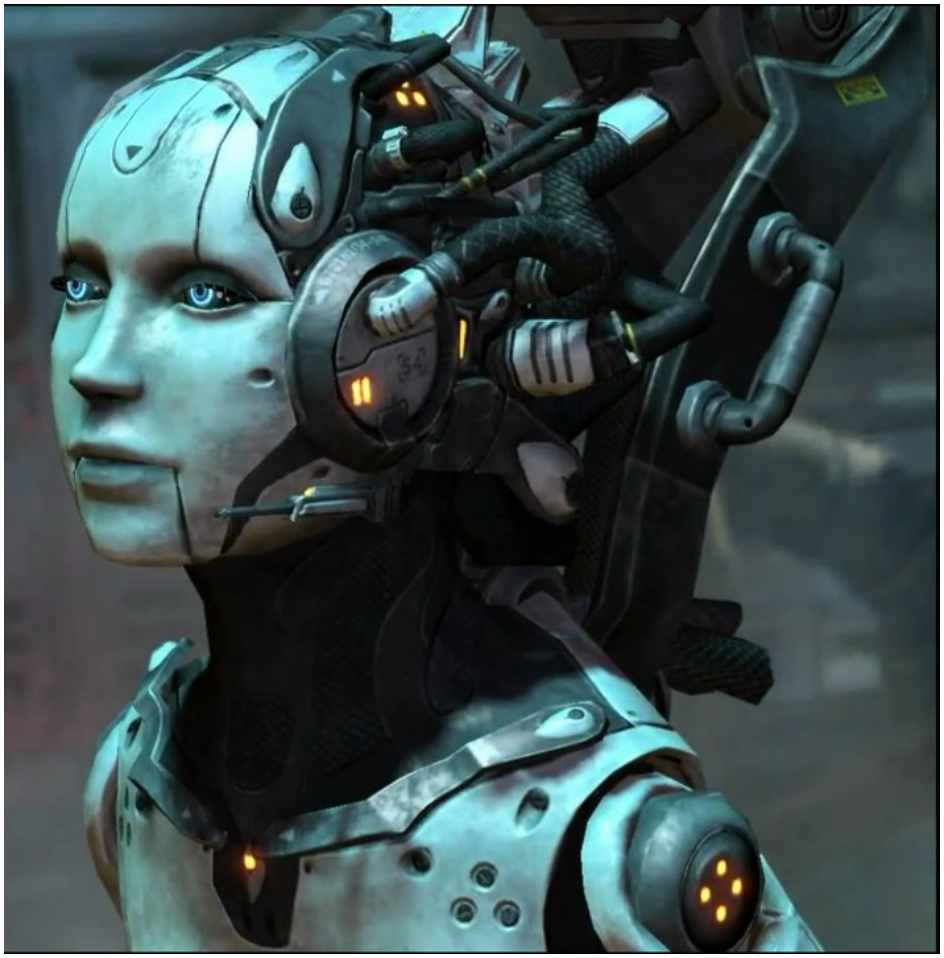


Developing a Self-learning Intelligent Agent in StarCraft II

Deep Reinforcement Learning with Imitation Learning

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Bachelor's thesis DATX02-19-83

Developing a Self-learning Intelligent Agent in StarCraft II

Deep Reinforcement Learning with Imitation Learning

Karl-Rehan Chiu Falck
Niclas Johansson
Emma Svensson
Markus Veintie
Franz Wang
Daniel Willim

CHALMERS
UNIVERSITY OF TECHNOLOGY

Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2019

Developing a Self-learned
Intelligent Agent in StarCraft II
Deep Reinforcement Learning with Imitation Learning

© Karl-Rehan Chiu Falck, 2019.
© Niclas Johansson, 2019.
© Emma Svensson, 2019.
© Markus Veintie, 2019.
© Franz Wang, 2019.
© Daniel Willim, 2019.

All figures are made by the authors, including the screenshots that are taken from the game StarCraft II developed by ©Blizzard Entertainment.

Supervisor: K V S Prasad, Computer Science and Engineering
Examiner: Carl-Johan Seger, Computer Science and Engineering.

Bachelor's thesis DATX02-19-83
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Adjutant, ©Blizzard Entertainment

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2019

Abstract

Knowledge of machine learning is becoming more essential in many fields. This thesis explores and outlines the basics of machine learning through the complex game StarCraft II with limited prior knowledge and resources. In particular deep Q-learning in combination with imitation learning was explored in order to reduce the time required for an agent to become capable of playing the game.

A few simpler environments were used as initial challenges before StarCraft II was explored. For all environments, the thesis reports a comparison of performance between the agents utilizing imitation learning and those that did not. In the cases of the simpler environments, agents using deep Q-learning combined with imitation learning showed significantly improved training time. Due to problems with the reward structure for the complex game StarCraft II no conclusion could be drawn about the implications of imitation learning in complex environments.

Sammanfattning

Kunskap om maskininlärning börjar bli allt mer väsentlig i många arbetsområden. Denna rapport utforskar huvuddragen av maskininlärning genom det komplexa spelet StarCraft II med begränsade förkunskaper och resurser. I synnerhet undersöktes djup Q-inlärning i kombination med imitationsinlärning för att reducera den tid som krävs för en agent att bli kapabel att spela spelet.

Ett par enklare miljöer användes som initiala utmaningar innan StarCraft II utforskades. En jämförelse av prestandan för de olika agenterna med eller utan imitationsinlärning återfinns i rapporten för alla miljöer. I de enklare miljöerna uppvisade agenterna som använde djup Q-inlärning tillsammans med imitationsinlärning signifikant ökad inlärningsförmåga. Dock gäller, till följd av problemen med belöningsstrukturen för det komplexa spelet StarCraft II, att ingen slutsats kan dras angående implikationerna av imitationsinlärning i komplexa miljöer.

KEYWORDS: Deep Q-Learning, Deep Q-Network, Imitation Learning, Machine Learning, PySC2, Reinforcement Learning, StarCraft II

Acknowledgements

This research was supported by Chalmers Univeristy of Technology. We thank our supervisor K.V.S Prasad, Associate Professor, Department of Computer Science and Engineering.

We thank Emilio Jorge, PhD student, Data Science and AI Division, Department of Computer Science and Engineering for his assistance, knowledge and expertise with machine learning methods and techniques.

We would also like to show our gratitude to Lars Norén, Research Engineer, Department of Computer Science and Engineering for his assistance of providing suitable equipment for the research.

Contents

Glossary

Agent is a single instance of the parameters of an *artificial neural network* together with a network architecture.

Artificial Intelligence (AI) is the simulation of intelligent behavior and human problem solving capabilities in machines and software. It includes learning, reasoning and predictive analysis.

Deep Learning is a *machine learning* method that constructs an *artificial neural network* structure similar to the function of the human brain and is used for predictive analysis.

Deep Q-Network (DQN) is a combination of *Q-learning* and *deep learning*. Replacing the Q-learning's table with a neural network that tries to predict the Q-values.

Epoch refers to one cycle of a full training session. Usually, training a neural network requires many epochs.

Imitation Learning is a branch of *machine learning*, it is the observation of an action and then the process of mimicking/reproducing it. Something that is instinctively to living beings, but which is difficult to reproduce in *artificial intelligence*.

Machine Learning is a field of computer science with the intent of teaching computers how to learn and be able to more accurately predict outcomes without being explicitly programmed.

Artificial Neural Network is a learning algorithm using a network based on the function of how neurons work together in the human brain to understand and process data.

Q-Learning is an algorithm which uses Q-values in a Q-learning table to improve an agent's optimal policy based on previous interactions and those effects on the Q-table.

Reinforcement Learning is a branch of *machine learning* and a training method based on rewarding desired behaviors and/or punishing undesirable outcomes.

Supervised Learning is a training method for *artificial intelligence* that uses known input data as well as correct output data as a learning basis for future data processes.

Unsupervised Learning is a training method for *artificial intelligence* using raw or unlabeled data that the algorithm has to act on with no additional information or guidance.

Glossary for StarCraft II

Actions per minute (APM), refers to the total number of actions a player can perform in one minute. It is used as a measurement of a player's skill.

Drone is a *Zerg* worker, collects *minerals*.

Fog-of-War is territory currently not seen by the player's units vision.

Hatchery is the *Zerg* command center, all *Zerg* units are produced from this facility.

Larva is a *Zerg* resource unit generated from the *hatchery*, all *Zerg* units are made from these, except the *queen*.

Marine is a basic *Terran* military unit.

Minerals are the primary resources in StarCraft II.

Mini-map is part of the user interface which shows the game map.

Overlord is a *Zerg* supply unit, increasing the overall supply capacity.

Protoss, one of three playable races. An alien quality over quantity race, few but powerful units..

Queen is a *Zerg* utility unit, can produce extra *larva*.

Spawning Pool a *Zerg* prerequisite building for producing the *zergling* and *queen*.

Supply Limit, a cap on how many units a player can have, this cap is extended up to 200. Can be increased for *Zerg* by building more *overlords*.

Terran is one of three playable races. A humanoid race with average quality and quantity of units.

Zerg is one of three playable races. A biological race using many units with below average strength.

Zergling is a basic *Zerg* military unit.

1 Introduction

In recent decades *artificial intelligence* (AI) has become a hot topic for research because of its unlimited potential to help and assist us, but also to outperform us in many ways. Tremendous advancements have already been made within the subject. *Machine learning* is a great tool for creating these intelligent agents and is thus a very important subject to study. Many things have already been mastered when it comes to machine learning but many complex problems are still too difficult for these algorithms to solve. In order to improve the algorithms, controlled environments are needed where the learning can be supervised and their success can be easily measured. Complex games are great such environments, where the algorithms can be developed and tested in a completely controlled way and their success is easily measured. This is seen by how much score is achieved or how many games they win, which can validate how well or poorly the agent performs. It has been proposed that StarCraft II would be a great domain for continued AI research because of its high complexity and large action space [1].

Many factors that can be found in complex games, such as situations with imperfect information, are relevant features of real life problems. By further researching and developing learning algorithms for these games, the hope is that these agents will be able to solve unexpected, real life situations better and more efficiently in the future. DeepMind is a company that has already used StarCraft II as a complex learning environment for intelligent agents [2]. Because of the limited resources available for this project, an interesting aspect is to explore if other, simpler ways, compared to how DeepMind solved StarCraft II, can be used to achieve similar results.

1.1 Purpose and Delimitation

This thesis aims to develop a self-learning, intelligent agent capable of playing the game StarCraft II with very limited resources. *Reinforcement learning*, a type of machine learning, will be used in combination with *imitation learning* to investigate how imitation learning can affect and reduce the amount of resources required for the training process.

The team has limited computing power and little prior knowledge of the subject. This is the main reason why the following delimitations will be applied in the scope of the project. Only a relatively small and simple network architecture will be tested for the learning algorithm. Using more complex networks require substantially more research and coding. Vast simplifications of the full game will also be made. These simplifications include limiting the features of the game (such as which race is played and on which map the game plays out) as well as the state and action space. As a result the data flowing through the network is more manageable with limited computing power. With these extensive delimitations the resulting agent should be able to play StarCraft II but will likely have difficulty beating advanced opponents.

1.2 Method

The project included a literature study on a wide range of basic machine learning topics as well as recent advancements made on the subject. A couple of projects focusing specifically on StarCraft II was also studied, mostly focusing on DeepMind's agent AlphaStar [2]. In order to gain further experience with machine learning, the algorithm was first developed in smaller, simpler games. Each iteration gave new insights about the network architecture and its hyper-parameters which led to improvements of the algorithm.

Frozen Lake was studied first. A simple game with perfect information with a single objective to move from start to finish on a static map containing a few obstacles. In the next iteration, the project used a mini-game called Move-To-Beacon provided by DeepMind to get accustomed to the StarCraft II environment before beginning with the full game. Through each learning platform the algorithm was improved and updated after each iteration, at the same time as the project group gained more experience using machine learning. With the recently gained knowledge the project was eventually able to build a functional agent for the full game. The results from the testing revealed what methods were useful and practical regarding the development of the agent and is discussed at the end of the report. The project was developed in Python and a complete list of libraries, versions and computer hardware can be found in Appendix C.

2 An Overview of Machine Learning

Machine learning is a wide subject within computer science with various implementations. Specific tasks are performed by algorithms and statistical models without explicit instruction. This way of defining machine learning stems from a highly cited article by Arthur Samuel from 1959 [3]. The general idea behind machine learning is to have a machine process data and learn to predict a response to it using patterns and inference. A distinction is commonly made between supervised and unsupervised learning. Implementations of machine learning where the algorithm's output is compared to known targets (correct outputs) is classified as supervised learning whereas techniques with non-predetermined outputs are typically classified as unsupervised learning [4].

Much progress has been made on the topic of machine learning in recent years. A large contribution to this is the company DeepMind, the world leader in artificial intelligence research. The company was acquired by Google in 2014 and Demis Hassabis, one of the founders, have announced that the company has the grand ambition of "solving intelligence, and then using that to solve everything else." [5]. Besides funding DeepMind other research teams at Google has made large contributions to the subject of artificial intelligence. One of the teams, Google Brain, developed a flexible data flow based programming model called TensorFlow [6] which was made open-source in 2015. TensorFlow has since become one of the most used libraries for machine learning development, foremostly used with Python. As of 2017 TensorFlow supports the high-level neural networks API Keras to allow for faster and easier implementations of neural networks [7].

2.1 Neural Networks as Learning Algorithms

A great way to develop machine learning algorithms is through artificial neural networks. The idea of these networks is to mimic the behaviour of the nerve cells (neurons) in mammalian brains, but in a much simplified way [8]. The network is made up of units (neurons) in different layers, going from an input to an output layer through any number of hidden layers, see Figure 1.

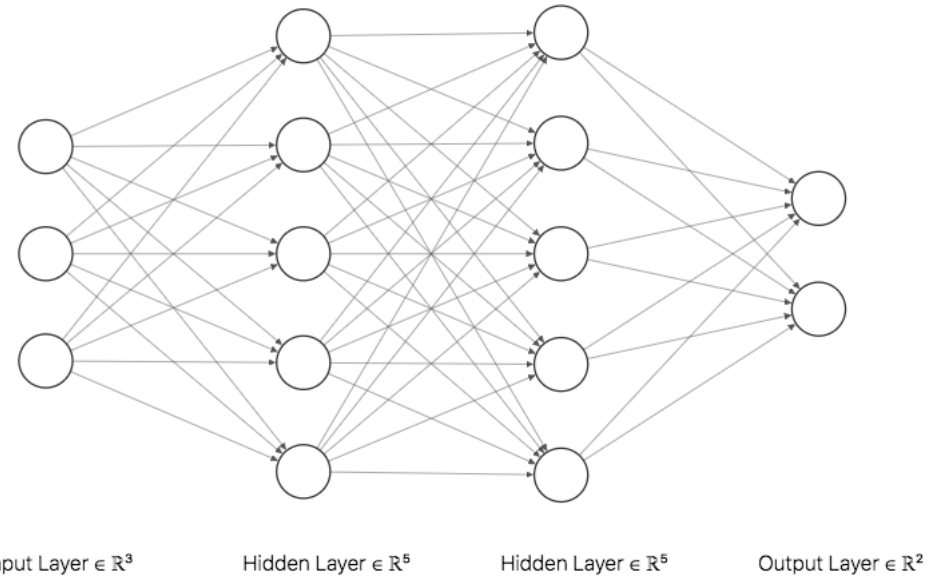


Figure 1: An example of how an artificial neural network can be constructed, with nodes feeding forward through weights (edges) from input layer (left) to output layer (right) through a number of hidden layers.

In a simple mathematical model of such network connections between units in different layers are modelled by unique weights, $w_{i,j}$, for connection from unit i to j. All inputs, a_i, to a unit, j, are multiplied by their respective weights and summed together. An activation function, g, is then applied to the sum resulting in the following expression for the output of a specific unit in the network,

$$a_j = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right).$$

As a result the inputs to a neural network can be processed and changed by each layer [4, Ch. 18.7.1]. The goal of training the neural network is thus to update these weights and other network parameters so that the output gradually improves toward the correct or optimal response called target. These parameters are then called a self-learning agent. When neural networks are applied to large data sets typically many layers are needed, with potentially millions of neurons, they are then called deep neural networks. This method can also be referred to as *deep learning* [9]. For deep neural networks the most typical way to update the weights is through *back-propagation*, where stochastic gradient-descent (SGD) is used to minimize the errors in the output defined by a *loss-function*. Each weight is then updated by

adding this gradient multiplied by a *learning rate*, α. The learning rate can be seen as the step size of the update [4, Ch. 18.7.4].

To set up a neural network in Python with Keras and TensorFlow, abbreviated tf, the following line is used.

```
model = tf.keras.Sequential()
```

In order to compile the network above a loss-function and how to optimize it has to be specified. Below the loss-function mean squared error (MSE) [10] is used together with the optimizer Adam and learning rate ALPHA. Adam is a first-order gradient-based optimizer [11].

```
model.compile(optimizer=tf.keras.optimizers.Adam(ALPHA),
              loss='mse',
              metrics=['accuracy'])
```

2.1.1 Setup of the Network Architecture

Many things need to be decided when constructing the architecture of a deep neural network, most of which have many different options each. It is important to consider the risk of *overfitting*, which can occur due to these networks' ability to learn very complicated mappings of inputs and outputs. Overfitting means that the learner has adapted to too much of the noise in the training data and there are many methods to deal with the problem. One method is called *dropout* where random units in the network are "dropped" during the training to prevent units from co-adapting [12]. Another, more simple technique that at least minimizes the risk of overfitting, is to not use unnecessarily large networks [4, Ch. 18.3.5].

There are many different ways of constructing the layers of units in a network, for example concerning how the units connect to each other between different layers. The most basic layer is the fully connected (dense) layer, where each unit from one layer feeds-forward through connections to each of the units in the next layer. The following line is used to add a dense layer with x number of units to a neural network in Keras.

```
model.add(tf.layers.Dense(x, activation='relu'))
```

The activation option will be explained shortly. Convolutional networks and cross-layer neurons are much more complex elements that can be used in the network structure. These elements have proven instrumental in any problem concerning image recognition but pose a much higher level of difficulty when training a neural network [13].

In order to add a memory aspect to a network, recurrent networks can be used. The idea here is that the network feeds its own outputs back in as its inputs. This way the network gets attributes of a dynamical system and can thus reach a stable state, exhibit oscillations or chaotic behaviour [4, Ch. 18.7.1]. Long short-term memory (LSTM) units, discovered in 1997 by Hochreiter and Schmidhuber [14], can be used in recurrent networks to achieve even better results. These units have a fixed self-connection just like units in recurrent networks but with highly complex, additional features to it. LSTM units have proven especially good in convolutional

network, according to a paper by a team at Google [15] where speech recognition performance was improved by combining a convolutional neural network, long short-term memory and a fully connected deep neural network into a unified network architecture. DeepMind's latest research project also shows that LSTM units work well for training intelligent agents in StarCraft II [2].

An important decision to make regarding the network is also what activation functions to use for each layer in the network. For the output layer the activation function controls the form of the output. If discrete values 0 and 1 are wanted one uses the heavy side function, or if continuous values between 0 and 1 are wanted the sigmoid function might be appropriate. Otherwise a simple linear function can be the best option. See Figure 2 for a few examples of activation functions.

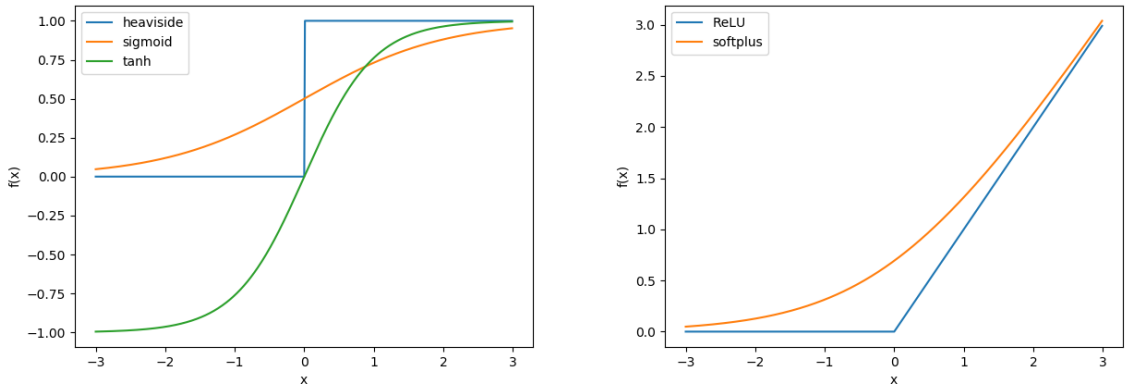


Figure 2: A few examples of functions that are typically used as activations in artificial neural networks. To the left the Heaviside, sigmoid and hyperbolic tangent functions are illustrated and to the right the softplus and ReLU functions are shown.

A function that is much used in hidden layers today is the rectifier function, units with this activation are called rectified linear unit (ReLU). This activation was used in the code-example above and can also be seen in Figure 2. The function is defined as

$$\text{ReLU}(x) = x^+ = max(0, x) \tag{1}$$

and was first introduced in 2000 from a biological stand point where the output of a neuron is an electrical current, positive if the neuron is active and zero otherwise [16]. A paper from Proceedings of the 14^{th} International Conference on Artificial Intelligence and Statistics 2011 [17] show that rectifying neurons in a deep neural network with 3 hidden layers or more, yield equal or better performance than hyperbolic tangent networks. Hyperbolic tangent is yet another established activation function, also shown in Figure 2. The paper also suggest that besides being biologically plausible rectifiers are computationally efficient.

2.2 Reinforcement Learning with Deep Q Networks

As previously stated this thesis concerns a third branch of machine learning, reinforcement learning, which is different from both supervised and unsupervised learn-

ing. Here the learner instead needs to map states (input) to actions (output) in order to maximize the received rewards yielded by the actions. This mapping is called an *action-policy* and it is a particularly useful way to learn how to play games [18]. Fundamental concepts in reinforcement learning are

- the agent which performs actions,
- the environment for example a game and
- the interpreter which translates the game state and reward from game to agent.

The most common implementation of reinforcement learning is Q-learning which predicts the expected future reward for each possible action, a_t, given the current state, s_t. The update rule or learning rule for these predictions is defined as

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)], \tag{2}$$

where r_t is the immediate reward, $\gamma \in [0, 1]$ discounts the delayed reward impact and α is the learning rate [4, Ch. 21.3.2]. A great advantage of this technique is that it includes delayed rewards and can therefore handle more long-term strategic planing, whilst at the same time discounting the impact of the delayed reward. This way the algorithm is not encouraged to wait too long for a future reward or to wait forever on a reward that may never come due to the environment terminating[18].

A challenge with reinforcement learning is the balance between *exploiting* the network's predicted best action, called greedy action, and *exploring* new actions that might have never been tested. The problem with only performing greedy actions is that optimal solution might not be found resulting in a lower long-term reward. Exploratory actions however prevents the agent from maximizing the short-term rewards and since a single action can't be both exploitation and exploration at the same time the two are in conflict. A simple way to deal with this is to choose a random action with some probability ε, called the ε-greedy policy. In many applications it is a good idea to have high exploration in the early stages of training and work toward either no exploration or only a small percentage of exploratory actions later on. Early in the training process the algorithm has not seen enough of the environment to have qualified predictions of which action is the best anyway but when most of the actions have been tested the greedy actions are more likely to be the optimal ones. For this reason ε is often chosen high in the beginning and then decreased with some function or by some step-size as the training progresses. Whether ε should be decreased to zero or not could for example depend on whether the environments conditions are static or changes. If the conditions are static there is no need for exploration once an optimal solution has been found whereas if the game keep changing some exploration might be beneficial[18].

For more complex problems deep Q-networks (DQN) are used to implement Q-learning. This is basically a deep neural network (see previous section) which takes in the current state and gives back the algorithm's prediction for each possible action, called Q-values. Consider again Figure 1, but with a state representation of the environment as the network's input space and the Q-values as its output space. In 2015 DeepMind published an article on deep Q-learning in Nature[19]. They refer

to the technique as deep reinforcement learning and suggest it has the potential to be successful in situations approaching real-world complexity. The report points out that reinforcement learning can be unstable when deep Q-networks are used but propose ways to deal with these instabilities. The instabilities mentioned are

- correlations in sequences of observations,
- significant changes to the policy due to small updates of Q and
- correlations between Q-values and target values $r + \gamma \max_{a'} Q(s', a')$.

In order to address the first two instabilities they suggest drawing random samples from previous experiences when training. This removes correlations in observation sequences and smooths over the changes in policy. DeepMind refers to this technique as *experience replay* but it can also be thought of as the algorithms memory, storing relevant information from each step and drawing the random samples from this data structure, see Algorithm 1 in Section 4.1. Regarding the third instability, DeepMind suggests that correlations between the Q-values and target values are reduced by only updating the Q-values in the target periodically.

One way to train a deep Q-network in Keras is with the fit function,

```
model.fit(state, target, epochs=1, verbose=0)
```

where target is set to the predicted Q-values for all actions besides the one that was actually performed, this action's Q-value is set according to the Q-learning rule, see Equation 2.

2.3 Imitation Learning

When trying to learn something it is often helpful to take inspiration, or learn directly, from the experiences of others. This idea has been used for many years in developments of autonomous humanoid robots[20]. The concept have also been used in combination with reinforcement learning in various fields of artificial intelligence and has come to be known as imitation learning. In surgical robotics a project was recently successful in using both human expert demonstrations (imitation learning) and reinforcement learning criterion to train their robot[21]. Another research project showed that when a learning algorithm can exploit past good experiences it can indirectly drive deep exploration resulting in better policy optimization[22].

In DeepMind's latest project[2], an agent capable of learning to play StarCraft II were developed, where they utilized imitation learning to significantly improve the performance of their agent. They implemented the imitation as supervised learning by training the network to predict the outcome of actions and also the game outcome from replays of humans playing the game. Reinforcement learning was then added to the agent which showed tremendous potential.

2.4 Visualization of a Self-learning Intelligent Agent

As learning algorithms get better, they become more and more like black boxes, libraries are available to create neural networks with limited knowledge of how they

actually work. This can easily become problematic when it comes to finding errors and debugging the algorithms. To address this issue a great way to further understand the networks is to visualize them in some way. Validation of learning algorithms might not be so difficult when it comes to supervised learning, since the algorithm's outputs are then compared to the known targets where the errors can be quantified. With unsupervised or reinforcement learning validation can be more difficult and here visualizations can be instrumental in analyzing what the agent is doing and whether this is correct or not.

These visualizations can be done either during the training process or on a trained agent's network. Another distinction can be made between visualizing the activation of neurons in the network and looking at the actual output of the network. Activation of individual neurons give a better understanding of what calculations are made within the network. This can be used to analyze how the network processes the inputs, which can lead to conclusions regarding whether the network structure is working properly[23]. With deep Q-learning, the Q-values (network's output) are a measure of how the network values different actions given the current state (input). These values can be directly visualized as a heat map portraying what the network is thinking. If this is done during the training process it can give insight about whether the agent is learning anything at all and if applied to the trained network it shows exactly what the agent has learned. In a situation where there are correct actions to take in a given state, one can then conclude whether or not the agent knows what these correct actions are. Here, the visualizations work as a tool for validation. On the other hand if there are no correct actions to take, these heat maps can serve as a way to analyze the strategies that agent has learned[2].

3 StarCraft II as a Learning Environment

Games have proven to be appropriate environments to develop machine learning algorithms in. Here they can be tested in a completely controlled way and their success is relatively easy to measure and validate. Recent projects have focused on the much more complicated game, StarCraft II [2], [24]. There are many useful resources available supplied by the developer of StarCraft II, Blizzard Entertainment, such as the API and the learning environment PySC2 that can assist with the research. The reason that StarCraft II is a great environment for improving machine learning algorithms is that it is a real-time, long-term strategy game with imperfect information. There are many new challenges to explore here. The following sections will explain the game and the progress that have already been made by other research projects.

3.1 The Complexity of StarCraft II

StarCraft II is a *real-time strategy game*, developed by Blizzard Entertainment. The overall goal in this game is to defeat the opponent by destroying all their buildings, which is achieved by amassing resources, exploring the map, building facilities and units, and then assault the opponent.

Classic games consists of two identical factions that compete against each other, for example the black and white pieces in Chess or Go. However in StarCraft II there are three different and distinctive races to choose from with pros, cons and different playstyles.

In contrast to perfect information games like GO or Chess where you can see all your opponents pieces and moves. Using this information a player can easily react and plan a counter strategy. Compared to StarCraft II, first of all it is a game with imperfect information, where parts of the map is hidden by a so called *fog-of-war*, see Figure 3. The only information available is whatever your own units can see around them. This makes the game far harder to predict and it is difficult to gain information about the opponents whereabouts, bases of operation, units, etc. It is required to explore and scout the opponent in order to gain any valuable information. This makes the game more complex and means that the player has to constantly predict what the opponent is doing, adapt to new situations and change strategies.

Figure 3: The mini-map of StarCraft II, the white window in the bottom right corner is the players camera, the red dotted area is unseen territory, the *fog-of-war*. The blue icons are resources and the green are player objects.

Secondly, the game is not turn based like Chess. StarCraft II is played in real time, which means that moves and actions are made at the same time between the players. This means shorter reaction time which can affect the decision making. Because the player does not have to wait for the opponent to make their moves, more proactive decisions can be made. The players are only limited by how many actions per minute (APM) they can make. The amount of APM a player can produce usually indicates the level of skill, where a higher APM is usually better. Professional players have an average of approximately 190+ APM [25], which is probably the higher limits of a human player's capabilities.

Another factor in StarCraft II is the large action space. There are approximately 10^8 combinations of different possibilities to use the cursors point-and-click interface [2]. A player has many different options to play with, building workers, units or structures, unique abilities with certain units and full control of their movements and actions are just a few examples. Within this large action space there are also countless of "useless" actions that does not provide any additional advantage for the player. For example trying to build something but having insufficient resources or

trying to traverse over an obstacle such as water with land locked units.

Lastly, short- and long term decisions and rewards need also to be taken into consideration in a complicated game such as StarCraft II. The length of a StarCraft II game is mostly undetermined due to the various strategies and counter strategies that can be played. Actions made in the early stages of the game can affect the later stages, thus it is hard to foresee if initial moves have good or bad consequences. An example is in the early stages of every game when one has to choose whether to expand early or not, which allows greater access to resources but leaves you vulnerable to enemy attacks. These are important decisions with opportunity costs that can have short- or long term consequences.

3.1.1 StarCraft II Gameplay

The gameplay of a StarCraft II game can be divided into two parts, macro management and micro management. Macro refers to the overall planning and decision making of a player which includes the economical and production management. Micro is the control of troops and the combat aspect of the game which is more local.

Figure 4: The players user interface for StarCraft II showing basic units and structures of the race Zerg.

Macro management is an important part of the game and is based around the players economy and management of the player base, see Figure 4. Usually the player with more resources has a higher probability of winning due to being able to produce more. This constitutes to gathering resources with worker drones, where minerals is the primary resource in StarCraft II. The decision on when to expand to a new source of income is also part of this and is critical in order to keep the production going. The other aspect of macro is the production management, it is essential to produce drones as they are the work force but also overlords as they

increase the supply count which allows the production of more units. Although all units are produced from the hatchery, certain buildings are required to unlock more advanced units. Constructing the spawning pool will unlock the first military unit that Zerg can produce, the zergling.

When it comes to micro management, combat units need instructions to perform actions. They need orders such as where to go or attack and which units need to go, without input they will just stand by idly. This is where critical decision making comes into play, the player has to determine for example which path its units should use with the limited information available to it because of factors such as fog-of-war. During encounters with hostile units the player has to decide if it can win or not and if it would be worth the losses.

3.2 Previous Machine Learning Research in StarCraft II

Much progress have already been made with machine learning in StarCraft II. Most recently DeepMind demonstrated their AlphaStar agents which managed to defeat professional human players 10-1. AlphaStar was composed of several different interconnected networks with different architectures and focus areas [26]. The networks were fed a combination of spatial information such as screen and mini-map features and non-spatial information such as mineral count [2].

Figure 5: The user interface for PySC2 displaying different information layers. On the right side are the spatial layers representing different types of information from the camera and mini-map. On the left some of the non-spatial information can be seen, for example number of minerals.

DeepMind used a combination of imitation learning and a league system to develop the AlphaStar agents. Imitation learning was used in order for the agents to reach intermediate capabilities. After this the agents were admitted into the league where they competed against each other and those who won progressed through the

league, facing more capable agents whilst the ones who lost frequently were removed from the league and replaced by mutations of successful agents in the league [27]. All in all, the average agent was trained with up to 200 years worth of game time [26].

As a part of DeepMind's research into StarCraft II PySC2 was developed, an API used for machine learning in StarCraft II. PySC2 provides information about the state of the game and actions for interacting with the game. The information is presented through different layers. There are spatial layers for information you can observe in the game and non-spatial layers for information presented in the GUI of the game [2]. See Figure 5 for the graphical representation of the layers.

Another example of progress that have already been made with machine learning in StarCraft II is the agent TStarBot1 developed as a joint project between Tencent AI Lab (China), University of Rochester (USA) and Northwestern University (USA). Within this project they made extensive research on the action space of the race Zerg. They combined all of the race's possible actions into 165 macro actions. The purpose of these macro actions is to simplify the otherwise all too complicated action space and allow for more manageable training times for agents [24].

4 Development of Deep Q-learning Algorithm

The following sections go through the development of the self-learning algorithm as a process through a number of learning environments. First the general, baseline algorithms is described. After this two simple learning environments are studied, with different input and output spaces, for the purpose of testing the algorithm in a few different settings and finding the optimal hyperparameters. The two games are; Frozen Lake and Move-To-Beacon.[1]With the latter being a very simple variant of the full game of StarCraft II focusing on a single task from the full game. At the end of the section the full game is explored, the simplifications applied are explained in detail and the result is presented. Throughout all three environments a deep Q-network was used with the baseline algorithm, initialized and compiled as in the code examples in Section 2. Only dense layers were used, with ReLU activation for all layers except the output layer which had a linear activation. The number of hidden layers and neurons used in each layer is specified for each environment throughout the section.

4.1 Baseline Learning Algorithm

A general overview of the learning algorithm used to train this project's agents can be seen in Algorithm 1. For each epoch a single round of the game is simulated, Line

[1]Tic-Tac-Toe was also tested as a learning environment for the purpose of pinning multiple agents against each other. The problem was solved and the trained agents could achieve perfect game play (all games ending in a tie) when tested against perfect opponents. Perfect results were however not reached every time due to the possibility of one agent getting the upper hand early on and thus resulting in no progress for either of the agents. It was decided to not proceed with multiple agents playing against each other and thus these results are omitted from the thesis.

5–15, and after this the agent's parameters are updated, Line 17–23. This update is what can be called the training of the agent.

The algorithm chooses actions based on the ε-greedy policy. Recall from Section 2.2 that this policy deals with balancing exploration and exploitation. During exploration the action is chosen randomly and during exploitation the action is predicted based on the agent's past experience. As stated in Section 2.2, exploration is needed when the agent has yet to experience many different states, its prediction of the rewards is inaccurate and the results of different actions needs to be explored. As the training goes on the agents predictions become better and therefore ε is decreased by $\varepsilon_{\text{decay}}$ at the end of each epoch. Do note however, once you stop exploring, training might appear to slow down or stop completely altogether as the network converges to an optimal solution and thus the parameters change at a slower rate or not at all.

Algorithm 1
Learning algorithm for deep Q-network with ε-greedy action-policy

1. Initialize network parameters $w \sim U(0, 1)$
2. Initialize memory $\mathcal{M} \leftarrow \emptyset$ with capacity N
3. Initialize done $\leftarrow$ **false**
4. **for all** epochs **do**
5. **while** game **not** done **do**
6. Read state, s_t, from game environment
7. **if** exploration **then**
8. Random action, a_t
9. **else if** exploitation **then**
10. Let agent predict next action, a_t
11. **end if**
12. Perform a_t
13. Receive reward, r_t, and observe new state, s_{t+1}
14. Store step in $\mathcal{M} \leftarrow \mathcal{M} \cup \{s_t, a_t, r_t, s_{t+1}, \text{done}\}$
15. **end while**
16.
17. Set mini-batch $\leftarrow$ random sample of steps from $\mathcal{M}$
18. **for all** steps $\in$ mini-batch **do**
19. Set target $\leftarrow$ predicts all Q-values for s_t
20. Set target$[a_t] \leftarrow r_t + \gamma \times \text{argmax}_a$(predicted Q-values for s_{t+1})
21. Fit s_t to target
22. **end for**
23. Decrease ε by $\varepsilon_{\text{decay}}$ as $\varepsilon* = \varepsilon_{\text{decay}}$, until $\varepsilon \leq \varepsilon_{\text{to}}$
24. **end for**

The game is simulated in steps where one step is one action. For each step the state, s_t, is observed, Line 6, and an action, a_t, is chosen according to the action-policy, Line 7–11. The agent then performs the action on the environment and receives a reward, r_t, and a new state, s_{t+1}, is observed. Afterwards, the step is stored in the algorithm's memory, Line 14, to be used during the training process.

The purpose of the memory is to deal with the instabilities that can otherwise come with deep Q-learning, see Section 2.2.

The training, beginning at Line 18, starts by randomly selecting a sub-set of steps, called a mini-batch, from the memory, Line 17. For each of the steps in the mini-batch a vector, target, is first set to the agent's predicted Q-values for s_t, Line 19. After this the Q-value in target corresponding the action that was actually performed, a_t, is updated using the Q-update rule, Equation 2 in Section 2.2, Line 20. Target is then used to update the network using back-propagation, using the **fit**-function in Keras, Line 21 in the algorithm.

4.2 Frozen Lake

First the algorithm was applied to the learning environment Frozen Lake. This game consist of a 4×4 static map, shown in Figure 6, which the player must cross to reach the goal without falling into holes in the ice. A neural network with one

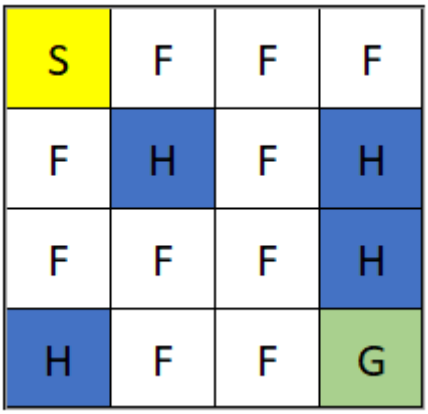


Figure 6: Static map of game board for Frozen Lake. The player should move from start, "S", to goal, "G", via only frozen patches, "F", avoiding holes, "H".

hidden layer, consisting of eight neurons, was used. The state representation was chosen to be the current state of the board, 16 input nodes, one for each tile in the game board. The state was also transposed to a 16×1 vector with a one representing the players position and zeros everywhere else. The action space was limited to four actions, since the player can only move in the four directions; up, down, right, left. The reward structure was setup so that the agent received one point if it reached the goal and zero points if it fell in to a hole or moved outside of the map along the way.

4.2.1 Testing Hyperparameters and Validating the DQN Agents

To find the best hyperparameters in Frozen Lake, regarding $\gamma, \varepsilon_{\text{to}}$ and mini-batch size, agents were trained with each combination of hyperparameter values from Table 1. After 200 completed epochs of training the agents were validated by playing the game 10 times with $\varepsilon = 0$. Five training sessions were performed for each combination of hyperparameters and all results in the section are averages over

these five runs for the purpose of eliminating some of the variance in the results. For all tests the memory capacity was set to 1000, $\varepsilon_{\text{from}} = 1$, $\varepsilon_{\text{decay}} = 0.995$ and $\alpha = 0.001$.

The training of the agents is visualized in Figure 7, as accumulated wins as a function of epochs. A win refers to the number of times the goal is reached by the agent. The figure shows the average results for all combinations of hyperparameters but the five best results are highlighted and their respective hyperparameters are stated in Table 1. An agent based on a random action-policy is also shown in Figure 7 as a baseline. The result from the validations can be seen as *heat maps* of the win rate. There is one heat map for each mini-batch size with respect to ε_{to} and γ. Only the ones for size 2 and 32 are shown in Figure 8, the full set of heat maps can be found in Appendix A.

Table 1: Values tested for hyperparameters γ, ε_{to} and minibatch size in Frozen Lake and the five best combinations of them based the result in Figure 7.

Hyperparameter	γ	ε_{to}	Mini-Batch Size
Values Tested	0.5:0.1:1	0:0.1:0.3	2,8,16,32,64
DQN Agent 1	0.7	0	16
DQN Agent 2	0.7	0	32
DQN Agent 3	0.8	0.1	64
DQN Agent 4	0.8	0	32
DQN Agent 5	0.9	0	32

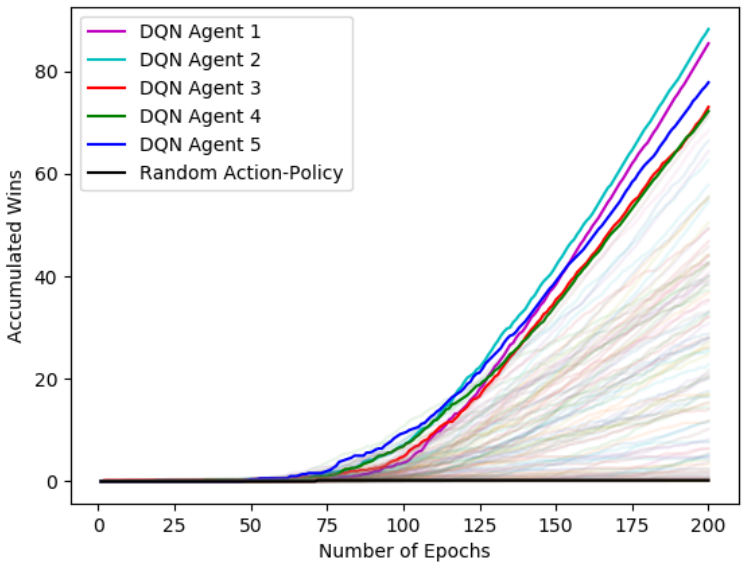


Figure 7: Accumulated wins from training DQN agents in Frozen Lake with different combinations of hyperparameters as stated in Table 1. The five best agents are highlighted and an agent based on a random action-policy have also been added as a baseline. *No prior imitation learning.*

It is clear in Figure 7 that the five best agents struggle initially during training, but once they have found a way to reach the goal they simply repeat this sequence of actions and keep on winning from this point on. This happens roughly around epoch 100 and after that the function is more or less a linear identity mapping. The result of the remaining agents are scattered from learning nothing to learning almost as good as the five best agents. The highest total score was achieved with $\gamma = 0.7$ and $\varepsilon_{\text{to}} = 0$. That ε should be decreased to 0 is not a surprise since once the agent has found a correct solution there's no need for further exploration. In this learning environment it also makes sense to use a γ which is not too high, since the goal should be reached in as few steps as possible. With no discount on delayed rewards ($\gamma = 1$) the algorithm would have no understanding of the fact that running around in a circle a thousand times before reaching the goal is not a good idea. Extended time running around on the game board with $\varepsilon \neq 0$ significantly increase the risk that exploration drives the player into an obstacle resulting in a loss.

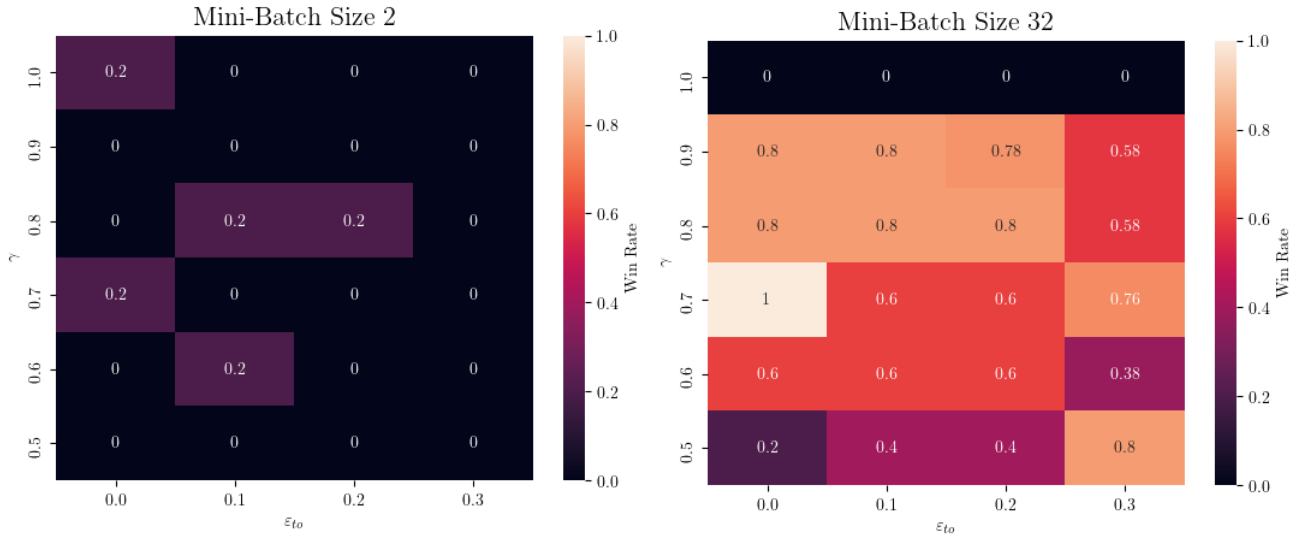


Figure 8: Heat maps of the win rate from 10 validation games for the DQN agents in Frozen Lake. The result for mini-batch size 2, (a), and mini-batch size 32, (b), has been chosen as they show the best and worst results. The full set of heat maps, for all mini-batch sizes that were tested can be found in Appendix A. *No prior imitation learning.*

The conclusions drawn from Figure 7 are corroborated by the heat maps in Figure 8 and Appendix A. Results are terrible for $\gamma = 1$ and the best agents from the training also outperforms the other agents during the validation. Regarding the size of the mini-batch the results are similar for all sizes of 8 and above. The best overall results are found with mini-batch size 32, which got a perfect score with $\varepsilon_{\text{to}} = 0$ and $\gamma = 0.7$ (same as for training). Mini-batch size 16 also show some perfect scores, see Appendix A. The minimum number of steps needed to reach the goal on the map used here is six, thus it seems reasonable that a mini-batch size of less than six could be problematic. With a mini-batch of 2 sample steps from the

game the training doesn't have any chance of taking a full game into consideration and ultimately it becomes hard for the agent to learn anything at all. This can be seen in heat map (a) in Figure 8, as the trained agent achieves close to no results at all during validation.

4.3 Move-To-Beacon

Next the game Move-To-Beacon was used as a learning environment. This is a mini-game of StarCraft II released together with PySC2 by DeepMind [2]. Move-To-Beacon focus on a single task of the full game, namely to control one unit (marine) and to move it to a specific location. This location is indicated by a beacon on a map represented as a 40×40 matrix, see Figure 9. When the marine is moved into the beacon area the player gets one point and the beacon is moved to a new random location, this continues for a set period of time and at the end the player is rewarded with the score it accumulated during the time period.

A neural network with one hidden layer, consisting of 1600 neurons, was used. The coordinates of the map were used to represent the state fed as inputs to the network, pre-processed and transposed into a 1600×1 vector. The state was also pre-processed such that a piece of the beacon were represented by a one and all other coordinates represented by zeros. Similarly, the action space was chosen to be which coordinate on the map to click next, transposed to a 1600×1 vector in accordance with the state representation. This can be considered a spatial-action policy. The reward structure followed the in-game score, receiving one point each time the beacon was reached.

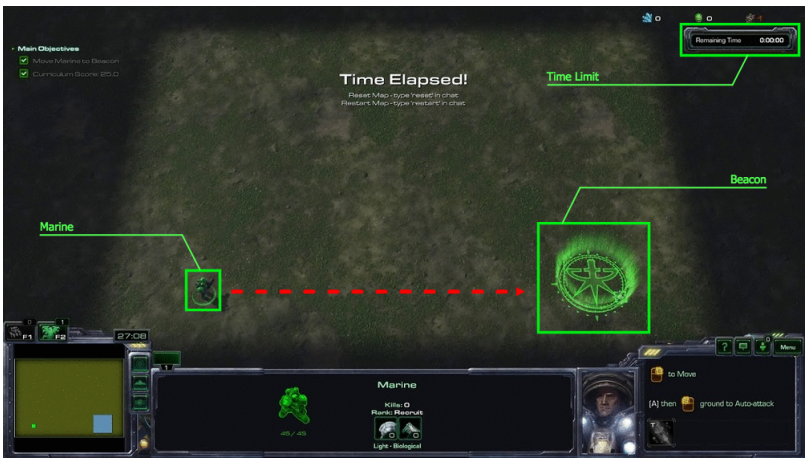


Figure 9: In Move-To-Beacon the goal is to move the marine to the beacon (red dotted line). Reaching the beacon is rewarded with one point and the beacon is moved to a new random location. The player can continue to move the marine until the time is up.

4.3.1 Testing Hyperparameters and Validating the DQN Agents

To find the best hyperparameters in this learning environment agents were trained and validated four times with each combination of values from Table 2. The results

throughout the section are averages over the results for these four agents. The agents were trained for 2000 epochs with a memory capacity of 200, $\varepsilon_{\text{from}} = 1$, $\varepsilon_{\text{decay}} = 0.9985$ and $\alpha = 0.001$. And after this they were validated through 100 games with $\varepsilon = 0$.

Table 2: Values tested for hyperparameters γ, ε_{to} and mini-batch size in Move-To-Beacon and the five best combinations of them based the result in Figure 10.

Hyperparameter	γ	ε_{to}	Mini-Batch Size
Values Tested	0:0.25:1	0:0.1:0.2	32,64,128
DQN Agent 1	0.25	0.1	128
DQN Agent 2	0.25	0.2	64
DQN Agent 3	0.5	0.1	128
DQN Agent 4	0.5	0	128
DQN Agent 5	0	0.1	64

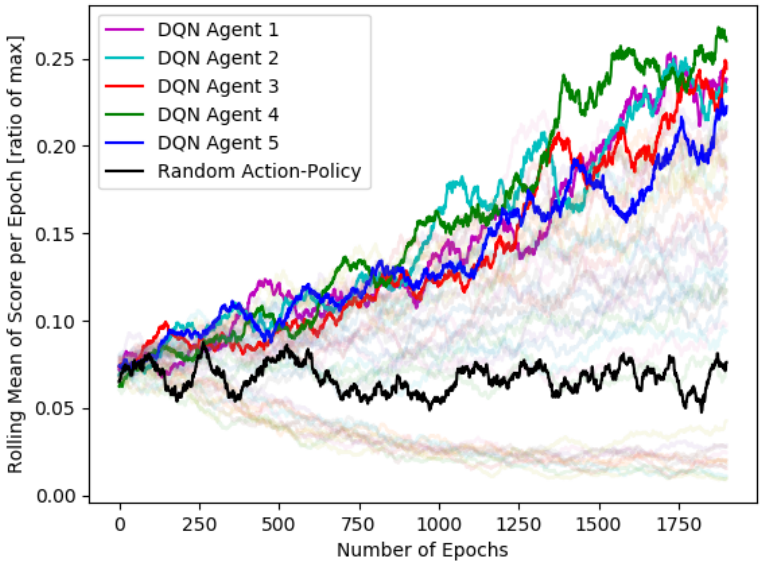


Figure 10: Rolling means, with a window of 100 epochs, of in-game score per epoch as a ratio of the maximum score per game from training DQN agents in Move-To-Beacon with different hyperparameters as stated in Table 2. The five best agents are highlighted and an agent based on a random action-policy have also been added. *No prior imitation learning.*

The result from training the agents is visualized in Figure 10. In the figure the five best results are highlighted and the values of their respective hyperparameters are stated in Table 2. A *rolling mean*, with a window of 100 epochs, has been chosen to show the results. This mean is taken over the in-game score received per epoch and as the ratio of the maximum score for a single game or the maximum number of times the beacon can be reached within the time span of a single game. The results from validating the agents is visualized as heat maps in Figure 12. One for each

mini-batch size, showing the average ratio of maximum in-game score with respect to ε_{to} and γ.

The agents in Figure 10 who's score decreases as the training progress have the common denominator that their γ is one or close to one. In the validation data the same trend can be seen, with poor performance for $\gamma = 1$ and $\gamma = 0.75$. It can also be seen in Figure 10 that none of the agents have learned a perfect, or close to perfect, action-policy after 2000 epochs of training. The performance of the best agents has reach 25% of full score but it is still progressing as this point and there's no indication that the learning is slowing down. In order to prove that the agents have the potential to reach better performance given enough time to train a longer run was performed for the DQN Agent 4 in Figure 10. In this test the agent was trained for 200000 epochs and the result is shown in Figure 11 as a rolling mean with a window of 2000. The agent's performance is not a 100% even after training for this long but it has reached almost 70%, which is a much better result. It should also be noted that, 200000 training epochs are still only a fraction of the training time that DeepMind's agents went through. Their agents reached perfect gameplay in Move-To-Beacon [2], but Figure 11 shows that almost as good results can be achieve with significantly less resources.

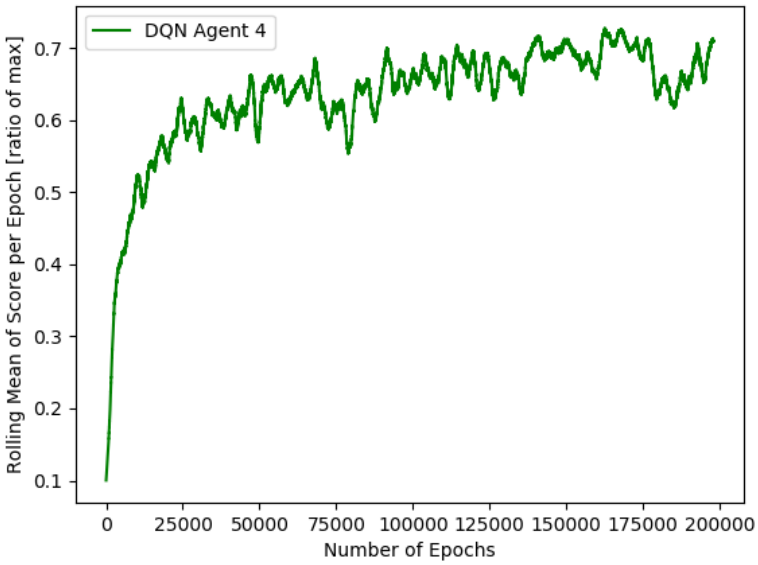


Figure 11: Rolling mean, with a window of 2000 epochs, of in-game score per epoch as a ratio of the maximum score per game from training DQN Agent 4 from Figure 10 for 200000 epochs. *No prior imitation learning.*

From both the results of training and validation the best performance was achieved with $\gamma = 0.5$, $\varepsilon_{\text{to}} = 0$ and a mini-batch size of 128. Yet almost as good were $\gamma < 0.5$ and mini-batch size 64 with various ε_{to}. The most notable difference from Frozen Lake is that a much lower γ is optimal for Move-To-Beacon. A possible explanation for this is that in Move-To-Beacon the next state after a beacon is reached, can't be foreseen because of its random re-placement on the map, and so

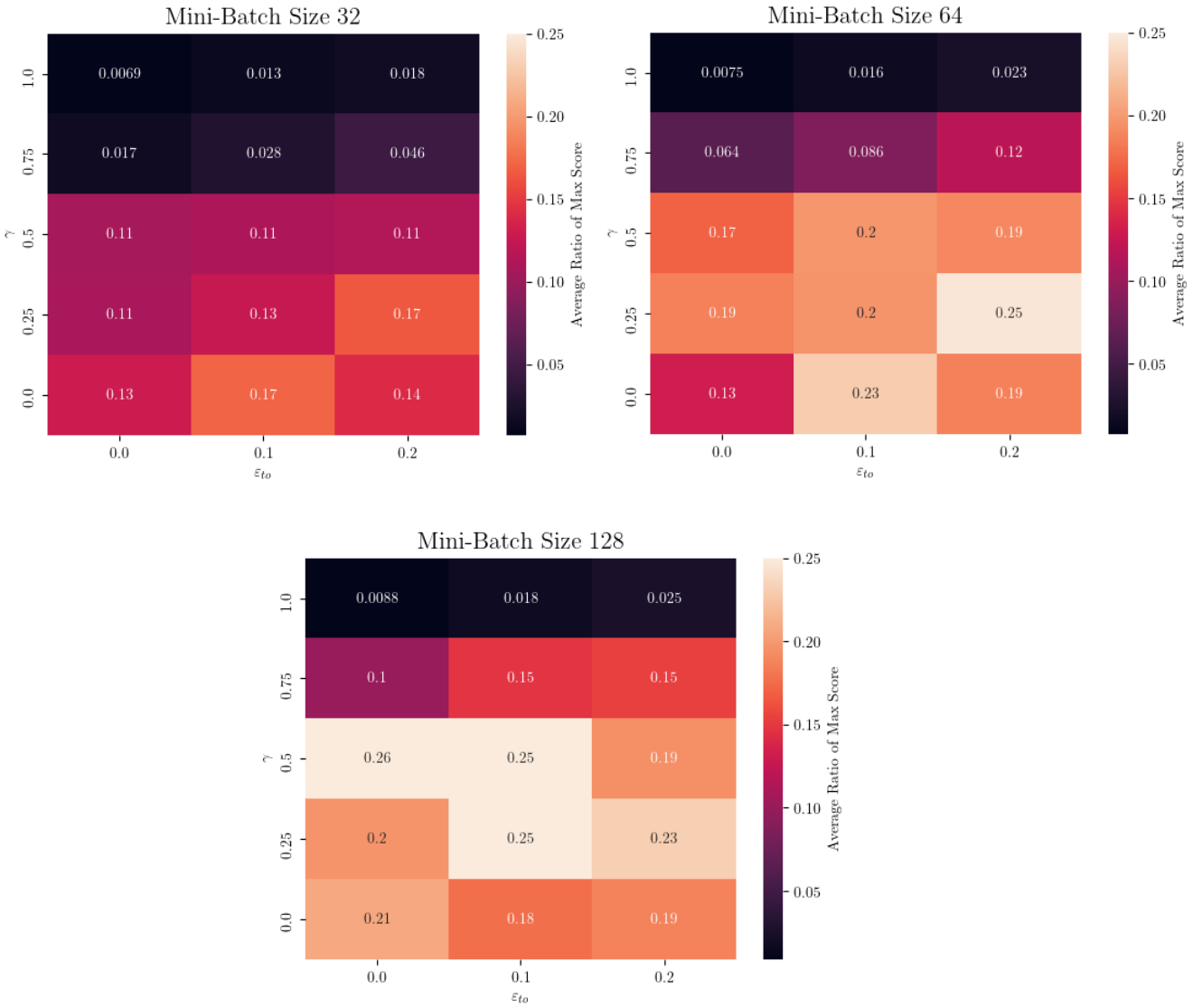


Figure 12: Heat maps of in-game score divided by maximum score per game, from 100 validation games for the agents in Move-To-Beacon. One for each mini-batch size with respect to ε and γ. *No prior imitation learning.*

future rewards aren't as important in this case. The player is however allowed to take any number of steps before reaching the beacon, whereas the optimal solution is always only one move to reach each beacon. This might explain why γ shouldn't be too low either.

Regarding the mini-batch size, a much larger size is optimal in Move-To-Beacon compared to in Frozen Lake. This is not surprising since the state and action spaces are much larger in this environment. It might even be the case that even larger mini-batch sizes are optimal but this is a trade off with the training times and was not tested in the scope of this thesis. When it comes to ε the result is not clear about which value is optimal since the difference in performance for the different values are similar between all three of them. Recall that the function of ε_{to} is to

decide whether or not to keep exploring during the later stages of the training. If there exists a correct action-policy, ε should likely be decreased to zero but otherwise $\varepsilon_{\text{to}} > 0$ can be optimal. One should also aim for ε to be ε_{to} only when the training has come far enough that a reasonably good action-policy has been reached. In Move-To-Beacon there is a correct action-policy (clicking directly on the beacon) and so a small ε_{to} should be optimal, but in the results shown here none of the agents had achieved especially good action-policies after the training was finished and so the value of ε_{to} isn't of much interest.

Also worth noting is that, even though Move-To-Beacon is an environment with a correct action for every state, the map consists of 1600 pixels, with a beacon of approximate size 6×6 thus the number of possible states is quite large (greater than 10^3). With a simple network (such as the one used here) it seems like the agent has to have seen at least close to every state in order for it to have a reasonable prediction about it. This might explain why the over all results are so poor for all agents and why such long training time seem to be needed for the agent to learn this game.

4.4 The Full Game of StarCraft II

Finally the learning algorithm was applied to the full game of StarCraft II, but with massive simplifications to the state and action spaces, as explained below. In this environment, one hidden layer with 16 neurons was used in the deep Q-network. The reason for using less neurons compared to Move-To-Beacon is that with the applied simplifications to the states and actions the total number of input and output neurons were much smaller than for Move-To-Beacon. Recall that Move-To-Beacon had 1600 inputs and 1600 outputs. Another simplification was made in this environment, namely that the agents were only allowed to play one of the races, Zerg.

4.4.1 Simplifications of State Space and Action Space

Using a spatial state space for the full game, as was used for Move-To-Beacon, was deemed too demanding for this thesis and instead a very limited non-spatial state representation was used. The state contains some values representing the game. An explanation of these values are listed below in the same order as they are contained. They only represent the players own progress and thus gives no perception of the opponent and no visual context of the game. This due to the representation being non-spatial.

0. **Minerals:**
   Minerals are used to build structures and produce units. They can be gathered from mineral veins with drones.

1. **Supply Limit:**
   Supply limit is the maximum supply a player can have. This value can be increased by producing overlords.

2. **Supply Used:**
   Some units require a set amount of supply. This value is the total amount of supply currently used.

3. **Supply for Military Units:**
   The amount of supply currently be-

ing used by military units.

4. **Amount of Workers:**
   The number of drones.

5. **Amount of Military Units:**
   The number of military units.

6. **Number of Spawning Pools:**
   The number of spawning pools. Because there is only one place the agent can build a spawning pool, this value can only be 0 or 1.

7. **Number of Queens:**
   The number of queens.

8. **Amount of Energy:**
   The amount of energy from the queen with most energy.

The action space was also simplified to a small sub-set of the full action space. The set of actions was made up of a number of custom made, high-level macro actions that the agent could choose from. Each macro action accomplished one task and was made out of several actions from the PySC2 API. PySC2 actions can interact with the game in different ways, but in order for something useful to happen a particular sequence of these actions are required. When the agent calls a macro action, a number of PySC2 actions gets stored in a queue. The number of actions stored depends on the macro action and the order is predetermined. At each step the next action in the queue is executed until there are no more actions and the agent chooses another macro action. A description of every macro action can be found in the list below with the same index as used in the code.

0. **Do Nothing:**
   This is important to have because feedback from the game is not given immediately. So the agent need to be able to wait and not be forced to do something at each step.

1. **Build a Drone:**
   Creates a worker drone from a larva selected through the hatchery, the main Zerg production facility. Requires 50 minerals, one larva and one supply. With this action the agent can expand the economy by increasing the rate resources can be harvested as well as construct buildings. *Requires 3 steps.*

2. **Build an Overlord:**
   Creates an overlord from a larva selected through the hatchery. Requires 100 minerals and one larva. For the agent to be able to create more units, a higher supply capacity is required, which you get 8 units of from the overlord. *Requires 4 steps.*

3. **Build a Spawning Pool:**
   Selects a random drone and builds a spawning pool at a predetermined position. Requires 200 minerals and one worker. The agent needs an army to be able to defeat the enemy and the simplest military units requires a spawning pool as a prerequisite. *Requires 2 steps.*

4. **Build Zerglings:**
   Creates two zerglings from a larva selected through the hatchery. Requires a spawning pool, 50 minerals, one larva and one supply. Zerglings are the basic army unit. *Requires 4 steps.*

5. **Attack Opponent's Main Base:**
   Selects every army unit and attacks the enemy's main base. To be able to win the agent need to destroy the enemy's buildings. To do that, an action for sending an attack command to the agent's army aimed at those buildings is required. *Requires 2 steps.*

6. **Attack Opponent's Second Base:**
   Selects every army unit and attacks the enemy's (potential) second base (also known as the enemy's natural expansion). To be able to win the agent needs to destroy the enemy's buildings and to do that an action for for sending an attack command to the agent's army to those buildings is required. *Requires 2 steps.*

7. **Defend Player's Second Base Site:**
   Selects every army unit and attacks the site of the closest resource point to our base. In normal games this area would be referred to as the player's natural expansion. The action is sometimes needed as the enemy may decide to expand or rally their armies there. *Requires 2 steps.*

8. **Defend Player's Main Base**
   Selects every army unit and sends an attack command aimed at the agent's main base. Used in case the enemy raids the agent's main base when its own army is on the offensive in order to protect it's vital structures. It performs somewhat less than satisfactory however as the army will attack any enemy unit it meets on the way back lacking any form of disengagement and thus may get held up by those things while the main base falls resulting in a loss. *Requires 2 steps.*

9. **Build a Queen:**
   Creates a queen from the hatchery. Requires a spawning pool, 150 minerals and two supply. Queens are useful for defending the base and to increase unit production. *Requires 2 steps.*

10. **Spawn Larva:**
    Creates 3 larva by selecting a queen and uses the ability "Spawn Larva" on the hatchery. Requires a queen. With more larva more units can be produced. *Requires 2 steps.*

Not all of these macro actions are viable at all times during a game. Some need to be preceded by others and some have requirements regarding current supply of minerals for example. These dependencies were not hard-coded together with the macro actions. Instead illegal actions resulted in no action performed. It was then left to the agent to figure out which actions can be performed during the training process.

4.4.2 Reward Structure and Results

Even with the added simplifications the training was very time consuming in this environment due to the game interface. Thus no testing of hyperparameters was done. Instead the results from the simpler environments helped in deciding reasonable values to use here. Regarding ε_{to}, this is an environment without a correct solution meaning exploration should not be entirely eliminated and so $\varepsilon_{\text{to}} = 0.2$ was chosen. Future rewards should be accounted for and thus $\gamma = 0.9$ was used. It was observed that the agent made approximately 400-900 actions in this environment, but only half of these actions were saved in the memory. For this reason the mini-batch size was set to 512 in order to stay above the number of actions stored in the memory for a single game, as per the results from Frozen Lake. In addition, the memory capacity was set to 5000, $\varepsilon_{\text{from}} = 1$, $\varepsilon_{\text{decay}} = 0.99$, $\alpha = 0.001$ and the training was done over 300 epochs.

Three types of reward structures were tested in this environment, described in the list below.

- **Sparse reward (1/0):**
  Giving all action one point if the game was won and zero points otherwise. Same as was used in Frozen Lake.

- **In-game score:**
  Giving each action the difference in in-game score between the new state and the previous state of that action, regardless of game outcome. The in-game score is based on the amount of resources that a player has, creating more units and buildings generates more points and loosing units impacts the score negatively.

- **In-game score, added by 2000 if game won:**
  Same as in-game score reward, but also adding 2000 to each reward if the game was won.

The result from training DQN agents with these three reward structures can be seen in Figure 13 together with an agent based on a random action-policy as a baseline.

With a sparse reward the agent won no games at all, the same result as for the random agent. The most likely explanation for this is that in such a complex environment, with a vast number moves done each game, the probability of winning a game on chance is very small. With this reward structure a win would be needed for the learning to start since without wins the network is never updated as all rewards are zero. It might be the case that the agent would have learned something given much longer time to train but this has not been proven.

The best result was achieved with the in-game score reward structure, the agent won its first game already after about 25 epochs. After this the agent kept winning occasionally up until around epoch 120. Note however that, after this point the agent won only a few times for the remainder of the training period. A possible explanation for why this agent stops winning can be found by looking at the final score received at the end of each game throughout the training process, shown Figure 14. This figure shows a rolling mean of the in-game score per epoch, with a

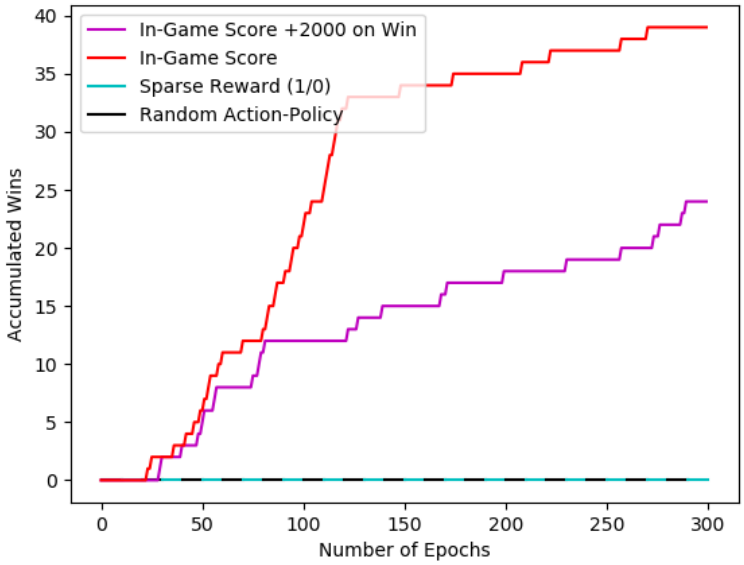


Figure 13: Accumulated wins from training DQN agents on the full game of StarCraft II, based on three different reward structures. An agent based on a random action-policy has been added as a baseline. Note that the result of the random agent is identical to that of the DQN agent based on a sparse reward structure. *No prior imitation learning.*

window of 10 epochs. The wins and draws have been marked with green and yellow dots respectively and the random agent's result is also shown. It is clear in this figure that the agent is learning something, to maximize the score. However, the score goes down when attacking the opponent due to losing units and so eventually the agent learns that a defensive strategy is better for this purpose which means a significantly smaller chance of winning in a game like StarCraft II. In conclusion this reward structure is fine to train an agent with in the game but it does not teach the agent to win since it gives no feedback about the game outcome.

The third reward structure was constructed in an attempt to fix the problems with using only in-game score. The idea was to make sure that winning was always advantageous whilst still giving the agent feedback throughout the game to allow for progress up to the first win. As can be seen in Figure 13, this structure didn't exactly work either, with less total wins than for only in-game score reward. It should however be noted that even if the total number of wins are fewer by the end of the training, the learning has not stalled in the same way it did with only in-game score reward. There is possibility that this agent would beat the other one given longer time to train. This hypothesis was never tested due to limited time.

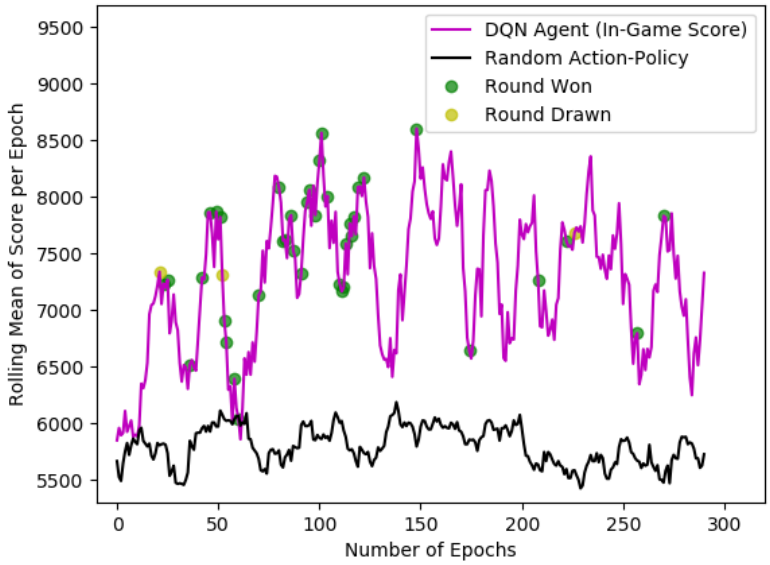


Figure 14: Rolling mean, with a window of 10 epochs, of the in-game score per epoch of training a DQN agent based on the in-game score reward structure. The wins and draws have been marked with green and yellow dots. *No prior imitation learning.*

5 Implementation of Imitation Learning

After developing a functional basic agent with deep Q-learning another problem became apparent. The agents required long training times with poor results even after these long training sessions for Move-To-Beacon and the full game of StarCraft II, see results in Section 4. As mentioned in Section 2.3 imitation learning is a much used method in combination with reinforcement learning. In game environments this lets the agent learn from a sample of correct or good actions, with the intent of improving performance and greatly reducing required training times. The idea is to help the agent master the basics of an environment faster. Then, once it has gotten sufficient basic knowledge the imitation learning is stopped and the agent is free to explore whatever strategies it predicts to be the best.

In order to add initial imitation learning to Algorithm 1 an alternative way of choosing actions was added. For a set number of initial epochs this alternative action selection was used after which the action policy continued as before. After imitation learning had been done ε was set to ε_{to} for the remainder of the training, since less exploration should be needed after imitation learning. The actions were chosen differently for the three environments, but the general idea was to generate correct actions if possible and otherwise good actions. These actions were then added to the memory per the original algorithm and trained on in the **for**-loop, Line 17–21.

5.1 Frozen Lake

In Frozen Lake there are a number of optimal solutions to the static map, one such sequence of actions was chosen and fed repeatedly during the imitation learning as mentioned above. Only the hyperparameters with the best result from Section 4.2 were used here, namely $\gamma = 0.7$, $\varepsilon_{\text{to}} = 0$, mini-batch size 32. Figure 15 show the

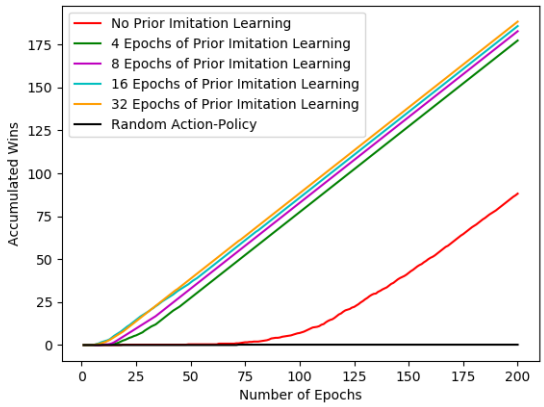


Figure 15: Accumulated wins from training DQN agents in Frozen Lake with 4,8,16 and 32 epochs of prior imitation learning. The best result from Figure 7 with no prior imitation learning has been repeated as well as the agent based on a random action-policy.

accumulated wins during 200 epochs of training with 4,8,16 and 32 prior epochs of imitation learning. The same result for training without imitation learning has been repeated from Figure 7. What is of interest in this result is the number of epochs before the optimal solution was reached. Without imitation learning the agent needed more than 100 epochs and with only a few epochs of imitation learning the optimal solution was reached by epoch 25. For increasing amounts of prior imitation learning the time becomes even shorter. It is clear that imitation learning substantially shortens the time it takes for DQN agents to master a simple game such as Frozen Lake.

5.2 Move-To-Beacon

Move-To-Beacon, similarly to Frozen Lake, has a correct action to every state. Technically any of the coordinates covered by the beacon suffices as correct. Therefore Algorithm 1 was changed in much the same way as for Frozen Lake, except for how this correct action was found. Here the location of the beacon for a given state was read directly from PySC2 and translated to the given action space by the following lines of code.

```
beacon = [unit for unit in obs.observation.feature_units
                    if unit.unit_type == 317]
action = beacon[0].x + BOARD_SIZE_X * beacon[0].y
```

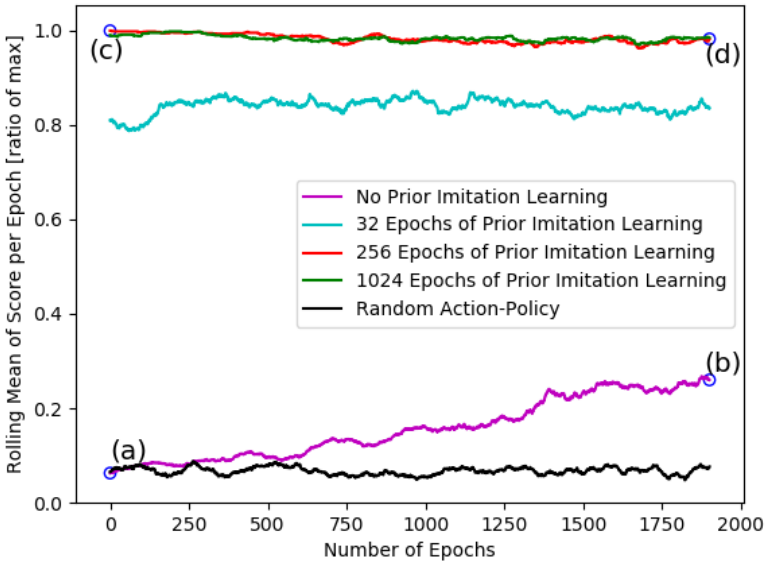
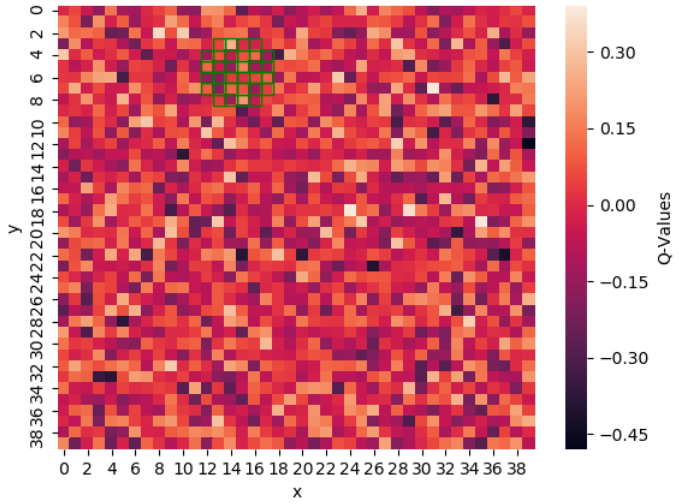


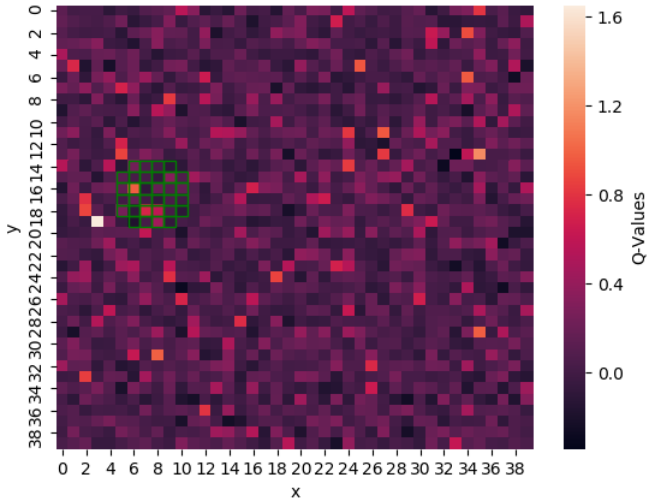
Figure 16: Rolling means, with a window of 100 epochs, on in-game score per epoch as a ratio of the maximum score per game from training Q-learning agents in Move-To-Beacon, with 32 and 1024 epochs of prior imitation learning. The best result from Figure 10 with no prior imitation learning has been repeated as well as the agent based on a random action-policy. (a), (b), (c) and (d) in the figure refers to specific states shown in Figure 17.

The hyperparameters that had the best result from Section 4.3 were used for testing imitation learning, namely $\gamma = 0.5$, $\varepsilon_{\text{to}} = 0$ and mini-batch size 128. Figure 16 shows the result of 2000 epochs of training the DQN agents with 32, 256 and 1024 epochs of prior imitation learning. The result for the agent with no prior imitation learning has been repeated as well as the result for the random agent. Same as in Section 4.3, the results are shown as a rolling mean of the in-game score per epoch with a window of 100 epochs and as a ratio of the maximum score possible per game. With 256 and more epochs of prior imitation learning the agents reach a close to perfect action-policy. This result is reached directly after imitation learning is done. For 32 epochs of prior imitation learning the result is much better than without imitation learning but only around 80% accuracy is reached. Recall that no exploration was done after imitation learning in these tests, this explains why the performance isn't improved during the 2000 epochs of training after imitation learning.

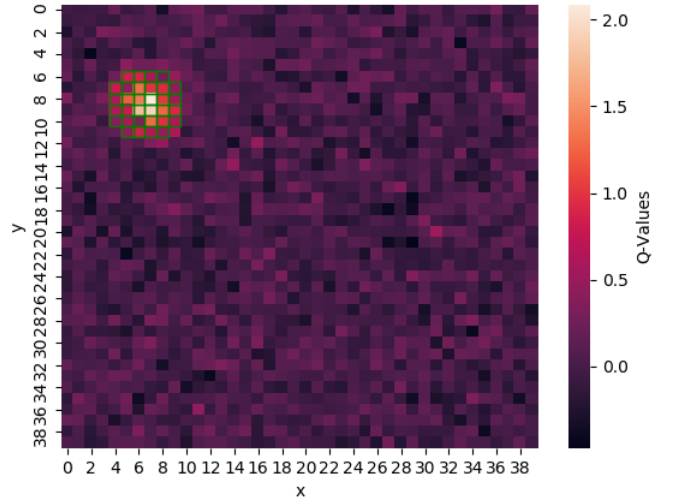
In order to further analyze the agent during the training process the Q-values for a sample of states has been visualized as heat maps in Figure 17. Recall from Section 2.2 that the Q-values are the agent's prediction about future rewards for each action for a given state. In the heat maps the state, where the beacon is located, has been marked in green and thus higher values in this area as opposed to the rest of the map means that the agent has achieved a good action-policy. The top heat maps in Figure 17 are samples from training without imitation learning and the bottom ones are are from training with 256 epochs of prior imitation learning.



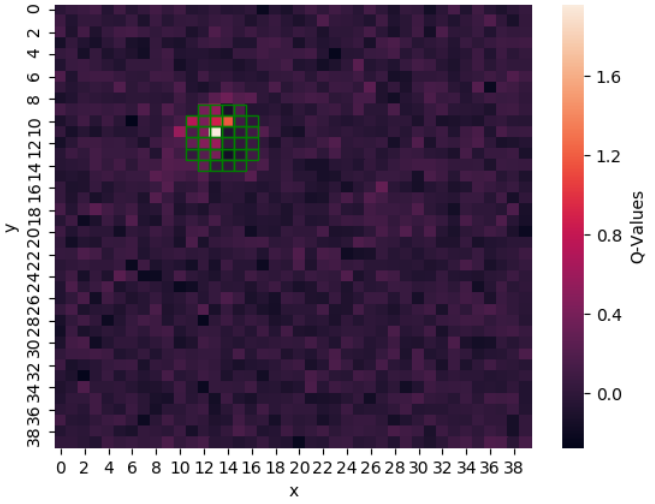
(a) Epoch 1 without prior imitation learning.



(b) Epoch 2000 without prior imitation learning.



(c) Epoch 1 with prior imitation learning.



(d) Epoch 2000 with prior imitation learning.

Figure 17: Heat maps of DQN agents' Q-values for a sample of states from their training in Move-To-Beacon, where x and y are coordinates on the map. The coordinates covered by the beacon, in each state, have been highlighted in green. (a) and (b) are taken from the first and last epoch of training the DQN without prior imitation learning in Figure 16. (c) and (d) are taken from the first and last epoch of training the DQN agent with 256 prior epochs of imitation learning from Figure 16. As explained in Section 2.2, higher Q-values imply the agent predicts there's a higher probability of the beacon being located at the given position. Note that the color axis differ between the figures, mostly between (a) and the rest.

The leftmost heat maps show states from the beginning of training (after imitation learning) and the rightmost ones show states from the last epoch.

For the first epoch with no prior imitation learning the Q-values are basically a result of the random uniform initialization, no updating of the network has been

done and so the agent has no idea where the beacon could be. In line with the result for this agent in Figure 16 the agent has not learned the game completely by epoch 2000 and so in this state, (b) in Figure 17, the agent is wrong about where the beacon is and wants to click slightly left of where it actually is. As expected the agent trained with imitation learning has a much better understanding of the game. Already directly after imitation learning, (c) in Figure 17, it is able to pinpoint the beacon perfectly and by 2000 epochs of extra training, (b), it is even more sure of one particular coordinate that is within the boundary of the beacon. This can also be seen in (c) and (d) where the area surrounding the beacon is darker representing that it is less confident in those positions.

5.3 The Full Game of StarCraft II

Unlike Frozen Lake and Move-To-Beacon, StarCraft II has no correct solution, this made it harder to add imitation learning to the problem. As stated in Section 2.3, DeepMind implemented imitation learning for their agents by using replays of human players [2]. Due to the simplifications of the state and action space in this

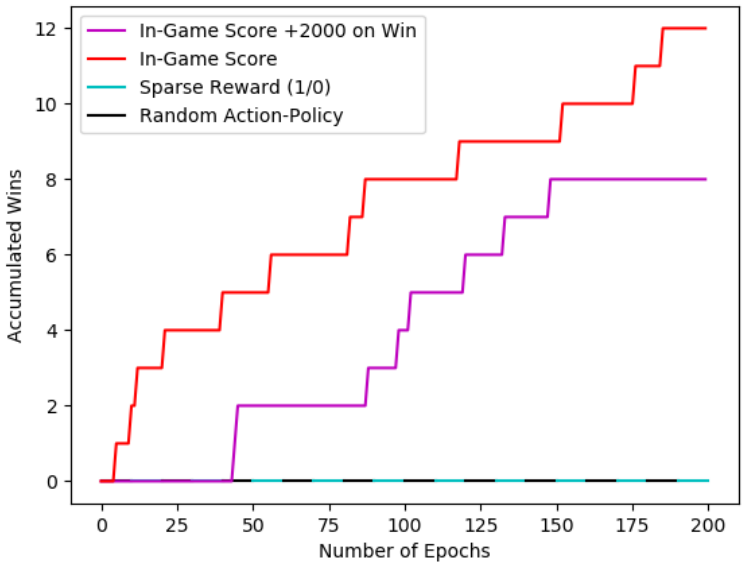


Figure 18: Accumulated wins from training DQN agents on the full game of StarCraft II with 100 epochs of prior imitation learning, based on three different reward structures. Note that the result of the random agent is identical to that of the DQN agent based on the sparse reward structure.

project, the same could not be done here, instead a rule-based agent was developed which can win by following a few simple rules. Firstly, it makes sure to have enough supply and one spawning pool in order to be able to build military units. To increase production of units it then, when possible, builds a queen and uses it to spawn more larva. After this the agent builds zerglings until there are 20 of them and then it sends them to attack the enemy. Appendix B shows the code behind generating

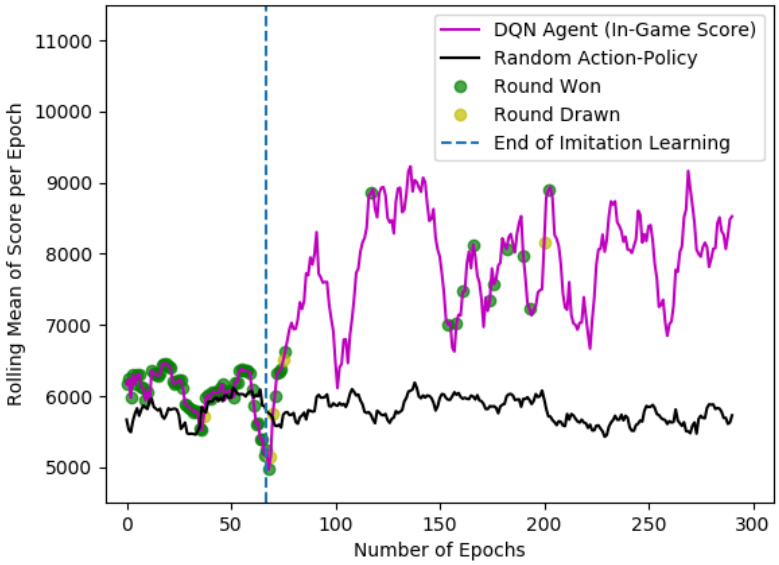


Figure 19: Rolling mean of the in-game score per epoch, with a window of 10 epochs, for training a DQN agent on the full game of StarCraft II. The DQN agent is based on the in-game score reward structure with 67 epochs of imitation learning. Here the epochs of imitation learning have **not** been omitted like they have been before, as they contribute to the analysis of the result. Where the imitation learning ends has been marked by a blue, dotted, vertical line. Wins and draws are again highlighted in green and yellow respectively. The result of an agent based on a random action-policy has been added as a baseline.

these actions. These sparse simple rules are enough to win against the "very easy" built-in AI, but the agent cannot win against opponents of higher difficulty.

Imitation learning was tested for each of the three reward structures with the same hyperparameters, as were used in Section 4.4. This result can be seen in Figure 18 as accumulated wins. With the sparse reward structure no improvement was made with imitation learning, the agent still did not win any games. The result of adding prior imitation learning to the agents with the two reward structures built on in-game score showed better results but not in comparison to the result without imitation learning in Figure 13. For some reason none of these results, with prior imitation learning, exceeded the result without prior imitation learning.

For the agent based on the in-game score reward structure, the score at the end of each game was also visualized. Figure 19 shows this result, in the same way the results where shown in Figure 14. Note that the epochs of imitation learning has not been omitted in this figure as they have been previously in this thesis. An interesting conclusion can be drawn from this figure, namely that the reward structure and the scripted agent are in conflict. During the epochs controlled by the scripted agent most games are won and an in-game score of around 6000 is achieved. When imitation learning ends and the agent starts to maximize the in-game score instead, the score goes up and the win rate decreases. Therefore it is clear that

this scripted agent did not work with this reward structure. It has already been established that the reward structure is faulty when it comes to learning how to win the game. Thus it might still be the case that the scripted agent would have worked if applied to an agent with a fully working reward structure, but this could not be proven in this thesis.

6 Discussion and Analysis of the Results

The purpose of this thesis was to investigate whether or not similar results compared to current scientific research with very limited resources could be achieved. The hypothesis was that vast simplifications combined with imitation learning would help this cause. What follows is a discussion on the impact of adding imitation learning as well as the problems which surfaced during the development of an intelligent agent in StarCraft II due to the simplifications. Finally a review and a breakdown of pros and cons for the use of imitation learning, discussing where it worked well and where it did not and why.

6.1 The Development Process and Overall Results

Throughout the project the result of training the agents were compared to random action-policy agents, present in most figures. This served as an important benchmark to prove that the DQN agents were learning. The results of the DQN agents started out as random and improved over time. This, compared to the performance of the random agents which did not improve over time, was a clear indicator that the DQN agents did learn something.

The final results of the agents in this thesis are mixed. For the simpler games, Frozen Lake and Move-To-Beacon, the best agents were able to master the respective games. This can be seen in the result from validating these agent, Figure 8 and Figure 17, as they achieve perfect gameplay. For the more complex game, StarCraft II, the results were less than desirable, possibly due to the problems with the reward structure.

Regarding the process of the project, each iteration through the simpler games did give valuable insights to the agents development such as an intuition for the hyperparameters. However the time spent on these games took longer than expected and left the project with less time than anticipated for the complex game StarCraft II. This in turn resulted in not enough time to thoroughly test this environment to a satisfactory level, resulting in the poor outcomes. Given more time further tests could have been done regarding the reward structures before the implementation of imitation learning.

6.2 Problems with the Reward Structure in StarCraft II

In the full game of StarCraft II the random action-policy agent could not win a single game. The regular DQN agents without imitation learning based on the in-game score reward structure, see Figure 14, showed promising progress initially but after a while the accumulation of wins started to diminish. It was established that this

behaviour happened due to the agent trying to maximize the in-game score rather than wins, a possible consequence of the reward structure is that it, among other things, did not distinguish between wins and losses. This problem persisted with the agents utilizing imitation learning, which can be seen in figure 19.

The sparse reward structure used in this thesis did not win any matches during training. The reason for this might be that the limited resources prevented the agent to run the game for more than 300 epochs and it is possible that a longer running time could have improved the acquired results. Because of the long games and the complex nature of StarCraft it is impossible to tell which action is the game winning action and it is therefore impossible to give the sparse reward in the same way as done in Frozen Lake and Move-To-Beacon. Therefore with limited resources this would not be a feasible solution. The preferred approach was still the algorithm based on the in-game score, which was not able to give significant performance in the game. Nonetheless Figure 13 shows that after only 25 epochs of training the agent starts to win, which favours the purpose of this thesis. These results were achieved with massive simplification of the environments and show that good results are achievable even with major simplifications.

6.3 Impact of Imitation Learning

As seen in the results from Section 5, the use of imitation learning for the simpler games shows that the time it takes for the agents to reach sufficient mastery of a game are substantially shorter when using imitation learning as an initial boost to the agent's learning. Not only would this reduce the training time, but also the cost of operation when results can be made at a shorter time. This would also mean that more training can now be done in less time which could further speed up the progress in the field.

The results indicate that imitation learning is indeed a good method for machine learning. However the same thing can not always be said with more complex environments, where more factors have to be considered. The results from the simple environments show that for similar but less complex environments it is possible to implement imitation learning and the algorithm correctly to improve training time. Due to the problems discussed with the reward system for StarCraft II no real conclusions can be drawn on the impact of imitation learning.

A worry one might have when it comes to basing intelligent agents on human experiences is that we might miss out on the new and improved solutions that only the agents can otherwise come up with. This is however not the case, in the above mentioned examples, imitation learning is only used in a way that complements other machine learning techniques. In a demonstration of DeepMinds StarCraft II agent [26] it was clear that the agent had learned completely new and unique strategies despite having been trained on the human replays.

7 Future Work

Due to the sheer complexity of StarCraft II and the limited resources significant simplifications had to be made of the game in the scope of this thesis. For this

reason there are many possible areas to improve on and explore further. In the following section some of these are listed.

Improved State and Action Space. The macro actions developed were limited to only the most basic functions that are valid during the early stages of a game. These actions could be expanded and improved upon to allow the agent to build more advanced units, which can be used during the later stages of the game. For this to be viable the current state representation would also need to be improved.

Moreover, the current agent uses no spatial actions or states and has hard-coded coordinates for where the agent should attack and build structures. A spatial action and state space could be implemented to allow for better micromanagement of units and the ability to add imitation learning using human replays. However this would require a more advanced network architecture which will be discussed further in the coming paragraphs. An example of an agent that used both spatial and non-spatial features, are DeepMind's AlphaStar [2].

A More Advanced Network Architecture. A convolutional neural network that uses image recognition might be one way to incorporate spatial features into the agent as it could use the screen to detect the number of units visible and their positions to make it easier to understand the current state better. This is a much more complex method, that has the drawback of being difficult to use. Not to mention that it requires additional computational power in relation to simpler networks.

To meet the needs of a larger state and action space a more complex network architecture would be needed. In this project a neural network with one hidden layer were used, where the number of nodes depended on the size of the state representation and action space of each game. On the simpler games with perfect information one hidden layer might suffice but on more complex games with large quantities of data more hidden layers might be more suitable. Here the possibility of overfitting needs to be taken into account as well, a side effect of more intricate networks, see Section 2.1.1.

A More Complex Reward Structure. Because of the complications, outlined in Section 6.2, with the reward structures tested in this thesis, more research is needed regarding how to best implement a reward structure that is in line with the goals of the game. In StarCraft II it is hard to determine whether an action is good or bad for the game outgame. A suggestion is to analyze actions and have subgoals that gives the agent more well defined feedback on its actions.

Multi-Agent Environments and Generic Algorithms. Something that this project explored during its early stages was the classic game Tic-Tac-Toe. The hypothesis was that this would be a good way to learn about how to have multiple agents compete against each other whilst developing their strategies and counter strategies similarly to StarCraft II. Due to complications where one agent got the upper hand and both agents stopped learning this part of the thesis was discarded. But it is an interesting subject to explore whether agents learning by playing against each other would reduce the need for external data to train on.

One solution to the problems experienced with Tic-Tac-Toe is genetic algorithms which is based on the natural selection process. In principle, you train several different agents against each other in something similar to a league and after a set amount of epochs, a portion of the agents that performed badly are overwritten with the data of the best agent which in essence turns them into mutated copies of that agent [28]. DeepMind used this algorithm to train their agents. Their implementation ensured that the winners fought against each other in a contest of survival of the fittest. Thus, strong agents would compete and learn from equally difficult opponents which improved the learning progress [27].

8 Conclusion

After the analysis some results were as expected, others were not. What can be derived is that on the deterministic games, the use of imitation learning improved the results by a large margin. This indicates that imitation learning is a good method for machine learning in these kinds of environments. However the same thing cannot always be said for complex environments, where more factors have to be considered. If implemented correctly imitation learning might enhance the training process, but this could not be proven in the scope of this thesis.

The StarCraft II learning environment is a great opportunity for further machine learning studies, the increased complexity compared to more classic games is a great challenge and requires extensive research. One aspect that needs to be further improved upon is the reward structure. Ideally a solution that could work well with imitation learning since it is proven to be able to enhance an agents performance with limited resources.

The findings in this thesis allows smaller organizations to get into complex machine learning and the results show that, in some cases, small amounts of imitation learning can be used to achieve good results.

References

[1] M. Campbell, "Mastering board games," *Science*, vol. 362, no. 6419, p. 1118, 2018. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/30523099

[2] O. Vinyals *et al.*, "StarCraft II: A New Challenge for Reinforcement Learning," 2017. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2017arXiv170804782V

[3] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[4] S. Russell and P. Norvig, *Artificial Intelligence : A Modern Approach*, 3rd ed., ser. Prentice Hall Series in Artificial Intelligence. Pearson Education Limited, 2016.

[5] T. Simonite, "How Google Plans to Solve Artificial Intelligence," March 2016. [Online]. Available: https://www.technologyreview.com/s/601139/how-google-plans-to-solve-artificial-intelligence/

[6] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2016. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2016arXiv160304467A

[7] keras.io. You have just found Keras. [Online]. Available: https://keras.io/#you-have-just-found-keras

[8] B. Mehlig, "Artificial Neural Networks," 2019. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2019arXiv190105639M

[9] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/26017442

[10] E. L. Lehmann and G. Casella, *Theory of point estimation. [electronic resource]*, 2nd ed., ser. Springer texts in statistics. Springer, 1998.

[11] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," 2015.

[12] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[13] Z. Yu, N. Yu, Y. Pan, and T. Li, "A Novel Deep Learning Network Architecture with Cross-Layer Neurons," in *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*. IEEE, 2016, Conference Proceedings, pp. 111–117.

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

[15] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak, "Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, Conference Proceedings, pp. 4580–4584.

[16] R. H. R. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, no. 6789, pp. 947–951, 2000. [Online]. Available: https://doi.org/10.1038/35016072

[17] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, G. Geoffrey, D. David, and D. Miroslav, Eds., vol. 15. PMLR, 2011, Conference Proceedings, pp. 315–323. [Online]. Available: http://proceedings.mlr.press

[18] R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction*, ser. Adaptive computation and machine learning series. The MIT Press, 2018.

[19] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/25719670

[20] S. Schaal, "Is imitation learning the route to humanoid robots?" *Trends in Cognitive Sciences*, vol. 3, no. 6, pp. 233–242, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364661399013273

[21] X. Tan, C.-B. Chng, Y. Su, K.-B. Lim, and C.-K. Chui, "Robot-Assisted Training in Laparoscopy Using Deep Reinforcement Learning," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 485–492, 2019.

[22] J. Oh, Y. Guo, S. Singh, and H. Lee, "Self-Imitation Learning," 2018. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2018arXiv180605635O

[23] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding Neural Networks Through Deep Visualization," 2015. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2015arXiv150606579Y

[24] P. Sun *et al.*, "TStarBots: Defeating the Cheating Level Builtin AI in StarCraft II in the Full Game," 2018. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2018arXiv180907193S

[25] M. Krajewski. (2014) Some StarCraft II Stats. Medium. [Online]. Available: https://medium.com/@mkrajew/some-starcraft-2-stats-7a32ece48caf

[26] DeepMind, "DeepMind StarCraft II Demonstration," 2019. [Online]. Available: https://www.youtube.com/watch?v=cUTMhmVh1qs

[27] O. Vinyals *et al.*, "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II," 2019. [Online]. Available: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/

[28] A. Meyer-Baese and V. Schmid, *Pattern Recognition and Signal Analysis in Medical Imaging*. Oxford: Academic Press, 2014.

A Complete set of data visualizations for Frozen Lake

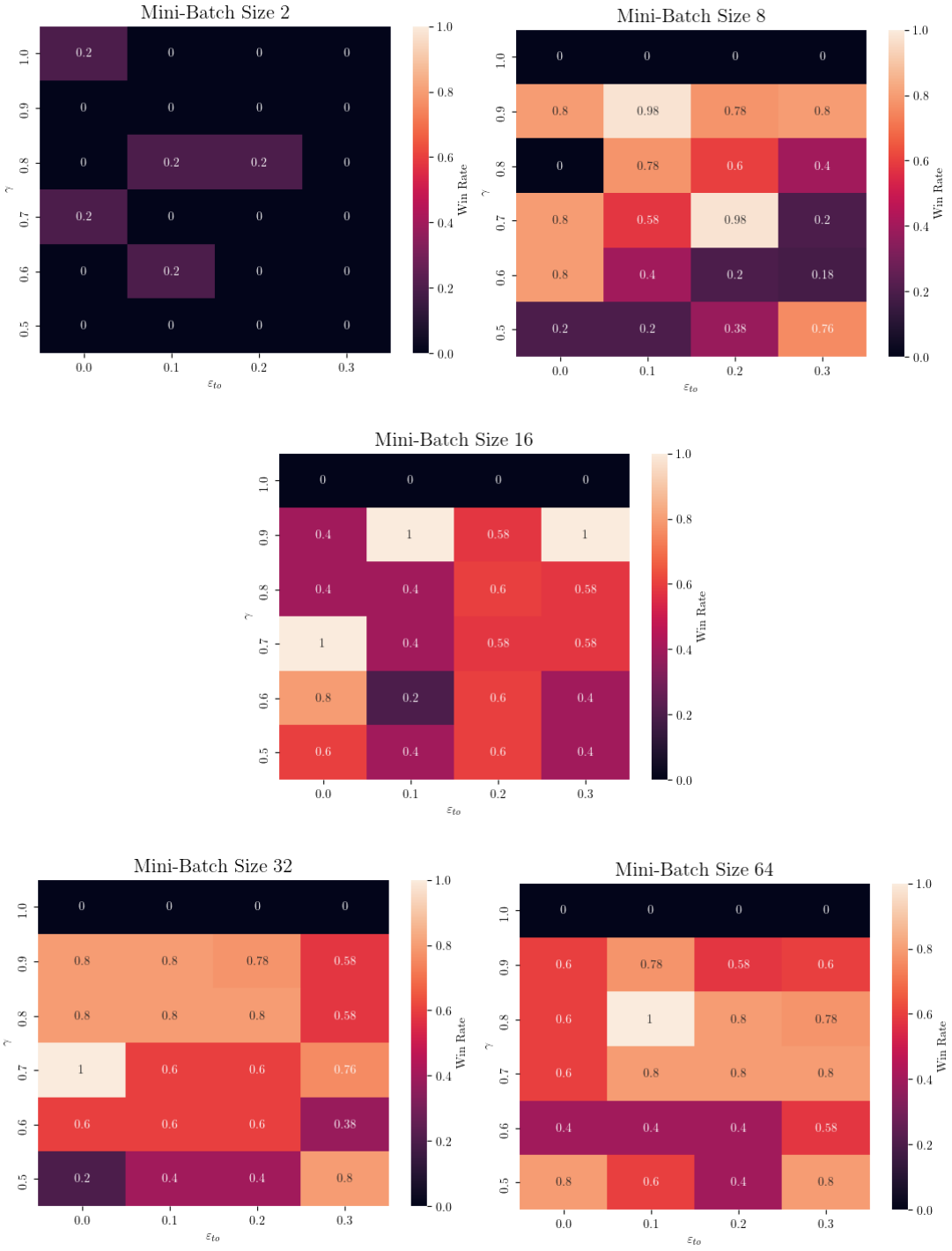


Figure 20: Heat maps of the win rate from 10 validation games for the DQN agents in Frozen Lake. One for each mini-batch size with respect to ε_{to} and γ.

B Code from Scripted Agent

Below is the code showing how the scripted agent chooses an action, used for generating imitation learning data.

```
# Build overloard when supply is full
if suply_limit - total_suply <= 1:
    #Build overloard
    action = 2
# When possible make sure to have atleast one spawning pool
elif minerals > 200 and nr_spawn_pools < 1:
    #Build Spawnpool
    action = 3
# Build drones until you have 16
elif workers < 16:
    #Build Drone
    action = 1
# When possible build a Queen (if you don't already have one)
elif minerals > 150 and nr_spawn_pools >= 1 and nr_queens < 1:
    #Build Queen
    action = 9
# When possible use a queen to spawn larva to get more units
elif nr_queens > 0 and amount_energy > 25:
    #Spawn Larva
    action = 10
# When you have a suffisient force, attack the enemy
elif army > 20:
    #Attack enemy base
    action = 5
# When in doubt build zerglings
else:
    #Build zergling
    action = 4
```

C List of libraries and computer hardware

Python libraries

Below is all Python libraries and versions used during the project, note that most of these libraries are just dependencies to the main libraries. The main libraries used are **tensorflow** and **Keras** for machine learning, **PySC2** for connecting to StarCraft II, **numpy** for mathematical calculations, **pandas** for large scale data handling and lastly **matplotlib** and **seaborn** for generating graphs. During the project Python 3.6.8 have been used.

Libary	Version
absl-py	00.07.01
astor	00.07.01
certifi	2019.03.09
chardet	03.00.04
cycler	00.10.00
enum34	01.01.06
future	00.17.01
gast	00.02.02
grpcio	01.19.00
h5py	02.09.00
idna	2.8
Keras	02.02.04
Keras-Applications	01.00.07
Keras-Preprocessing	01.00.09
kiwisolver	01.00.01
Markdown	03.00.01
matplotlib	03.00.03
mock	02.00.00
mpyq	00.02.05
numpy	01.16.02
pandas	00.24.02
pbr	05.01.03
pip	18.1
portpicker	01.03.01
protobuf	03.07.00
pygame	01.09.04
pyparsing	02.03.01
PySC2	02.00.02
python-dateutil	02.08.00
pytz	2018.9
PyYAML	5.1
requests	02.21.00
s2clientprotocol	4.8.3.72282.0
scipy	01.02.01
seaborn	00.09.00

setuptools	40.06.02
six	01.12.00
sk-video	01.01.10
tensorboard	01.12.02
tensorflow-estimator	01.13.00
tensorflow-gpu	01.12.00
termcolor	01.01.00
urllib3	01.24.01
websocket-client	00.55.00
Werkzeug	00.14.01
wheel	00.33.01
whichcraft	00.05.02

Hardware

CPU: Intel Core i7 9700k @ 3.6GHz
GPU: Nvidia GeForce RTX2060