's blog: A History of Real-Time Strategy Games* [online]. [retrieved on 2018-05-08] Available at: <https://web.archive.org/web/20110427052656/http://gamespot.com/gamespot/features/all/real_time>

[2] GERYK, B. In: Bruce Geryk's blog: *A History of Real-Time Strategy Games* [online]. [retrieved on 2018-05-08] Available at: <https://web.archive.org/web/20110721213420/http://www.gamespot.com/gamespot/features/all/real_time/p2_01.html>

[3] Wikipedia: *Turn-based Strategy* [online]. [retrieved on 2018-05-08]. Available at: <https://en.wikipedia.org/wiki/Turn-based_strategy>

[4] CHURCHILL, D.: *Heuristic Search Techniques for Real-Time Strategy Games*. Edmonton, 2016. PhD thesis. University of Alberta, Department of Computing Science. [retrieved on 2018-05-08] Available at: <https://era.library.ualberta.ca/items/c3589c0d-9b9e-46d5-b1a1-b4004379faad/view/9d9a23d0-7dc1-46ad-afd3-ba8f8cc49c39/thesis_pdfa.pdf>

[5] VINYALS, O.; GAFFNEY, S.; and EWALDS, T.: DeepMind's blog: *DeepMind and Blizzard Open StarCraft II as an AI Research Environment, 2017* [online]. [retrieved on 2018-05-08] Available at: <https://deepmind.com/blog/deepmind-and-blizzard-open-starcraft-ii-ai-research-environment/>

[6] Wikipedia: *StarCraft* [online]. [retrieved on 2018-05-08] Available at: <https://en.wikipedia.org/wiki/StarCraft>

[7] *AIIDE StarCraft AI Competition* [online]. [retrieved on 2018-05-08] Available at: <http://www.cs.mun.ca/~dchurchill/starcraftaicomp/>

[8] BURO, Michael, PhD.: *Call for AI Research in RTS Games. In Proceedings of the AAAI Workshop on AI in Games, 2004*. San Jose: McEnery Convention Center. [retrieved on 2018-05-08] Available on: <http://www.aaai.org/Papers/Workshops/2004/WS-04-04/WS04-04-028.pdf>

[9] *StarCraft II* [online]. [retrieved on 2018-05-08] Available at: <https://github.com/deepmind/pysc2/blob/master/docs/environment.md>

[10] VINYALS, O. et al.: *StarCraft II: A New Challenge for Reinforcement Learning arXiv:1708.04782, 2017* [online]. [retrieved on 2018-05-08] Available at: <https://deepmind.com/documents/110/sc2le.pdf>

[11] Github: [davehurchill] *CommandCenter: AI Bot for Broodwar and Starcraft II* [online]. [retrieved on 2018-05-08] Available at: <https://github.com/davechurchill/commandcenter>

[12] SAMUEL, A. L.: *Computer Games I.* NY: Springer, New York, 1988. [retrieved on 2018-05-08] ISBN 978-1-4613-8718-3 also available at: <https://link.springer.com/chapter/10.1007%2F978-1-4613-8716-9_14>

[13] Wikipedia: *Reinforcement Learning* [online]. [retrieved on 2018-05-08] Available at: <https://en.wikipedia.org/wiki/Reinforcement_learning>

[14] Watkins, C.J.C.H. & Dayan, P. Mach Learn (1992): *Machine Learning (Q-Learning).* [retrieved on 2018-05-08] ISSN 0885-6125 also available at: <https://rd.springer.com/article/10.1007/BF00992698#citeas https://rd.springer.com/content/pdf/10.1007%2FBF00992698.pdf>

[15] HOEIJMAKERS, N.: *Possibilities for Applying Decision-making Processes and Reinforcement Learning with Units in a RTS Game*. Enschede, NL. Bachelor's this. University of Twente. [retrieved on 2018-05-08] Available at: <http://referaat.cs.utwente.nl/conference/5/paper/6745/possibilities-for-applying-decision-making-processes-and-reinforcement-learning-with-units-in-a-rts-game.pdf>

[16] Github: [MorvanZhou]: *Reinforcement Learning with Tensorflow.* [online] [retrieved on 2018-05-08] Available at: <https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow>

[17] McCULLOCK, J.: *John McCullock's Tutorial: Q-Learning* [online]. [retrieved on 2018-05-08] Available at: <http://mnemstudio.org/path-finding-q-learning-tutorial.htm>

[18] MELO, F. S.: *Convergence of Q-learning: A Simple Proof*. Institute for Systems and Robotics, Lisbon. [retrieved on 2018-05-08] Available at: <http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf>

[19] JIANG, J.; TRUNDLE, P.; and REN, J.: *Medical Image Analysis with Artificial Neural Networks.* Digital Media & Systems Research Institute, University of Bradford. [retrieved on 2018-05-08] Available at: <https://pdfs.semanticscholar.org/bf3c/5e84e462e19940e025abe0f1687d85842fac.pd>

[20] STERGIOU, C.; and SIGANOS, D.: *Neural Networks* [online]. [retrieved on 2018-05-08] Available at: <https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#What%20is%20a%20Neural%20Network/>

[21] CHUNG, M.; BURO, M.; and SCHAEFFER, J.: *Monte Carlo planning in RTS Games. In IEEE Symposium on Computational Intelligence*

and Games (CIG). Colchester: Essex University, 2005. [retrieved on 2018-05-08] Available at: <http://citeseerx.ist.psu.edu/viewdoc/similar?doi=10.1.1.125.7452&type=cc>

[22] WILSON, A. R.: *Masters of War: History's Greatest Strategic Thinkers, 2012* [Audio/Video Course] [online]. [retrieved on 2018-05-08] Available at: <http://www.thegreatcourses.com/courses/masters-of-war-history-s-greatest-strategic-thinkers.html>

[23] Github: *actions.py* [online] [retrieved on 2018-05-08] Available at: <https://github.com/deepmind/pysc2/blob/e9d12938afa0f28fb9285e5b0acf4a5c1a8f7289/pysc2/lib/actions.py>

[24] Github: [tewalds] *StarCraft II Learning Environment* [online] [retrieved on 2018-05-08] Available at: <https://github.com/deepmind/pysc2>

[25] Github: [sheikyabooti] *StarCraft II Client - protocol definitions used to communicate with StarCraft II* [online] [retrieved on 2018-05-08] Available at: <https://github.com/Blizzard/s2client-proto>

[26] NIELSEN, M. A.: *Neural Networks and Deep Learning*. Determination Press, 2015 [online]. [retrieved on 2018-05-08] Available at: <http://neuralnetworksanddeeplearning.com/chap1.html>

[27] PEDREGOSA et al.: *Scikit-learn: Machine Learning in Python JMLR 12, pp. 2825-2830, 2011* [online]. [retrieved on 2018-05-08] Available at:<http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html>

[28] NIELSEN, M. A.: *Neural Networks and Deep Learning*. Determination Press, 2015 [online]. [retrieved on 2018-05-08] Available at: <http://neuralnetworksanddeeplearning.com/chap2.html>

[29] [3Blue1Brown], (2017, Nov 3) *Backpropagation calculus | Appendix to deep learning chapter 3* [Video File] [online]. [retrieved on 2018-05-08] Available at: <https://www.youtube.com/watch?v=tIeHLnjs5U8>

[30] WERBOS, P., J.: *Backpropagation Through Time: What It Is and How to Do It. In Proceedings of the IEEE, 1990.* VOL. 78, NO. 10, Oct 1990. [retrieved on 2018-05-08] Available at: <ftp://143.54.11.3/pub/SIA/refer%C3%AAncias/BPTT.pdf>

[31] GOODFELLOW, I.; BENGIO, Y.; and COURVILLE, A.: *Deep Learning.* MIT Press. [retrieved on 2018-05-08] ISBN 9780262035613 also available at: <https://www.scribd.com/document/318441546/Ian-Goodfellow-Yoshua-Bengio-Aaron-Courville-Deep-Learning-pre-pub-version-MIT-Press-2016-pdf>

[32] SOREN, D.: *Soren D's Blog: Teaching a Neural Network to Play a Game Using Q-learning, 2017* [online]. [retrieved on 2018-05-08] Available at: <https://www.practicalai.io/teaching-a-neural-network-to-play-a-game-with-q-learning/>

[33] WENDER, S.; and WATSON, I.: *Applying Reinforcement Learning to Small Scale Combat in the*
Real-Time Strategy Game StarCraft:Broodwar 2012. Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG'12). [retrieved on 2018-05-08] Available at:

<https://pdfs.semanticscholar.org/1308/c8326203caec91a7c44ffd1dfe86dd227c7f.pdf>

[34] CHURCHILL, D.; SAFFIDINE, A.; and BURO, M.: *Fast Heuristic Search for RTS Game Combat Scenarios. Proceedings of the AIIDE Conference,2012*. Stanford, Palo Alto, California. [retrieved on 2018-05-08] Available at: <https://skatgame.net/mburo/ps/aiide12-combat.pdf>

[35] Github: [skjb] *Tutorials for Building a PySC2 Bot* [online] [retrieved on 2018-05-08] Available at: <https://github.com/skjb/pysc2-tutorial>

[36] KLEIN, B.: Bernd Klein's course: *Python Machine Learning Tutorial: Neural Networks with scikit.* [online] [retrieved on 2018-05-08] Available at: <https://www.python-course.eu/neural_networks_with_scikit.php>

Appendices

Appendix A

CD contains electronic version of this report, source code and folder containg all the experiments.