Universiteit Leiden

Master Computer Science

Deep Reinforcement Learning for Micro Battles in StarCraft 2

Name: Jelmer Prins
Student ID: 2773058

Date: 01/07/2023

Specialisation: Artificial Intelligence

1st supervisor: Mike Preuss
2nd supervisor: Marcello A. Gómez-Maureira

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

***Abstract*—StarCraft 2 (SC2) is a popular testing ground for Reinforcement Learning (RL). We introduce MicroStar, a Deep RL model designed for unit control (micro) in SC2 games. MicroStar is able to show strong results on all the micro minigames introduced with the SC2LE, while being trained solely on a consumer-grade PC. The MicroStar architecture was inspired by AlphaStar but is much more simplified and uses only a small fraction of the learning time. We improved on the performance by DeepMind's FullyConv on one of the three tested minigames whilst only using 7.5 days of in-game time where they used 673 years of in-game time for training.**

I. Introduction

Recent years have seen a surge of achievements by Reinforcement Learning (RL) which were previously thought to be too challenging such as the super-human performance in the game of Go [1]. With Deep Reinforcement Learning (DRL) seemingly having great potential for solving varied problems not only in games but in many fields such as robotics [2], finance [3], chemistry [4] and healthcare [5].

Games are often used for testing DRL network architectures and algorithms. We have seen more and more complex games being successfully tackled by DRL not just board games but also video games such as Atari games [6] and Dota 2 [7]. In 2017 DeepMind worked together with Blizzard to create the StarCraft 2 Learning Environment (SC2LE) [8] which allowed the game of StarCraft 2 (SC2) to be used as a RL environment. The SC2LE allows for training on the full game but also came with 7 minigames that each test different types of skills needed for SC2. There have been several papers published that tackle these 7 minigames with the most successfully being from DeepMind themselves [9].

There have been large successes in using DRL for solving the full game of SC2. Mainly by AlphaStar, a DRL implementation by DeepMind that achieved GrandMaster level play [10]. This however required a large team, lots of engineering and used a large amount of compute (50,400 CPU cores and 3072 TPU cores for 44 days [10]). Which makes it hard to replicate for smaller teams with less resources. Nevertheless some attempts have been made, mainly by SCC [11] and TStarBot-X [12]. But in the case of SCC it still required a network with 49m parameters, a large dataset of human replays and 30 years of in-game time per agent. Which is still impressive as that is about an order of magnitude less computational resources than AlphaStar used.

DeepMind's work in solving the minigames that came with the SC2LE use an exorbitant amount of computational resources [8], [9]. This is a big problem in the field of DRL as this need for large GPU clusters creates a high barrier of entry for any party that might be interested in deploying DRL for similarly complex problems. We seek to implement a neural network capable of achieving strong results whilst only being trained on a single consumer-grade PC. We achieve this by designing a network that leverages many modern network architectures and is specifically designed to control multiple units. Unit control is a sub-problem of SC2 often called micromanagement by SC2 players. Micromanagement or micro for short is a players ability to utilize its units efficiently in fights. We implemented a DRL model called MicroStar inspired by the architecture of AlphaStar [10] but much simpler and smaller. We benchmark MicroStar on some of the same problems DeepMind did and find competitive results at a fraction of the computational cost. The implementation was done in PyTorch [13] and is open-sourced and publicly accessible here: GitHub Link.

We will first give some background on RL and its application to SC in section II. We will then give some more details on the SC2LE and how SC2 is used as an RL environment in section III. Then we will introduce MicroStar and go into detail about the network architecture explaining how the different parts of the network function and interact in section IV. We will then discuss how MicroStar was trained in section V and the results in section VI.

II. Background/Related Work

A. Deep Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning where agents learn to interact with an environment. A RL agent influences the environment by selecting actions based on its (partial) observation of the environment state. After each action, the agent receives a reward that indicates its performance and the state transitions to a new state. Through multiple interactions, the agent aims to learn a policy that maximizes cumulative future rewards. This policy is a mapping from states to actions, and the ultimate objective of reinforcement learning is to find the optimal policy that maximizes the cumulative reward. Games have been widely used as a common testbed for RL techniques due to their well-defined environments, complexity, challenge, and built-in score systems that can serve as reward functions.

Deep Reinforcement Learning (DRL) employs deep neural networks as policy functions. Notably, DeepMind demonstrated the effectiveness of this approach by using Deep Q-Networks to achieve state-of-the-art performance on various Atari 2600 games [6]. The utilization of neural networks offers several advantages, such as the ability to handle high-dimensional inputs and reducing the need for manual feature engineering [14]. Neural networks also enable generalization to unseen states and the representation of nonlinear and complex policies. The optimization of neural network weights is typically accomplished through gradient descent, facilitating the learning process.

In recent years, DRL has achieved notable advancements, examples include AlphaGo and AlphaZero, which achieved superhuman-level play in Chess, Go, and Shogi [1], [15]. These games were previously considered computationally challenging due to their vast state spaces (e.g., 10^{170} for Go). Beyond gaming domains, DRL has made significant contributions to algorithm optimization through both AlphaTensor and AlphaDev [16], [17]. There have also been advancement made in robotics [2] and autonomous driving [18]. These achievements illustrate the progress made by DRL in addressing complex problem-solving tasks across diverse domains.

B. StarCraft

There has been a rich history of using both StarCraft 2 (SC2) and its predecessor StarCraft (SC) as a testing grounds for AI [19]. The games fall in the Real-Time Strategy (RTS) genre. Some characteristics of this genre that make it challenging for RL is the need for long-term planning, large action and state spaces, sparse rewards, incomplete information and the need for time-sensitive reactions. The goal of the game is to destroy all your opponents buildings, this is generally done by building up an economy in the early game and then gaining an advantage over your opponent by attacking in favorable positions until you can fully overpower them. This is however not as easy as it sounds as there are many complex strategies that involve tricking your opponent or punishing certain play-styles.

Different aspects of the game have often been separated into different sub-problems with 2 major ones being micro-management (micro) and macromanagement (macro). The first sub-problem, micro, is all about unit control when fighting enemy units. This requires mostly short-term planning and fast response times. Macro on the other hand is the act of building up infrastructure, which means efficiently balancing the economy and unit production thus playing the long-term strategy. These problems presented in SC/SC2 have been tackled by many different fields from ML namely Search-Based Algorithms, Supervised Learning, Evolutionary Computations and Reinforcement Learning and sometimes combinations of these. What we will be focusing on in this paper is micromanagement using Reinforcement Learning which can be seen in purple in figure 1.

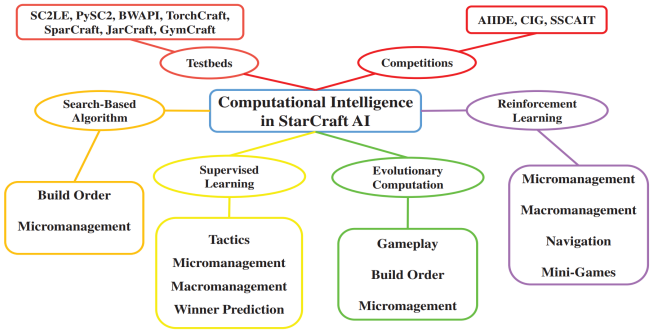


Fig. 1: Overview of Computational Intelligence in SC [19].

III. Environment

In this study, we utilize parts of the SC2LE as our reinforcement learning environment. SC2LE is a collaborative effort between DeepMind and Blizzard, introduced in 2017 [8]. To interact with SC2 and effectively use it as a RL environment, we make use of sc2client-proto [20], a SC2 API provided as part of the SC2LE.

For the ability to give unit commands in the way we want we employ a Python API wrapper by BurnySc2 [21] instead of the PySC2 wrapper provided by DeepMind. This community created API wrapper has the added benefit of more easily being able to adapt community-created scripted bots. Although we ultimately did not utilize the scripted bots feature, this wrapper still facilitates the possibility of training against these community sourced bots in the future. We also build out some of our own functionality for the environment in order to effectively collect all the data needed for training.

A. Minigames

There are 7 special minigame maps provided by Blizzard which allow us to train our RL agents in a very standardized way that makes it easy to compare to previous results. These 7 minigames cover a range of scenarios both related to micro and macro and provide their own reward signal. We chose to focus on three of these minigames *Find And Defeat Zerglings*, *Defeat Roaches* and *Defeat Banelings and Zerglings* since these are purely related to micro.

1) Find And Defeat Zerglings: *Find And Defeat Zerglings* gives the agent control over 3 marines which are placed in the center of a flat game map, all over the map there are spread out zerglings placed in the fog of war. A zergling is a small melee unit and is generally weaker than a single marine. An in-game view of this scenario can be seen in Figure 2. A reward of 1 is given for every destroyed zergling and a reward of -1 is given for every lost marine, there is also a time limit of 3 minutes. A general strategy for this would be to keep all 3 marines together while they explore the map looking for zerglings. If all 3 marines focus on the same zergling they can take them out without taking (much) damage. Due to the time pressure it is also important to explore the map in an efficient way without back-tracking over an already explored area.

Fig. 2: In-game view of *Find And Defeat Zerglings* minigame. 3 Marines can be seen in the center of the screen and some zerglings at the edge of their vision range.

2) Defeat Roaches: The *Defeat Roaches* minigame has the agent fighting 4 roaches using 9 marines, and overview how this looks in-game can be seen in Figure 3. A roach is a slightly more advanced and stronger ranged unit compared to a marine. A reward of 10 is given for each destroyed roach and a reward of -1 for each lost marine. In the case that all 4 roaches are destroyed the agent is given another 5 marines and a new 4 roaches are spawned. The only way for 9 marines to win

against 4 roaches is if they all attack together and focus fire on a single roach at a time to quickly reduce their numbers. This roach minigame however has no fog-of-war which makes it simpler in some regards and really puts the emphasis on effective focus fire.

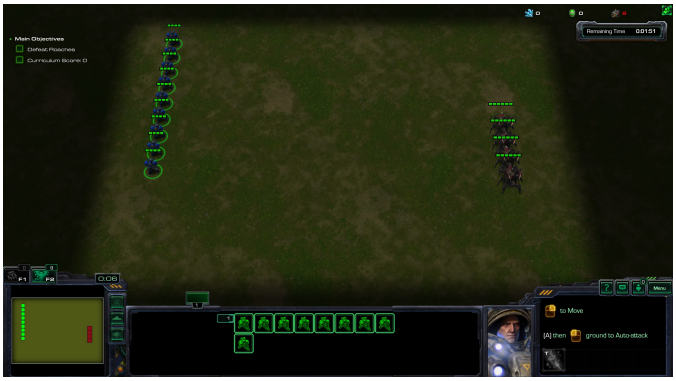


Fig. 3: In-game view of *Defeat Roaches* minigame. The 9 marines can be seen on the left and the 4 roaches on the right. It can also be seen that there is full vision of the battlefield with no fog-of-war.

3) Defeat Banelings and Zerglings: The *Defeat Banelings and Zerglings* minigame is similar to the *Defeat Roaches* minigame but now the opponent has a mix of zerglings and banelings. An in-game view of this minigame can be seen in Figure 4. Banelings are a unit that self-destruct upon contact with the enemy and deal a large amount of damage around themselves, this can have catastrophic consequences for the marines if they are standing too close together. A reward of +5 is given for each zergling and baneling that gets destroyed.

Fig. 4: In-game view of *Defeat Banelings and Zerglings* minigame. The marines are standing in a line on the right whilst the zerglings and banelings are clustered together on the left of the screen.

4) standard maps: We are also able to play on standard game maps which allows us to train with and against any unit composition we desire. This also enables play against different types of scripted bots instead of only Blizzard created ones. It also requires us to define our own reward functions which can be both a blessing and a curse as it can lead to some loss in generality and makes it harder to compare results with previous solutions. In order to speed up training time we also made it possible to do many RL episodes back-to-back within a single game, so without the need to reload the map for every episode (which has to be done for the minigame maps).

B. Observations

Instead of relying on rendered screen pixels, sc2client-proto provides raw data from the SC2 game engine whilst still respecting the fog-of-war. All available observations can be found on the sc2client-proto GitHub [20]. For our implementation, we selectively utilize specific portions of the data which we believed to be relevant for micro in order to limit the network size.

1) Spatial Data: Spatial data is divided into screen features and minimap features, each containing various categorical and scalar features. Screen features provide a detailed view of the camera's focus area, including information about unit types and their health points. Minimap features, on the other hand, offer an overview of the entire map with basic data such as the presence of friendly or enemy units and visibility range, but without specific unit types or HP.

In our implementation, we chose to focus on the minimap features and exclude the screen features, as utilizing the screen features would require the network to control the camera movement. We employed spatial observations at a resolution of 64x64, which, according to Wang [11], achieves similar performance as a 128x128 resolution but with lower computational cost. The minimap feature layers we utilized are *player_relative*, *visibility_map*, *height_map* and *pathable*. The *player_relative* feature layer is presented as 5-dimensional categorical data, indicating the presence of enemy/ally/self units. Since there are no neutral or allied players in our scenarios, we reduced this to 2-dimensional data by distinguishing only between own and enemy units. An example of what some of these feature layers look like can be seen in Figure 5.

2) Scalar Data: In addition to spatial data, a significant amount of scalar data is available, which can be further categorized into score, player, and unit data. The score data includes information about the total score accumulated throughout the game and can be used as a reward signal. Player data provides fundamental information such as the player's current minerals, gas, population, and population cap. Lastly, unit data contains comprehensive information about every known unit, including both units and buildings.

In our implementation we only use unit data. Player data can be disregarded entirely, as its information is relevant only for macro or the full game and not for micro. Score data is only useful as a reward signal, so we do use it, but only when playing minigames and only as a reward signal, not as part of the observation. However, unit data is highly relevant to us, as it encompasses all the information about each unit in our vision. Since we do not use the screen features, this is the only way to acquire details such as unit HP, unit type, and specific unit locations. Our network relies on unit type, unit location,

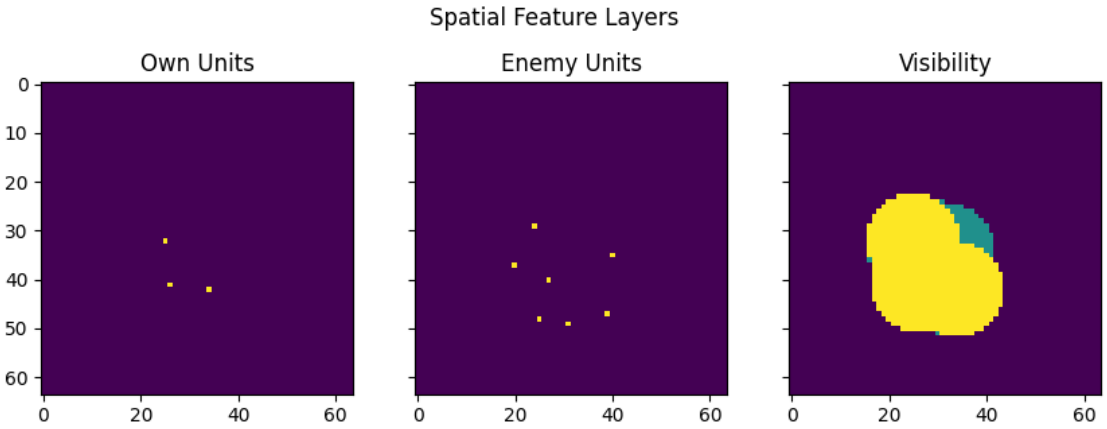


Fig. 5: Three of the five spatial feature layers just a few seconds after the start of the *Find And Defeat Zerglings* minigame. We can see our own three marines in the *Own Units* feature layer (three dots in the center). The enemy zerglings that have been discovered so far are visible in the *Enemy Units* layer. On the *Visibility* layer we can see which part of the map we currently have vision on in yellow and an area we had vision on before in blue.

health, shield, energy, and weapon cooldown information from the unit data.

C. Actions

The action space in SC2 is massive with many of the actions requiring multiple arguments and being depended on what unit is currently selected. The BurnySC2 API Wrapper however allows for unit control without the need to deal with selecting units, it also allows multiple different action to be taken at every time step. This allows our agents to pick a separate action for each unit on every step. The main actions that we need for unit control are Move, Attack Move, Hold Position and Attack Target, see table I for descriptions of these actions. Our agents are thus for example able to move unit 1 to location A whilst simultaneously ordering unit 2 to hold its position and unit 3 to attack an enemy unit. Something our units are however not able to do is use special unit-specific active abilities. For that reason, all the units we used in our experiments do not posses such special active abilities.

Our agent acts every 16 game loops, the game is played at 22.5 game loops per second so this equates to an action about every 0.7 seconds of in-game time. We choose this rate to be at a good balance between being able to react to changes in the game state whilst still making actions meaningful and not making the rewards too sparse. This gives the agent a maximum of 84 actions per minute (APM) per unit being controlled.

Action name	**Explanation**	**Extra Arguments**
Move	Unit moves to location, ignores all enemies.	Position 2D
Attack Move	Unit moves towards location, stops and attacks enemies if they enter attack range.	Position 2D
Hold Position	Unit stops moving and defends position without moving from location.	None
Attack Target	Unit moves towards target and attacks it once it is within range, ignores all other enemies.	Unit
No Action	No orders issued to unit, keeps working on previous orders or idles in case of no active previous orders.	None

TABLE I: An overview of all the possible actions a unit can execute using our network and that are needed for basic control.

IV. Network Architecture

Our neural network architecture, which we named MicroStar, was inspired by AlphaStar [22] but simplified in many ways and only keeping parts that we believed to be relevant for unit micro, a direct comparison between the network architectures can be seen in Appendix A. MicroStar has 2.3 million parameters where as AlphaStar has about 139 million parameters. Even with this reduced format the complex action space of SC2 necessitates at least three different outputs in order to simultaneously select Actions, Positions and Enemy Units, see table I. This in combination with both spacial and scalar observations makes our overall network architecture quite complicated with many different parts. We will now do our best to describe the workings and purpose of each of these parts. Starting from the inputs and working our way up through the network to the outputs. An overall overview can be seen in Figure 6 and the number of parameters per part of the network can be seen in Table III.

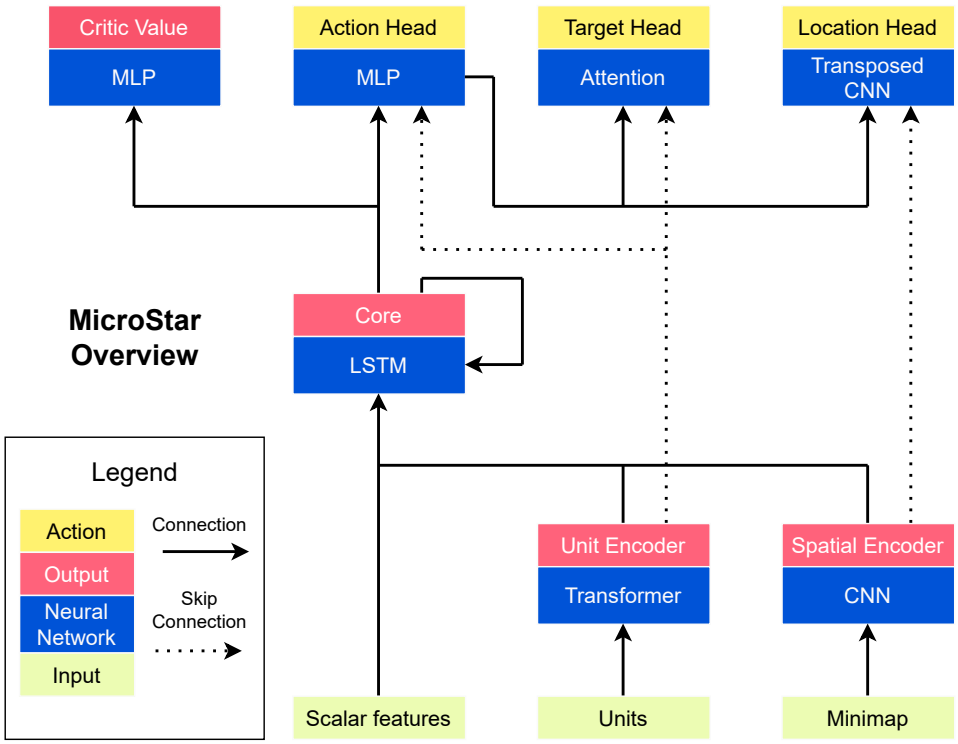


Fig. 6: Overview of MicroStar Network Architecture.

A. Terminology

In order to effectively explain the full architecture we will first explain some of the terminology we will be using. MLP stands for Multilayer perceptron which consist of multiple linear layers. CNN, short for Convolutional Neural Network, is a specialized type of neural network proficient in computer vision and other spatial tasks [23]. By employing convolution operations, CNNs extract localized features from input data, facilitating the detection and understanding of spatial patterns. Moreover, CNNs possess the ability to effectively alter the resolution of the input data through downsampling or upsampling techniques. LSTM stands for long short-term memory networks and utilizes recurrent connections to model and process sequential data. Unlike traditional neural networks, LSTMs are specifically designed to handle temporal dependencies and learn long-term patterns [24]. The Transformer model specializes in sequence-to-sequence tasks and rely on self-attention mechanisms to capture contextual relationships between items in a sequence.

B. Settings

The network was designed to use settings to easily variate the exact sizes of different parts of the network. An overview of these settings can be seen in Table II. The *max_entities* setting which dictates the maximum number of unit data that is able to flow through the network, so any units past this number will not be controlled by MicroStar. This setting does not actually change any part of the architecture so it could be changed dynamically but we chose to keep it constant during runs to prevent jagged tensors when using batches and instead use padding and masking where needed. Many of the settings in Table II were chosen arbitrarily using some intuition and limited to be smaller or equal to the AlphaStar architecture, we are not claiming these to be optimal and much more testing and optimization can likely be done on these.

C. Unit Encoder

1) Inputs: Every unit is encoded as 54-dimensional tensor which contains a one-hot encoding of the *unit_type*, a binary encoding of the position, ratios for health, shields and energy, weapon cooldown and an one-hot encoding of combined health + shields. The amount of possible *unit_types* has been greatly reduced to mostly just basic units but this can easily be expanded upon if needed. The input to the unit encoder ends up being two *max_unit* x 54 tensors, one for enemy and one for friendly units.

2) Internals: These two tensors are concatenated and fed through a Transformer Encoder [25] which is made up of a self-attention and feedforward network, we use an implementation by PyTorch [13]. The resulting tensor is then split again into friendly and enemy units. We call these tensors *own_unit_embeddings* and *enemy_unit_embeddings*.

Name	Default Value
unit embedding size	128
unit transformer feedforward size	128
unit transformer nheads	2
unit transformer layers	2
spatial embedding size	256
map skip channels	16
core layers	1
core output size	256
autoregressive embedding channels	4
target head attention size	64
dropout	0
max units	16
location action space resolution x	62
location action space resolution y	62

TABLE II: An overview of all the MicroStar network settings and their default values.

Name	Parameters
Total	2.343.047
Unit Encoder	206.464
Spatial Encoder	844.464
Core	792.576
Action Head	305.765
Location Head	28.273
Target Head	66.688
Critic	98.817

TABLE III: An overview of the number or parameters (weights) of each part of the MicroStar model.

3) Outputs: The Unit Encoder has 4 outputs, firstly *own_unit_embeddings* and *enemy_unit_embeddings* both being *max_entities* x *unit_embedding_size* tensors. And then 2 tensors called *own_embedded_unit* and *enemy_embedded_unit* which are created by taking the mean over their respective *unit_embeddings* tensor, resulting in two 1 x *unit_embedding_size* tensors.

D. Spatial Encoder

1) Inputs: We pull our spatial observations from SC2 at a resolution of 64x64 pixels. We use 5 feature layers in total, those being own units, enemy units, height, pathable, and visibility. So our input to this part of the network ends up being an 5x64x64 tensor.

2) Internals: The input tensor is projected to 64 channels by a 2D convolution using a kernel of size 1. We then have 2 convolutions layers that downsample by doing 2D convolutions with kernel size 4 and stride 2. These convolutions also lower the number of channels down to *map_skip_channels* and this tensor is called *map_skip*. It is then flattened and fed through a linear layer. This final tensor is called *embedded_spatial*.

3) Outputs: The outputs are *map_skip* and *embedded_spatial*. The map skip output is meant to preserve some spatial information and is shaped as *map_skip_channels* x 14 x 14. The *embedded_spatial* tensor has a shape of 1 x *spatial_embedding_size*

E. Core

1) Inputs: The input to the Core is a concatenation of *embedded_unit*, *embedded_unit_enemy*, *embedded_spatial* and *scalar_features*. The first 3 of these come from the encoder parts of the network and the *scalar_features* come directly from the observation space and consist of two integers indicating the amount of friendly and the amount of enemy units. The size of this concatenation of tensors is 1 x (2 · *unit_embedding_size* + *spatial_embedding_size* + 2). The internal state of the LSTM is also provided when possible.

2) Internals: The Core is implemented as a LSTM with a single layer and a hidden size of *core_output_size*. An implementation from PyTorch was used for the LSTM. There are also optional Dropout layers implemented before and after the LSTM.

3) Outputs: The output is given by a single 1 x *core_output_size* tensor. We also save the internal state of the LSTM to re-use for the next environment step.

F. Action Head

1) Inputs: The Action Head receives both the *core_output* and *own_unit_embeddings*. The *core_output* is repeated *max_entities* times to match the shape of *own_unit_embeddings* and these tensors are then concatenated into a single *max_entities* x (*core_output_size* + *unit_embedding_size*) tensor.

2) Internals: The Action Head is a MLP which maps from the input size to a tensor with a length equal to the number of available actions, which is our case is 5 as can be seen in Table I. This tensor is then put through a softmax function to produce *action_logits*. The actions not belonging to any unit are masked out.

3) Outputs: The outputs of the Action Head are *action_logits* and *autoregressive_embedding*. *action_logits* contain a weighted chance for each of the units to execute each of the 5 actions, the tensor has the shape *max_entities* x 5. *autoregressive_embedding* refers to the tensor that is produced just before the last linear layer.

G. Location Head

1) Inputs: The inputs for the Location Head are *autoregressive_embedding* and *map_skip*. The former contains information coming from the Core/Action Head and the latter contains spatial information from the Spatial Encoder at a resolution of 14x14.

2) Internals: First the *autoregressive_embedding* is reshaped from flat to have the same spatial shape as *map_skip*. Then *map_skip* is repeated *max_entities* times and these two tensors are concatenated. This resulting tensor is projected to 64 channels by a 2D convolution using a kernel of size 1. We then have 1 convolutional layer with kernel size 3 followed by two transposed convolutional layers with kernel size 4 and stride 2 to up-sample to 62x62. This tensor is then flattened and has a softmax applied to it to get our *location_logits*.

3) Outputs: The *location_logits* contain the probability for every location to be chosen. It is shaped as *max_entities* x 62 x 62, which after flattening is *max_entities* x 3844. Note that this output is only used when the corresponding unit has picked either the *Move* or the *Attack Move* action.

H. Target Head

1) Inputs: The inputs to the Target Head are *autoregressive_embedding*, *own_unit_embeddings* and *enemy_unit_embeddings*.

2) Internals: The Target Head uses an attention mechanism to calculate the attention from each friendly unit with respect to every enemy unit. This is done using Scaled Dot-Product Attention [25] without multiplying with the Values.

$$Attention(Q, K) = softmax(\frac{QK^T}{\sqrt{d_k}})$$

Here Q are the queries and K the keys. d_k is the dimension of the keys and queries. The queries are calculated by first concatenating *autoregressive_embedding* and *own_unit_embeddings* and then applying a linear layer. The keys are calculated from putting *enemy_unit_embeddings* through another linear layer. Both the linear layers have *target_head_attention_size* dimensions. After transposing the keys and doing the matrix multiplication we get a *max_entities* x *max_entities* tensor that maps the attention from each friendly unit to each enemy unit. We then apply a mask to prevent targeting non-existent padding units and finally apply a softmax to receive our *target_logits*

3) Outputs: The only output is *target_logits* which is a *max_entities* x *max_entities* tensor mapping from every friendly unit to every enemy unit. This tensor can then be sampled to choose which enemy unit will be attacked. Note that this output is only used when the *Attack* action was chosen by the Action Head.

I. Critic

1) Inputs: The critic only uses the Core output as input. This is a single 1 x *core_output_size* tensor.

2) Internals: The Critic is a MLP which uses 2 linear layers to map from its input to a single scalar value. The critic essentially tries to predict the future rewards and is used in training.

3) Outputs: The output is a single scalar value.

V. Training

A. Data collection

In order to train a neural network, data is essential. In RL this data comes from the agent interacting with the environment through multiple episodes. An episode refers to a complete sequence of interactions starting from an initial state and continuing until a terminal state. In order to efficiently collect data we launch multiple games of SC2 at the same time. Each of these games is controlled by a *Data Collector* whose purpose it is to save the observations, network outputs and rewards. After every completed episode the *Data Collectors* send their data to the *Learning Manager* and retrieve the latest version of the neural network. The *Data Collectors* also deal with the logic of initializing and ending episodes and on which game steps to run the agents.

The *Learning Manager* has a replay buffer which stores the agent's experiences, collected during interactions with the environment. The replay buffer has a maximum size, and when new data needs to be added the oldest data in the replay buffer is removed.

B. Network Updates

Every few seconds the *Learning Manager* trains the network for 3 epochs. It randomly samples batches from the replay buffer to use for the network updates. Sampling random batches is important for breaking temporal correlations. In RL environments, consecutive experiences tend to be highly correlated. If we train the agent directly on sequential data, it can lead to unstable learning by constantly over-fitting to different parts of the problem.

For updating the network we use PPO [26] which is a type of actor critic method which means the agent is learning both a policy π (actor) and an value function (critic). The critic is essentially giving feedback to the actor by estimating the expected reward. We can then calculate the advantage function, $\hat{A}_t$, which estimates how advantageous an action is compared to the average action in a given state. It is calculated as the difference between the critic value and the actual received reward. PPO is a type of policy gradient method which utilizes a Clipped Surrogate Objective to the actor policy in order to constrain the policy update to a specified range, preventing drastic policy changes that can lead to unstable training. The clipped loss, denoted as $L^{CLIP}(\theta)$, is defined as follows:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

In this equation, θ represents the parameters of the policy π, $\pi_\theta(a_t|s_t)$ denotes the probability of selecting action a_t in state s_t under the policy π_θ, and $\pi_{\theta_{old}}(a_t|s_t)$ represents the corresponding probability under the previous policy. The term $r_t(\theta)$ is the probability ratio, which measures the ratio of probabilities between the current and previous policies:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

Name	Default value
learning rate start	0.0001
learning rate decay	0.999
minimum learning rate	0.00005
time between optimizations	10
gamma	0.99
ϵ start	0.2
ϵ decay	0.999
ϵ min	0.05
replay buffer size	2048
batch size	256
epochs	3

TABLE IV: An overview of the training hyperparameters.

The purpose of clipping $r_t(\theta)$ in the surrogate objective is to limit the policy update to a specified range. By constraining the policy update, PPO prevents overly large policy changes that can destabilize the training process. The hyperparameter ϵ determines the extent of the clipping. The above clipped loss gives us the loss for the actor but our total loss also consists of the critic loss, since they share a part of the network, and an entropy loss.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]$$

Here c_1 and c_2 are coefficients, S is the entropy loss and L_t^{VF} is the critic loss given by a squared-error loss $(V_\theta(s_t) - V_t^{targ})^2$ where V_θ stands for the value network which we mostly refer to as the critic. We then optimize the total loss using ADAM [27], iteratively updating the parameters θ to improve its performance over time.

C. Hardware

All training was done on a single consumer-grade PC with an i5-12400 CPU and a RTX 3070 ti GPU (8GB VRAM). During training we were mostly bottlenecked by the CPU which is used to run SC2. We used anywhere between 4 to 8 instances of SC2 simultaneously for sampling the environment. The 8GB of VRAM also limits the maximum size of the network and the batch size during training.

VI. Results

We trained MicroStar on several different micro scenarios. First of all we used three minigames provided by Blizzard; *Find and Defeat Zerglings*, *Defeat Roaches* and *Defeat Zerglings and Banelings*. This provides us with 3 scenario's that have been used by many other research teams thus allowing us to make good comparisons. We then also trained MicroStar on a scenario similar to those used by Meta AI Research [28] for SC1, namely the 5 marines vs 5 marines scenario. All data was collected using the architecture settings from II and learning settings from IV. For validation after training we ran the final agent for 1000 episodes.

A. Find and Defeat Zerglings

The average rolling reward during training for the *Find and Defeat Zergling* minigame can be seen in Figure 8. The total training time was 20.000 episodes which equates to about 650 hours of in-game time and 14 hours real life time.

The general found strategy by MicroStar was to keep its three marines close while they circle around the center of the map in a clock-wise motion, a simplified view of this can be seen in Figure 7. When a marine gets about half-damaged it keeps it standing still to avoid losing it. When it loses a marine or all marines are damaged it keeps all marines standing still to wait out the 3 minute timer. This ends up making each episode take the full 3 minutes with the most interesting part happening at the first minute. This made training a bit slower than the other minigames and inflated to total in-game time needed for training. A video showcasing a full episode can be viewed here and for the untrained version of MicroStar here.

Fig. 7: Simplified overview of the strategy used by MicroStar in *Find and Defeat Zergling*. The marines move in a clock-wise movement around the map until they lose too much HP and MicroStar keeps them stationary so they don't get killed.

B. Defeat Roaches

In the *Defeat Roaches* minigame the agent controls 9 marines and is tasked with defeating a group of 4 roaches. Efficient target fire where all marines attack the same roach is very important for this minigame. This minigame was explained in more detail in section III-A2. The average rolling reward during training can be seen in Figure 10. The total training time was 15.000 episodes which equates to about 160 hours of in-game time and 6 hours real life time.

The found strategy involves immediately running up on the roaches and target firing them down one at a time, see Figure 9 for a simple overview. This is a good basic strategy but it is lacking some finer control like moving marines that are about to die away from the roaches and then back in once they are not being targeted anymore.

There is also some chance involved in how the roaches target fire, sometimes its possible to lose less than 5 marines

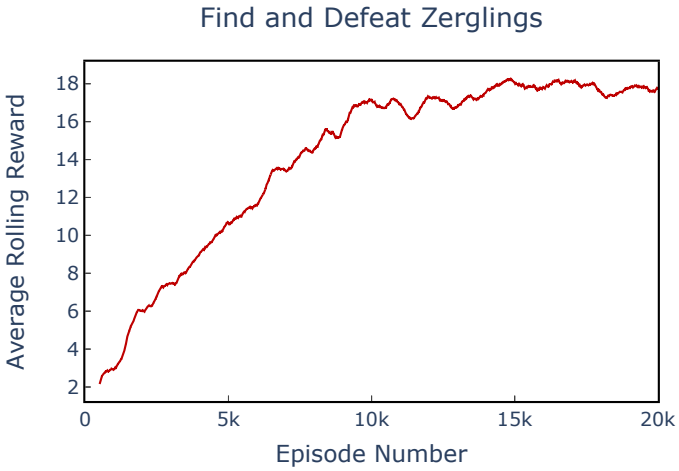


Fig. 8: Rewards during training, rolling average with window size of 500.

	Average Reward	**STD (σ)**	**Max Reward**
MicroStar (Ours)	16	2.6	22
FullyConv [29]	8	2.4	14
FullyConv + 3D Conv [29]	22	3.8	40
A3C [30]	∼6	-	16
FullyConv (DeepMind) [8]	45	-	56
FullyConv LSTM (DeepMind) [8]	44	-	57
Atari-Net (DeepMind) [8]	49	-	59
Relational DRL (DeepMind) [9]	**62**	-	-
GrandMaster Player (human) [8]	61	-	61

TABLE V: Rewards for the *Find and Defeat Zerglings* minigame achieved by us and others.

in a round of the minigame which means that when MicroStar receive 5 more marines for the next round they have made a net profit on marines. This can sometimes allow the agent to snowball its advantage over the roaches and reach very high scores like 300+. An example of the trained behavior can be found here and for the untrained version here.

C. Defeat Zerglings and Banelings

In the *Defeat Zerglings and Banelings* minigame the agent uses a group of marines to fight zerglings and banelings.

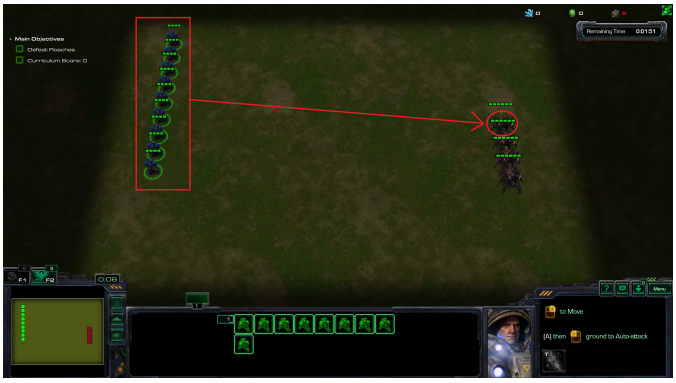


Fig. 9: Simplified overview of the strategy used by MicroStar in *Defeat Roaches*. The marines all focus the same enemy roach.

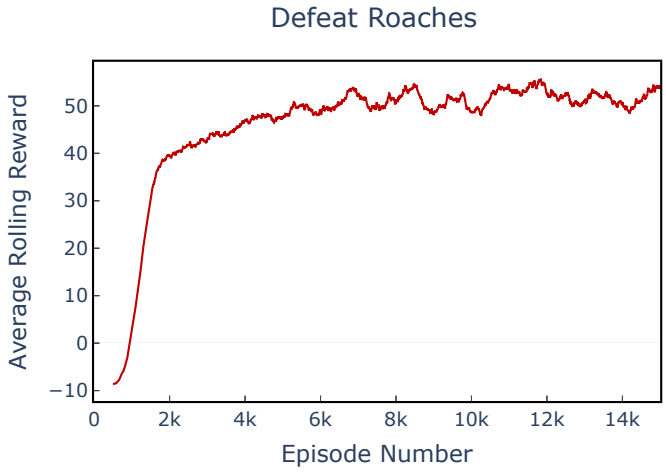


Fig. 10: Rewards during training, rolling average with window size of 500.

banelings explode themselves when they contact enemies dealing area damage around them, this makes it vital to spread out your units or target down the banelings before they reach you. This minigame was explained in more detail in section III-A3. The average rolling reward during training can be seen in Figure 12 and a comparison to other work can be seen in Table VII. The total training time was 15.000 episodes which equates to about 185 hours of in-game time and 7 hours real life time.

The strategy employed by MicroStar involves utilizing the Attack Move command to spread its units. This approach allows some marines to act as sacrificial units, tanking banelings while ensuring the survival of other marines to deal with the zerglings. Figure 11 shows a simplified overview of the strategy. While sacrificing units may sound risky, due to the behavior of the zerglings it almost always works out, making it a solid and effective strategy. Keeping the marines clumped together and target-firing down the banelings whilst moving

	Average Reward	STD (σ)	Max Reward
MicroStar (Ours)	53	23	252
FullyConv [29]	23	19	121
A3C [30]	∼20	-	42
FullyConv (DeepMind) [8]	100	-	355
FullyConv LSTM (DeepMind) [8]	98	-	373
Atari-Net (DeepMind) [8]	101	-	351
Relational DRL (DeepMind) [9]	**303**	-	-
GrandMaster Player (human) [8]	215	-	363

TABLE VI: Rewards for the *Defeat Roaches* minigame achieved by us and others.

backwards might allow for a higher max score but it is also very risky and hard to learn.

Fig. 11: Simplified overview of the strategy used by MicroStar in *Defeat Zerglings and Banelings*. The Marines all move around spreading themselves out and fire at the enemies when they get close.

Additionally, during training, we noticed that MicroStar occasionally learned to keep low-health units stationary at the rear. This strategy makes some sense since a full hp Marine can just barely survive a baneling explosion, enabling them to eliminate up to 2 banelings before succumbing. However, it comes with some risks as poorly spread-out low-health marines are vulnerable if a baneling does reach the back row. In the final found policy, this behavior was not retained. A video of this final behavior can be found here, towards the end of this example video (1:01) MicroStar fails to spread out its marines properly and it can be seen how devastating this can be as it loses 5 marines very quickly which leads to the complete defeat one round later. The untrained version can be found here, at first it seems sort of similar to the trained version as MicroStar's natural tendency is to spread its units but it still fails to spread out enough and quickly loses.

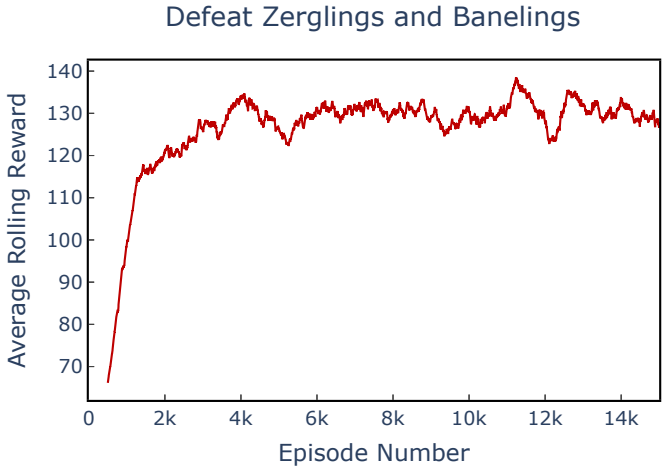


Fig. 12: Rewards during training, rolling average with window size of 500.

	Average Reward	STD (σ)	Max Reward
MicroStar (Ours)	131	65	418
A3C PlusFC [30]	∼70	-	130
A3C Self-Attention [31]	136	-	-
FullyConv (DeepMind) [8]	62	-	251
FullyConv LSTM (DeepMind) [8]	96	-	444
Atari-Net (DeepMind) [8]	81	-	352
Relational DRL (DeepMind) [9]	**736**	-	-
GrandMaster Player (human) [8]	727	-	848

TABLE VII: Rewards for the *Defeat Zerglings and Banelings* minigame achieved by us and others.

D. 5 Marines vs 5 Marines

For this scenario we implemented ourselves we battled 5 marines against 5 other marines. The enemy marines are controlled by a scripted bot we implemented called AMoveBot. AMoveBot simply Attack Moves (A-Moves) all its units

toward the enemy units. This is a basic but effective strategy often used by SC2 players. This scenario gives full vision of the map so there is no need to deal with the fog-of-war.

We created the reward function ourselves and it gives +2 for every enemy unit killed, -2 for every unit lost, a small amount of reward for every point of damage dealt and a bonus reward of +2 if the episode is won by destroying all enemy units.

Something we found during testing is that the size of the network heavily impacts the performance of the agent so we will be comparing between four different network settings of MicroStar (base, A, B and C), the exact changes can be seen in Table VIII. Each agent was trained for 5000 episodes which takes about 16 hours of in-game time and 40 minutes real life time. The win rates during training can be found in Figure 13 an the final evaluations of the networks were done over 1000 episodes and can be seen at the bottom of Table VIII.

The strategy found by MicroStar was very similar to the *Defeat Roaches* minigame, it tries to target-fire down one enemy unit at a time. A big difference however is that this fight happens much faster as the total health pool is smaller thus a single wrong action can have larger consequences.

We found a large difference in performance between the networks Base, A, B and C. Especially B and C performed much better which seems to be due to a larger Unit Encoder IV-C, which uses 16 headed attention instead of the 2 headed attention used by the base MicroStar. This seems to help with target-firing as it seemed units were more likely to pick the same target.

VII. Discussion

A. Training Time

We find our MicroStar model to place somewhere in between FullyConv by DeepMind and some of the other published SC2-Micro papers. Only on *Defeat Zerglings and Banelings* do we find we can improve on the performance from FullyConv. It should however be noted that DeepMind used much more computational power for their FullyConv model; each of their agents trained for 8x600m = 4.8b game steps which is about 59000 hours of in-game time (2460 days). They then did this 100 times for each agent whilst varying hyper-parameters to find their optimal performing agent, this equates to about 673 years of SC2 per final agent. This is significantly more than 650 hours (27 days) of SC2 for our longest run.

The Relational DRL [9] from DeepMind crush all other competitors and achieve results comparable to or better than a GrandMaster human player. DeepMind is not exactly clear on their training times but they mention 10 billion optimization steps per agent with a batch size of 32. If we assume a similar action rate as FullyConv this comes out to be about 3600 years of SC2 per agent and then they trained 100 agents per minigame. Although we can not be sure of this number it is clear that is exceeds the computational time from their FullyConv agents. Their mean scores for the minigames were also only taken over 30 episodes which is about 1 hour of SC2 time which seems really quite low compared to their training time.

This imbalance in training time does of course go both ways as we did also use about 3x more training time than the implementation of FullyConv from Dumitrescu [29]. Wheras the A3C implementation from Alghanem [30] seemed to use about equal or more computational time to us.

B. Defeat Roaches

A problem we faced on the *Defeat Roaches* minigame which we did not yet discuss is that in some runs our agent would get stuck in a local minima. This would happen when the agent found the strategy of just standing still with all its units to avoid the roaches and wait out the timer. This can lead to a reward of 0 which is higher than the reward of -9 that you get when losing all your marines without killing a single roach. When training just starts this result of -9 is very common and killing any roaches at this stage requires some luck. So if this luck doesn't happen it will simply learn to just save all its units by keeping them away from the roaches. This would happen for about half the runs we tried during testing but could quickly be identified by looking at the rewards during the first 15 minutes of a run.

C. PPO

PPO is an on-policy learning algorithm which means the policy that interacts with the environment should be the same policy as the policy that is optimized. In our implementation we actually go slightly off-policy and re-use some trajectories that were sampled with an old policy. This going slightly off-policy did not seem to hurt performance. We speculate this is due to the relatively small replay buffer and since we take small steps in the policy space the old policy is still close enough the current policy that we are almost on-policy. We did also experiment with clearing the replay buffer after each optimization step and going fully on-policy but this did not improve performance. It actually increased the total learning time needed since we have to wait longer for more new trajectories to be sampled and we were already being bottlenecked by the environment sampling.

D. Environment

We used the BurnySC2 API wrapper [21] in order to easily allow us to give multiple orders on the same game loop. We later learned there should also be a way to do this with PySC2 created by DeepMind. Since we also did not end up using any community created bots for BurnySC2 using PySC2 would likely have run faster, although we did not confirm this.

E. Environment

We used the BurnySC2 API wrapper [21] in order to easily allow us to give multiple orders on the same game loop. We later learned there should also be a way to do this with PySC2 created by DeepMind. Since we also did not end up using any community created bots for BurnySC2 using PySC2 would likely have run faster, although we did not confirm this.

Name	Base MicroStar	MicroStar-A	MicroStar-B	MicroStar-C
total parameters	2.343.047	4.255.495	2.803.591	4.109.575
unit embedding size	128	**256**	128	**256**
unit transformer feedforward size	128	128	**1024**	**1024**
unit transformer nheads	2	2	**16**	**16**
unit transformer layers	2	2	2	2
spatial embedding size	256	**512**	256	256
map skip channels	16	16	16	16
core layers	1	1	1	1
core output size	256	256	256	256
autoregressive embedding channels	4	4	4	4
target head attention size	64	64	64	64
win rate	0.29	0.33	0.75	**0.86**

TABLE VIII: An overview of different MicroStar architectures tested on our 5 marines vs 5 marines scenario.

VIII. Future Work

For future experiments it would be interesting to see how much we can improve the MicroStar model by doing a hyper-parameter search over the different network architecture settings we mention in Table II. We already saw large swings in performance from changing the Unit Encoder size in the 5 marines vs 5 marines experiment VI-D, it is possible similar performance gains could be made on the minigames.

Something the AlphaStar architecture did that might also be of interest to us is giving the Critic network access to hidden information not visible to the Actor. This becomes particularly relevant in scenarios involving fog-of-war, such as the *Find and Defeat Zerglings* minigame. This can also contain information about the current cumulative score which might help the critic in predicting future scores.

Another thing we would like to experiment with, is what DeepMind called their *scattered entities* layer. This was used in AlphaStar's Spatial Encoder as an extra feature layer together with the minimap features. It was created by taking their *unit_embeddings* and placing them in their appropriate positions on a spatial feature layer. This gives the Spatial Encoder access to a lot of specific data about each unit which might lead to stronger spatial reasoning which seemed to be one of MicroStar's weakest areas judging from the *Find and Defeat Zerglings* results. This was currently not done as it will add quite a bit of GPU memory usage but we could experiment with first embedding the *unit_embeddings* down to a smaller number of dimensions.

Exploring additional experiments beyond the Blizzard-provided minigames is also of interest. For instance, testing with more diverse armies from all races, instead of predominantly relying on marines, could provide valuable insights. It is also possible to change the network to also allow units to use their active abilities. We also have a scenario already setup that allows for a capture the flag style minigame where an agent wins by destroying an enemy flag structure. These experiments would require the network to make more tactical decisions, moving beyond pure micro. The training could be

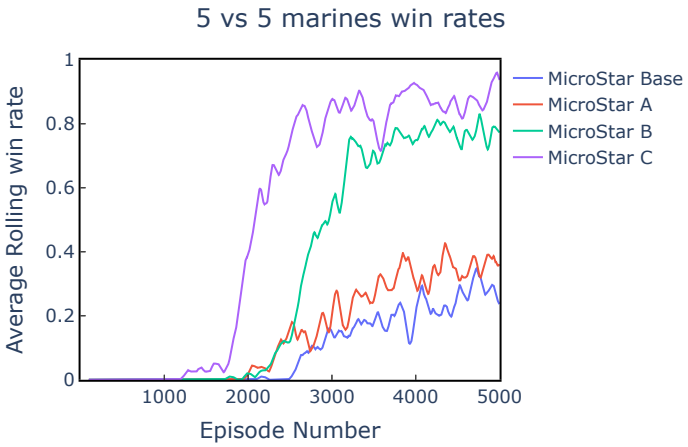


Fig. 13: Comparison in win rates between different MicroStar architectures from Table VIII during their training. Rolling average with window size of 100

done using a league like setup similar to AlphaStar playing either against some pre-defined scripted bots or against other RL agents. The problem would be benchmarking these models as there is no standardized way to do this. It would possibly require to be tested against human players.

It might also be interesting to train a second network for the macro part of the game and then use that in combination with MicroStar to play full-length games of SC2. This would require quite a lot of adaptions in how MicroStar is trained though as it is currently only focused on fighting but in actual full-length games unit control is more about positioning tactically and choosing when to fight correctly, which is also a very interesting problem but quite different to the scenarios discussed in this paper.

IX. Conclusion

In summary, the MicroStar model shows promising results for micro in SC2 while only being trained on a single consumer-grade PC. It is able to challenge DeepMind's FullyConv [8] model while only using a fraction of the computational power. MicroStar was shown to be capable of learning a solid strategy that makes logical sense on every scenario we tested. This might make it an interesting model for smaller teams with less resources. Another large advantage of MicroStar is that the full implementation is open source on GitHub so it can be quickly adapted and improved on without needing to re-implement everything. MicroStar might also still have some potential that can be unlocked with a hyperparameter search as shown by the 5 marines vs 5 marines experiment.

While more work is still required, we have shown that it is possible to achieve competitive results on the sub-problem of unit micro without the need for a large team and using only a single consumer-grade PC. This was done by utilizing modern neural network architectures and taking inspiration from previous works in the field.

References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[2] T. Miki, J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, "Learning robust perceptive locomotion for quadrupedal robots in the wild," *Science Robotics*, vol. 7, no. 62, p. eabk2822, 2022.

[3] A. Charpentier, R. Elie, and C. Remlinger, "Reinforcement learning in economics and finance," *Computational Economics*, pp. 1–38, 2021.

[4] Z. Zhou, S. Kearnes, L. Li, R. N. Zare, and P. Riley, "Optimization of molecules via deep reinforcement learning," *Scientific reports*, vol. 9, no. 1, p. 10752, 2019.

[5] C. Yu, J. Liu, S. Nemati, and G. Yin, "Reinforcement learning in healthcare: A survey," *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–36, 2021.

[6] 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.

[7] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[8] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser *et al.*, "Starcraft ii: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[9] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart *et al.*, "Relational deep reinforcement learning," *arXiv preprint arXiv:1806.01830*, 2018.

[10] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[11] X. Wang, J. Song, P. Qi, P. Peng, Z. Tang, W. Zhang, W. Li, X. Pi, J. He, C. Gao *et al.*, "Scc: An efficient deep reinforcement learning agent mastering the game of starcraft ii," in *International conference on machine learning*. PMLR, 2021, pp. 10 905–10 915.

[12] L. Han, J. Xiong, P. Sun, X. Sun, M. Fang, Q. Guo, Q. Chen, T. Shi, H. Yu, X. Wu *et al.*, "Tstarbot-x: An open-sourced and comprehensive study for efficient league training in starcraft ii full game," *arXiv preprint arXiv:2011.13729*, 2020.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[15] 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.

[16] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R Ruiz, J. Schrittwieser, G. Swirszcz *et al.*, "Discovering faster matrix multiplication algorithms with reinforcement learning," *Nature*, vol. 610, no. 7930, pp. 47–53, 2022.

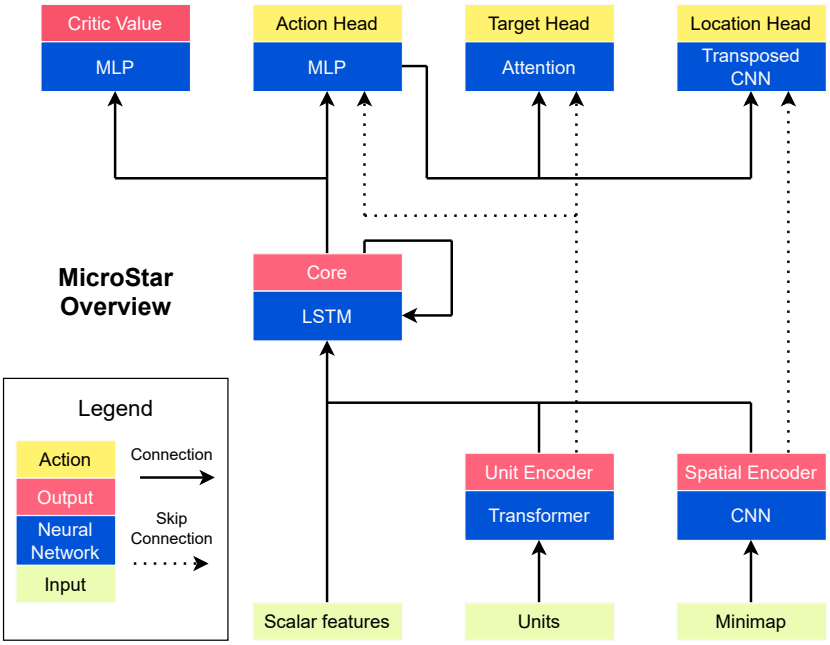
[17] D. J. Mankowitz, A. Michi, A. Zhernov, M. Gelmi, M. Selvi, C. Paduraru, E. Leurent, S. Iqbal, J.-B. Lespiau, A. Ahern *et al.*, "Faster sorting algorithms discovered using deep reinforcement learning," *Nature*, vol. 618, no. 7964, pp. 257–263, 2023.

[18] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 6, pp. 4909–4926, 2021.

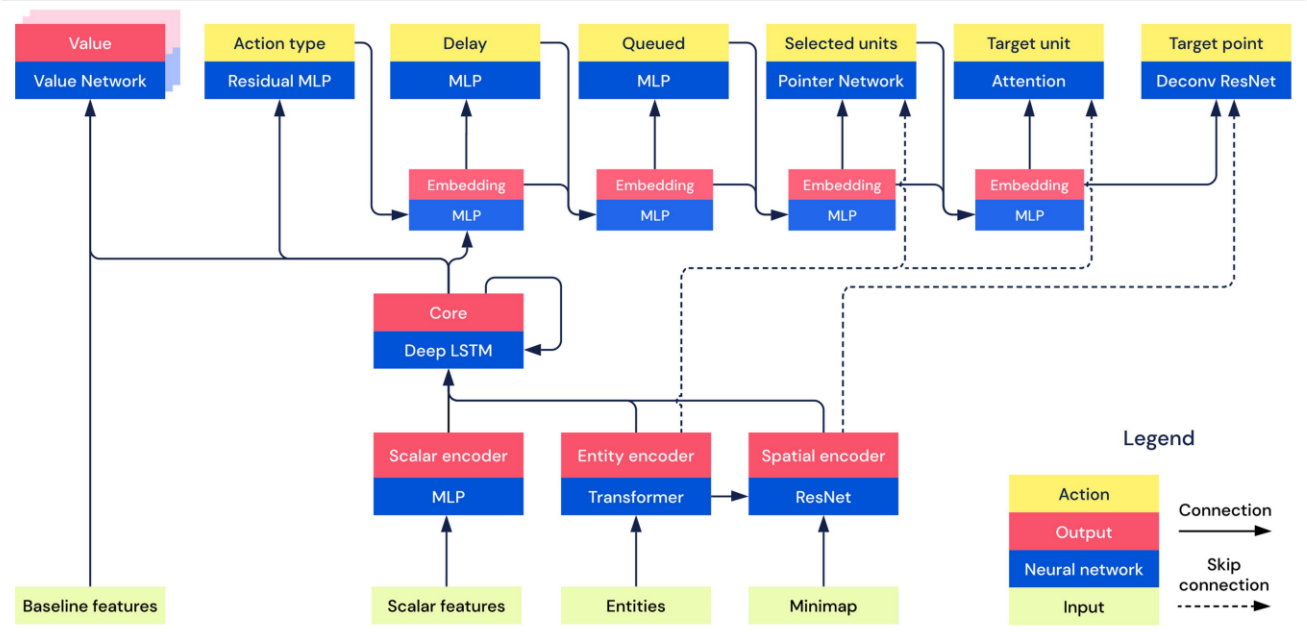
[19] Z. Tang, K. Shao, Y. Zhu, D. Li, D. Zhao, and T. Huang, "A review of computational intelligence for starcraft ai," in *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2018, pp. 1167–1173.

[20] Blizzard, "s2client-proto," https://github.com/Blizzard/s2client-proto, 2023.
[21] BurnySc2, "python-sc2," https://github.com/BurnySc2/python-sc2, 2023.
[22] R.-Z. Liu, Z.-J. Pang, Z.-Y. Meng, W. Wang, Y. Yu, and T. Lu, "On efficient reinforcement learning for full-length game of starcraft ii," *Journal of Artificial Intelligence Research*, vol. 75, pp. 213–260, 2022.
[23] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, 2021.
[24] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
[26] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.
[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[28] N. Usunier, G. Synnaeve, Z. Lin, and S. Chintala, "Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks," *arXiv preprint arXiv:1609.02993*, 2016.
[29] A. Dumitrescu and T. Rebedea, "Reinforcement learning for building starcraft 2 agents." in *RoCHI*, 2022, pp. 137–144.
[30] B. Alghanem *et al.*, "Asynchronous advantage actor-critic agent for starcraft ii," *arXiv preprint arXiv:1807.08217*, 2018.
[31] X. Shen, C. Yin, and X. Hou, "Self-attention for deep reinforcement learning," in *Proceedings of the 2019 4th International Conference on Mathematics and Artificial Intelligence*, 2019, pp. 71–75.

APPENDIX A
MICROSTAR AND ALPHASTAR COMPARISON



(a) General overview of the architecture of MicroStar.



(b) General overview of the architecture of AlphaStar [10].

Fig. 14: Comparison between MicroStar and Alphastar architecture. Most notably the Delay, Queued and Selected units heads were not implemented but even all parts that were implemented were down-scaled considerably.