CHALMERS
UNIVERSITY OF TECHNOLOGY

Machine Learning in StarCraft II

Lowering the Difficulty Threshold of Starting From Scratch

Bachelor's thesis in Computer Science and Engineering

SVANTE BENNHAGE, ERIC GULDBRAND,
OMAR OUEIDAT, MATTIAS TORSTENSSON,
SILAS ULANDER, ERIK WALLHEDE

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

Bachelor of Science Thesis

Machine Learning in StarCraft II

Lowering the Difficulty Threshold of Starting From Scratch

SVANTE BENNHAGE
ERIC GULDBRAND
OMAR OUEIDAT
MATTIAS TORSTENSSON
SILAS ULANDER
ERIK WALLHEDE

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Machine Learning in StarCraft II
Lowering the Difficulty Threshold of Starting From Scratch
SVANTE BENNHAGE, ERIC GULDBRAND, OMAR OUEIDAT, MATTIAS TORSTENSSON, SILAS ULANDER, ERIK WALLHEDE

© SVANTE BENNHAGE, ERIC GULDBRAND, OMAR OUEIDAT, MATTIAS TORSTENSSON, SILAS ULANDER, ERIK WALLHEDE, 2019.

Supervisor: Jonas Duregård, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Morten Fjeld, Sven Knutsson, Department of Computer Science and Engineering

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

Cover: An Orbital Command, a Terran building in StarCraft II.

Typeset in LaTeX
Gothenburg, Sweden 2019

Acknowledgements

Special thanks to our supervisor Jonas Duregård for providing us with assistance and valuable feedback throughout the project. We also thank the DATX02-19-81 group for bringing to our attention and helping us with the Sc2Reaper tool. Our gratitude also to the Division for Language and Communication for their help with the writing process as well as to the Department of Computer Science and Engineering for making this project possible.

Machine Learning in StarCraft II
Lowering the Difficulty Threshold of Starting From Scratch
SVANTE BENNHAGE
ERIC GULDBRAND
OMAR OUEIDAT
MATTIAS TORSTENSSON
SILAS ULANDER
ERIK WALLHEDE
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Artificial intelligence research is currently a hot topic within many industries. In terms of research, games such as StarCraft II provide a good testing ground due to its accessibility. However, getting started can still be more difficult than it should be.

This paper aims to facilitate the development of a machine learning agent for StarCraft II by designing tools for data collection, making a simple API built on top of PySC2 to facilitate interaction with the game and by analyzing a few different types of artificial neural networks with respect to StarCraft II.

It is concluded that defining reward functions for reinforcement learning can give rise to unexpected behaviors. A further conclusion is that convolutional neural networks tend to be more resource intensive than non-convolutional networks and that they are thus less suited for anyone without access to large computational power. Lastly, a network is trained on collected data to continuously predict the win chance for players in a StarCraft II match. Unfortunately the network does not become successful in its task, likely in part due to the simplicity of the network.

Keywords: Artificial Neural Networks, Machine Learning, StarCraft II, Reinforcement learning, Supervised learning

Sammandrag

Artificiell intelligens är ett högaktuellt område inom många forskningsfält. I forskningen är spel som StarCraft II en utmärkt utgångspunkt på grund av sin tillgänglighet och sina tydliga avgränsningar. Tyvärr kan det för nybörjaren ändå vara svårare att komma igång än det borde vara.

Denna rapport syftar till att underlätta uppstarten av maskininlärning genom att utveckla verktyg för datainsamling, bygga en API ovanpå PySC2 som underlättar interaktion med spelet och genom att analysera några olika typer av artificiella neurala nätverk med hänsyn till StarCraft II.

Slutsatsen som dras är att många definitioner av belöningsfunktionen (eng: "reward function") kan ge upphov till oförväntade beteenden. Vidare dras slutsatsen att konvolutionella neurala nätverk tenderar att vara mer datorkraftsintensiva än icke-konvolutionella och är därmed mindre lämpliga för den med tillgång till mindre kraftfull hårdvara.

Slutligen tränas ett nätverk på insamlad data för att kontinuerligt förutspå vinstchansen för spelarna i en StarCraft II match. Däremot lyckas nätverket inte ge några användbara resultat, troligtvis på grund av nätverkets alltför enkla struktur.

Contents

1
Introduction

Recent years have seen much development within the field of artificial intelligence (AI) [1]–[3], with a long-term goal of developing artificial general intelligence (AGI) [4]. AI is a popular field of research within a multitude of industries, some examples being autonomous driving, camera technology and medicine.

A common concept within the field of AI is *agents*. An agent is an autonomous component which reacts to inputs received from the surrounding environment, whether it is the real world, a game, or some other simulation [5]. The output is decided by some sort of scripted logic or an *Artificial Neural Network* (ANN), which is a computational learning system inspired by human biology [6]. ANNs are commonly trained with methods such as *reinforcement learning* or *supervised learning*. The response from an agent should be the next action to take. For example, an agent designed to play chess would look at the current state of the board in order to contemplate its next move. An agent designed for autonomous driving could use sensory data to build an understanding of the car's surroundings, thus being able to decide if it could or should switch lanes.

In AI research, games are seen as one of the most prominent challenges, as well as an excellent environment to train ANNs in [7]. Computer games have also been referred to as the best platform to use in order to reach the long-term goal of creating an AGI [7]. As recent as 2017, an agent created by DeepMind was able to convincingly beat the world champion program in traditional board games such as Chess, Shogi, and Go [8], paving way to further research in other genres of games.

Despite this breakthrough, there is still a relatively large area unexplored in gaming. Shogi, Go, and Chess all have a smaller action space compared to genres such as *Real Time Strategy* (RTS) games, which makes it harder to create agents for, requiring higher amount of computing power and more efficient algorithms.

A popular game within the RTS genre is *StarCraft II* (SC2), which also is a primary candidate for research in AI using games. SC2 is a game that revolves around gathering resources, managing one's economy and building an own army with the end goal of destroying the opponent's base. There are many reasons why SC2 poses a major challenge for AI, one of them being the aspect of imperfect information. A player can only see what is happening in areas where friendly units or buildings are located, meaning that information about the enemy's movements and advancements has to be acquired by the player's own accord. SC2 has been the focus of research for some time and this year (2019), an agent, created by DeepMind, was able to beat two professional SC2 players ten times while only losing once [9].

1.1 Purpose

This paper aims to lower the difficulty threshold of starting out with machine learning in StarCraft II. It analyzes different artificial neural networks and provides understanding of several tools associated with creating an agent that can learn and play StarCraft II.

1.2 Problem description

Creating an able machine learning agent for StarCraft II requires several types of networks in order to incorporate different types of data. This paper aims to alleviate the difficulty of creating such an agent by a few key steps.

First, two kinds of reinforcement agent are evaluated on StarCraft II in terms of their received score, training time and behavior. This is to gain a basic understanding of reinforcement agents in a complex game environment.

Secondly, a path is staked for training SC2 agents on replays by modifying and combining existing libraries for replay collection, sorting and parsing. This is a necessity for the supervised agent, since no ready solution seems to already exist. An easy-to-use API for any type of SC2 machine learning is also developed so that it will be easier for others to get started with SC2 machine learning. The hope is that this API, its documentation and this report can serve as a stepping stone for anyone who is starting out with AI development so that they can quickly learn and build more advanced agents without spending too much time on researching the basics themselves.

Finally, an ANN is trained with supervised learning to predict the current chance a player has to win by looking at replays of human games. This is relevant since a previous study by DeepMind [10] indicates that it is difficult for an SC2 reinforcement agent to learn without knowing if its actions improve or deteriorate its chances of winning.

1.3 Limitations

Result analysis is limited to only a subset of StarCraft II, namely by restricting play and replay analysis to a single of the game's three races (terran) on SC2 version 4.8.3.

Furthermore, reinforcement agents are only evaluated on a mini-game representing a sub-problem of SC2 (collecting minerals) and have their action space restricted. This includes grouping multi-step actions or actions with positional arguments into single-stage actions with no arguments. This limitation is motivated by DeepMind's results [10] on how reinforcement agents perform poorly on SC2's large action space.

Networks are only given cursory analysis of how well they utilize computational resources and are not optimized to take advantage of all available hardware during training. A proper analysis of these aspects would require more in-depth knowledge regarding tools and libraries used for training, and optimization has not been a priority due to the relatively small benefit of being able to utilize the resources available.

1.4 Social and Ethical Aspects

The research and development of AI has both pros and cons: the technology has been used to help diagnose diseases [11] while having the potential to be used for mass surveillance [12]. It could be used to help automate jobs or be used in self driving cars, while at the same time it could be used in unmanned combat or to help oppressive governments. This makes AI the topic of ethical debates as well as the subject of potential new laws.

This thesis only explores creating a basic agent for a game, which is a small sub-problem of creating AGI. This would therefore not have any large-scale ethical consequences in real life. However, there is the possibility that AI would affect the game in a negative way. When AI was introduced to Chess, with the ability to beat the world champion, there was no direct negative impact. Creating an agent for this thesis will therefore probably not affect the game negatively, but rather perhaps introduce a nuanced way of playing the game which has not been thought of before.

2
Background

This chapter intends to make it easier for the reader to understand all the different concepts regarding how to create an agent for SC2 using artificial neural networks. It includes describing the different tools used, previous work done on the subject, as well as the general topic of machine learning and AI.

2.1 StarCraft II

StarCraft II is a mostly deterministic [10] RTS game. Players can choose to play as one of three races: terran, protoss or zerg. Each race has access to different units and buildings which leads to different strategies being popular for each race. There are multiple game modes, the most common one involves two players competing against each other in a one-on-one duel. Players start with a single base building and have to collect resources, construct buildings, create an army and destroy all the opponent's buildings. Games often last anywhere from 5 to 40 minutes depending on the strategies employed.

The SC2 action space is vast, since a player can at any moment select any number of their buildings and units, and then issue commands that build something, attack, or move them to any point on the map. In addition some unit types are able to use abilities which can cause drastic changes to the game's state. An example would be teleporting in order to bypass obstacles or by calling down a nuclear strike, destroying most units in the area.

Blizzard Entertainment, the company behind SC2, recently created a public API for interacting with the game [13]. This allows for easier development of agents that can play the game. Along with this, DeepMind created the StarCraft II Learning Environment [10] PySC2. PySC2 maps the Blizzard API for the Python [14] programming language which makes it easier to integrate with tools like *TensorFlow* [15].

Games like SC2 provide opportunities to explore the behaviour of machine learning and artificial intelligence in a setting where every piece of crucial information is not continually available, and has to be acquired by choices and actions made by the network it concerns [9], compared to games like Go and Chess where this is not the case.

Actions in Go and Chess always lead to exactly the same result given the state of the game, while this is mostly true for SC2 there are some caveats. First is the asymmetry of information, players do not always have access to the entire state of the game but instead see a subset of it at any one time, thus, the perceived state

of the game could differ drastically from the actual state. Second, there are some minor aspects of the game, such as the unit update order, which are random [10]. Last is the real-time component of the game, while it is possible to execute the same actions at the same time it becomes highly unlikely as a match continues. However, if both players act identically to a previous match, at the same point in the match, the outcome will be the same assuming the random seed is the same.

2.2 Artificial intelligence and machine learning

To quote George F Luger [16], "Artificial intelligence (AI) may be defined as the branch of computer science that is concerned with the automation of intelligent behavior". In later years, this concern has found application and a useful testing ground with computer games [17].

Machine learning is a sub-field of artificial intelligence where a computer follows a data parsing algorithm that allows it to learn how to solve a problem related to the parsed data, rather than by following an algorithm incorporating domain specific knowledge to solve a problem directly [18]. The study of machine learning is defined by Tom M. Mitchel in his 1997 book "Machine Learning" [19] as "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E". For instance, the chess engine Deep Blue was developed using a more classical approach to artificial intelligence and uses an evaluation function that could not have been created without previous knowledge of chess [20]. Classical machine learning like the Viola-Jones 2001 face-detection algorithm [21] however, does not add any domain knowledge specifically, but leaves it to the computer to figure out what features are important for distinguishing faces.

2.3 Artificial Neural Networks

Modern machine learning research is to a great deal based on artificial neural networks (ANN) which are constructed using layers of artificial neurons connected by weighted edges [6]. Figure 2.1 shows a fully connected neural network (each node in one layer is connected to each node in the next) with one hidden layer. This is a typical look of a neural network although the number of neurons(nodes) in each layer can vary greatly. The number of hidden layers can also vary, with any network using multiple layers being classified as a deep network [22].

The simplest form of artificial neuron is the McCulloch-Pitts [23] neuron, which only has two states, either it is active or inactive [6]. It calculates a weighted sum of its input values and if that weighted sum is greater than the neuron's *threshold* (also called *bias*), it will activate. A single neuron is described by

$$H(\sum_j w_j x_j - \theta), \tag{2.1}$$

where w_j represents the weight for the input x_j, θ is the threshold of the neuron

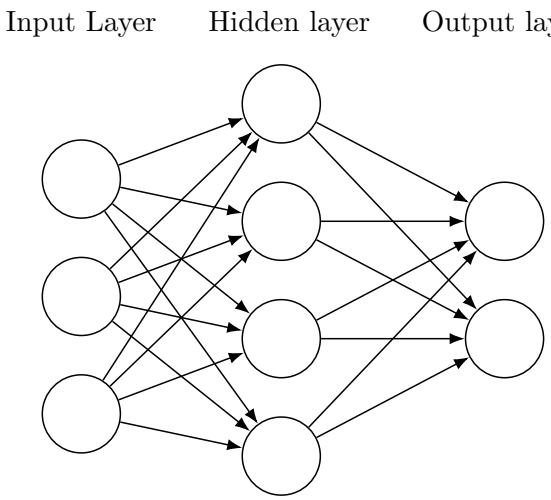


Figure 2.1: *A fully connected (dense) neural network with one hidden layer.*

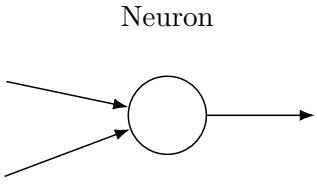


Figure 2.2: *The representation of a McCulloch-Pitts neuron. The two left arrows is the input and the output is the arrow to the right.*

and H is the Heaviside step function

$$H(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}. \tag{2.2}$$

The Heaviside function is the neuron's *activation function* and modern, more advanced neurons have continuous output values. These are facilitated by other activation functions such as ReLU(see section 2.3.2).

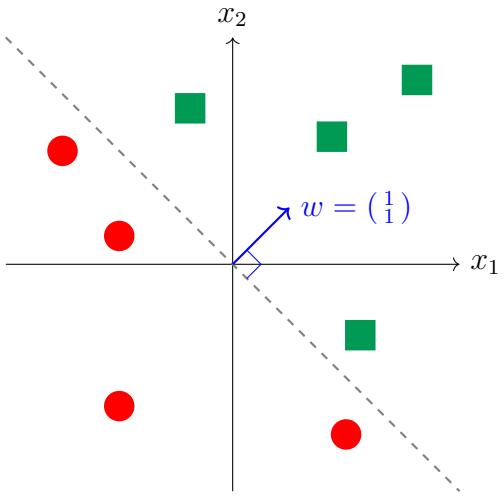


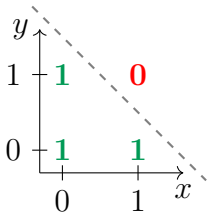
Figure 2.3: *A decision boundary representing a neuron with two inputs with both weights set to one and the threshold set to zero. The dotted line is the boundary classifying two different kinds of data, represented by green squares and red circles. The blue arrow is the representation of the weights of the neurons in vector format.*

In the case of a single neuron with two continuous inputs and a binary activation function as shown in Figure 2.2, one can interpret the neuron geometrically as a line going through the input plane, separating points on either side to one of the possible outputs [24]. This is shown in Figure 2.3, which shows the *decision boundary* of a neuron with two inputs, both with weights $w_1 = w_2 = 1$, and threshold $\theta = 0$. It can be seen that with both weights equal to 1 the weighted sum would be 0 when $x_1 + x_2 = 0$, on the line perpendicular to the weight vector. In other words the direction of the vector made up by the weights of each input to a neuron decides what angle the decision boundary has. Similarly the value of the threshold shifts the position of the line. Single neurons with N inputs can be interpreted similarly but instead of a line separating the space there is instead a $(N-1)$-dimensional hyperplane [24].

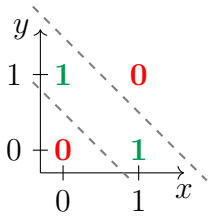
Since a neuron separates the input space with a single hyperplane, a single neuron can only classify inputs correctly if they are linearly separable. For instance, the XOR function

$$XOR(x, y) = \begin{cases} 0 & x = y \\ 1 & x \neq y \end{cases} \tag{2.3}$$

can not be classified by single neuron, as demonstrated in Figure 2.4a.



(a) *Incorrect XOR function classification*



(b) *Correct XOR function classification*

Figure 2.4: *Figure 2.4a shows an unsuccessful classification of the XOR function because of only using one neuron. The Figure 2.4b shows a successful classification of the XOR function thanks to using more than one neuron.*

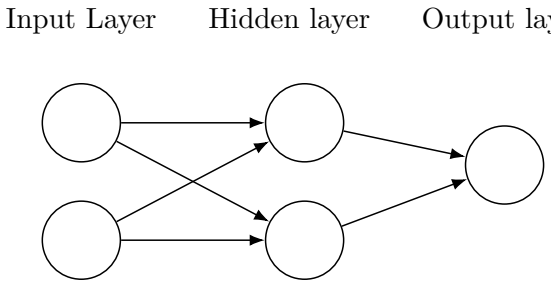


Figure 2.5: *Network of neurons used for successfully classifying XOR. Weights and biases of links are not shown in this figure.*

To successfully classify complex input, more neurons need to be introduced. Figure 2.5 shows a layered feed-forward structure that can do classify XOR correctly. Here, each neuron receives input from the previous and then feeds its output forward to the next. For such networks, the first two neurons can create decision boundaries as seen in Figure 2.4b, each activating and outputting the value 1 for three of the points. The last neuron can then have its weights and thresholds set up so that it activates if both of the previous neurons are active, thus making the final output only activate for the points between the decision boundaries, successfully classifying the XOR function.

With more layers and more neurons in each layer, the network will be able to correctly classify data that has more complex boundaries. However, if the network becomes too complex it can lose its ability to generalize in favor of memorizing. This is called *overfitting* [25, p. 93].

2.3.1 Optimization Algorithms

In machine learning, the optimization algorithm is how the program can learn by itself. For an ANN, learning is based around updating the weights and biases between neurons. The performance of an ANN is determined by the *loss function*, a measure of how close the actual output of the ANN is to the target output.

One loss function is the squared error

$$E[w] = \frac{1}{2}\sum_{i\mu}(t_i^\mu - O_i^\mu)^2. \tag{2.4}$$

where O_i^μ is the output from output neuron i for pattern (input) μ and t_i^μ is the expected output of that neuron for pattern μ, its label. This sum generally has a positive value, but if the network labels input correctly it approaches zero [24]. Since O_i^μ is a neuron it is defined similarly to (2.1), but with a different activation function and an extra index to specify which neuron in the layer the weights and threshold are connected to. It is described by

$$g(\sum_{ij} w_{ij}x_j - \theta_i). \tag{2.5}$$

The loss function has a global minimum when all patterns are correctly classified. Since it is a function of the threshold and weights for each neuron inside the network, the performance of the network can be improved by changing the weights according to

$$\Delta w_{ij} = -n\frac{dE}{dw_{ij}}. \tag{2.6}$$

That is, the change in each weight, Δwij, is proportional to the gradient of E [24, p. 104]. The threshold for each node in the network should also be updated, this can be done similarly to how the weights were updated, but through derivation with respect to the threshold instead.

By repeatedly feeding input through the network and updating the weights and thresholds, the network should converge towards a local minimum, however by adding some randomness to how patterns are fed one hopes it converges towards a global minimum where it can correctly classify all input patterns. This is called stochastic gradient descent.

Ideally, the network can also generalize to successfully classify patterns that it has not seen before. Whether this happens or not in practice depends on many factors, such as if the learning rate n is properly set or if the network structure is appropriate for the problem. The learning rate is the factor of which the network updates its weights and thresholds. There are a lot of different techniques that can be used in order to potentially improve the network's learning ability, for example by lowering the learning rate over time.

An extension to stochastic gradient descent is the optimization algorithm called *Adam*[26], which has its roots in two other algorithms, *RMSProp* and *AdaGrad* [27].

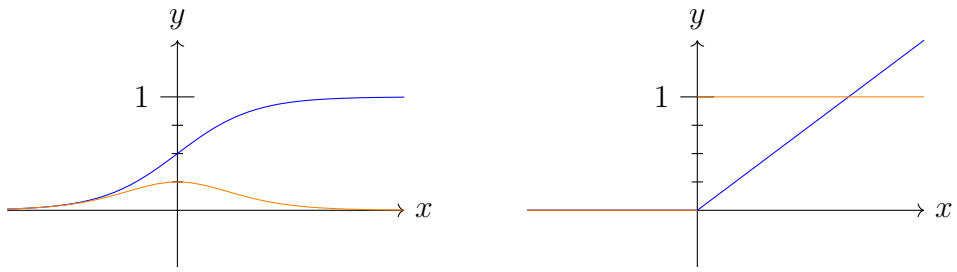
The principle behind AdaGrad is to give parameters which receive big updates a smaller effective learning rate while the effective learning rate is increased for the parameters which received a smaller update [28].

RMSProp is described by Dahal, P [28] to offer a solution to the problem related to the monotonically increasing part of AdaGrad where the learning rate is steadily decreasing and can thus approach a point where it stops learning altogether.

Adam is the combination of adaptive learning and momentum learning with an additional bias term, which is explained by Dahal, P [28].

2.3.2 Activation functions

In artificial neural networks all nodes use an activation function. Commonly used functions are Sigmoid, Tanh, and different types of ReLU. The purpose of the activation function is to modify the output of each neuron to follow a certain behavior. An example of one such behaviour is using the sigmoid function as an output of a neural network since it can be interpreted as a probability.



(a) *Sigmoid* **(b)** *ReLU*

Figure 2.6: *Graphical visualization of the Sigmoid and the ReLU funtion. The blue curve is the ordinary function, $f(x)$, and the orange is the derived function, $f'(x)$.*

Sigmoid and Tanh are used when the desired output of the neurons are values between zero and one, respectively one and negative one. A common problem associated with both the sigmoid (Figure 2.6a) and the tanh as an activation function is the *vanishing gradient problem* (see section 2.3.3).

ReLU stands for rectified linear unit. The derivative of the ReLU (Figure 2.6b) functions is constant, namely 1 and 0. This reduces the risk of the vanishing gradient problem, at the cost of not having a upper bound of the output values of the neurons.

2.3.3 Vanishing Gradient Problem

Often, it is not a good idea to create too deep networks due to the vanishing gradient problem. This problem often occurs when a network becomes too deep. A vanishing gradient refers to an exponential decrease of the size of the gradient, thus diminishing the effects of the neurons in the layers [25] and slowing down the learning process. A proposed solution to the problem is *Long Short Term Memory networks* [29]. The ReLU function (see section 2.3.2) is believed to partially negate the vanishing gradient problem since it does not saturate (approach a limit) as its input grows [25].

2.3.4 Long Short Term Memory Network

A Long Short Term Memory network (LSTM) is a variant of RNN that is designed to solve the issue of preserving long term dependencies that appears in ordinary RNNs [30]. LSTMs contain a memory cell that stores and updates a state of information [31], [32] that the network itself deems to be relevant to the context of what it is currently trying to predict. For each step in time, the LSTM forwards the current

state of its cell to the next iteration or step in time, allowing it to to take previous data into consideration when making decisions.

The state of a cell is managed through three different structures called gates, each with their own purpose [30]. There is the forget gate, the input gate, and the output gate. The forget gate learns to decide which data in the cell state is not necessary to store anymore. The input gate decides on which values in the cell state to update, as well as which new values should be entered. Lastly, the output gate learns to determine which part of the cell state we want to output.

The LSTM architecture is designed specifically to solve the vanishing gradient problem [33]. As mentioned in the previous section, this issue arises when a value that is meant to alter the importance of an event from a previous state of time is multiplied over and over with each other. LSTM networks monitor this issue with its gates, and the flow of information in and out of the hidden state is guarded by these gates [29]. Because of the gates ability to continuously update the gradients value, or even shut the gate off by setting the gradient to 0, thus allowing no new data to enter or to be removed from the hidden state, they make sure that the importance of the the hidden state does not end up as undervalued.

2.3.5 Convolutional Neural Networks

Humans can quickly identify an object in a picture after having learned what the object looks like, even if the picture overall is drastically different from previous instances that the object has been seen in. For computers this is difficult, since a picture is nothing but a matrix with values representing each pixel rather than observable objects with context. Teaching machines to identify objects quickly and accurately takes a lot of computing power given a densely pixelated picture. To solve this problem a neural network that uses *convolution* was created by Yann LeCun and his team in 1994 [34].

Convolutional Neural Networks (CNN) are networks that are built hierarchically with *convolution layers* and *pooling layers* that are altered as seen in Figure 2.7.

Convolutional Layers take an image as input, and outputs a set of feature maps. The image input can be perceived as a 3-dimensional matrix or a set of 2-dimensional matrices, where each set represents a channel in the image (e.g 1 channel for grayscale images, 3 for RGB etc). Similarly, the feature maps are multidimensional as well, with each index being the output of a single neuron from the layer. In contrast to fully connected layers, convolutional layers are not fully connected, instead each neuron is only connect to a small square section of the input layer. The weights of the neurons are different as well, with a convolutional layer's neurons sharing their weights with each other. This allows the network to detect features in the image without being influenced by the coordinates that the feature has [35]. One can interpret the convolutional layer as a square, called a *filter*, moving across the input and doing calculations with the data inside of the square. Parameters for the layer change the square's behaviour, for example *stride* changes how many pixels it moves at a time. An example of how the input is mapped to the output of a convolutional layer can be seen in Figure 2.8, where a small 4x4 image with only one channel is connected to a convolutional layer using a 2x2 filter. For more complex convolutional

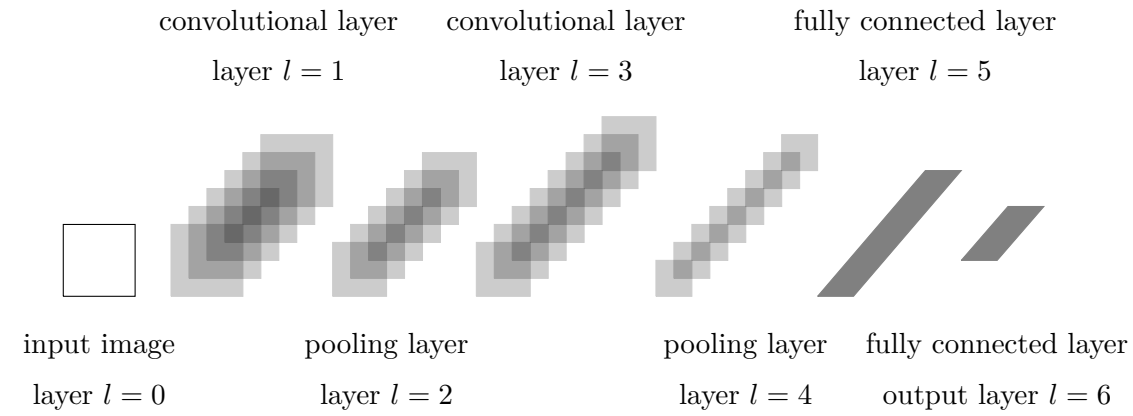


Figure 2.7: *The architecture of the original convolutional neural network, as introduced by LeCun et al. (1994). Each layer of pooling reduces the size of the feature maps by filtering out only the spatial information. The feature maps of the final pooling layer are then fed into the actual classifier consisting of an arbitrary number of fully connected layers.*

layers each channel of input has its own set of weights. Furthermore it is possible to have multiple filters and feature maps, effectively connecting the input to an additional convolutional layer. For further reading, see [35].

Pooling layers are layers that run through the feature maps produced by a convolution layer. The pooling layer merges the semantically similar features in the feature map [36]. This layer is often placed after a convolution layer.

The pooling layer usually used is max-pooling, which also uses the same principle as the convolution layers of having a window that run through the feature maps, but instead of performing discrete convolution it takes the maximum value of each of the filters. The output is then a densely computed layer which density is determined by the filter size and the stride length.

2.4 Supervised learning

Supervised learning is when a machine learns from already labeled data by trying to classify it with the same labels [37], [38]. Models trained with supervised learning are typically used for classification or regression [37], [38]. Classification being when data points are classified as one of several groups, such as when recognizing images of different kinds of fruit, and regression being when a value from a continuous scale is estimated, such as estimating the value of a house given its floor plan.

In comparison to *reinforcement learning*, a drawback with supervised learning is the need for large amounts of labeled data. Labeling data is usually very labor intensive which creates a bottleneck for the development process. The quality and variety of the data itself is also critical.

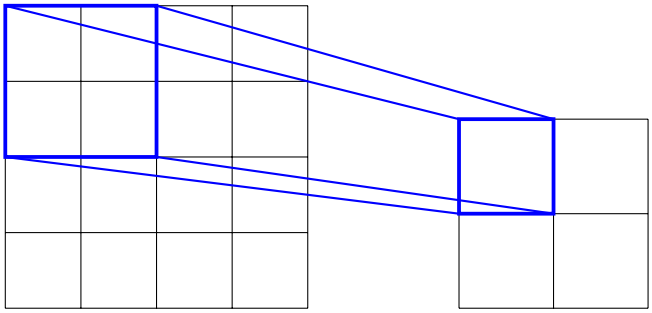


Figure 2.8: *Example of how a convolutional layer maps input (left) to output (right). The size of the filter changes the size of the blue square in the input, while the stride changes how it moves. This example uses a stride of 2, this maps the other quadrants of the input to their respective output quadrants. Using a stride of 1 would result in shifting the square one step at a time, increasing the output size to 3x3.*

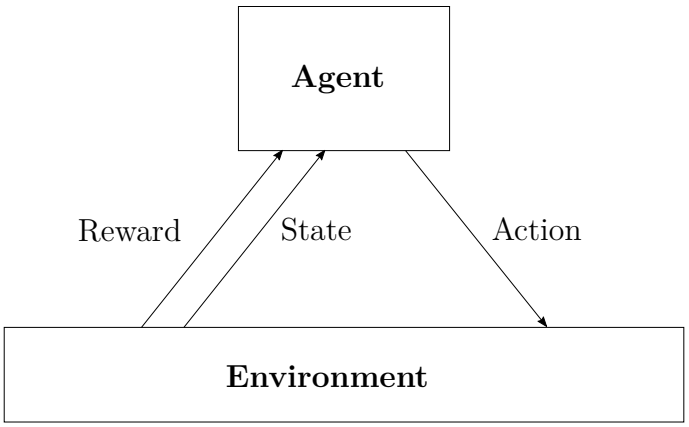


Figure 2.9: *An agent using reinforcement learning, having reward and state as inputs from the environment and action as an output to the environment.*

2.5 Reinforcement learning

Reinforcement learning (RL) is the process where an agent optimizes its decision-making by maximizing a reward through trial and error [39], rather than being told what to do. This is complicated by the fact that the greatest rewards often are hidden behind stretches of low or no reward. As such, the agent must learn to take certain actions even when the reward is delayed. In the book Reinforcement Learning [39] trial and error, and delayed reward are recognized as "the most important distinguishing features of reinforcement learning".

As visualized in Figure 2.9, a reinforcement learning agent is a function that takes two inputs: the reward, and the current state of the environment. It will then produce some output action that in turn affects the environment state.

RL has been proficient in areas like games when there is a distinct reward system for each action executed [40]. There are also different kinds of reward systems one can implement into the learning phase. An example would be Super Mario where the goal could be to reach the finish line, of which the reinforcement algorithm

will give positive rewards when Mario goes forward and negative when he moves backwards. Another reward system for the same game could be how many points you can achieve, of which the positive rewards increases every time Mario collects coins.

2.6 Deep Q-Learning

Deep Q-Learning (DQL) [40] is a reinforcement learning algorithm derived from a simpler model called *Q-learning*. Q-learning is a way to provide the agent with the ability to learn to act optimally in a Markov chain, this is done by experiencing the consequences without building a complete map of the domain and could be viewed as asynchronous dynamic programming [41].

Algorithm 1 Deep Q-Learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode= 1,M **do**
  Initialize sequence $s_1 = x_1$ and pre-processed sequenced $\phi 1 = \phi(s1)$
  for t=1, T **do**
    With probability ϵ select random action a_t
    otherwise select $a_t = max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and pre-process $\phi_{t+1} = \phi(s_t + 1)$
    Store transition (ϕ_t,a_t,r_t, ϕ_{t+1}) in D
    Sample random minibatches of transitions (ϕ_j,a_j,r_j, ϕ_{j+1})
    Set $y_j = \begin{cases} r_j, & \text{for terminal } \phi j + 1 \\ r_j + \gamma max_a Q^*(\phi(s_{t+1}), a'; \theta) & \text{for non terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to an equation
  end for
end for

Figure 2.10: *Algorithm for Deep Q-Learning with experience replay as described by DeepMind in their 2013 paper [40]. This algorithm is used the reinforcement agent in this paper.*

In DQL one trains an artificial neural network to approximate the reward, r, that an agent would get for performing an action, plus a factor of the maximum future reward if the agent plays optimally according to the network. In normal Q-learning this is achieved by constructing a table of every single state as well as what cumulative reward the agent would get for taking every possible action in those states, rather than using a network [41].

DQL utilizes *experience replay*, a form of data that saves the agent's experiences

at each time step into a replay memory. This memory is used during training so that the agent utilizes previous experiences in its training [40].

A Deep Q Network (DQN) is a network that uses DQL for learning. DQN can have different policies, the paper however focuses on the ϵ-greedy policy. This policy integrates *exploitation* and *exploration* decision-making, which helps with the training of the agent. Exploration is the probability that the agent chooses a random action in order to explore different areas. Exploitation makes the agent perform the action that it currently believes is the best. A typical DQN uses a degrading exploration rate to make sure that the agent tries many random choices in the beginning, gradually optimizing the decision-making with the help of exploitation.

The main algorithm in Figure 2.10 was used for Deep Q-learning by DeepMind in their 2013 paper "Playing Atari with Deep Reinforcement Learning" [40] and is also used for the reinforcement agent in this paper.

2.7 Tools

TensorFlow [15] and **Keras** [42] are APIs initially written for Python that allows for easy creation of different neural network models. Both APIs use modular components to allow for fast experimentation. This means its possible to create high performing neural networks with a few lines of code.

PySC2 is a tool which implements the Blizzard API to make it easy to create and train agents with the functionality it includes out of the box. One of its most prominent features is the multiple image layers (called feature layers) with different game information that one can extract, these range from which units occupy certain pixels on the screen to which parts of the map that are visible. One can use these images as inputs to an agent, which makes it easier for the agent to interpret the game as it is an abstraction of it. Another feature that PySC2 provides is the ability to send actions to the game client, allowing agents to act.

Sc2reaper [43] is a tool for extracting game data from replays. With the agents using the game's state as input, and replays only containing a list of actions performed in a match, this program allows one to convert the game's replays into game states. The data is stored inside of a database using MongoDB.

MongoDB [44] is a relatively new type of database, NoSQL, that has removed the concept of table, rows, SQL, and schemas. In contrast to older types of databases, such as MySQL and PostgreSQL, MongoDB has no transactions, foreign keys and other typical features. MongoDB is a document-oriented database [45]. MongoDB stores its data in flexible BSON documents, which are Binary JSON documents [44]. This means that fields can contain other documents as well as data structures that can change over time. BSON is efficient both in storage and scan-speed [46].

3
Method

During the project several agents were created. First a scripted agent to test the functionality of the API was created and then an agent made for gathering minerals was created and trained using reinforcement learning. Finally an agent was created for predicting the chance of winning a game in any given time. This was all done on hardware heavily limited in power, generally using normal consumer laptops or desktops instead of servers or processing units made for the task. The hardware used for training the networks included an Nvidia GTX1070, Nvidia GTX1080, an Intel Core i5-6440HQ CPU with 3.5GHz, an HDD, and an SSD.

3.1 Creating artificial neural networks

Input data from the game came in two different formats: layer data such as screen data and minimap data, and numerical values such as the player's amount of gas or mineral. The input motivated using a model that could receive both numeric values and layer data. A combination of a dense network and convolution network would solve this problem, thus this structure was chosen for any network that also receives images as input. All the models provided in this project were created using Keras.

3.2 Reinforcement Learning

Several different reinforcement agents were created. Initially a scripted bot was created to test the newly created simplification of the PySC2 API, and to serve as a reference for how well the trained agents should perform. The testing was done on a mineral collecting mini-game, which is a single player game mode with the goal of collecting as many minerals as possible. In SC2, minerals are one of two resources for creating units, but in this mini-game, they are mainly used as score. The possible actions used by the agents were: No operation (Do nothing), build SCV (worker unit), build supply depot (increases the amount of workers the agent can have in total) and send a single worker to harvest minerals. These four actions were continuously used throughout the reinforcement learning process. During training, a game from start to finish is referred to as an *episode*.

The scripted bot followed a very simple set of rules:

```
if justSelectWorker:
    harvestMinerals
else if areWorkersFree:
    selectWorker
else if SupplyFree < 3 and Minerals > 100:
    buildSupply
else:
    buildSCV
```

This made the bot immediately send any free workers to harvest minerals while always producing new workers, with the exception of a small gap near the start of each episode as the first supply depot is being built.

With the given action space the scripted bot's actions should be close to the optimal strategy, which should be to always build workers while making all of them work as quickly as possible.

3.2.1 The first agent

Following the scripted bot, a simple Deep Q-network was used with an ϵ-greedy policy. The initial exploration rate used was 1, this was then multiplied by 0.9999 each frame, with a minimum exploration rate of 0.05.

This agent had a simple reward function that gained a positive reward when selecting a worker-unit, and a negative reward when performing any other action. It was made with the purpose of learning to only select different workers, ensuring that the constructed neural network, in conjunction with the DQL algorithm, was actually working.

3.2.2 A smarter agent

The agent was further developed by repurposing it to play the CollectMineralAndGas mini-game, where the objective is to collect as much minerals and gas as possible within the time frame. However, because the production of terran units within our scope did not require any gas, we opted to have the agent exclusively gather minerals.

This agent used a fully connected dense network with a DQL algorithm for training. Its reward function was

$$R = |W_{F(i-1)} - W_{Fi}| * (0.5 + W_i/50), \tag{3.1}$$

where W_{Fi} is the number of free workers at step i and W_i is the total number of workers at step i. It calculates the absolute difference between the amount of idle (free) workers on the previous frame and current amount of idle workers, meaning that the network gets a reward whenever a worker starts a task, finishes a task or is created. This is then multiplied by a value based on the amount of workers that the agent has in order to further incentivize creating workers.

Table 3.1: *The data used for supervised learning. Screen and minimap data is collected from the PySC2 graphical feature layers. Numeric data is collected from the PySC2 structured data.*

Screen Data	Minimap Data	Numeric Data
Factions	Factions	Frame Number
Units	Selected	Minerals
Health	Vision	Vespene Gas
Selected		Supply Total
Height		Supply Used
Vision		Supply Army
		Supply Workers

3.2.3 Extending the agent

Extending the agent for more complex tasks, the dense network was replaced with a CNN in order to take advantage of the layer data that PySC2 provides, which hopefully allows for more nuanced decision-making. The CNN was initialized with nine channels, presented in the Screen Data and the Minimap Data coloumns in Table 3.1. This input was retrieved using the simplified API created in the earlier stages. When initializing the network the size of the frames from PySC2 was chosen to 64x64, which makes the layer sizes individually 64x64 as well.

3.3 Supervised Learning

To supplement the DQN, another network was built with the purpose of determining the probability of a player winning when given that player's perceived game state. With its output serving as a clear indicator if a given state is beneficial or not the network would then be utilized as the reward function for a neural network using reinforcement learning.

In order to train the network, a large amount of training data was required. The intuitive course of action was to make use of prerecorded replays since they should provide a wide variety of different game states for the network to practice evaluation on.

Since StarCraft II stores replays of games as the sequence of the actions the players executed, it is impossible to directly extract a game state from the replay file without first simulating the match up to that point. In addition, simply running the replay directly with StarCraft II does not make the data available in the format PySC2 uses. Instead, a tool to extract and format the data was required.

Sc2reaper was used to extract individual frames of a match store it in a MongoDB database. However, due to the default version of Sc2reaper not storing data in the same format that PySC2 wants, Sc2reaper was modified in order to make a version

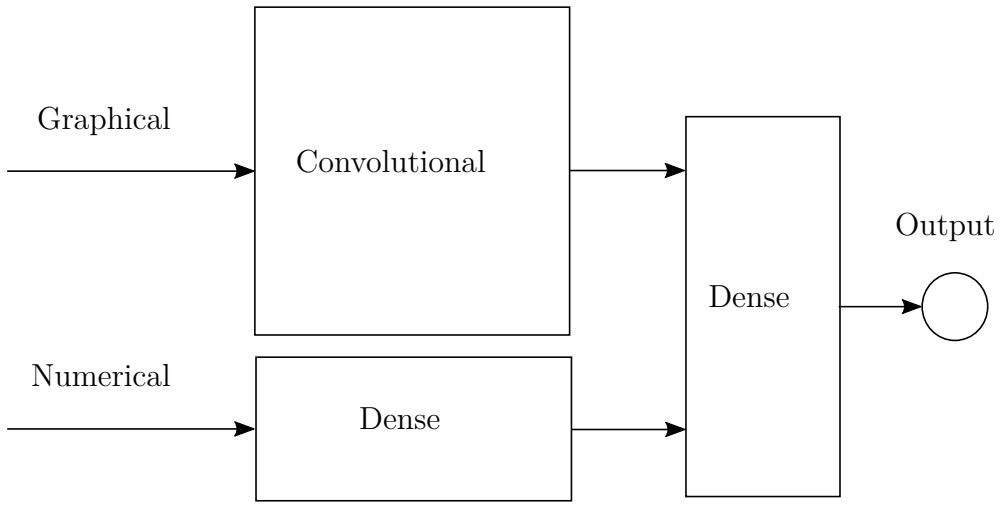


Figure 3.1: *An overview of the convolutional network model used for reinforcement and supervised learning. The convolutional part has a graphical input from SC2, while the first dense network takes numerical data (number of minerals etc.). The second dense network merges the two previous mentioned networks to create one single large network.*

which does use the correct format. In addition, functionality to parse an entire folder of replays was added to reduce manual labor.

To acquire the training data, SC2 replays of game version 4.8.3 were scraped from Spawningtool [47], a website where users submit their own replays. Due to a small fraction of replays actually being version 4.8.3 from Spawningtool even though they were tagged as such, we had to scrape more replays, this time from GGGReplays [48]. This yielded just over 1000 usable replays in total, with the total number scraped being 10 000. The modified version of Sc2reaper was then used to parse the replays, sequentially adding each one to the database.

After parsing about 240 replays, gathering the data seen in Table 3.1, a patch for StarCraft II was released. This automatically updated the game on the computer building the database which made all subsequent replays crash upon being parsed preventing the remainders from being used. Due to time constraints, this version incompatibility problem was not fixed.

Training was done on a subset of the replays that were put on an SSD. This was approximately six times faster than a normal HDD where they were otherwise stored. Due to space limitations approximately half of the matches were used as training data on the SSD, while the other half was left to use as validation data.

A network was then created, which had a convolutional part handling the image data from the screen and minimap as well as a dense part handling the numeric data, shown in Figure 3.1. Both parts were then connected to a final dense part before the output layer. Different versions of this network with varying hyperparameters were then trained on the dataset. Each new version of this network had its hyperparamters changed in an attempt to reduce the complexity of the network, generally

neuron count in the dense layers was reduced as well as reductions to the amount of feature maps used by the covolutional layers. The output from the network was one node using the sigmoid activation function to get values between zero and one as the probability of victory.

4
Results

This chapter presents performance metrics of a few agents based on different ANNs and describes the function of the created PySC2 wrapper API. All relevant code is available on our GitHub repository[1].

4.1 A simplified API for PySC2

A wrapper for PySC2 has been created to handle both getting data from the game and sending player actions to it. Specifically, functions have been created that return select data from the PySC2 observation object. This removes the need to dig through PySC2 code to figure out how to get data that can now be accessed via the simple API. If PySC2 data is required that can not be accessed via the simple API, looking at the API code and documentation can help with understanding how to write the PySC2 query for that data too.

Common terran actions are exposed as functions by the simple API to make it easier to use them with correct parameters. This was done since those parameters are often poorly documented in PySC2 and require a deeper understanding of SC2 and PySC2 than should be necessary for most cases.

In order to prevent crashes, the previously mentioned functions check if each action is available and legal. For example making sure any units exist before issuing a select all command to the game. If the action is not available, no operation is performed instead. This may not always be optimal, but it is a reasonable default behavior when starting out with AI development in SC2.

Finally there is a framework for defining composite actions that consist of multiple PySC2 actions. This makes it possible to expose the agent to actions on a higher level of abstraction, such as defining one "build unit" action rather than requiring the network to learn that it can build a unit by taking several specific actions after one another.

4.2 PySC2 data extraction from replays

A web scraper to collect replays and a tool for sorting replays by matchup (such as terran vs terran) and game version has been created to make it easier to gather replays from an up-to-date SC2 version.

A modified version of Sc2reaper has been created so that it is now trivial to extract

[1] `https://github.com/ecen/starcraft2-ai`

and save game states in a format that PySC2 can read. In addition, functionality has been added to automatically run Sc2reaper on multiple replays in sequence so that the database can be filled with minimal user effort. Finally, there is documentation on the parts that have been edited, how to edit them further in order to customize what data is stored, and on how to solve or circumvent issues that we encountered when trying to parse replays.

4.3 Supervised Learning

The victory prediction network that was supposed to determine if a player is winning when given a state from the game, has not shown much success. While the network's loss function, mean absolute error, has gone as low as 0.14 on the training set before training was terminated, there has been no significant improvement on the validation set for any version of the network.

As expected, running the network on a live match of StarCraft II yields some weird results. In general the network predicts a win chance very close to either 0 or 1, with values closer to 0.5 being rare. No clear pattern has been found regarding the network's output relative to its input either. Instead the network acts in seemingly unpredictable albeit often humorous ways.

One such behaviour includes predicting with a certainty of 1 that the player will win while looking at their base containing an army of a dozen marines. Slightly later into the match when the opposing player's army arrived, the network still had a firm belief that the first player was guaranteed to win the game. A few seconds later, when the first player's army had been decimated and every building except one had been destroyed, the network still had full faith in the player's ability to recover and win an easy victory.

Another example of the network's behaviour is the network fully believing that the player would win upon seeing a battle which had just started, only to suddenly flip the prediction to a guaranteed loss when the same player obliterated the opposing army.

4.4 Reinforcement Learning

Reinforcement agents showed a clear increase in performance compared to random action selection in a time-limited mineral collection mini-game. Figure 4.1 shows the score achieved in the mini-game in each episode of training for the best network. The score is equal to the total number of minerals collected in one episode. When selecting random actions at every step, the agent received scores of 1000 to 3000. In comparison, after training for 300 episodes using reward function (3.1), the network scored between 3000 and 5000. This is a clear increase in performance for the trained network. As can be seen in the same figure (Figure 4.1), the trained network's max value is very similar to that of a good scripted agent, but its minimum value is lower.

Reward function (3.1) was continuous, that is, it was applied after each step, rather than sparse and applied only at the end of each episode. Its performance can be compared to those of a sparse reward function, R, where R is the total minerals

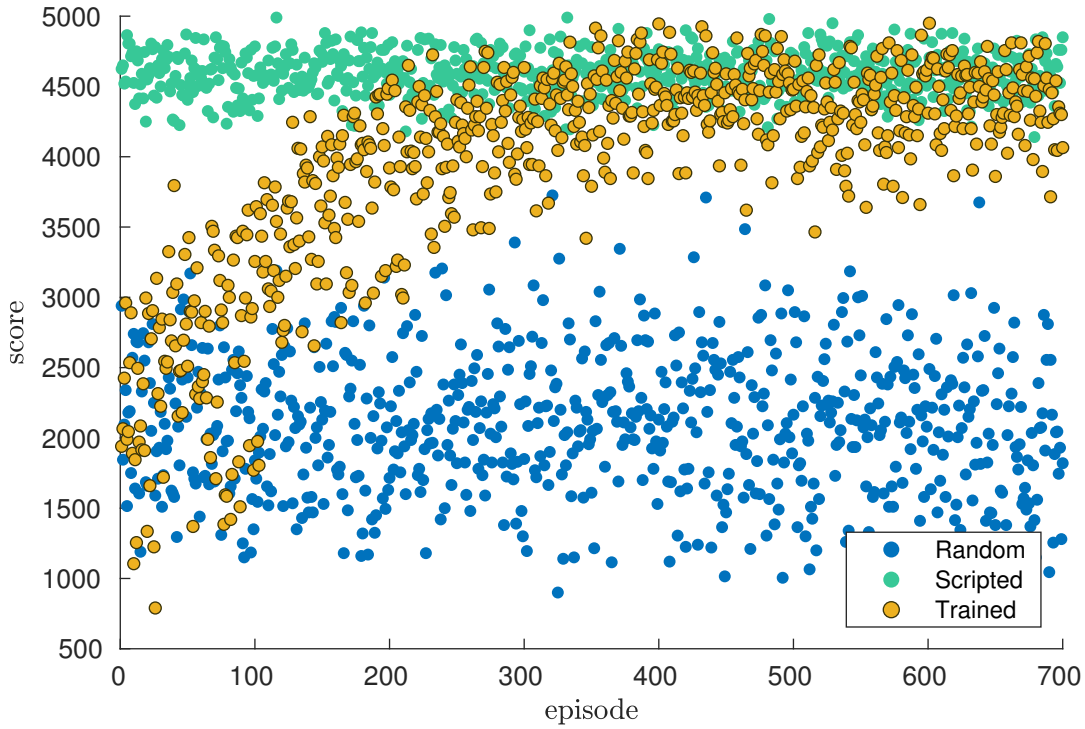


Figure 4.1: *Score versus number of episodes trained on the mineral collection mini-game for two networks and one scripted agent. The first network always performs random actions, the scripted agent acts in a way that should be close to optimal, and the second network is trained with a reward function* (3.1)*. This shows an improvement for the trained network over selecting random actions but with more spread downwards than the scripted agent.*

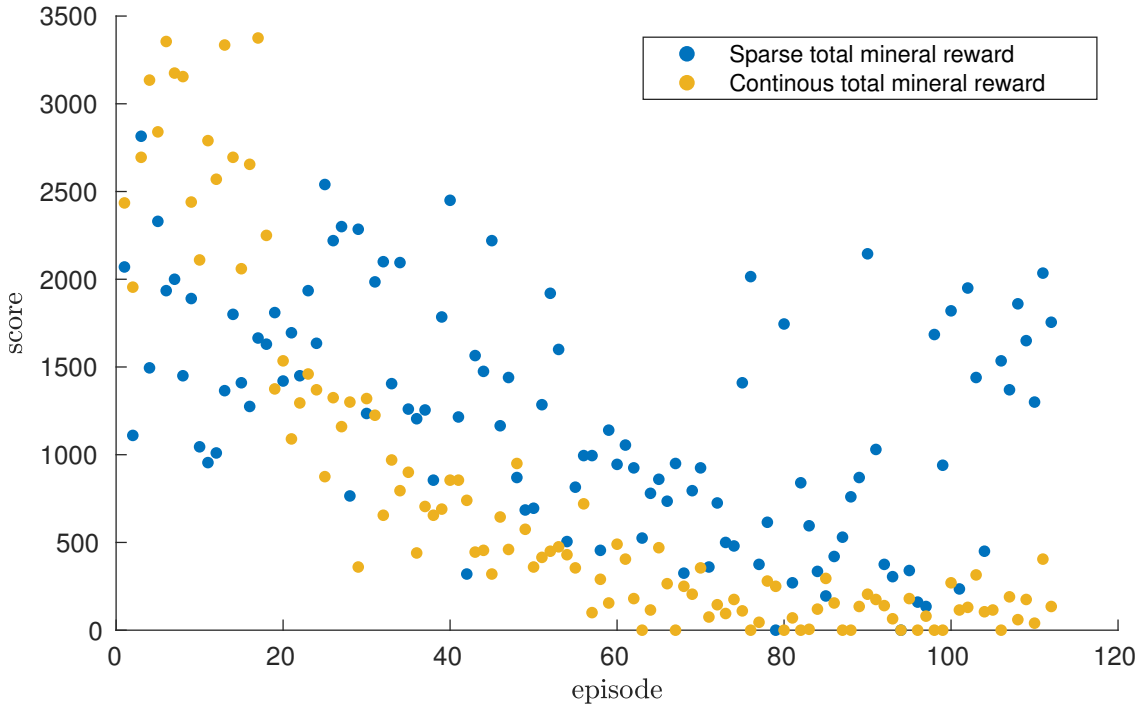


Figure 4.2: *Score versus number of episodes trained for a short time on the mineral collection mini-game for two networks using different reward functions. Both reward functions set R to the total number of minerals collected so far during the episode but the first is sparse (awarded at end of episode) and the second is continuous (awarded at end of each step). Both these reward functions seem to perform worse than that in equation* (3.1)*. Compare with Figure 4.1.*

collected during the episode, and a continuous reward function were R is the total minerals collected so far, see Figure 4.2. Note that these networks were trained using a steeper exploration rate decay per step (a factor 0.99990 instead of 0.99999).

A more in-depth representation of the trained network from Figure 4.1 is shown in Figure 4.3, but trained for more episodes. Data points in the figure are colored to show how many workers have been built by the end of each episode. The solid line and the right y-axis shows how large the exploration rate is. The dashed line highlights where exploration rate reaches its minimum of 5%. The figure only shows the first 1500 episodes of training, but this network was trained for a bit more than 3000 episodes. The performance is very similar for all episodes from episode 1000 and onward.

A small cluster of episodes where the network only uses 15 workers appear around episode 200. In these episodes, the network is not building any supply depots, seemingly using all workers for mining rather than building supply depots at the cost of mining slower for a while. This behavior appeared in all networks trained using reward function (3.1) but it manifested itself in several different ways.

A notable behavior of the network in Figure 4.3 is the sudden change in performance once the network hits its minimum exploration rate of 5%, signified by a dashed line. This did not happen in the same way for other networks, not even for networks with identical hyperparameters. Figure 4.4 shows 1500 episodes of a

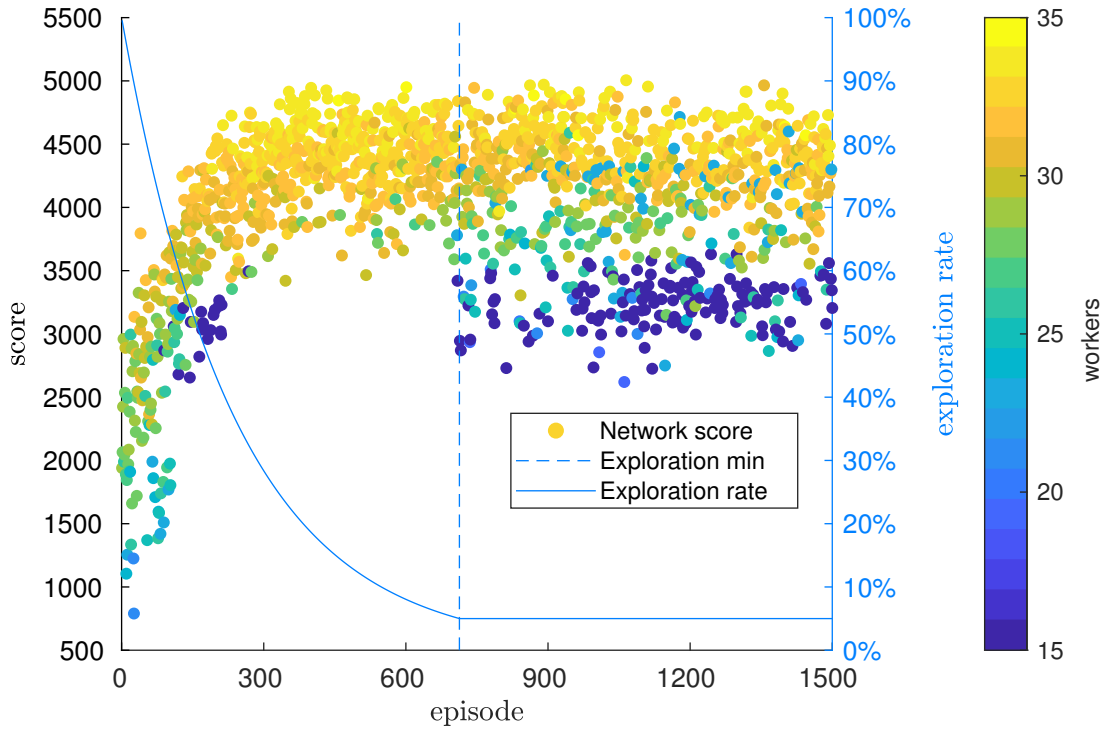


Figure 4.3: *The trained agent from Figure 4.1 but each data point is colored to visualize how many workers are available at the end of each episode. The line and the right y-axis shows the exploration rate at each episode. The small data point cluster to the left with very few workers was prevalent in many versions of the network with the reward function in* (3.1) *and represents the usage of as many workers as possible without building any supply depots. This behavior disappears but returns abruptly just as the exploration rate reaches its minimum value. The performance remains very similar from episode 1000 to at least 3000, after which the training was stopped.*

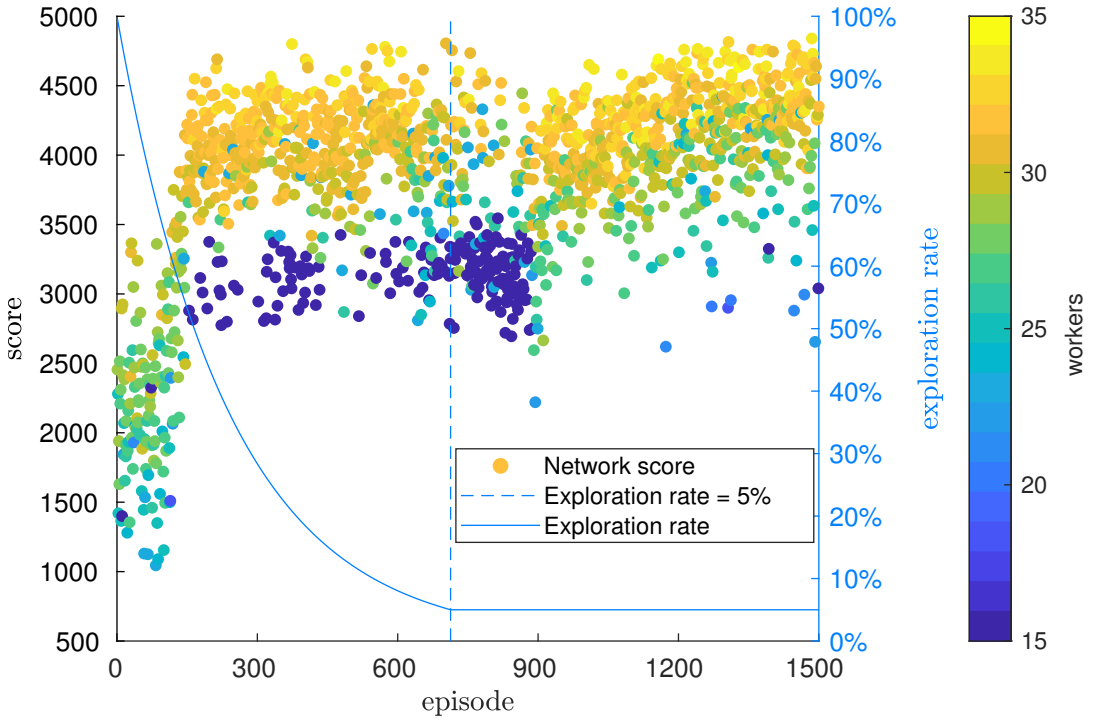


Figure 4.4: *A repeat training of the network from Figure 4.3. Even though all the hyper parameters are the same, this network performs very differently.*

network trained with the exact same parameters as Figure 4.3, and Figure 4.5 shows 3000 episodes of a network trained with the same parameters again but using a minimum exploration rate of 0%. The difference in episode count is due to the training process crashing before planned training time was over.

Considering the first 1500 episodes, Figure 4.5 is quite similar to Figure 4.4 since both have a band of episodes with 15 worker units quite early on. However, Figure 4.5 has a less prominent dip after the 5% exploration rate line. Both behave differently from Figure 4.3 which only has a small cluster of episodes with 15 worker units early, and then starts using a low worker amount after the 5% line. In addition, Figure 4.5 has a large dip around episode 1800 that was not present for the network in Figure 4.3. Figure 4.5 also shows a lower performance minimum after being trained than the other networks, including a worker minimum of 12, whereas other networks build at least 15.

Moving on to the convolutional agent described by Figure 3.1, this agent provided fewer results compared to the non-convolutional agent in previous figures that used only numeric data. Figure 4.6 shows the longest run of the convolutional agent. At the end of this figure, training terminated due to insufficient memory. Compared to the non-convolutional agent every episode took longer to train: 80 seconds instead of 20 on the same hardware (an Intel Core i5-6440HQ CPU with 3.5GHz). The non-convolutional agent consumed 35% of the CPUs capacity on average, while the convolutional agent consumed 60% on average. Running the convolutional agent on a GPU (GTX1080) yielded no improvement in speed on this simple network. However, initial testing showed that convolutional networks with more keras filters also took 80 seconds per episode on the GPU, but 220 seconds on the CPU.

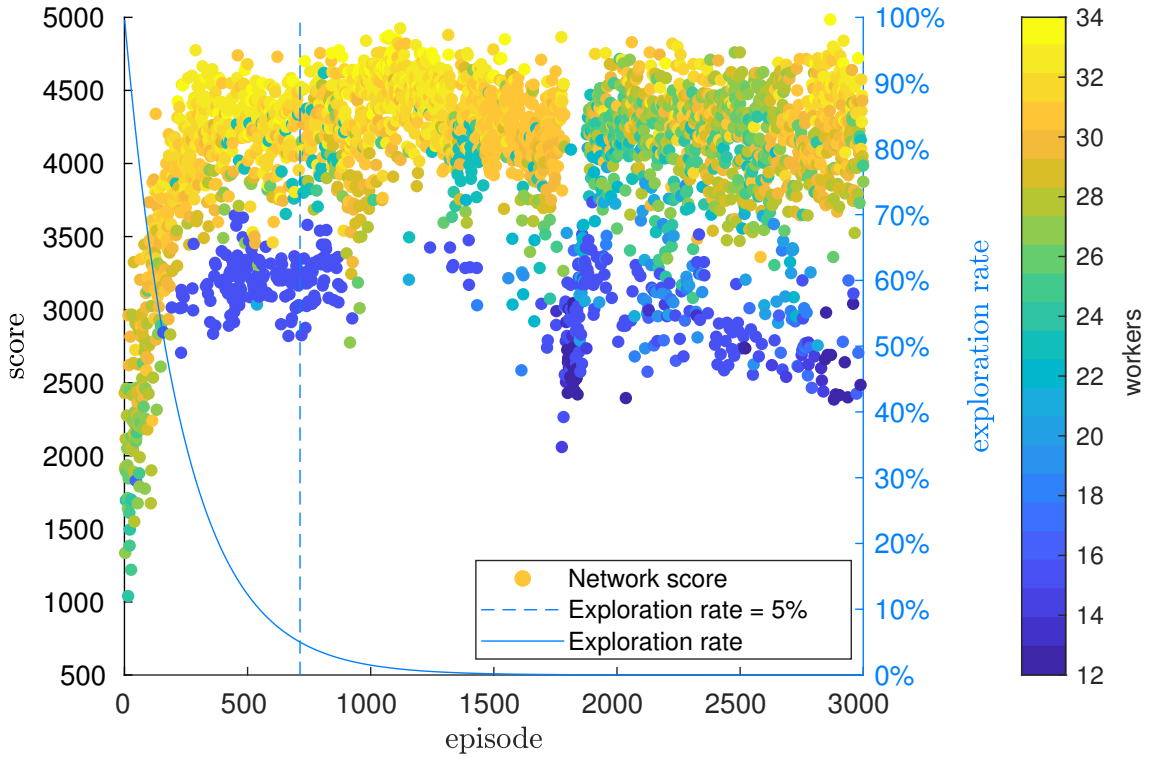


Figure 4.5: *A repeat training of the network from Figure 4.3 but with a minimum exploration rate of 0%. Even though all hyperparameters but the minimum exploration rate are the same, this network performs very differently from both Figure 4.3 and 4.4. Note that this figure shows 3000 episodes, not 1500. Note also that this network has a few episodes where it uses only 12 worker units. Previous networks used a minimum of 15.*

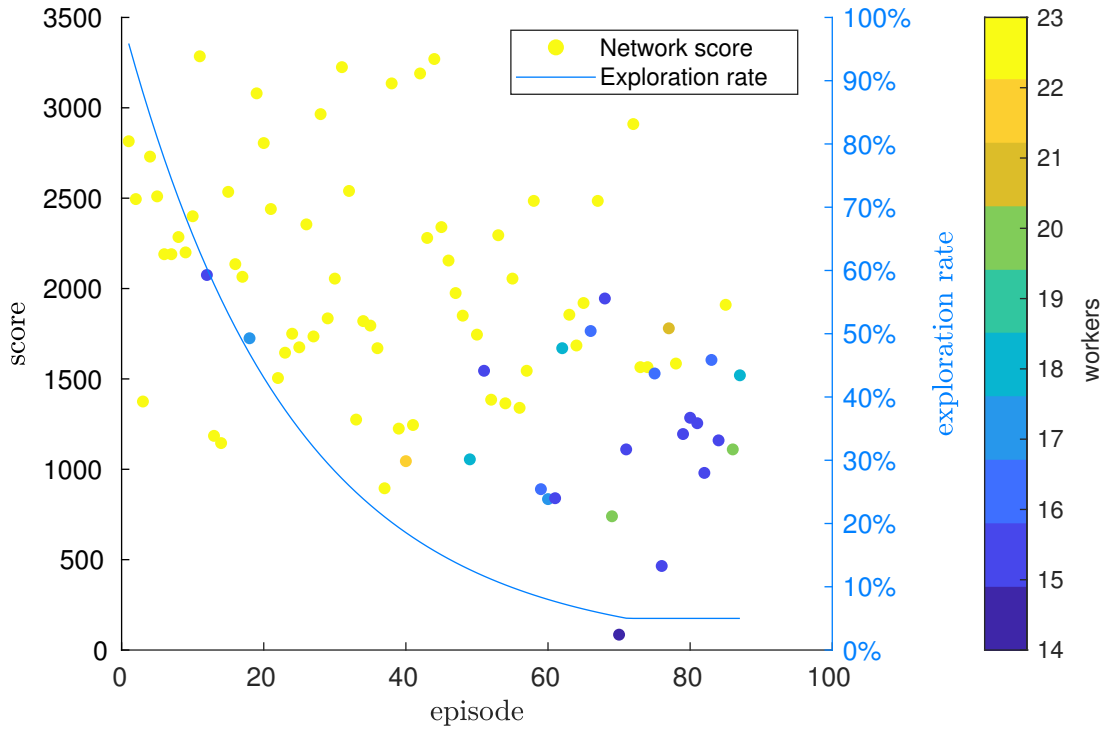


Figure 4.6: *Score and worker unit count at the end of each episode on the mineral collection mini-game for a reinforcement agent built with the convolutional network. This network was trained for fewer episodes than other networks since it ran out of memory after this point. It shows no improvement but might have done if it had trained for longer.*

5
Discussion

This chapter provides an analysis of the achieved results, clarifies the thought process behind certain decisions and discusses some problems that arose during development. Furthermore, improvements and suggestions for future research and development are given.

5.1 Replay data extraction issues

Significant time was invested into solving issues regarding replay collection and parsing. Sc2reaper allowed reading replay data, but extracting that data in a format that could be used to train an agent was more difficult.

One of the larger issues with Sc2reaper was that it failed upon trying to parse a replay from a version of SC2 that differs from the live version. This drastically reduced the amount of replays available since it would only be possible to use replays from the last two months, in contrast to potentially using replays from several years back. If multiple game versions had been used, balance changes to the game would probably affect prediction accuracy to some extent, but it seems reasonable to assume that the overall prediction would not be too far off. A network should still be able to predict who is winning a battle if one side has a large advantage.

After parsing approximately a quarter of the acquired replays, StarCraft II updated to version 4.8.4. Even though this patch contained no significant changes to the game, it immediately invalidated all remaining replays because Sc2reaper started throwing exceptions. Since it was a new patch there were not any replays available to download yet, restricting the training of the prediction agent to the 247 replays that had already been parsed for the rest of the project. This also locked in the types of data that was parsed since it was no longer possible to change what data was to be saved and then re-parse a replay.

Furthermore, the parser that was being used was not the intended final version, but rather an unfinished version where there was no support for a validation set yet. As a side effect of this there is not a validation set being used during training to identify overfitting and give a better overview of the network's ability to generalize. Instead what we have done is train on part of this database and then manually run the network on the other part to see if the loss value was significantly lower than the expected untrained value of approximately 0.5.

5.2 Supervised Victory Prediction Network

The supervised network saw neither an improvement in its validation loss function nor on manual test runs on live matches, while the training loss did decrease. This generally indicates that the network is overfitting the training data and failing at generalizing [25, p. 94].

The reason behind the overfitting is currently unknown and there are multiple possible reasons and solutions, including but not necessarily limited to having too few replays and data points, biased replays, the network structure, or lack of regularization and bad input.

It is worth mentioning that beyond filtering the replays to only include matches between two terran players, no further investigation was done into what the replays actually look like. Therefore, while somewhat unlikely, there is the possibility that the replays used contained matches which were biased in some way. A recurring pattern with the trained networks is that they tend to predict a high win chance when looking at the base on the right side of the Automaton map. This could indicate that most replays on that map happened to have the player that spawned on the right side as the winner, and since players spend a lot of time looking at their own base that area could then have been recognized as something that causes the player to win, even though the maps are symmetrical.

This could potentially have been mitigated by flipping the image data sent to the network depending on the player's spawn location, exploiting the symmetry to effectively make the input data always have the players base in the same spot.

Regularization was not used during the training but has been shown to help avoid overfitting [25]. It is also possible that the network itself could be the issue, the network's overall structure as well as its hyperparameters such as neuron and layer count could simply be the primary cause for the overfitting.

While training the different iterations of the network, different attempts have been made to solve the issue of overfitting, primarily by changing the parameters of the convolutional layers and the amount of neurons in the dense layers, each new iteration trying to make the network less complex than before. However, as a side effect of training around 48 hours per iteration, so that there is a wide range of versions to validate, there have only been around 15 iterations of the network in total. In contrast to the amount of possible choices that exist, with regards to neuron and layer count, very few options have been tried.

5.3 Reinforcement agent

The reward function (3.1) turned out to be effective. A few other functions were tested, but with little to no success. The two variants in Figure 4.2 represent two naive implementations of reward function that show no improvement for short training times. However, it is possible that at least the sparse reward of total minerals collected could yield a performance increase if trained for a longer time and with a slower exploration rate decay.

The network's tendency to sometimes use only 15 worker units was a surprising behavior, since it is far from the maximum amount of workers that can gather

resources at the same time. Figures 4.3, 4.4 and 4.5 demonstrates this behavior with clear dark blue bands representing episodes were only 15 worker units were created. In almost all cases this is likely to mean that no supply depots were built since the network is very likely to build worker units if there is free supply and it is exploring. Interestingly, Figure 4.5 shows that in a few episodes, the network with a minimum exploration rate of 0% built no worker units at all, choosing to remain with 12.

A network's tendency to not use supply depots often had a sudden start or end, and it did not seem to be gradually affected by training time or exploration rate with the exception for at the very start. With an initially high exploration rate, not building supply depots or worker units is very unlikely, despite the fact that episodes using 34 worker units almost always outperform episodes using fewer workers.

When an agent runs out of supply, there is no dramatic difference between states where it has built supply depots and where it has not. This is believed to be the reason for why the agent is inconsistent with building these structures. With the input to the agent consisting of the amount of minerals, free workers, free supply and if it selected a worker on the previous frame, an agent which uses all of its workers will have the variables for free workers and free supply stuck at 0 when all supply is used. Similarly, if the network selects a worker in this state the select-worker input changes to 1. If this does not cause the network to build a supply depot then three of the inputs effectively deadlock, with the only input still able to change being the amount of collected minerals. The input becoming mostly static makes it potentially impossible for the game to reach a state where the network decides to build a supply depot.

It is worth noting that in some cases it is possible for a smaller set of workers to outperform a large set by chance, a side effect of how the harvest minerals action was implemented. The map has two mineral lines with different distance from the main base, making it inefficient to use the far mineral line until the one close by is fully saturated by workers. However, when the network orders a worker unit to harvest minerals it will be sent to a random mineral node. This explains some of the variance between episodes, in particular that of the scripted agent which always performs actions in the same order.

However, this source of randomness does not explain all variance for the reinforcement agent. Figure 4.1 shows that the reinforcement agent has a larger variance than the scripted, but with similar max value.

Interestingly however, the network in 4.5 shows two differences to other networks. The first is that it sometimes does not build any workers, but remain with using 12. This is only possible when the exploration rate is very low. It is not clear why the network would do this since it performs worse in comparison to strategies that do build workers, and should have no negative effects or costs compared to not building them.

The second interesting phenomenon is that after the downward spike around episode 1800, the network starts performing quite well using somewhere around 25 worker units. The convolutional reinforcement agent required much longer training time than its non-convolutional counterpart due to the increased computational requirements, and might have required the slower exploration rate decay that most of

the non-convolutional networks used. Unfortunately, this could not be tested further due to difficulties with restricting how much memory was needed for training.

For experience replay, the state of the last million game frames and the resulting state after an action was performed is stored. This worked without issues while running the non-convolutional agents which only used four decimal values as input. In comparison, convolutional network's state contains multiple 64 by 64 matrices of data. Their size was not directly measured, but equivalent data stored in the database used for the supervised network had an average size of 150KB (although this might include some overhead). With the memory storing two of these per game frame, 300KB, trying to store a significant amount of states so that the network has access to a large pool of data to draw from in this fashion is not viable. As a reference point the laptop that ran the network had 8 GB of RAM, while a million of these states, assuming they share the same size as the database version, would take up 300GB. Therefore either a revision of how the memory is handled, or a massive RAM upgrade is required.

5.4 Future Work

This section is dedicated to discussing tracks that could be of interest for future work within the scope of this paper.

5.4.1 An SC2 Agent with a contextual network

A further improvement that can be done for the product is the ability to consider previous states of the game when deciding its next choice of action. A human player is expected to remember previous events and information in an ongoing match, because of how important it is to consider the opponents game plan when making your own decisions. For example, if the enemy's army is not currently visible, but a moment earlier it was seen entering a hidden area between the two bases, that is likely to be indicative of an upcoming attack.

This could be possible by integrating some variant of a LSTM. As explained in section 2.3.4, LSTM is a variant of RNN designed to solve long term dependencies. One popular variant of LSTM networks is the *General Recurrent Unit* (GRU), which has a slightly less complicated model [30]. The proposed idea was to have one of these network variants be responsible for decision making. The network would receive its input from a CNN which was used to compress the image layers from PySC2. From the information that was gathered, we were under the impression of that the output would be one of two cases. The first case would be a representation of how the next desirable state would look like. In this case there would have to be some separate implementation of a tool or network that could interpret this representation and execute the action that leads to this next state. The second case would be that the network's output would be the the action itself, without any extra implementation required.

In either case, the general consensus was that information related to previous states of the game is a necessity to process in order for an agent to learn to adapt to many different scenarios. This also lead us to believe that it would be worthwhile

to test if a LSTM network could be used for win prediction.

5.4.2 An SC2 agent which plays the real game

The end goal that one would strive to accomplish using our groundwork would be to create an agent that would be able to play the game as good, or better, than an average player.

One step towards that end goal is to teach an agent to play the game using supervised learning. This might be especially useful to create a base agent that can then learn further using reinforcement learning. By "pre-learning" from human play the agent can hopefully be put on the right path, rather than blindly exploring SC2's large action space.

Another step would be to use the victory prediction network as the reward function for a reinforcement agent. This seems like one of few viable approaches for reinforcement learning due to the difficulty of building good reward function as discussed in section 5.3 and concluded in the PySC2 release paper [10]. Before this approach can be taken however, more work is required on the victory prediction network, also discussed in 5.3, since it does not yet provide any useful data.

The final training stage for the agent that we strove to build was supposed to be reinforcement learning in a specially designed tournament environment, similar to how DeepMind trained the AlphaStar agent. The idea is to have many different iterations of the same base agent compete and learn from each other [9]. The winning agents would move forward in the tournament while learning from previous matches, and losing agents would sometimes be reintroduced to prevent the more successful agents from forgetting how to deal with less optimal strategies.

DeepMind proved without a doubt that a tournament system is a successful approach for simultaneous training of several agents with reinforcement learning. However, our initial purpose was to do research on how well you could train an agent with a limited resource of computing power. Given that DeepMind used 16 Tensor Proccessing Units (TPU) [9] per agent, where each TPU corresponds to roughly 15 GPUs, such resource restraints were not taken into consideration by them.

It is debatable whether or not this approach would prove to be useful in our particular case. Considering that training a single convolutional agent on a GTX1080 GPU utilized 30% of its total capacity, it would only support so many different processes. This seems to indicate other bottlenecks: CPU, memory speed, or something else. Similarly, training a non-convolutional agent on a 3.5 GHz 4-core processor consumed about 35% processing power, suggesting a bottleneck for this agent that is not the CPU. As such, a tournament system would only really be useful if more hardware was available. Ideally complemented by optimizations to the training process.

As a response to this limitation, a side track that would have been both interesting and useful to explore is the possibilities of optimizing the training process. For example, simple things such as preloading the upcoming batch of training data in parallel with the actual training itself could speed up the process. If the process of a single agents training can be sped up by a significant rate, the prospect of having an agent perform at a decent level does not seem too far fetched.

6
Conclusion

Our work has made it easier to get started for someone interested in starting out with machine learning in StarCraft II, but the results of the analyzed networks were worse than expected.

The non-convolutional reinforcement agent clearly outperformed random and was on par with the scripted agent, although with some odd behaviors that meant it did not receive top score as consistently as would have been preferred. We believe that these behaviors are related to unforeseen aspects of the reward function or side-effects of the design of the agent's action space. Others are encouraged to explore other designs.

In contrast to the non-convolutional agent, convolutional agents do not seem to be an ideal choice for anyone new to the field or using poor computational power and storage space. They need to be run on one or several GPUs if training time is not to become unreasonable, which in turn requires more optimization and consideration of potential bottlenecks. Due to the size of image data, limited storage space may also restrict how much data can be used for training, likely decreasing how much the network can learn.

The victory prediction network, also using the convolutional network model, proved to be ineffective as well. While its training loss was improved, its performance on data that was not from the training set did not. This could be due to biased or too little training data, network structure or one of many other reasons. This agent would need to be tested and possibly developed further to be of any use.

Fortunately, the replay tools did work quite well, as did the simple API and both are ready to be used more in the future. The only known remaining issue is that the toolkit is unable to parse replays that are not of the latest SC2 version. This means that replays have to be collected and parsed within a reasonable amount of time to minimize the risk of an SC2 update halting progress. It seems reasonable that a solution to this problem can be found in the future.

Bibliography

[1] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[2] D. Balduzzi, M. Garnelo, Y. Bachrach, W. M. Czarnecki, J. Pérolat, M. Jaderberg, and T. Graepel, "Open-ended learning in symmetric zero-sum games", *CoRR*, vol. abs/1901.08106, 2019. arXiv: 1901.08106. [Online]. Available: http://arxiv.org/abs/1901.08106.

[3] J. H. Holland *et al.*, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[4] P. Wang and B. Goertzel, "Introduction: Aspects of artificial general intelligence", in *Proceedings of the 2007 conference on Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms: Proceedings of the AGI Workshop 2006*, IOS Press, 2007, pp. 1–16.

[5] S. Franklin and A. Graesser, "Is it an agent, or just a program?: A taxonomy for autonomous agents", in *International Workshop on Agent Theories, Architectures, and Languages*, Springer, 1996, pp. 21–35.

[6] DeepAI. (). Neural network, [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/neural-network.

[7] J. T. Georgios N. Yannakakis, *Artificial Intelligence and Games*. Springer, 2018.

[8] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm", *arXiv preprint arXiv:1712.01815*, 2017.

[9] Team AlphaStar. (2019). Alphastar: Mastering the real-time strategy game StarCraft II, [Online]. Available: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/ (visited on 2019-05-16).

[10] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "StarCraft II: A new challenge for reinforcement learning", 2017. eprint: arXiv:1708.04782.

[11] F. Amato, A. López, E. M. Peña-Méndez, P. Vaňhara, A. Hampl, and J. Havel, *Artificial neural networks in medical diagnosis*, 2013.

[12] S. Hoffman, "Managing the state: Social credit, surveillance and ccp's plan for china", vol. 17, 11 Aug. 17, 2017. [Online]. Available: https://jamestown.org/program/managing-the-state-social-credit-surveillance-and-the-ccps-plan-for-china/ (visited on 2019-02-13).

[13] Blizzard. (2017). S2client-proto, [Online]. Available: https://github.com/Blizzard/s2client-proto (visited on 2019-04-15).

[14] Python Software Foundation. (2019). Python language reference, version 3.7, [Online]. Available: http://www.python.org (visited on 2019-04-01).

[15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *Tensorflow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[16] G. F. Luger, *Artificial intelligence: structures and strategies for complex problem solving*. Pearson education, 2005.

[17] J. Laird and M. VanLent, "Human-level ai's killer application: Interactive computer games", *AI magazine*, vol. 22, no. 2, p. 15, 2001.

[18] D. Fagella. (2019). What is machine learning?, [Online]. Available: https://emerj.com/ai-glossary-terms/what-is-machine-learning/ (visited on 2019-05-16).

[19] T. M. Mitchell, *Machine learning*. McGraw Hill, 1997.

[20] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue", *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[21] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features", IEEE, 2001, p. 511.

[22] L. Deng, D. Yu, *et al.*, "Deep learning: Methods and applications", *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014.

[23] A. C. Lagandula. (2018). Mcculloch-pitts neuron — mankind's first mathematical model of a biological neuron, [Online]. Available: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1 (visited on 2019-05-16).

[24] J. A. Hertz, *Introduction to the theory of neural computation*. CRC Press, 2018.

[25] B. Mehlig, "Artificial neural networks", *arXiv preprint arXiv:1901.05639*, pp. 100–122, Feb. 2019.

[26] J. Brownlee. (Jul. 2017). Gentle introduction to the adam optimization algorithm for deep learning, [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ (visited on 2019-05-16).

[27] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", *arXiv e-prints*, arXiv:1412.6980, Dec. 2014.
[28] P. Dahal. (2017). Solving the model - sgd, momentum and adaptive learning rate, [Online]. Available: https://deepnotes.io/sgd-momentum-adaptive (visited on 2019-05-16).
[29] M. B̃. Pascalu, "On the difficulty of training recurrent neural networks", pp. 1–9, 2013.
[30] C. Olah. (Aug. 2015). Understanding lstm networks, [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on 2019-03-29).
[31] Greff, Srivastava, Koutník, Steunebrink, and Schmidhuber, "LSTM: A search space odyssey", 2017.
[32] H. Sak, A. Senior, and F. Beaufays, *Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition*, 2014. [Online]. Available: https://arxiv.org/abs/1402.1128.
[33] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[34] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", 1998.
[35] C. Asawa. (2015). Cs231n convolutional neural networks for visual recognition, [Online]. Available: http://cs231n.github.io/convolutional-networks/ (visited on 2019-05-14).
[36] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. DOI: 10.1038/nature14539.
[37] D. Soni. (2018). Supervised vs unsupervised learning, [Online]. Available: https://towardsdatascience.com/supervised-vs-unsupervised-learning-14f68e32ea8d (visited on 2019-05-16).
[38] X. Zhu and A. B. Goldberg, "Introduction to semi-supervised learning", *Synthesis lectures on artificial intelligence and machine learning*, vol. 3, no. 1, pp. 1–130, 2009.
[39] R. S. Sutton and A. G. Barto, *Reinforcement learning*, 2nd ed. The MIT Press, 2017.
[40] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning", *arXiv preprint arXiv:1312.5602*, 2013.
[41] C. J. Watkins and P. Dayan, "Q-learning", *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
[42] 2019. [Online]. Available: https://keras.io/.
[43] M. G. Duque. (2019). Sc2reaper, [Online]. Available: https://github.com/miguelgondu/sc2reaper (visited on 2019-04-01).
[44] 2019. [Online]. Available: https://www.mongodb.com/what-is-mongodb (visited on 2019-05-16).
[45] E. Plugge, P. Membrey, and T. Hawkins, *The definitive guide to MongoDB*. Apress, 2010.
[46] [Online]. Available: http://bsonspec.org/ (visited on 2019-05-16).

[47] K. Leung and D. Paskert. (2013). Spawning tool, [Online]. Available: https://lotv.spawningtool.com/ (visited on 2019-04-24).
[48] GGGReplays. (2016). Gggreplays, [Online]. Available: https://gggreplays.com/ (visited on 2019-04-24).