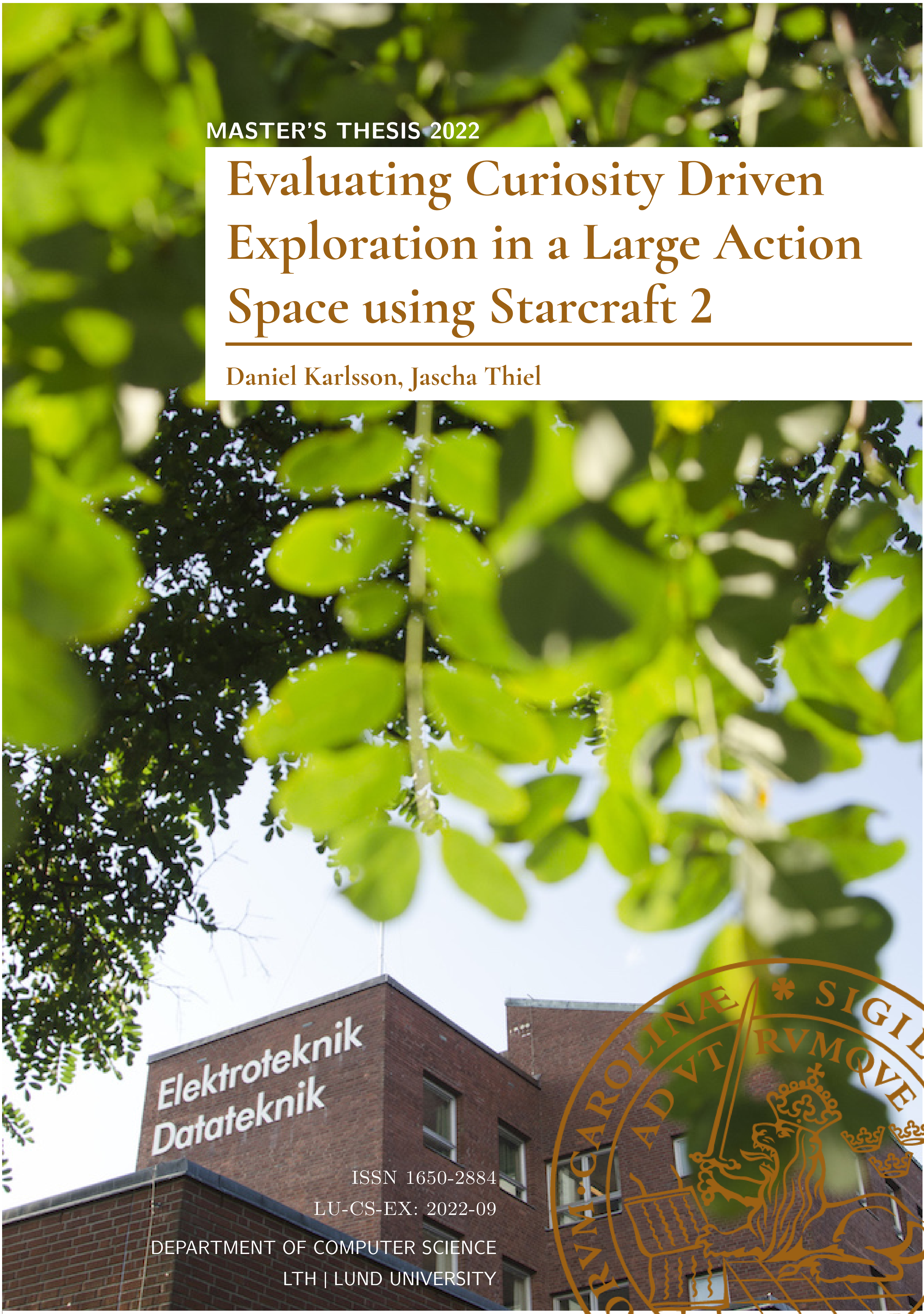
MASTER'S THESIS 2022

Evaluating Curiosity Driven Exploration in a Large Action Space using Starcraft 2

Daniel Karlsson, Jascha Thiel



ISSN 1650-2884
LU-CS-EX: 2022-09

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-09

Evaluating Curiosity Driven Exploration in a Large Action Space using Starcraft 2

Daniel Karlsson, Jascha Thiel

Evaluating Curiosity Driven Exploration in a Large Action Space using Starcraft 2

(Evaluating Curiosity Driven Learning in environment with a large, complex action space)

Daniel Karlsson
`dat14dka@student.lu.se`

Jascha Thiel
`dat14jth@student.lu.se`

March 15, 2022

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Hampus Åström, `hampus.astrom@cs.lth.se`
Volker Krueger, `volker.krueger@cs.lth.se`

Examiner: Elin Anna Topp, `Elin_Anna.Topp@cs.lth.se`

Abstract

Starcraft 2 is a PC game that has been addressed more and more within the research field of artificial intelligence (AI) and reinforcement learning. One research field of interest is the concept of Intrinsic Curiosity Exploration, a reward generated by an Intrinsic Curiosity Model that incentivizes exploration called Intrinsic Curiosity. This model defines the agent error in predicting the outcome of its action based on some feature space learned by an inverse dynamics model. Starcraft 2 presents an interesting problem since the action space is ample, and many actions have a similar outcome. For this thesis, we implement an Intrinsic Curiosity Module with a custom loss function to work with a well-established reinforcement learning agent, the Deep Q-network. Our research aims to allow an agent to learn and improve while playing Starcraft 2.
Agent playing video:
https://www.youtube.com/watch?v=07FUgVS9jQkfeature=youtu.be

Github:
www.github.com/dat14dka/ThesisProject

Keywords: Reinforcement Learning, Curiosity Driven Exploration, Intrinsic Curiosity Module, Starcraft 2

Acknowledgements

We want to thank Volker Krueger and Hampus Åström at LTH for working tirelessly with guiding and correcting our work. We would also like to thank Elin Anna Topp at LTH for feedback on our rapport.

I, Daniel Karlsson, would also like to thank Magnus and Caroline Karlsson for providing support, housing, and electricity for the later part of the project. I would also like to thank my sister Felicia Karlsson, who inspired me to keep working even when the task felt too overwhelming by showing great courage in the face of adversity. Finally, I would like to thank Alexandros Sopasakis at LTH for awakening my interest in learning and mathematics.

I, Jascha Thiel, would like to thank Gunnar and Hjördis Thiel for supporting me throughout this project. I would also like to thank all my professors, lecturers, and teachers at LTH, who deepened my interest in technology, programming, and machine learning through their way of teaching.

Contents

Chapter 1
Introduction

In recent years reinforcement learning has made great progress, from beating the highest-ranked GO player in the world [1], to playing at a consistently professional level in Starcraft 2 [2]. Using reinforcement learning algorithms, an agent can learn and improve upon its performance autonomously, given a chance to act in specific environments [3]. One such algorithm is the Deep Q-network (DQN). This network iteratively improves its ability by evaluating actions, then incorporating the best-perceived action to reach a higher ability [4], and shows great results using this approach[4]. However, one problem with this system is that in environments with sparse rewards (low reward environment), the DQN learns slowly [12], without any explicit reward, the DQN chooses its actions more or less randomly, making this an inefficient way of learning.

An Intrinsic Curiosity Module (ICM) [5] is a component that adds additional rewards based on Intrinsic Curiosity and improves the learning speed for the DQN in sparse environments. Intrinsic Curiosity defines the agent error in predicting the outcome of the action based on some feature space learned by an inverse dynamics model. The purpose of this additional reward is to guide the activities of the reinforcement learning agent when no extrinsic reward is provided [5]. This approach may therefore improve performance for the reinforcement learning agent in sparse reward environments [5].

The algorithms mentioned above solve complex problems, and one such problem is playing the strategy PC game Starcraft 2 (Figure 1.1). The game can be accessed and played through the API PYSC2, an API that can use a reward structure that gives no extrinsic reward for long periods of time[11]. In the game, the player controls friendly soldiers, while the opponent controls the opposing soldiers. All players can distribute different commands to the soldiers, for instance, "attack".

There also exist subtasks within Starcraft 2, which are called minigames [11] and they present a lesser challenge than the entire Starcraft 2 game. The player receives a total score in each minigame based on how well the player performs. Using this score, one can examine if adding Intrinsic Curiosity to the DQN can improve its ability to learn in these minigames. This thesis aimed to determine how the process of learning to play the Starcraft 2 minigame "DefeatRoaches" differs for the DQN, with and without Intrinsic Curiosity.

1.1 Questions Evaluated by the Thesis

- How will the Deep-Q Network (DQN) perform with and without Intrinsic Curiosity in large action space, such as Starcraft 2?
- How will the two algorithms, DQN with and without Intrinsic Curiosity perform with sparse rewards?
- Will the DQN with Intrinsic Curiosity outperform the DQN?

1.2 Exploring the Research Questions

There already exist a developed DQN agent who can play and improve performance in Starcraft 2 [19], which is used with modifications (hyperparameters were rebalanced for the sparse environment) to save time and resources. However, to explore our research question, the ICM model must be implemented and integrated with the DQN. Furthermore, to analyze how the different models perform during different reward environments, we also plan to implement other reward structures in Starcraft 2.

Figure 1.1: A game of Starcraft 2

1.3 Contribution Statement

While exploring these research questions, the thesis illustrates how the ICM and DQN could be used to achieve very promising results within the field of Starcraft 2. The game presented a very difficult and unique, previously unexplored kind of challenge for the ICM, since the action space is very large and many actions have the same outcome [16]. Previously the ICM has been used to play simpler arcade games, such as Mario [5]. Finally, an alternative activation function for the Inverse Network is proposed (Mean Square Error activation function) and showed better results than the traditionally used Softmax activation function in this specific context.

1.4 Motivational Statement

The challenge of a complex game with a large action space is interesting because it can be related to a number of real world problems, even though the game of Starcraft 2 itself is not very realistic. One example would be automated driving. Similar to a player in Starcraft 2, a driver has a large number of possible actions available at all times, with only a relatively small number of "right" choices. Another similarity is that many actions, e.g reducing or increasing the speed slightly, often have very little effect, while other actions might have a drastic effect, suddenly turning or decreasing the speed abruptly for example. A second potential real world application could be the automated exploration and information gathering from huge maps, where an AI could be used to find certain types of information, while discarding non-relevant information. The similarity with Starcraft 2 here would be that the map initially contains lots of unknown information, with only a relatively small part being relevant while playing the game.

1.5 Methodology and Outline

The work developing the agent was a mix of experimental and analytic work. Once the initial programming phase was completed and the software worked as intended, the main focus was to choose the optimal value for the different hyperparameters. While background theory and related studies provided some guidance on how to choose these values, initially this was done experimentally. In other words, one or more parameters where changed and the results observed. Over time, this changed more in to an analytic approach, where parameters where changed with the intent to change or amplify a certain behaviour of the agent, rather than just observing the result of these changes. During the evaluation phase, only very small changes where made to certain parameters, as some effects only became clear when running longer training sessions. Once the configurations of the Agent was finalized, a purely analytic approach was used to compare the performance of the Agent in the different environments.

In the next Chapter, relevant information about Starcraft 2 and the PYSC2 API will be presented, followed by theory regarding Deep Q-Networks (DQN) and the Intrinsic Curiosity Module (ICM). In the third Chapter, a number of related and relevant studies are presented. This is followed by a discussion about the Systems used and developed in Chapter 4. In Chapter 5, the Results of our work is presented. In Chapter 6, the results and systems are discussed and evaluated. This chapter also contains sections discussing possible improvements, as well as suggestions for future work. Lastly, Chapter 7 contains the main conclusions from this thesis.

Chapter 2
Background Information

This report assumes that the reader is familiar with the concepts of Markov Chains, the fundamentals of Reinforcement Learning, and Neural Networks.

2.1 Starcraft 2

Starcraft 2 is a real-time strategy video game where players issue commands to units with different skill sets. The player can select one of three races, which then determines what type of units the player can control. The players use the units to build structures, farm resources, and attack enemy units to achieve the game's goal of defeating the enemy by destroying their base. [11].

2.1.1 Rules of Starcraft 2

The player needs to decide which actions their units should take to destroy the enemy base. Each unit can use a variety of actions and different skills, with the ability "attack" being a common skill for most units [11]. However, the damage an attack leads to is different for different units, which unit determines the degree of damage caused. Also, note there is no randomness or miss-chance involved in the different attacks.

The player determines their units' actions in order to defeat their opponent. To achieve this goal, the player needs to complete sub-goals, such as collecting resources, building units, and killing enemy units [11]. Within the game of Starcraft 2, there are minigames included that include specific sub goals [16], which allows the player to practice and become more proficient at achieving the particular subgoals. For this report, we only focus on the minigame "DefeatRoaches".

DefeatRoaches

In the "DefeatRoaches" minigame, the human player opposes a computer-controlled opponent. The initial state in this minigame includes 9 marines (controlled by the human player) placed in a vertical line, randomly placed either on the left or the right side of the map, and 4 roaches (controlled by the computer) placed in a vertical line on the opposite side of the map (Figure 2.1). In the game, the marines try to kill all roaches, if they succeed, the game spawns 5 new marines and 4 new roaches with full health, and the surviving marines from the just played episode remain alive without having their health restored. When a new episode starts, all units move back to opposing positions of the map, and a new episode starts [16]. The computer-controlled roaches will act deterministically. That is, they will stay in place, and if marines attack them, the roaches will fight back until they are dead.

The human player has a complete overview of the total area covered in the game in the minigame and a complete overview of the actions of each roach and each marine, which means no information is hidden. The player and the computer can not move units to the edge of the gamespace (Figure 2.1). The added score for killing a roach is 10 points, while every defeated marine results in a lost point (-1 point). The game ends when the roaches defeat all marines or 120 seconds have elapsed [16]. The final score of an episode is the sum of all earned points, where the goal of each game is to receive as many points as possible.

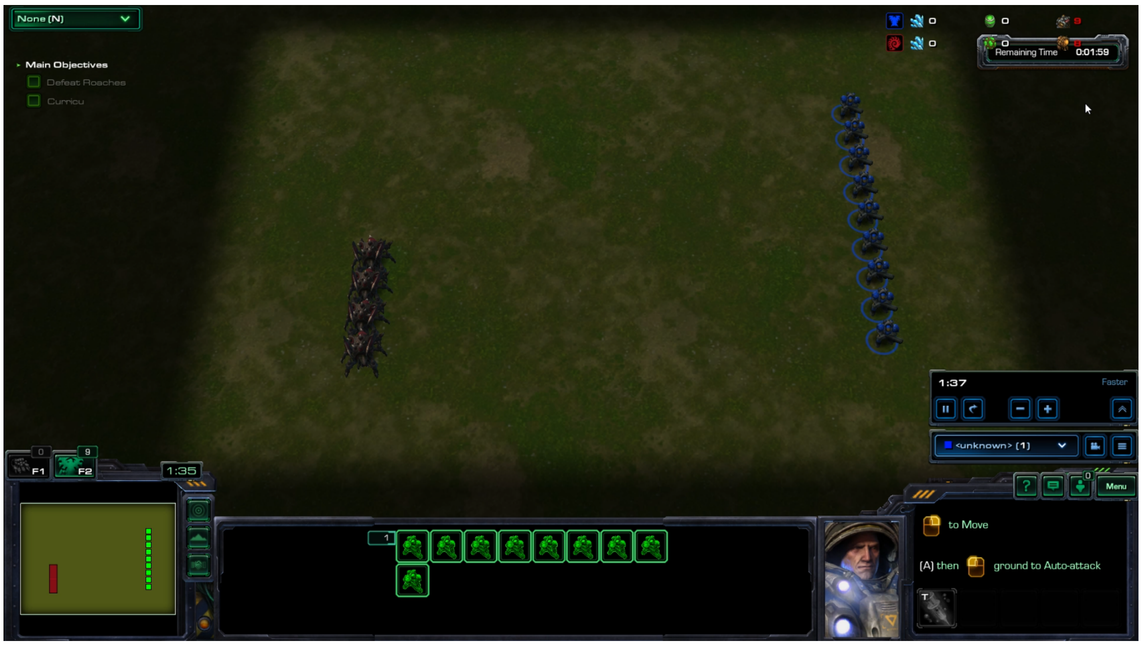


Figure 2.1: Screenshot of the minigame *DefeatRoaches*. The blue units on the right are the user controlled *Marines*, the *Roaches* (enemy units) are on the left.

2.1.2 Game Tactics

To achieve as high a score as possible, human players use specific tactics, two of these, "Focus fire" and "Kiting", will be briefly described below to give some understanding of how to play the game optimally. Unfortunately, no complete list of these tactics exists. However, one can observe professional players play to get an idea of the optimal way of playing [1].

"Focus fire" can best be described as focusing fire on a single enemy unit, which is the fastest way to eliminate an enemy unit since the more friendly units that attack an enemy unit, the more damage is done to the enemy unit. The faster the number of enemy units is reduced, the less damage they can do to the Marines, which results in the player receiving a higher scoring. The "Focus fire" strategy is easy to understand and perform. However, when playing the minigame optimally, the player should combine multiple strategies.

One of the tactics commonly combined with "Focus fire" is "Kiting". It can be described as constantly moving the units controlled by the player, with the goal being to avoid taking damage from the enemy units and still being able to deal damage to them. While the concept of "Kiting" is not very complicated to understand, it is more difficult to do effectively, especially in combination with "Focus fire".

2.2 PYSC2

PYSC2 is an open-source environment written in Python that provides an API for reinforcement learning agents to interact with Starcraft 2 while optimizing the interaction between the Starcraft 2 environment and the reinforcement learning agent [16]. The PYSC2 environment links Starcraft 2 and the machine learning agent written in Python and exposes the Starcraft 2 Machine Learning API to the python code. PYSC2 makes it possible to start and select minigames as well as train a reinforcement learning agent [16].

2.2.1 Game Sequence

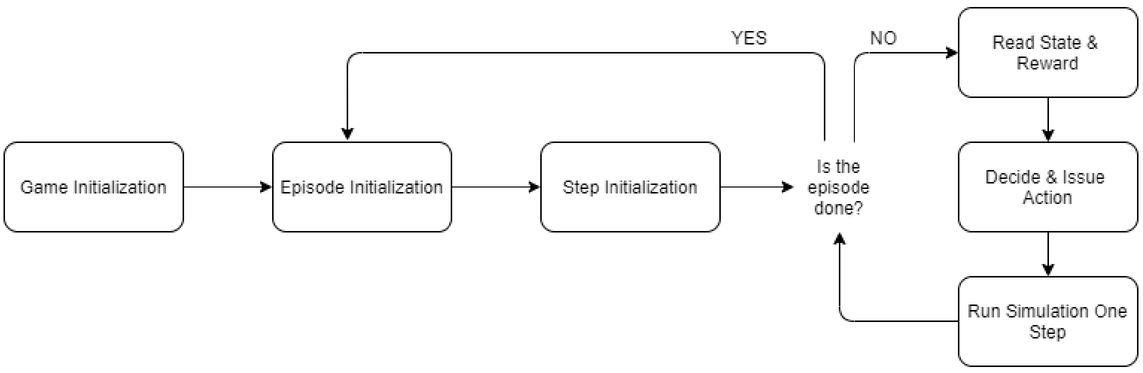


Figure 2.2: Model of the PYSC game sequence

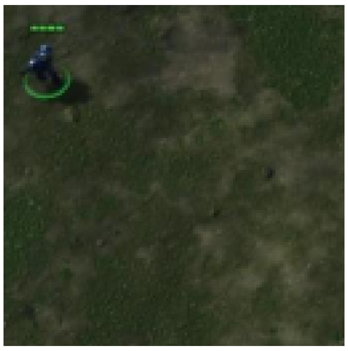
[1]To see a display of advance tactics, see the following link, where the high ranking Starcraft 2 player Kevin Johansson plays "DefeatRoaches":

Semi-Professional playing DefeatRoaches: youtube.com/watch?v=fNJks7uMInY

For an overview of the sequence that occurs when using PYSC2 to play a Starcraft 2 minigame, see Figure 2.2. Firstly the Game Initialization phase occurs. Any custom code by the developer runs, the game is booted, and all necessary modules for running the game initializes. Then the game loop starts and runs until the developer terminates it. The game loop begins with the Episode Initialization phase [28]. Any custom code by the developer runs, and all the variables related to each episodic run reset (such as game score). The API then receptively executes Step Initialization until the episode is terminated (for DeafeatRoaches, this is a loss or the time running out). A step is equivalent to letting one time-step occur in the game simulation. When a step occurs, the API reads a state and a reward from the game. After which, the agent decides some action and issues it to its units. Lastly, the agent's and other players' actions occur, and the game simulates which affect the actions had on the environment. This cycle is then continuously repeated, as stated beforehand, to improve agent performance.

2.2.2 States and Rewards

PYSC2 provides a state input consisting of several feature representations representing different aspects of the game (see Section 8.2 for complete list). For instance, one such feature representation is "player_relative", which determines whether units are friendly or hostile. A matrix with the exact dimensions as the game window expresses each feature representation [28], for instance, in a 4x4 pixel game window, each feature representation will consist of a 4x4 matrix. In the player_relative matrix, a friendly unit at pixel-coordinate (0,0) will cause the feature matrix to be equal to 1 at index (0,0), while all other values in the matrix will equal 0. Figure 2.3 shows an example of this player_relative matrix and the corresponding game window. In conjunction with receiving a state, the agent also gets a reward.



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.3: A approximately 21x21 game window and a simplification of the corresponding ***player_relative*** matrix, in reality the matrix would be in a higher resolution.

A reward is an integer given at each game step, where the reward provides feedback to the performance of the agent [28], for instance, the default reward structure for the "DefeatRoaches" gives 10 points (+10) when killing an enemy and a loss of 1 point (-1) when losing a friendly unit [16]. The developer can use the predetermined reward structure or create a custom-designed reward structure.

2.2.3 Actions

After the state and the reward are registered, an action is issued [28]. The developer needs to specify the type of action and the arguments required for that type of action[2] to issue the action, one such category of actions is screen actions [28]. Screen actions are commands issued in conjunction with coordinates, for instance, the command “move” ("move to coordinate x,y") [11]. The developer must specify x- and y-coordinates and the action type "move" to issue the move command. When the command is issued, the units start to move toward the defined (x,y) coordinates. To determine which action to take, the developer can use a DQN.

2.3 Deep-Q Networks (DQN)

Deep-Q Network is a value estimation reinforcement learning algorithm that enables an agent, acting in some space, to improve at a task with experience [4] and builds upon the concept of Q-learning and expands on it using Neural Networks.

2.3.1 Q-learning

Q-learning supposes a world model based on Markov Decision Processes. However, it adds the concept of an action-value function, modeled by Q, which answers the question: "How profitable is the action a when I am in the different states?". Q-value for some state s_t and action a_t is calculated iteratively like so:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}} \tag{2.1}$$

In this algorithm, γ and α are constants referred to as "the discount value" and "the learning rate" [4]. The agent uses a learning rate to avoid overfitting, which reduces how quickly the Q-values are updated. At the same time, a discount factor helps to account for the uncertainty of relying on future rewards. Commonly, a Q-table keeps track of each Q-value, where the Q-table updates as the amount of experience grow (or equivalently as t grows over time). Experience is the set of actions, states, and rewards for each time step the agent observes. Using a football game as an example, the agent can shoot, run, or stand still, and the possible states in which the agent can exist are "possessing the ball" and "not possessing the ball". In our example, the agent starts with "possessing the ball" and each Q- value initializes at 0. Furthermore, γ is set at 0.95 and α to 0.1. Figure 2.4. presents the Q-table from the above example.

[2]For a complete list of actions and action types offered by PYSC2 see https://github.com/deepmind/pysc2

	shoot	run	stand still
poss	0	0	0
not_poss	0	0	0

Figure 2.4: An example of a Q-table for an agent playing soccer at initialization

The match carries on, and the agent runs and then shoots the ball, scoring and receiving a reward of 10. In turn, this results in the following rewards, calculations and corresponding Q-table.

1. $Q^{new}(poss, shoot) \leftarrow 1 = 0 + 0.1 \cdot (10 + 0.95 \cdot 0 - 0)$
2. $Q^{new}(poss, run) \leftarrow 0.095 = 0 + 0.1 \cdot (0 + 0.95 \cdot 1 - 0)$

	shoot	run	stand still
poss	1	0.095	0
not_poss	0	0	0

Figure 2.5: An example of a Q-table for an agent playing soccer after update

2.3.2 Q-learning using Neural Networks

In many real-life problems, the state and the action space are too large for modeling all q-values in a table [4]. As an alternative, it is possible to create a model of the Q-function by using a Convolutional Neural Network.

Input to the network consists of the current state (s_t), the network itself can consist of any layer best fit for the task, and the network outputs a predicted Q-value for the input state (s_t) and each possible action (a_t). Below are the calculations for calculating the target used for predicting the Q-values:

$$Q^{target}(s_t, a_t) \leftarrow \underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \max_{a_{t+1}} [\underbrace{\hat{Q}(s_{t+1}, a_{t+1})}_{\text{predicted Q-values}}] \quad (2.2)$$

Then as the loss function $L(Q^{target}, \hat{Q})$ converges, the network correctly estimates the actual Q-values, and the agent chooses the action with the largest Q-values, which converges to a policy where the reward of the agent is maximized [4]. However, with this approach, there are several problems.

The first problem includes convergence and is caused by the target value depending on the output of the network, as seen in Equation 2.2. Since the network continuously weights and changes the output, so do the target values [4]. One solution is to use two networks: one network predicting current Q values (here called the Q-network) and one network predicting the next state Q-values (target-network). The target network will then have its weights frozen and only periodically copy the weights from the Q-network [4].

The second problem includes overfitting to the most recent set of data points by the DQN. Since the DQN continuously receives and trains on inputs, the network runs the risk of overfitting to the last encountered experiences if there exists significant covariance between the experiences which are closely related in time [4]. However, adding "Experience replay memory" reduces this problem by having a memory that saves experiences, and then the agent chooses a new random set of experiences for each training iteration. Algorithm 1 presents the final algorithm.

Algorithm 1: DQN Obtained from Mnih et al. (Human-level control through deep reinforcement learning)

```
Input: States and reward
Output: Q action value function (from which the policy is obtained and the action
        is selected).
Initialize experience replay memory $D$; Initialize action-value function $Q$ with
  random weight $\theta$;
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$;
for episode = 1 to M do
    Preprocessed state $\phi_1 = \phi(s_1)$;
    for t = 1 to T do
        Select $a_t = max_{a_t} Q(\phi(s_t), a; \theta)$;
        Execute action $a_t$ and observe reward $r_t$ and state representation $s_{t+1}$;
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$;
        Sample random minibatch of transition $(s_j, a_j, r_j, s_{j+1})$ from $D$;
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j + 1} \\ r_j + \gamma \cdot max'_a \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
        Perform a gradiant descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
          network parameters $\theta$;
        Every C steps reset $\hat{Q} = Q$;
    end
end
```

The presented algorithm does not account for another common reinforcement learning problem: exploration versus exploitation.

2.4 Exploration versus Exploitation

The agent must choose which action to select when interacting with the environment, and it uses a specific policy to do so [3]. The most apparent strategy is to choose the action that provides the highest expected reward, or as in the case of the DQN, always select the action that provides the largest Q-value. However, if the environment changes or the agent has incomplete information, the agent may act based on false information. For instance, an agent with the task to buy apples for the lowest price in an environment with multiple stores that have apples with different prices will receive a point (+1) reward if finding an apple costing 2$

and +2 if finding an apple costing 1$. The agent starts its search and discovers a fruit store that the agent has not seen before, with apples priced at 2 $. The agent would then not search for other stores since it would only act on its previous knowledge of available stores and prices, which would not include knowledge of the store selling apples for 1$. One problem is thus how to balance each choice and decide when an agent should choose exploitation (greedy) or exploration. If an agent wants to estimate the state, reward, and action space more accurately, it should select an exploratory action to learn more, not a greedy one. However, if the agent keeps choosing exploratory actions all the time, the policy would never converge, and if the agent acts too greedily, it might never discover the optimal policy [13]. One way of solving this for the DQN is to add ϵ-greedy Select (for further description of ϵ-greedy Select, see below).

2.5 DQN with ϵ-greedy Select

As mentioned in section 2.3, there exists in reinforcement learning an exploitation versus exploration dilemma [13]. One way to incorporate exploration is using the algorithm ϵ-greedy Select, which means to select with the probability of ϵ an action at random and with the probability 1-ϵ, the action with the highest expected reward or for the DQN the highest Q-value [4]. ϵ-greedy pairs with multiple different reinforcement learning algorithms.

A typical implementation of ϵ-greedy uses "exponential decay". "Exponential decay" refers to ϵ starting at some ϵ_{max} and then decays exponentially over the number of episodes [33]. The idea behind this approach is that, given infinite time and the agent exploring the entire action space, it will find an optimal solution. Algorithm 2 presents the DQN with ϵ-greedy Select.

Algorithm 2: DQN Obtained from Mnih et al. (Human-level control through deep reinforcement learning)

```
Input: States and reward
Output: Q action value function (from which the policy is obtained and the action
        is selected).
Initialize experience replay memory $D$;
Initialize action-value function $Q$ with random weight $\theta$;
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$;
for episode = 1 to M do
    Preprocessed state $\phi_1 = \phi(s_1)$;
    for t = 1 to T do
        With the probability of $(1 - \epsilon)$ select $a_t = max_{a_t} Q(\phi(s_t), a; \theta)$ else select a
          random action ;
        Execute action $a_t$ and observe reward $r_t$ and state representation $s_{t+1}$;
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$;
        Sample random minibatch of transition $(s_j, a_j, r_j, s_{j+1})$ from $D$;
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j + 1} \\ r_j + \gamma \cdot max'_a \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
        Perform a gradiant descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
          network parameters $\theta$;
        Every C steps reset $\hat{Q} = Q$;
    end
end
```

ϵ-greedy Select alleviates the problem of exploration versus exploitation. However, when looking at something called sparse problems, this strategy of exploring can be insufficient.

2.6 Sparse Rewards

Some environments rarely provide rewards, and when an environment provides rewards that are mostly 0, the rewards are said to be sparse [5]. Such an example is a soccer game where the game rewards the agent +1 when winning, -1 when losing, and in all other cases 0 (example for ending the game in a draw, receiving yellow or red cars, or scoring goals) [5]. Most rewards would, in this case, be 0. In Figure 2.6 there is one example of a sparse and one example of a none-sparse reward structure when playing a soccer game. In the DQN, this reward system becomes problematic since the rewards vanish when repetitively multiplying them with the discount factor, then using the DQN results in most states having similar Q-values. Furthermore, this essentially makes the acting policy closer to random selection, further reducing the efficiency of the DQN.

It is, however, possible to alleviate the problem by adding an intrinsic reward, where the agent generates the intrinsic reward by itself [5]. The agent uses the intrinsic reward to guide behavior when an extrinsic reward is absent. One algorithm that implements such an idea is the Intrinsic Curiosity Module.

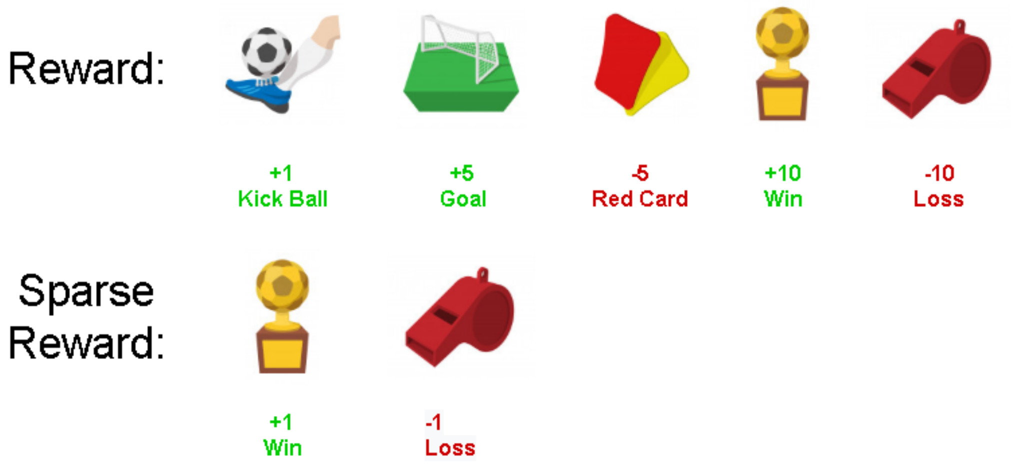


Figure 2.6: Example of Sparse and Not Sparse Rewards for an agent playing soccer

2.7 DQN with ϵ-greedy Select and Intrinsic Curiosity Module (ICM)

The Intrinsic Curiosity Module (ICM) aims to provide Curiosity as an intrinsic reward ("Intrinsic Curiosity") for the reinforcement learning agent to perform exploratory actions [5]. The module adds an extra incentive to explore on top of the native algorithm exploration incentive (ex: ϵ-greedy), called Intrinsic Curiosity, which is defined as the error when the agent predicts the outcome for its action in a space defined by its inverse model [5]. The system lets a Forward Model predict the next state, and the prediction error becomes the intrinsic reward, then adds the intrinsic reward to the extrinsic reward. The Forward Model bases its prediction on the state space defined by the Inverse Model [5].

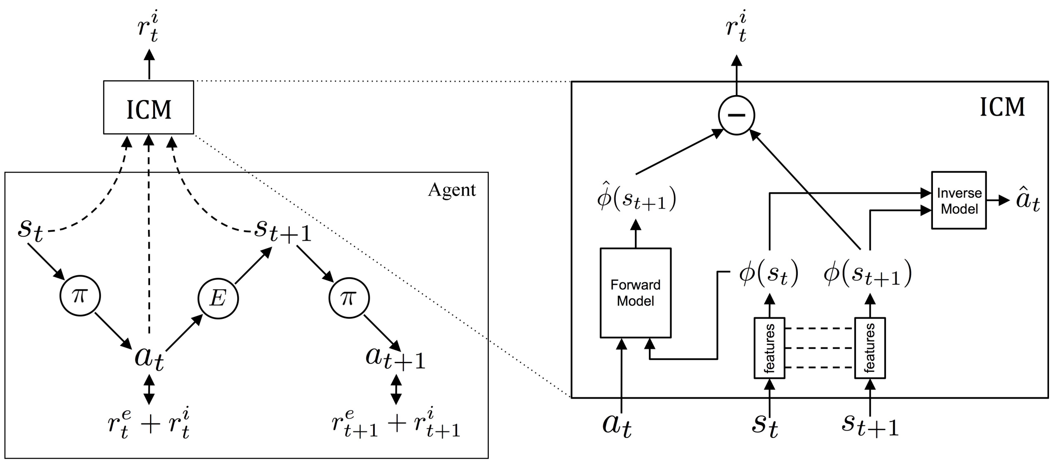


Figure 2.7: Overview of an agent and the ICM

Multiple neural networks called the Inverse Network models what is called The Inverse model [5]. These network include as input the current state (s_t) and the next state (s_{t+1}), fed into a neural network which encodes both s_t and s_{t+1} into two separate embedded vectors ($\phi(s_t)$ and $\phi(s_{t+1})$) [5]. These vectors are concatenated and then fed into an additional neural network which outputs the probabilities that each action a_t was taken in order for state s_t to transition into state s_{t+1}.

The purpose of the Inverse Network is to produce state representations ($\phi(s_t)$ and $\phi(s_{t+1})$) to feed into the Forward Network [5], and these embedded networks then feed into the Forward Network that predicts actions based on the provided vectors from the Inverse Network. The embedding network will embed state information to only contain relevant data for predicting actions based on states $\phi(s_t)$ and $\phi(s_{t+1})$)). This filtering happens through embedding state information regarding things that the agent can affect and things the agent cannot affect, but that affect the agent state [5]. One does not wish to embed information about things that do not affect the agent and that the agent cannot affect, and The Inverse Network uses a soft-max activation function and the previously mentioned network structure to achieve this goal.

A neural network models the Forward model and is called the Forward Network that takes as input the action at time t (a_t) and the embedding vector of the current state ($\phi(s_t)$), provided by the Inverse Model [5] and its output tries to predict the embedding vector of the next state, $\hat{\phi}(s_{t+1})$. The Forward Network traditionally uses a Mean square error activation function and the previously mentioned network structure to achieve this goal.

Lastly the agent calculates the intrinsic reward for each time-step t :

$r_t^i = |\hat{\phi}(s_{t+1}) - \phi(s_{t+1})|$.

The agent calculates total reward as:

$r_t = r_t^i + r_t^i$, where r_t^e the extrinsic reward provided by the environment at time t.

The ICM can be observed in Figure 2.7.

The question then arises – what does this system achieve? In the beginning, the networks will perform poorly, the Curiosity will be high, and so will the incentive to take actions that result in a higher intrinsic reward. However, as time progresses, the Forward and Inverse networks converge and provide a smaller and smaller intrinsic reward [5], the agent will then gradually act more and more on the extrinsic reward. Although with this approach, there exists a problem referred to as "The White Noise Problem".

2.8 The White Noise Problem

Deephak and colleagues [5] addressed "The White Noise Problem" when proving that if the ICM encounters an environment that is hard to predict, as if the agent watches a TV playing only white noise, the ICM will not be able to learn to predict the outcome of the tv accurately. Following this, the ICM continuously receives a high intrinsic reward that overshadows any extrinsic reward [5]. The intrinsic reward will motivate the DQN to learn to predict the white noise behavior, which it never will be able to do entirely, and with no success in this matter, the ICM does not choose to try any other action (Figure 2.8).

Figure 2.8: The agent is at risk of getting stuck watching white noise.

2.9 Further improvements

Apart from the previous improvements mentioned in this thesis, additional ones can be made. This section covers strategies to improve performance for reinforcement learning algorithms, which are not exclusive to the DQN or the ICM, but used in a wider capacity within the field of machine learning.

2.9.1 One-hot Encoding

When working with categorical information, such as colors, there is a need for the data to be converted into numerical values, so it can be fed into a neural network since the input of the neural network needs to be in a numerical format. One approach would be to assign each color a number, for instance, 1 could be red and 2 could be blue. However since 1 is smaller than 2, this would imply a relationship between the colors which do not exist, that blue is somehow "larger" than red. One way of avoiding this problem is using one-hot encoding. The strategy of one-hot encoding entails creating a vector with the same length as the number of categories and then having each element be 1 if the input is of the corresponding category, and else 0 [30]. For instance in the case of the red and blue colors, red might be (1, 0) and blue might be (0, 1). Now no false relationship is implied between the categories since each category inhabits its dimension on the vector.

2.9.2 Log Normalization

When using different inputs with a difference in magnitudes neural networks may have a problem with converging to the optimal solution. The problem occurs since the inputs are multiplied with the weights and then the result is used as output for the network, then the larger input may end up having a bigger impact when determining the output of the network [31]. The large output could then hinder the network when it tries to converge to the optimal solution since the input of a large magnitude might not always be the most important factor when determining a correct output [31]. How can one solve this?

One solution to this problem is to normalize all inputs, so that all inputs are between 0 and 1, effectively giving all inputs a similar magnitude. One way to normalize the inputs is to log normalize them [32], this is especially effective when the distribution is skewed since log normalizing gives a larger spread of values. When log normalizing each input value is preprocessed before being fed to the network, according to the following formula: $log\_norm(x) = log(1 + x)$. If each element of a matrix is log normalized in this way, it is called matrix log-normalization **??**.

Chapter 3
Related Works

Below are the most critical findings in the vast published knowledge base we reviewed before initiating our research.

3.1 Grandmaster level in Starcraft 2 using multi-agent Reinforcement Learning

AlphaStar is a multi-agent reinforcement learning agent that has achieved good results when used on the entire game of Starcraft 2 and was rated above 99.8% of all officially ranked human players and at Grandmaster level for all three races [2]. When playing Starcraft 2, the agent uses a combination of neural network architectures, imitation learning, reinforcement learning, and multi-agent learning. Using imitation learning, the agent learns by watching professional Starcraft 2 players when they play minigames [2].

The scope of the research behind AlphaStar and the complexity of the architecture is larger than the scope of this study. However, it is still relevant to review parts of this work for several reasons. AlphaStar achieves good results when playing Starcraft 2 with reinforcement learning. Those results provide a high benchmark to compare the results from smaller projects like ours while keeping in mind that the resources used in our study and the AlphaStar development are not comparable. The existence of massive projects, like AlphaStar, in combination with a significant number of smaller projects that use StarCraft 2, reiterate the relevance of using the game in the context of evaluating reinforcement learning approaches. Furthermore, used combinations of learning algorithms give hints about possible improvements to the agent created in this work [2].

One exciting aspect when reviewing the research behind AlphaStar is how the challenge of discovering novel strategies was addressed. Rather than using an "algorithm related" approach to this, e.g. using randomness or curiosity, they utilized data from human players, also called *imitation learning*.

3.2 Solving Starcraft 2 minigames using Deep Reinforcement Learning

In the report *Solving Starcraft 2 minigames using Deep Reinforcement Learning*, Kevin Johansson and Patrik Persson present their results from training an agent for Starcraft 2 minigames. Their results came close to the results in Deepmind's, despite using much fewer resources [19]. Our project uses their project as a starting point. Rather than implementing a DQN-agent from scratch, we used an existing agent that we knew works with the PYSC2-environment, whereafter we added the ICM to the PYSC2-environment. The DQN-agent was also used as a comparison point to verify that the implemented ICM improved performance. The only change that we made to the DQN-agent was to set the reward setting to sparse (Section 2.6) to compare the two agents in both reward settings.

3.3 Curiosity Driven Exploration

The most relevant report to our research, more or less serving as the starting point of our study, is *Curiosity-driven Exploration by Self-supervised Prediction*, by Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell.

In their report, Pathak et al. introduce the idea of using curiosity as an intrinsic reward in situations where extrinsic rewards are sparse or non-existent. The researchers defined in their report curiosity as *"the error in an agent's ability to predict the consequence of its actions in a visual feature space learned by a self-supervised inverse dynamics mode"*, when they evaluate the intrinsic curiosity model by letting the agent play the games Super Mario Bros and Doom [5].

Introducing the aspect of curiosity in a reinforcement-learning context adds a way of rewarding exploration in the agent's behavior. While traditional reinforcement-learning agents have the risk of finding a strategy that solves the given task "pretty well" and sticking with it, the introduction of an intrinsic reward, i.e. curiosity, incentivizes the exploration of other strategies. When working well, this stops the agent from exploiting a known strategy that works rather than trying to find the optimal strategy.

Since Pathaket al.'s evaluation shows that the addition of curiosity can improve performance, there are essentially two major benefits that come with a curiosity module. The first is, as mentioned above, an increased incentive to explore the environment, which results in increased performance. The second, maybe even more important than the increased performance is that the curiosity module solves the problem of having to engineer an artificial extrinsic reward structure for real-world scenarios that do not provide naturally suitable or only relatively sparse extrinsic rewards. The elimination of the need to develop a problem-specific extrinsic reward structure makes the solution much more scalable.

3.4 Large-Scale Study of Curiosity-Driven Learning

In the paper *Large-Scale Study of Curiosity-Driven Learning* by Burda et al. the researchers performed the first large-scale study on curiosity-driven learning, with no extrinsic reward structure to complement the intrinsic rewards provided by the curiosity module. The curiosity model used in this study is the one presented by Pathak et al. [17], a model used to play different arcade games, i.e. from the Atari game suite. Our study uses this model.

The motivation for exploring the possibilities of using curiosity as an intrinsic reward in the context of RL is primarily the same as discussed in Section 3.3. Using dense, problem-specific, hand-designed extrinsic reward structures is not scalable, whereas intrinsic rewards would be.

The most severe limitation of the purely curiosity-driven, prediction-based approach in stochastic setups is the *"noisy-TV problem"*, or in more general terms, just *"white noise"*. This phenomenon includes the occurrence of random changes to the environment that drastically limits the performance of the purely curiosity-driven algorithm [17].

3.5 Attention-Based Curiosity-Driven Exploration in Deep Reinforcement Learning

In the report *Attention-Based Curiosity-Driven Exploration in Deep Reinforcement Learning*, P. Reizinger and M. Szemenyei explore the possibilities of adding the concept of attention to curiosity learning models. They compared the results of different models and the attention-based curiosity model while playing different arcade games. The results in their paper are promising, as they show consistently improved scores compared to when using baseline models [18]. While the study does not occur within the context of Starcraft 2, the results presented in the cited paper lead us to believe that a combination of attention and ICM, or even attention, A2C and ICM, could produce better performance than what the DQN and ICM achieve. Although the *"white noise"* problem did not occur in our project, the results regarding the handling of the problem are, in our opinion, very promising.

The noise problem (see 2.8, 3.4 and [17]), or *"noisy TV problem"*, is a situation where more or less random things, that the agent cannot fully control, occur in the environment. They can be triggered by actions taken by the agent or happen randomly. The solution presented by Reizinger and Szemenyei is to use an attention network on the forward loss function of the ICM module while ignoring the inverse loss [18]. The goal of this is to separate situations where curiosity is beneficial from situations where it is not, creating what they call rational curiosity [18].

Chapter 4

Systems

To decide upon a policy, two systems were developed using Python v.2.7 and Tensorflow v.2.7.8, a Deep Q-network using ϵ-greedy select and a Deep Q-network using e-greedy and an Intrinsic Curiosity Module. The game was played through the framework PYSC2 using a windows machine and an Anaconda environment.

4.1 PYSC2

Version v.2.1.0 of PYSC2 was used during development and testing. For this version, the game sequence occurs like described in Section 2.2.1.

4.1.1 Game Sequence

For the "Game init" phase, the resolution of the game window initializes as 84x84 pixels, and all network weights initialize with a Xavier uniform initializer [29]. The initializer sets each layer's weights so that the variance is the same for each network layer, which helps avoid exploding or vanishing gradients when optimizing the loss function, which would keep the network from converging [29]. No relevant custom code is run for the "Episode init" phase, so the state and the reward are read. In the "Decide Action" phase, each system runs its code, including saving transitions, updating weights, and generating values needed for training. Finally, the game step advances typically by one step each time but using a hyperparameter called "step_mul" the game step sets to advance 20 game steps instead of one, where the reward of all 20 steps gets added and given for $t + 1$ [28]. Section 2.2.1 presents an overview of the process.

4.1.2 States and rewards

As mentioned above, the game window initializes as 84x84 pixels, meaning that each state matrix is also 84x84, and all feature screen matrixes are used as input, resulting in 27 matrices in the dimensions 84x84x27 for an individual state input. Section 8.2 displays what each matrix conveys, as well as if the information is categorical or numerical. For exact details of how each matrix is encoded, please see below[1].

In conjunction with a state, PYSC2 also supplies rewards. For this study, two separate reward structures were developed. One was identical to the "DefeatRoaches" scoring, as being described in Section 2.1.1 and the second reward structure accumulated all scores and returned them to the agent at the last step of the episode. These two reward functions were called none-sparse respectively sparse, were the driving factors behind states, and rewards were the actions chosen by each policy.

4.1.3 Actions

The system only allowed for one action type, the "attack" action. As "attack" can take place at each pixel and as there are 84x84 pixels or 7056 total pixels, there are 7056 possible actions. The "attack" action order may induce different behavior based on an internal set of rules, where the top rules take precedence over the rules below.

1. If enemy units are at the location of the selected pixel, all friendly units will attack the enemy unit.
2. If your friendly units are fighting then nothing will occur, regardless of the pixel selected.
3. If your units are out of combat, all units move towards the selected pixel, if it encounters any enemy units on the way, all friendly units will attack the closest enemy unit.

4.2 DQN using ϵ-greedy Select

The DQN using ϵ-greedy Select forms the base for both algorithms and starts by checking if there are enough entries to fetch from the Replay memory. If not, it skips to choosing an action, but if there's enough, it starts by retrieving transitions from its replay memory. The size of the replay memory is set to 10 000 transitions(replay memory size) [28], and the number of transitions fetched for each step must be 16(batch size) [28]. Before training, additional preprocessing and generating target values occurs.

States and actions are preprocessed before being given to the network as input by having actions, and categorical input state matrices one hot-encoded, and numerical input state matrices log normalized. State matrix one hot-encoding ensures that actions are given a numerical value, so it can be processed by the networks, and that no false relation between

[1]https://github.com/deepmind/pysc2/blob/master/pysc2/lib/features.py

actions occurs (e.i action 1 should not be smaller than action 2) [30]. Log normalization ensures that input values are close to 0, since all matrix values are positive, this in turn ensures that all input values are scaled equally(irrelevant of unit of input), not causing any input to be unjustifiably emphasized [31]. The categorical and numerical input state matrices are then fed through an embedding network. All preprocessed state matrices are then concatenated. An overview of the process can be observed in Figure 4.1. After preprocessing the target network then generates Q-values for s_{t+1} and target values are calculated using the preprocessed actions for each entry(see Equation 2.2).

After this, it uses an RMSprop-optimizer, with a learning rate of 10^{-5}, updates the Q-networks weights based on the generated target values and the Q-networks predicted Q-values. The RMSprop-optimizer outperforms any other optimizer available from Tensorflows framework given our system and given the task of learning to play "Defeat Roaches". The loss function in use is Mean-Square-Error or $Loss_{dqn} = MSE(\hat{Q}, Q^{target})$ (see Section 2.3.2 for motivation). The Q-network and the target network are identical but with an individual set of weights. Both network structures can be observed in Figure 4.2). After training, the target network updates its weights, and this occurs every 5000th step (target update) [4][19] for the none sparse environment and 500th step for the sparse environment.

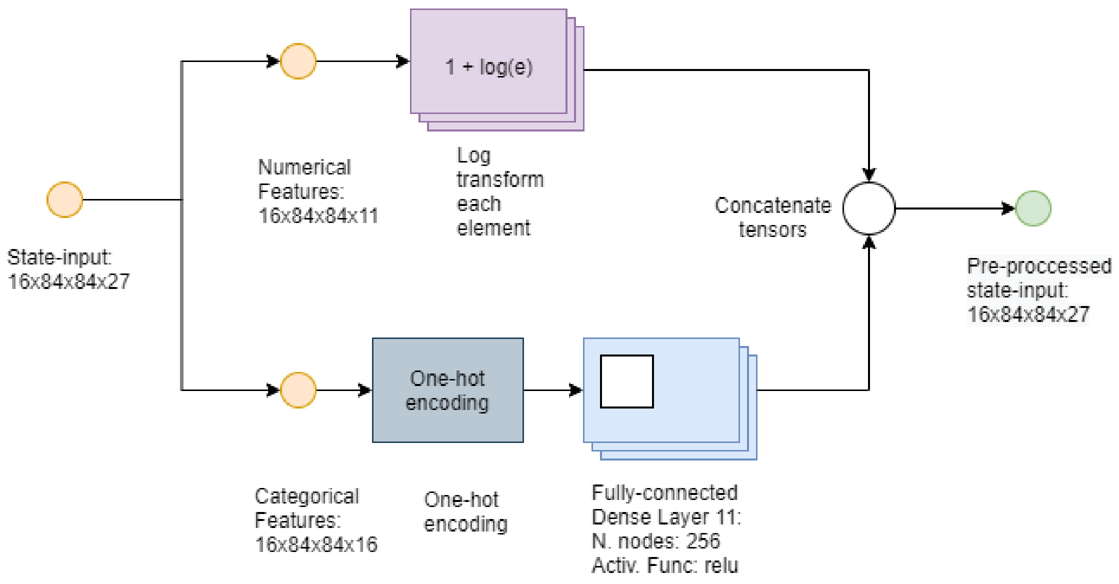

1 + log(e)
Numerical Features: 16x84x84x11
Log transform each element
Concatenate tensors
State-input: 16x84x84x27
Pre-proccessed state-input: 16x84x84x27
One-hot encoding
Categorical Features: 16x84x84x16
One-hot encoding
Fully-connected Dense Layer 11: N. nodes: 256 Activ. Func: relu

Figure 4.1: The process of preprocessing state inputs

After the network update, it picks an action. If its the first step of the episode, ϵ is updated according to

$$\epsilon(\underbrace{\sigma}_{\text{current episode}}) = e^{-0.001*\sigma}$$

Where with the probability of ϵ the agent picks an action according to a uniform distribution, and with the probability of (1-ϵ), it chooses the action with the highest predicted Q-value.

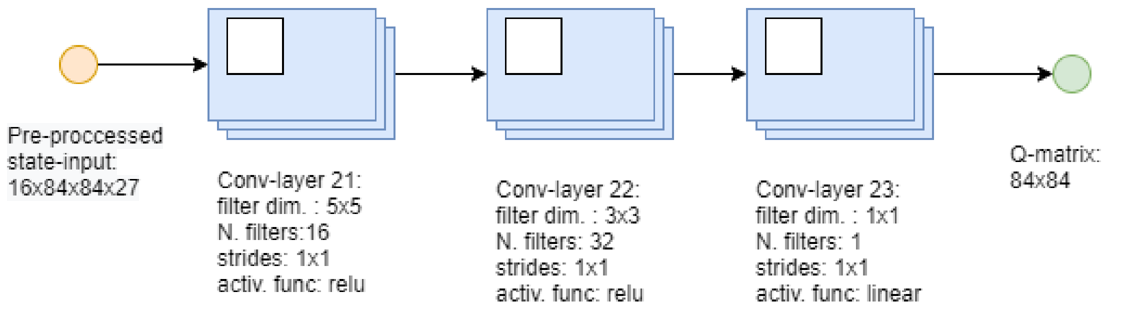


Figure 4.2: The structure of the Q and target network.

4.3 DQN using ϵ-greedy Select and ICM

The DQN using ϵ-greedy Select and ICM are identical to the DQN using ϵ-greedy Select with two exceptions, it adds an intrinsic reward, and it uses two additional neural networks(Inverse and Forward Network) to model Intrinsic Curiosity. Before the target values are calculated for the DQN, the intrinsic reward is calculated for each entry in the batch, then added on top of the extrinsic reward: $r_t = (\eta/2) * r_t^i + r_t^e$, and the target values are calculated with the newly calculated r_t, with $\eta = 0.1$. The next deviation occurs when training. Instead of minimizing the loss of only the DQN network, the loss of the Forward and Inverse Network is also minimized. The total loss is given as $loss_{total} = \lambda * loss_{dqn} + \beta * loss_{for} + (1-\beta) * loss_{inv}$, where $\lambda = 0.001$ for the none sparse environment and $\lambda = 0.1$ for the sparse environment. For both environments $\beta = 0.7$.

The Inverse Network takes as input $pre(s_t)$ and $pre(s_{t+1})$ and outputs state vectors $\phi(s_t)$ and $\phi(s_{t+1})$, with the length of 288 respectively. The vectors are then both fed into a set of fully connected dense layers which outputs a prediction of the x and y coordinate where the "attack" action occurred. The network structure can be observed in Figure 4.3 and the loss function is Mean Square Error or equally $loss_{inv} = MSE(\sqrt{x^2 + y^2}, \sqrt{\hat{x}^2 + \hat{y}^2})$. This is a custom loss function that differs from the original implementation, which used a softmax function for all given actions [5]. When using the original softmax loss function, the Inverse Network would not converge; however, it did using our custom loss function.

The Forward Network takes as input the current state vector $\phi(s_t)$, produced by the Inverse Network, and the current preprocessed action $pre(a_t)$. It outputs the predicted next state vector $\phi(s_{t+1})$.The network structure can be observed in Figure 4.4 and the loss function used is Mean Square Error or $loss_{for} = MSE(\phi(s_{t+1}), \hat{\phi}(s_{t+1}))$.

In summary this results in DQN using ϵ-greedy Select having 256 dense nodes, 462422016 dense weights and 1378 convolutional filter weights, while DQN using ϵ-greedy Select and ICM has 48772097 dense nodes, 12566061313 dense weights and 5026 convolutional filter weights.

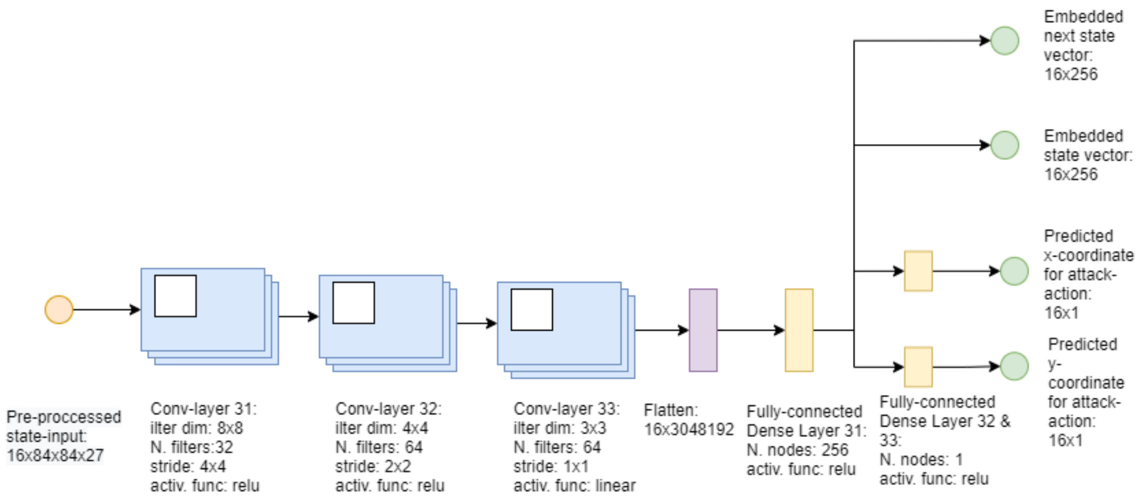

Embedded next state vector: 16x256
Embedded state vector: 16x256
Predicted x-coordinate for attack-action: 16x1
Predicted y-coordinate for attack-action: 16x1
Pre-processed state-input: 16x84x84x27
Conv-layer 31: ilter dim: 8x8 N. filters:32 stride: 4x4 activ. func: relu
Conv-layer 32: ilter dim: 4x4 N. filters: 64 stride: 2x2 activ. func: relu
Conv-layer 33: ilter dim: 3x3 N. filters: 64 stride: 1x1 activ. func: linear
Flatten: 16x3048192
Fully-connected Dense Layer 31: N. nodes: 256 activ. func: relu
Fully-connected Dense Layer 32 & 33: N. nodes: 1 activ. func: relu

Figure 4.3: The structure of the Inverse Network

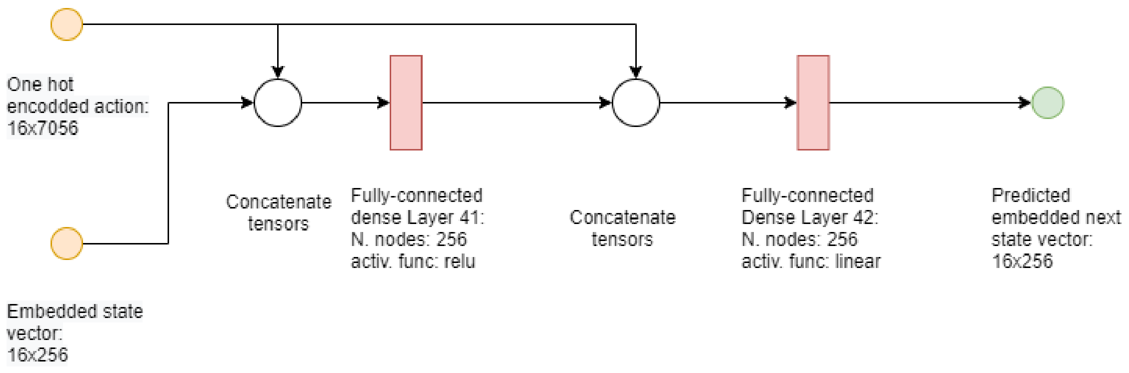

One hot encodded action: 16x7056
Embedded state vector: 16x256
Concatenate tensors
Fully-connected dense Layer 41: N. nodes: 256 activ. func: relu
Concatenate tensors
Fully-connected Dense Layer 42: N. nodes: 256 activ. func: linear
Predicted embedded next state vector: 16x256

Figure 4.4: The structure of the Forward Network

Chapter 5
Experiments

In this chapter, the experiments used to evaluate the agent, as well as the results from those experiments, will be presented. After that, a section regarding the loss of the inverse network of the ICM follows. In the last section, the observed behavior and playing strategies of the agent will be addressed.

The first experiments consist of the agent playing the minigame *DefeatRoaches* (see 2.1.1), with two different reward settings (sparse rewards and standard reward, see 2.6). To show the difference in performance, the achieved game scores are compared to scores achieved by the baseline DQN agent. This is followed by an analysis of the ICM's inverse network's loss in Section 5.2. Lastly, in Section 5.3 the observed behavior and strategies of the agent are presented.

Implementing the networks above results in a big difference of complexity for the networks, where the agent using DQN and ICM is vastly more complex, meaning the agent using DQN and ICM has a much larger amount of nodes and weights. The exact difference in complexity is shown in Section 4.3, but did the added complexity achieve any difference in performance?

5.1 DefeatRoaches

Below the results from experiments on the minigame *DefeatRoaches* (see section 2.1.1) with two different reward settings (see section 2.6) are presented.

5.1.1 Standard Reward

As discussed in sections 2.6 and 4.1.3, two different reward settings are used when evaluating the models. *Standard Reward* means that the agent receives rewards after each taken action.

Figure 5.1 shows the achieved mean score of the agent using DQN and ICM (blue), compared with the baseline DQN agent (red). A clear improvement in terms of game score can be seen. In the first 2000, some episodes the agent is improving faster and is also reaching a higher level with a narrower standard deviation, compared to the baseline DQN agent.

Figure 5.2 shows the five individual runs. Between about 2000 and 6000 episodes, there is a quite large spread in the scores. After about 6000 episodes the spread is getting smaller. While the average score is not improving significantly after this, the smaller difference means the agent is performing more consistently at this point, with less frequent spikes and dips.

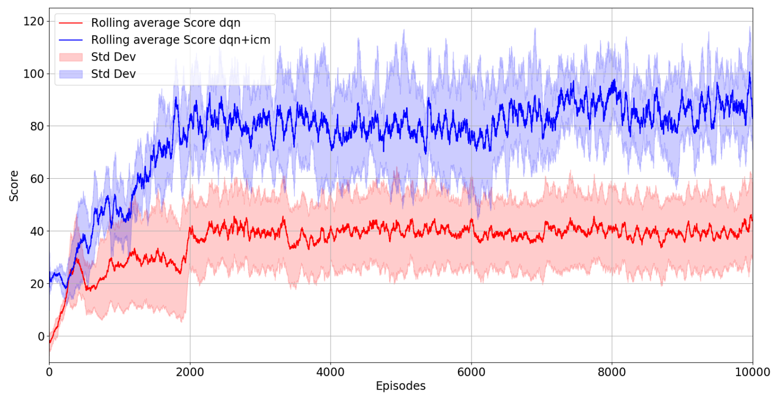


Figure 5.1: The mean scores of 5 runs (10000 episodes) for each model, on the minigame *DefeatRoaches* with the standard reward setting. The lines show the rolling average with a window size of 50 episodes, with one standard deviation (Std Dev) confidence interval (shaded areas). Note that score and reward are distinct from each other, except when only the extrinsic reward is used.

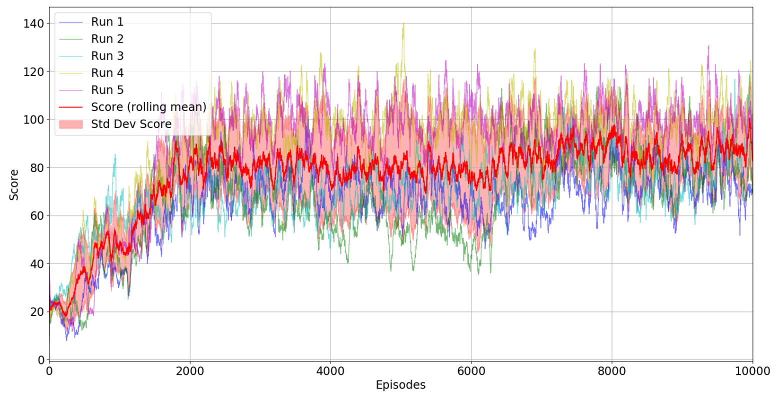


Figure 5.2: The rolling average score for an agent-driven by external and internal reward (DQN+ICM), with a window size of 50 episodes, of 5 runs (10000 episodes), as well as the mean of these 5 runs (red line), with standard deviation (Std Dev). The experiment is run on the minigame *DefeatRoaches* with the standard reward setting. Note that the scale of the y-axis (Score) has changed compared to previous figures. The score is not equivalent to the reward here.

5.1.2 Sparse Reward

The results below show how the DQN+ICM agent performed compared to the baseline DQN agent when playing the minigame *DefeatRoaches* (see Section 2.1.1) with sparse rewards (see Sections 2.6 and 4.1.2.

In figure 5.3 the graphs show that while neither of the models is improving over time, the agent combining DQN and ICM is at least not regressing over time and performs at a more consistent level. The individual runs of the agent are shown in Figure 5.4.

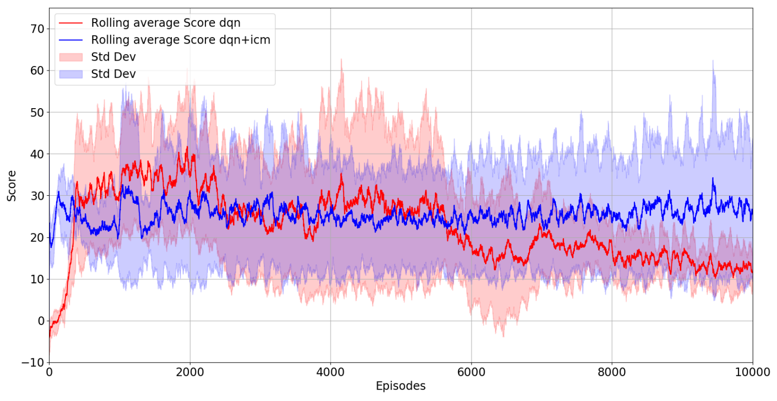


Figure 5.3: The mean scores of 5 runs (10000 episodes) for each model. The lines show the rolling average with a window size of 50 episodes, with one standard deviation (Std Dev) confidence interval (shaded areas). Note the changed scale of the y-axis (Score) compared to previous figures. The experiment is run on the minigame *DefeatRoaches* with the sparse reward setting. Score and reward are distinct from each other, except when only the extrinsic reward is used.

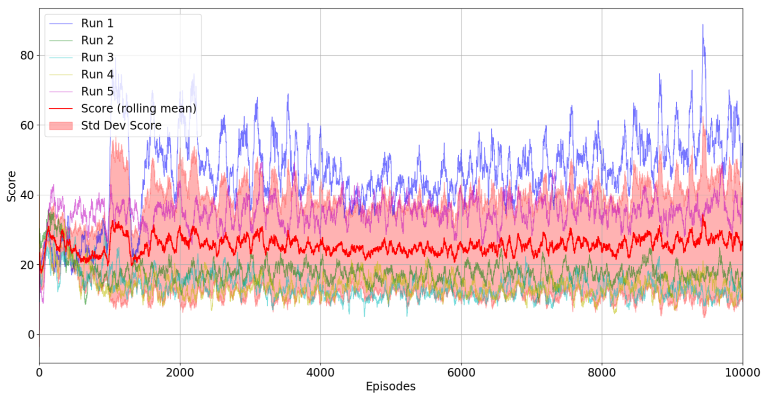


Figure 5.4: The rolling average score for an agent-driven by external and internal reward (DQN+ICM), with a window size of 50 episodes, of 5 runs (10000 episodes), as well as the mean of these 5 runs (red line), with standard deviation (Std Dev). The experiment is run on the minigame *DefeatRoaches* with the sparse reward setting. Note that the scale of the y-axis (Score) has changed compared to previous figures. The score is not equivalent to the reward here.

5.2 Network Loss for DQN with ICM

In this section, graphs of the loss of the ICM's inverse network are shown. Figures 5.5 and 5.6 show the loss when using standard and sparse rewards (see sections 2.6 and 4.1.2) respectively. The figures show that the loss of the inverse network is not converging in the same way and has a far larger spread when the sparse reward setting is used. This explains why the agent does not learn as well and does not achieve scores on the same level as when the standard reward setting is used (See the figures in sections 5.1.1 and 5.1.2).

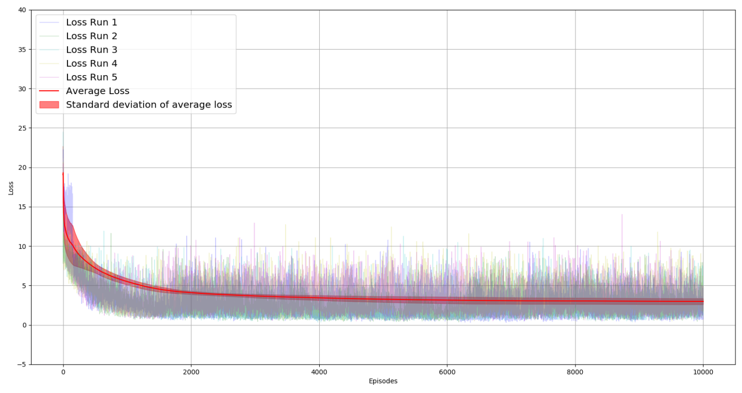


Figure 5.5: Loss of the inverse model of the DQN+ICM (Average and individual runs, 10000 episodes) with the standard reward setting. The agent used here is the same as in Section 5.1.1, playing *DefeatRoaches*.

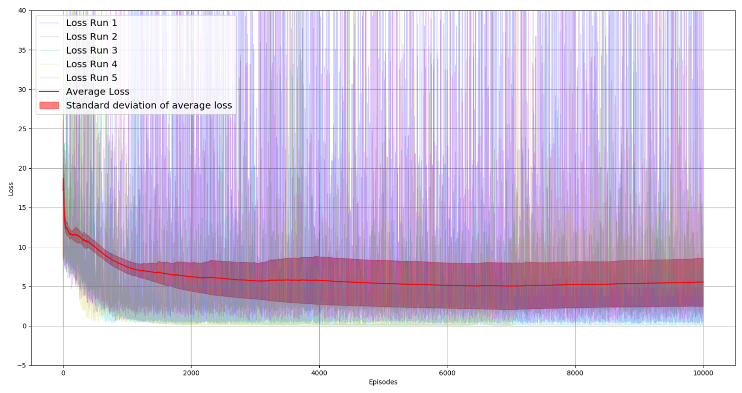


Figure 5.6: Loss of the inverse model of the DQN+ICM (Average and individual runs, 10000 episodes) with the standard reward setting. The agent used here is the same as in Section 5.1.2, playing *DefeatRoaches*.

5.3 The Agents' Strategies

Examples showing how the agents trained during this work play can be observed here:

`https://www.youtube.com/watch?v=07FUgVS9jQkfeature=youtu.be`
`https://www.youtube.com/watch?v=cP7mxN2A0Cgfeature=youtu.be`

Of the strategies discussed in Section 2.1.2, our agents only learned the "Focus fire" behavior. Some attempts at "Kiting" can be observed occasionally, but not consistently, and not with any success at particularly high levels. While the agent did not learn to kite, improvements regarding the chosen position to attack from can be observed as training progresses. It is possible that the agent could learn kiting with longer training. Due to time and resource limitations, it was not feasible for us to do significantly longer training to verify this.

Two other behaviors that can be observed are the "standing still and do nothing" and the "run around in circles" behaviors. These are, of course, not behaviors we want the agent to learn. Both of them are connected to the white noise problem (see section 2.8). Further discussions about how this problem was dealt with and how it could be further improved can be found in the following section.

Chapter 6

Discussion

Now some final reflections regarding the experiment will be discussed. We reflect on what went well, also possible improvements to the ICM in general and for this specific task, follow up on the discussion of possible improvements, and we suggest some further possible research.

6.1 Evaluating the ICM

From the result we reached, one can conclude that the ICM improved performance for the DQN agent, both in a sparse and none-sparse environment. By using the ICM, the agent learned faster and received a higher-end score. However, the improvement came at a cost since the neural networks used are larger and more complex when including the ICM.

When needing to minimize loss for the target and Q-network and needing to do the same for the inverse and forward network caused an increase in both training time and a more significant need for storage capacity.

Another issue caused by having a larger and more complex set of neural networks is that balancing all hyperparameters becomes vastly more complex since these networks do not only need to balance all hyperparameters for the Q and target network but also the Inverse Network and Forward Network. Furthermore, the developer also needed to balance the amount of intrinsic reward the agent should receive for each step, using the η parameter.

If one accepts the cost associated with increased performance, one also needs to be conscious of the fact that the ICM has limitations. As mentioned in Section 2.8, if the agent encounters "white noise", the ICM becomes a detriment to the agent performance. Another limitation that occurred when running DefeatRoaches, is that given a sufficiently larger and complex action space, the Inverse-network could not converge using the soft-max loss function (as used by Deephak and colleagues) [5] since the action space is complex, in the sense that many actions have the same outcome (see Section 4.1.3). The question then arises - how did our implementation perform?

6.2 Evaluating the Implementation

Firstly, the white noise problem (see Section 2.8) did not occur in any part of our implementation, most likely because the mini-game "Defeat Roaches" only has randomness when the game is placing the new roaches and marines. This behavior only occurs for a brief moment, and therefore the agent cannot wait to watch any white-noise behavior continuously.

When evaluating the loss for the DQN when an ICM is added while using a none sparse reward structure, one can observe that the DQN network only somewhat converges. Since it did not fully converge, there is a possibility that another policy would be found if the agent ran for more episodes. The DQN network loss did fully converge when not adding an ICM.

When evaluating the inverse network, it did not converge when using the soft-max function as the loss function. However, it did converge when using our custom loss function described in Section 4.3. It was too difficult for the Inverse Network to fit the underlying distribution for the large and complex action-state space. We hypothesized that this depends on the size of the action space and the fact that many actions have the same outcome (see Section 4.1.3), making it hard to predict for the Inverse Network what action causes the transition from s_t to s_{t+1}. However, further research is required before any conclusion can be drawn.

The Forward Network converge even if the Inverse Network converged or not, which is in line with what Deephak and colleagues stated in their report, that is, an Inverse Network that encode all information will also encode data inconsequential to the agent [5]. The inconsequential data effectively acted as noise for the Forward Network since encoding data out of the agent's control and which do not affect the agent is insignificant when the Forward Network tries to produce an output. However, the noise is easy to account for since "DefeatRoaches" is a deterministic environment, resulting in the noise not having any variance. The Forward Network was therefore able to produce a steady stream of intrinsic reward.

Since there is an increase in scoring when using the ICM, one can assume that the addition of the intrinsic reward supplied a signal beneficial for performance. Furthermore, since the Forward and Inverse Network converged, one can assume that the signal provided is one of Intrinsic Curiosity. The question then arose - is it possible to improve the agent further?

6.3 Possible Improvements

The main phenomenon in our attempt was that the DQN network did not completely converge in the none sparse environment. Running more episodes could have solved this problem, as the network would have had more time to converge. This hypothesis is supported by comparing our 10 000 runs with the team at Deepmind, who ran about 1 000 000 runs and saw development throughout.

There could also be an issue of over or underfitting. The original DQN was designed to reach scores close to 40, while our agent reached top scores of more than 300. This could be amended by changing the hyperparameters λ, learning rate, and/or the network structure of the DQN networks. Implementing these changes would be interesting for further research.

6.4 Suggestion for Further Research

As is mentioned in the previous section, it would be of interest to evaluate if the DQN not fully converging is inherent when adding the ICM. This could be done by running more episodes or by experimenting more thoroughly with hyperparameters.

Another issue raised by this thesis is the fact that the Inverse Network did not converge when using the traditional softmax function. Comparing the performance of our loss function to the traditional softmax loss function would be of interest. The new loss function used in this thesis exploits the relationship between actions. For instance, if the action attack occurred at (0,0), then predicting (1,0) would be better than predicting (81,81). Possible research would be to run the original Deephak and colleagues implementation [6] and use the idea of a loss function that exploits the relationship between different actions and compare performance. For instance, when playing Mario, if the action "move left" occurred, then predicting "jump" is probably more correct than predicting "move right". The specifics of the formulation of any loss function are left to the researchers.

6.5 Pitfalls of the ICM

In future research, it should be noted that there exist pitfalls when working with the ICM. One of these pitfalls is trying to balance all networks at the same time when working with the DQN and ICM. One should set hyper-parameters for each network separately, then once both converge, try to integrate both networks with each other. There still might be a big change in hyperparameters after integration, but this approach serves as a good starting point for setting hyperparameters. Finally, there exist some pitfalls specific to this implementation.

6.6 Pitfalls of the Implementation

Running our implementation takes significant computing power. In these experiments, an Nvidia RTX 2080 was used to run the software, and the program was barely able to run. If one wishes to continue this research, make sure to have significant hardware resources.

Chapter 7
Conclusion

For this thesis, an Intrinsic Curiosity Module was developed to play the game of Starcraft 2, a game with a very high amount of actions (for this thesis, 7056 possible actions). Then two different Deep-Q Network agents, one with an Intrinsic Curiosity Module and one without, were trained to play the minigame "Defeat Roaches" within the game of Starcraft 2. Finally, two different reward environments were developed, one sparse, giving a low amount of rewards, and one none sparse, providing a higher amount of rewards.

The results showed that the Intrinsic Curiosity Module learned faster, had its score plateau faster, and reached a higher end-score in both the sparse and also none sparse reward environment. However, the Deep-Q Network agent with the Intrinsic Curiosity Module never fully converged for some of the runs. This implies that there could be further change in that agent's strategy, meaning that the end score might differ if the agent is allowed to run for more episodes.

These findings point to the Intrinsic Curiosity Module improving training speed; however, it is unclear if the module improved end score since there could be a change in the Deep-Q Network agent's behavior if allowed to run for more episodes. Although all the networks related to the Intrinsic Curiosity Module managed to converge, proving that the Intrinsic Curiosity Module provided its intended purpose.

The hardest part of the thesis was balancing and running the agents since each run took a lot of computing power and time. It became hard to optimize the network since there was a lack of time after writing and balancing the ICM.

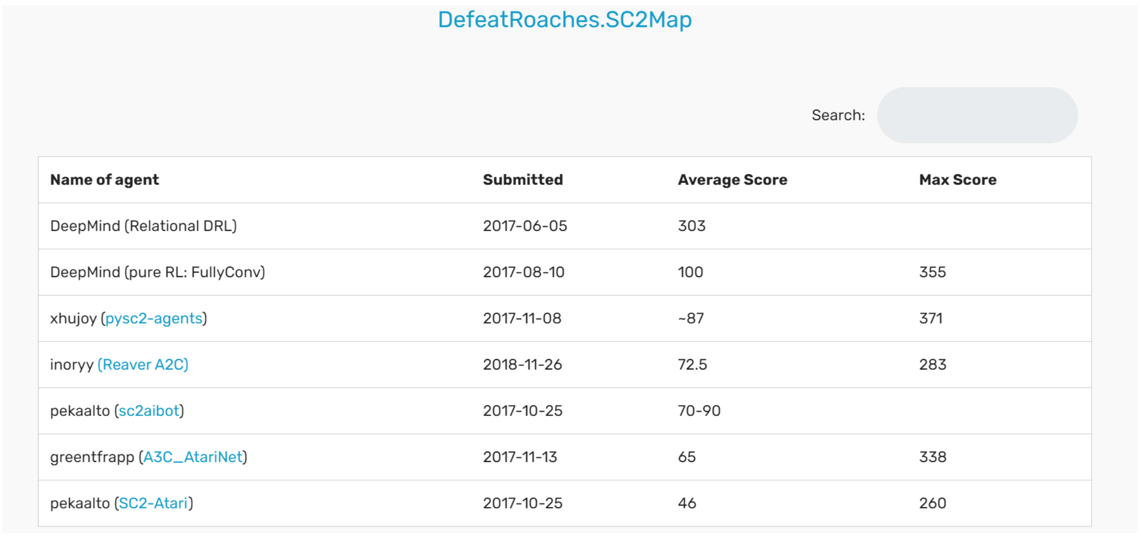
The thesis answered almost all of the research questions, however, to fully answer the research questions, the agents would have to run for more episodes, so that the DQN networks would fully converge for all runs.

Chapter 8

Appendix

8.1 PYSC2 Leaderboards

Below are screenshots of the *DefeatRoaches* and *DefeatZerglingAndBanelings* global leaderboards from the date 15/10-2020. The leaderboard only contains reinforcement learning agents (no human players). The current leaderboards can be found, at the time of this report, at the following link: `http://starcraftgym.com/`



DefeatRoaches.SC2Map

Search:

Name of agent	Submitted	Average Score	Max Score
DeepMind (Relational DRL)	2017-06-05	303	
DeepMind (pure RL: FullyConv)	2017-08-10	100	355
xhujoy (pysc2-agents)	2017-11-08	~87	371
inoryy (Reaver A2C)	2018-11-26	72.5	283
pekaalto (sc2aibot)	2017-10-25	70-90	
greentfrapp (A3C_AtariNet)	2017-11-13	65	338
pekaalto (SC2-Atari)	2017-10-25	46	260

Figure 8.1: Global *DefeatRoaches* leaderbord

8.2 PYSC2 State Representations

Below is a table which choose all Screen-feature representations used as state input for each experiment. In the table units and buildings are referred to as entities. All game-mechanics relevant for this table are discussed at www.Starcraft2.com.

Screen-Feature representations		
Feature-name	Feature-type	What information it conveys
height_map	NUMERICAL	A height map of the game area.
visibility_map	CATEGORICAL	A map showing what is visable to the player
creep	CATEGORICAL	The location of creep.
power	CATEGORICAL	The power of each entities
player_id	CATEGORICAL	Which player owns what entities
player_relative	CATEGORICAL	Location of each friendly and enemy entity.
unit_type	CATEGORICAL	Unit type of each unit.
selected	CATEGORICAL	Which units are selected and which are not
unit_hit_points	NUMERICAL	The absolute amount of health-points for each entity
unit_hit_points_ratio	NUMERICAL	The ratio of health-points left for each entity
unit_energy	NUMERICAL	The absolute amount of energy for each unit
unit_energy_ratio	NUMERICAL	The ratio of energy left for each unit
unit_shields	NUMERICAL	The absolute amount of shields for each unit
unit_shields_ratio	NUMERICAL	The ratio of shield left for each unit
unit_density	NUMERICAL	The amount of units in each pixel
unit_density_aa	NUMERICAL	The amount of units in each pixel
effects	CATEGORICAL	Every effect affecting every entity
hallucinations	CATEGORICAL	If friendly units are hallucinations or not.
cloaked	CATEGORICAL	If friendly units are invisible or not.
blip	CATEGORICAL	Location of every blip
buffs	CATEGORICAL	Every buff(power-ups) affecting every entity
buff_duration	NUMERICAL	The duration of each buffs for every friendly entity
active	CATEGORICAL	If friendly land-mines are active or not
build_progress	NUMERICAL	The time left before a building is complete
pathable	CATEGORICAL	A map of where units can move
buildable	CATEGORICAL	A map of where any units can build
placeholder	CATEGORICAL	Just a placeholder

8.3 Work Load Distribution

Daniel Karlsson (LTH) developed the addition of the ICM, balanced its hyperparameters, and integrated it to work with the DQN agent. Jascha Thiel (LTH) ported the existing DQN base agent from Tensorflow version 1 to version 2, explored and compared different environments to run the agent (Windows, Ubuntu, WSL and Docker), developed the code to extract and display the data seen in the graphs (using the Tensorflow framework) and gave minor assistance with balancing the hyperparameters for the ICM.

Regarding this thesis, Daniel Karlsson wrote the main part of the Theory, Systems, Introduction, Appendix and Discussion, while Jascha Thiel wrote the main part of the the Related Works and Result sections. Both Jascha and Daniel did language checking and minor changes to every section of the report.

8.4 Technical assistance

The PYSC2 installation guide is somewhat outdated(see https://github.com/deepmind/pysc2). Most notably, a step in the installation demands the user to put the mini-game files in the "maps" folder. Note, however, that the maps folder is no longer in the Starcraft 2 directory, so the user has to make a "maps" folder and then put the mini-game files there.

If the user wants to create custom sparse reward structures, the documentation states that the last step should produce +1(win), -1(loss), and 0(tie), however this does not occur if the developer sets the hyper-parameter *step_mul* to anything but 1. Since for each step PYSC2 sums the score for all the steps skipped, if any of the last steps have some score reward, it will be added to the last step's reward (for further explanation see Section 4.1.1)

References

[1] The Guardian. World's best Go player flummoxed by Google's 'godlike' AlphaGo, AI. https://www.theguardian.com/technology/2017/may/23/alphago-google-ai-beats-ke-jie-china-go

[2] Vinyals, O., Babuschkin, I., Czarnecki, W.M. et al. Grandmaster level in Starcraft 2 using multi-agent reinforcement learning. Nature 575, 350–354 (2019). https://doi.org/10.1038/s41586-019-1724-z

[3] Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd. ed.). Prentice Hall Press, USA. P.830-833.

[4] Volodymyr Mnih and Koray Kavukcuoglu and David Silver and Alex Graves and Ioannis Antonoglou and Daan Wierstra and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. 2013.

[5] Pathak, Deepak andAgrawal, Pulkit and Efros, Alexei A. and Darrell, Trevor. Curiosity-driven Exploration by Self-supervised Prediction. ICML. 2017

[6] Pathak, Deepak and Agrawal, Pulkit and Efros, Alexei A. and Darrell, Trevor. Curiosity-driven Exploration Github. 2021.

[7] Xilinx. "HDL Synthesis for FPGAs Design Guide". section 3.13: "Encoding State Machines". Appendix A: "Accelerate FPGA Macros with One-Hot Approach". 1995.

[8] Piech, C., Bassen, J., Huang, J., Ganguli, S., Sahami, M., Guibas, L. J., Sohl-Dickstein, J. (2015). Deep knowledge tracing. In Advances in Neural Information Processing Systems (P. 505–513).

[9] Tamhane, A., Arora, S., Warrier, D. (2017, May). Modeling Contextual Changes in User Behaviour in Fashion e-Commerce. In Pacific-Asia Conference on Knowledge Discovery and Data Mining (pp. 539–550). Springer, Cham.

[10] M. Shanker, M.Y. Hu, M.S. Hung. 1996. Effect of data standardization on neural network training, Omega, Volume 24, Issue 4, Pages 385-397

[11] Blizzard. GAME OVERVIEW. https://starcraft2.com/en-us/game. [2020]

[12] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L. Lewis, and Satinder P. Singh. Action-conditional video prediction using deep networks in atari games. In NIPS, 2015.

[13] Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd. ed.). Prentice Hall Press, USA. P.839-842.

[14] Volodymyr Mnih and Adrià Puigdomènech Badia and Mehdi Mirza and Alex Graves and Timothy P. Lillicrap and Tim Harley and David Silver and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. 2016. 1602.01783

[15] Shannon, Claude E. (July 1948). "A Mathematical Theory of Communication". Bell System Technical Journal. 27 (3): 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x.

[16] Vinyals, Oriol and Ewalds, Timo and Bartunov, Sergey and Georgiev, Petko and Vezhnevets, Alexander and Yeo, Michelle and Makhzani, Alireza and Küttler, Heinrich and Agapiou, John and Schrittwieser, Julian and Quan, John and Gaffney, Stephen and Petersen, Stig and Simonyan, Karen and Schaul, Tom and Van Hasselt, Hado and Silver, David and Lillicrap, Timothy and Calderone, Kevin and Tsing, Rodney.
Starcraft 2: A New Challenge for Reinforcement Learning. 2017
Github: github.com/deepmind/pysc2

[17] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, Alexei A. Efros. Large-Scale Study of Curiosity-Driven Learning. 2018

[18] Reizinger, Patrik and Szemenyei, Márton. Attention-based Curiosity-driven Exploation in Deep Reinforcement Learning. ICASSP. 2020

[19] Johansson, Kevin and Persson, Patrik. Solving Starcraft 2 minigames using Deep Reinforcement Learning. Date Unknown

[20] Gagniuc, Paul A. (2017). Markov Chains: From Theory to Implementation and Experimentation. USA, NJ: John Wiley Sons. pp. 1–235. ISBN 978-1-119-38755-8.

[21] Sean Meyn; Richard L. Tweedie (2 April 2009). Markov Chains and Stochastic Stability. Cambridge University Press. p. 3. ISBN 978-0-521-73182-9. Archived from the original on 23 March 2017.

[22] Li QL. (2010) Markov Reward Processes. In: Constructive Computation in Stochastic Models with Applications. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-11492-2_10

[23] Bellman, R. (1957). "A Markovian Decision Process". Journal of Mathematics and Mechanics.

[24] Abhishek Suran. On-Policy v/s Off-Policy Learning. https://towardsdatascience.com/on-policy-v-s-off-policy-learning-75089916bc2f.
[Jul 14 2020]

[25] Sutton, Richard S. and Barto, Andrew G. Reinforcement Learning: An Introduction. 2018.0262039249. A Bradford Book. Cambridge, MA, USA. P. 124 - 127

[26] Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd. ed.). Prentice Hall Press, USA. P.728-732.

[27] Yamashita, R., Nishio, M., Do, R.K.G. et al. Convolutional neural networks: an overview and application in radiology. Insights Imaging 9, 611–629 (2018). https://doi.org/10.1007/s13244-018-0639-9

[28] Deepmind. PYSC2 Github. https://github.com/deepmind/pysc2/blob/master/pysc2/lib/features.py. [27-06-2021]

[29] Siddharth Krishna Kumar. 2017. On weight initialization in deep neural networks.

[30] Brownlee, Jason. 2017. "Why One-Hot Encode Data in Machine Learning?". Machinelearningmastery. https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/ [15-01-2022]

[31] Bex.T 2020. "Logarithmic Scale in Data Science: How to Plot and Actually Understand It". https://towardsdev.com/logarithmic-scale-how-to-plot-and-actually-understand-it-c38f00212206 [15-01-2022]

[32] Ha, Thang and Lubo-Robles, David and Marfurt, Kurt and Wallet, Bradley. 2021. On weight initialization in deep neural networks.

[33] Aakash Maroti. RBED: Reward Based Epsilon Decay. 2019

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2021-12-21

EXAMENSARBETE Evaluating Curiosity Driven Exploration in a Large Action Space using Starcraft 2
STUDENTER Daniel Karlsson, Jascha Thiel
HANDLEDARE Volker Krueger (LTH)
EXAMINATOR Elina Anna Topp (LTH)

Intrinsic Curiosity Module förbättrar en artificiell intelligens

POPULÄRVETENSKAPLIG SAMMANFATTNING **Daniel Karlsson, Jascha Thiel**

Intrinsic Curiosity Module är en ny spännande artificiell intelligens algoritm, som tidigare har fått lära sig spela arkadspel (ex: Mario) med lovande resultat. I detta arbetet testar vi dess förmåga att hantera mer avancerade problem.

En artificiell intelligens (ai) kan idag lära sig att utföra olika aktiviteter, såsom att spela datorspel. Att låta en ai lära sig spela datorspel är ett billigt sätt att undersöka dess prestanda. En väl presterande ai kan sedan exempelvis användas för att utveckla självkörande bilar. Ett team från Berkley, USA utvecklade en ai vid namn Intrinsic Curiosity Module (ICM), som de kombinerade med en annan ai-algoritm för att bilda en algoritm som kunde spela tv-spelet Mario. Deras resultat visade att ICM:en kan användas för att förbättra inlärningsförmågan, genom att motivera ai:n till att testa olika handlingar så den utforskar spelets möjligheter mer, och hittar de bästa alternativen. Mario är ett simpelt spel i förhållande till moderna spel såsom Starcraft 2, exempelvis har Mario fyra möjliga handlingar medans Starcraft 2 har flera tusen. Starcraft 2 kan användas för att undersöka om ICM:en fortfarande kan förbättra inlärningsförmågan för spel med många handlingar, eller om dessa spel är för komplexa för att ICM:en ska lära sig och resulterar i att agenten spenderar all sin tid med att utforska spelet, utan att välja en optimal strategi. För att undersöka detta använde vi en existerande ai algoritm, Deep Q Network (DQN), och testade dess förmåga att lära sig spela Starcraft 2 med och utan ICM. Nivån i Starcraft 2 vi valde hade som mål att eliminera fiendesoldater,

En ai spelar Starcraft 2

och ju fler fiender man eliminerade, ju mer poäng fick man. Utöver detta fick agenten först mer direkt och sen mindre direkt feedback över hur den presterade. Ju mer direkt feedback agenten fick, ju enklare var det för den att avgöra vilka handlingar som gav högst poäng. Resulatet visar att tillförandet av en ICM förbättrar inlärningsförmågan, både dess inlärningshastighet och slutpoäng, oavsett vilken feedback agenten fick. Att Starcraft 2 var avsevärt mer komplext än Mario är inte ett hinder för ICM:en, vilket tyder på att ICM:en kan tillämpas på mer avancerade problem.