PYTORCH-BASED REINFORCEMENT LEARNING FRAMEWORK FOR STARCRAFT II ENVIRONMENTS

March 31, 2019

Nico Ott, Hendrik Vloet
University of Freiburg
Department of Neurorobotics

Contents

1 Introduction

Over the course of the last years, deep reinforcement learning (further referred to as RL) has impressively shown its potential in the field of artificial intelligence (AI) for both digital and analog games. A major breakthrough was achieved by Google Deepmind in 2013, solving multiple Atari games ([1]) with the help of Deep-Q-Networks (DQN).
Another milestone is AlphaGo ([2]), an RL bot which was trained on the ancient Chinese board game Go, which used human expert knowledge, self-play and a combination of classic machine learning techniques (like Monte-Carlo tree search to find promising moves) and RL methods. AlphaGo was able to defeat the reigning world champion Lee Sedol in March 2016 with a score of 4 to 1. In October 2017, a completely auto-didactic version called AlphaGo Zero was introduced, which was not pre-trained on human expert data. It was able to defeat the version of AlphaGo, that won against Sedol, within 3 days of learning. Similarly, DeepMinds chess/shogi bot AlphaZero was able to dominate conventional engines in December 2018. (AlphaZero was already successful in December 2017, but these matches were heavily criticized for granting AlphaZero an advantage).
Although board games are hard problems, they are overshadowed by digital video games in terms of complexity. Because of their manifold mechanics, huge action and state spaces and long horizons, video games pose the next great challenge for RL algorithms. Several successes in this area could be achieved already. OpenAIs multi-agent bot OpenAI Five managed to defeat strong teams in Dota 2, a game where team play is a crucial premise to victory. ([3]).
The most recent breakthrough in this field is DeepMinds RL bot AlphaStar[1] which was able to defeat professional players in the game StarCraft II. ([4]).

1.1 The StarCraft II Learning Environment (SC2LE)

StarCraft II is a space-war themed real-time-strategy game where two players compete to destroy each others bases. A player can select between 3 equally strong races, whose play styles are fundamentally different. Victory depends on the players ability to efficiently collect resources, build structures, train worker and military units and finally attack and defend with its army. StarCraft II is a complex game that takes many hours to learn. Many human players invest a vast amount of their lifetime to improve their game-play. Professional players are even competing against each others in international leagues.
The game‘s predecessor "StarCraft" has always been a subject to AI research. The algorithms that were created to play in so called bot leagues mainly focused on manually handcrafted rules and predefined scripts. In August 2017, the game developer studio Blizzard cooperated with DeepMind to release the StarCraft II Learning Environment (SC2LE) [5], a framework that can interact with the StarCraft II game API.

[1]AlphaStar was released during this work and thus not regarded as baseline.

The framework includes PYSC2, a Python-based library, that focuses on the usage of RL implementations. Playing the whole game was expressed to be a new challenge, since it poses inherently difficult environment properties for RL algorithms:

- State spaces are large and state information is only partially observable through a mechanic called fog-of-war.[2] (This makes scouting a crucial part of the game).
- Action spaces are huge and vary over time.
- Episodes have long horizons with thousands of decisions to be taken. (Depending on the players strategy a game of StarCraft II can usually last in the range from 5-40 minutes).
- Rewards are inherently sparse (win/lose/draw) and delayed. (Early decisions might have a big impact on later stages of the game).
- Some modes are team-based and therefore multi-agent problems.

The SC2LE framework also contains 7 minigame maps[3], which cover different objectives from the full game and serve as benchmark for research. The community was also encouraged to submit new minigames, that focus on other aspects of the game.

1.2 Hierarchical Reinforcement Learning (HRL)

RL algorithms struggle when environments have certain properties. Large action spaces, sparse rewards and unstable systems often lead to slow or unsuccessful training results. Even more sophisticated algorithms like actor-critic methods or advantage-based agent might fail. The easiest way to overcome these obstacles is to manually design simplifications, a strategy used in the recent achievements mentioned above: action spaces are limited on meaningful sizes, reward structures are adapted to input more refined information, exploration is guided through expert data.

Another possibility to tackle these issues is the use of hierarchical frameworks. The strategy is to hierarchically decompose the original problem into smaller sub-problems. These reduced domains are then solved by finding their respective sub-policies in a divide-and-conquer fashion. In general, this process uses a curriculum learning approach, where policies are learned independently on easy tasks and the experience is then used to gradually solve harder tasks.

1.3 Goal

The goal of this work was to implement a PYSC2 compatible framework, that uses RL agents programmed in Python with PyTorch as underlying neural network library. The framework needs to be modular, s.t. environments and agents modules can be swapped out freely. The framework should be able to use simple RL algorithms, but also be prepared for more sophisticated implementations like hierarchical reinforcement learning agents.

[2]Fog-of-war is a game mechanic that only grants player vision where they have units.

[3]The minigames used in this work are described in the background section.

2 Related Work

After its release in August 2017, several works have brought up notable results on the SC2LE framework:
Reaver (June 2018, [6]): Reaver is a modular RL framework which was designed to work with the PYSC2 library and could achieve good learning results on the SC2LE minigames. Standard library actor-critic agents have been used. Their neural networks where implemented with TensorFlow.
AlphaStar (December 2018, [4]): DeepMind announced its Starcraft II RL bot AlphaStar in a live stream. The early training was done with game replays from expert StarCraft II players. In later stages, bots were matched up against each other in an evolutionary bot league. Successful bots were used for the next training iteration to improve their policies.
An early stage of the agent beat two professional StarCraft II players with a score of 5-0. A newer version of the AI was able to beat one of the two players again with a score of 2-1. Despite the longer training time, the human player was able to figure out a counterstrategy in the final match. This showed that the trained agent was still vulnerable to creative strategies and exploits unseen in the training phase.
In general, AlphaStar was better at managing its army during fights, by efficiently cycling wounded units out to safe distances when they were close to dying. This was only possible due to superhuman reaction speed during fights. AlphaStar could reach more than 1000 APMs (action per minutes) during combat, while having fairly low APMs in the phases in between.
Full Game with HRL (February 2019, [7]): This recently published paper investigates a hierarchical approach on the full-length game. In the first step the vast action space is reduced to a meaningful "macro-action space" (e.g. build a certain building, train a certain unit, ...). The trajectories of human expert data were analyzed and frequently used action sequences were encapsulated in macro-actions. This reduces search complexity, improves learning and testing efficiency.
The second step is to design the agent's action policy as a two-layer hierarchical architecture. The top layer describes a controller that selects sub-policies based on the full observation input. The chosen sub-policy then defines the macro-action that the agent should execute. This hierarchical structure allows the sub-policies to be defined with smaller state spaces and action spaces, that specialize on their respective tasks. (e.g. a sub-policy focusing on building structures, does not rely too much on current positions of enemy units. A sub-policy specialized on combat might only value building information about defensive structures like towers).
This HRL bot was tested against the conventional built-in StarCraft II AI on several degrees of difficulty. The trained agent won all of its matches on the easiest difficulty. On the hardest difficulty, it achieved a respectable win rate of 43%. The authors analyzed that a single-layer action policy was already able to perform well on easier difficulties, which means, that the macro-actions are proven to be a powerful tool.

3 Background

3.1 Markov Decision Process (MDP)

RL environments are modelled as Markov Decision Processes (MDP) and are defined by a 5-tuple $< S, A, P_a, R_a, \gamma >$ where S is the set of states, A the set of actions, P_a the state transition probability matrix, R_a the reward function and γ the discount factor. The solution to an MDP is a policy $\pi : s \rightarrow a$, a function that maps from a state $s \in S$ to a probability distribution a over a set of actions A.

3.2 RL Control Loop

In an RL setting, an agent aims to learn an optimized policy that maximizes its expected return by reinforcing beneficial behavior while discouraging poor behavior. Experience is sampled with the basic reinforcement control loop.

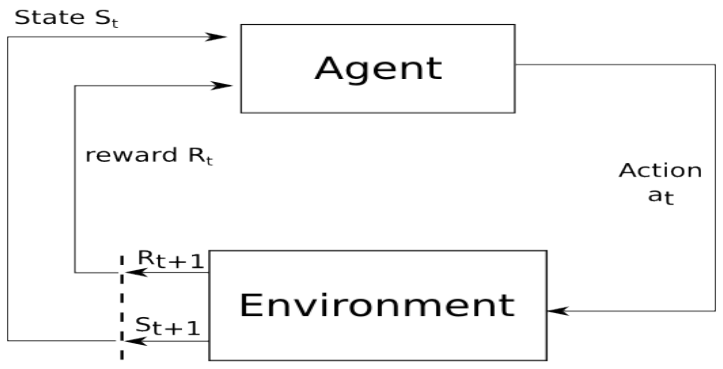


Figure 1: Basic reinforcement control loop

At every time step t, the agent selects an action according to a sample policy π, based on the current observed state s_t. It applies the selected action to the environment and receives a reward r_{t+1} and a new observed state s_{t+1}.
The optimal action-value function (or Bellman optimality equation) is another way to express a policy that maximizes the expected return:

$$Q^*(s,a) = \mathbb{E}[R_t | s = s_t, a = a_t, \pi^*] = \mathbb{E}[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a')]$$

The optimality has to hold for every timestep t (observing state s_t and executing an action a_t with an optimal policy π^*). a' is the action probability distribution of the policy in state s_{t+1}.

3.3 Deep Q Networks

The DQN-algorithm combines the Q-learning algorithm with deep neural networks as function approximators for the policy and the target. The neural network weights are optimized by minimizing the temporal difference error (TD error) of the optimal Bellman equation:

$$Q^*(s_t, a_t) = \mathbb{E}_{s_t,a_t,r_{t+1},s_{t+1}} \| Q_\Theta(s_t, a_t) - y_t \|_2^2$$

where

$$y_t = \begin{cases} r_{t+1} & \text{if } s_{t+1} \text{ is terminal} \\ r_{t+1} + \gamma \underset{a'}{max}\ Q_{\Theta_{target}}(s_{t+1}, a') & \text{otherwise} \end{cases}$$

DQN is an off-policy algorithm, that samples from an experience replay buffer (ER). The experience transistions are saved in the ER in the form of an observation tuples $< s_t, a_t, r_{t+1}, s_{t+1}, \gamma >$. Whenever the agent takes an optimization step, a mini-batch is randomly drawn from the replay buffer and used to update the network weights via gradient descent. DQN uses two different Q-networks: the current network with parameters Θ and the target network with parameters Θ_{target}. The parameters of the target network are set to Θ every N episodes.[4]

3.4 Semi-Markov Decision Process (SMDP)

Semi-Markov Decision Processes are defined by a five tuple $< S, \Sigma, P_S igma, R_S igma, \gamma >$ where S is the set of states, Σ the set of skills, $P_S igma$ the transition probability matrix, $R_S igma$ the received skill rewards at each time step and γ is the discount factor. A solution to an SMDP is a skill policy μ [8], [9].

3.5 Skills, Options, Macro-actions

The option framework [8] augments the set of primitive actions to a set of temporally extended options (further referred to as skills). A skill is defined as a triple $\sigma =< I, \pi, \beta >$ with I being the set of initial states, at which the skill can be started, π being the intra-skill policy, and β being the termination probability for the skill.
In the context of this work, skills can be understood as a sub-policy for simplified environments. In the scope of HRL, these skills can be learned in a curriculum learning fashion, where sub-policies are found for simplified tasks, which can later be applied as temporally extended actions for more complex tasks.

[4]For reference, the pseudo code taken from the DQN paper is included in figure 24) in the Appendix section.

3.6 Skill Policy

The skill policy $\mu : s \rightarrow \Delta_{\Sigma}$ maps the states to a probability distribution over skills Σ. In this context, the skill-value function $Q_{\Sigma} : S \times \Sigma \rightarrow R_{\Sigma}$ represents the long-term value of following the intra-skill policy of a skill $\sigma \in \Sigma$ starting in state $s \in S$. The action-value function is changed to a skill-value function that now incorporates skills instead of actions:

$$Q(s,\sigma) = \mathbb{E}[R_{\tau} | s = s_{\tau}, \sigma = \sigma_{\tau}, \mu^{*}] = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^{t} r_{t} | (s,\sigma), \mu]$$

With τ being a time step at which a skill σ is started. Likewise the skill reward is altered over a horizon of k steps denoted as:

$$R_{s}^{\sigma} = \mathbb{E}[r_{\tau+1} + \gamma r_{\tau+2} + ... + \gamma^{k-1} r_{\tau+k} | s = s_{\tau}, \sigma]$$

Given these definitions, the optimality of the skill-value function can be expressed through the optimal skill value function [10] which has to hold at every skill timestep τ.

$$Q_{\Sigma}^{*}(s,\sigma) = \mathbb{E}[R_{s}^{\sigma} + \gamma^{k} \max_{\sigma' \in \Sigma} Q_{\Sigma}^{*}(s', \sigma')]$$

3.7 Lifelong learning

Lifelong learning can be seen as the next logical evolution in RL [11]. The goal of lifelong learning systems is to sequentially retain learned knowledge in a knowledge base and transfer it to new tasks, i.e. new hypotheses and/or policies (Figure 2).

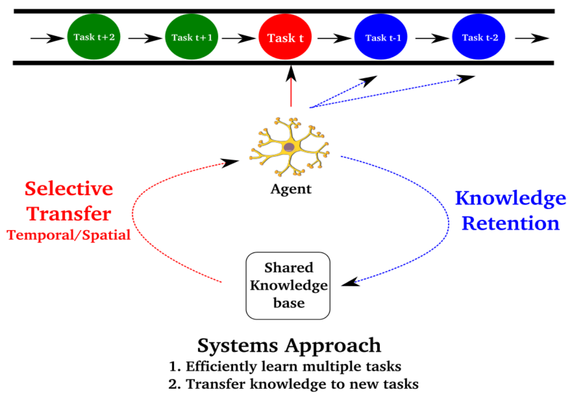


Figure 2: Lifelong learning and knowledge transfer [9]

The idea of lifelong learning seems reasonable for such a complex and evolving environment like StarCraft II. Players have to adapt constantly and learn new strategies in order to outplay their opponents.

3.8 HDRLN Framework

The HDRLN framework is a hierarchical RL framework which was first used on toy domains of the game "Minecraft" [9]. The basic hierarchical idea is the usage of skills inspired by the option framework. The HDRLN framework is roughly build upon the DQN architecture and augmented by:

- a Deep Skill Module that holds the knowledge base about the skills.
- a Controller, that acts as a skill policy, choosing either between skills or primitive actions.
- a Skill Experience Replay Buffer (SERB) that can store state transistion information when using skills.

In general, the HDRLN framework is an abstract concept, that allows to freely specify the underlying RL algorithms for the controller skill-policy and the intra-skill action-policies. Additionally, the Deep Skill Module can be implemented either as a simple array of Deep Skill Networks (DSN) or a single distilled Deep Skill Network.

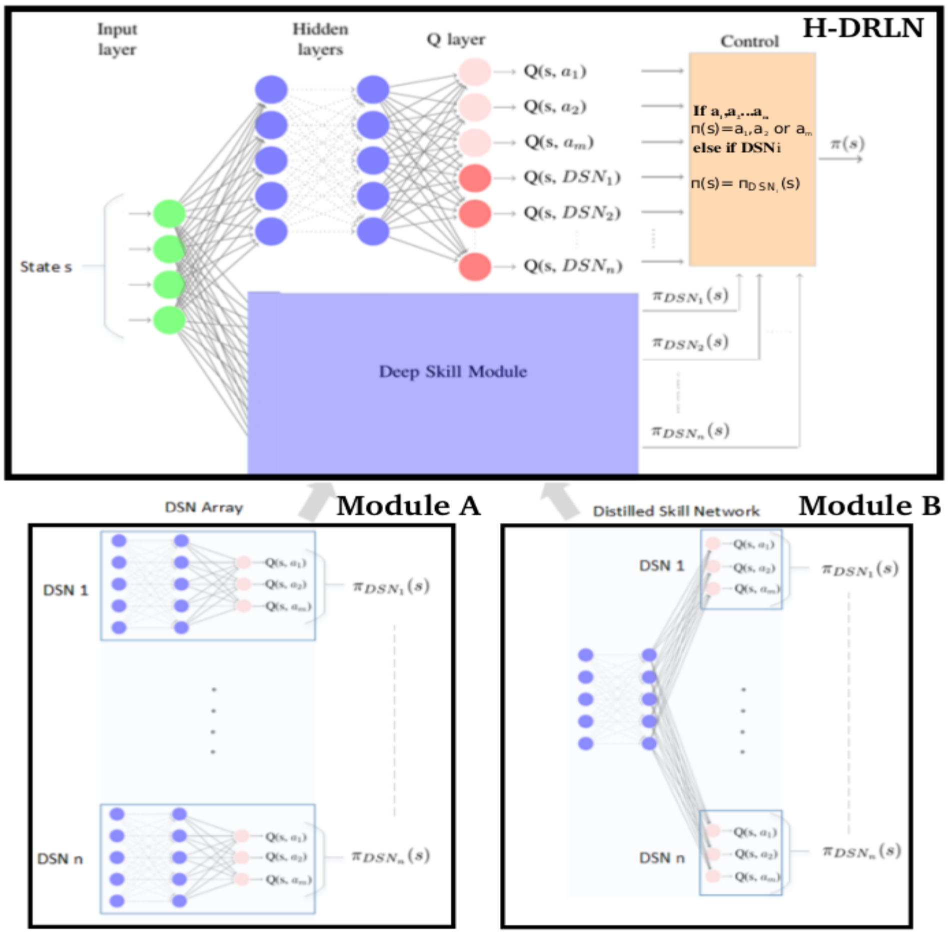


Figure 3: Framework of the HDRLN paper [9]

3.9 Policy Distillation

Policy distillation is used to transfer and efficiently retain knowledge from one or multiple teacher policies T to a student policy S. This can be done by supervised learning, e.g. when the teacher and student models are separate networks, the student network can be trained to predict the outcome of the teacher network (in this case the Q-values). In the original paper, the objective function [9], is defined as a slightly modified version of the Mean-Squared-Error (MSE) loss:

$$\|\mathrm{Softmax}_{\tau}(Q_T(s)) - Q_S(s)\|^2$$

where τ is the softmax temperature.
When multiple teacher models T_i and one single student network are used, the training of the student is done by round-robin iterations where every N iterations the students learns from one of the teacher models T_i.
The motivation for policy distillation is mainly to reduce the network complexity from a large teacher model to a simpler and more lightweight student model. Multi-policy distillation is used to condense the knowledge from N multiple sub-policies (from N different sub-tasks) into a single network with N different output layers, one for each sub-policy. Because policy distillation does not apply to the RL control loop, the HDRLN agent must use a separate script that distills the learned DQN skills of the sub-domains into a single distilled network.

4 Methodology

The insights on first programming tests with hard-coded bots were used to create a modular framework that is able to handle all necessary functions needed for a modular RL setup. The building blocks of the framework are described in detail in the following chapters.

4.1 Framework Architecture

4.1.1 Experiment Procedure

In this framework, RL agents are trained and tested in sessions which are further referred to as experiments. Independent of the used agent or environment, an experiment consist of two phases which use two separate scripts:

- a learning script (where the policy is trained) and
- a testing script (where the policy is evaluated).

Figure 4 shows, how these two phases interact with each other:

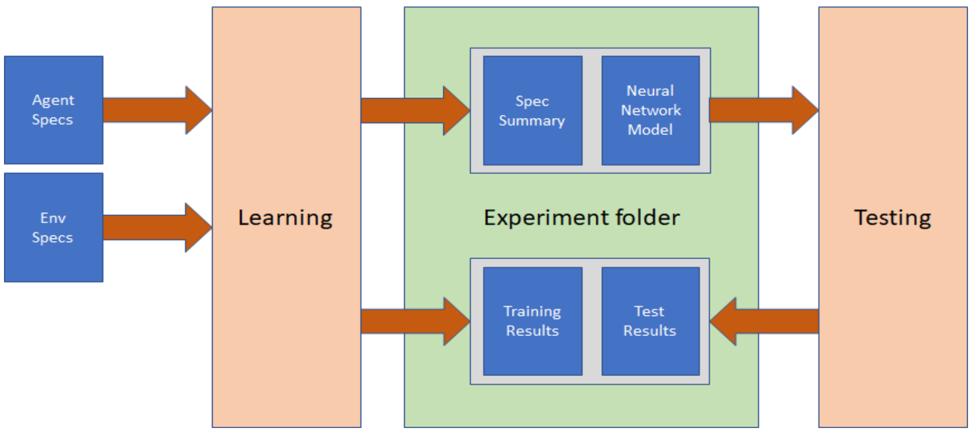


Figure 4: Experiment procedure

First, an experiment needs to be fully defined through specification files (further referred to as specs). When starting the learning phase, an experiment folder is created, which stores all data related to an experiment. This includes:

- agent and environment specifications merged into a single file (called spec summary).
- The neural network models and weights.
- Training and testing results (list and plot data).

In the testing loop, the spec summary and the model file are loaded to setup an exact copy of the setup used for training. However, the agent will be set to testing mode and therefore will not update its model parameters.
We chose this two step experiment setup instead of a single fully automated script (with intermediate tests), to shorten training time.

4.1.2 RL Main Loop

The training and testing script share a similar structure:

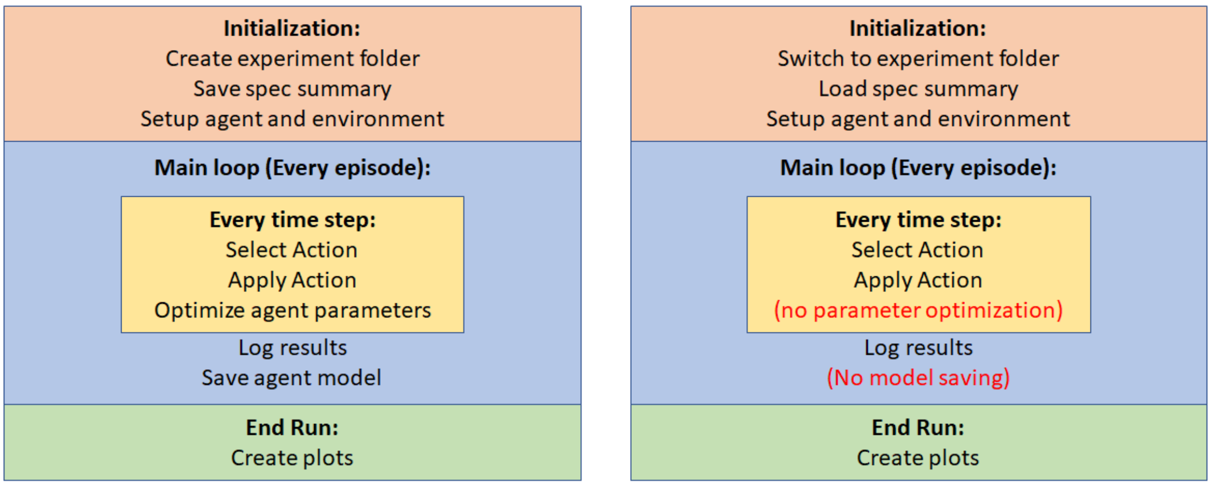


Figure 5: Schematic overview of the training and testing loop

However, the testing loop does not alter the model parameters and therefore does not save any intermediate models. Additionally, the specs for the testing file are loaded from the spec summary which was generated by the training script.
In the initialization phase the agent, environment and file manager classes are setup with the chosen hyperparameters. The agent is either set to learning mode (when training) or testing mode (when evaluating). In the main loop the agent selects actions according to its policy and applies it to the environment. In the case of the training script, the agents neural network weights are updated after every time step. When an episode is finished the episode results are logged and (if the agent is in training mode) the current agent model is saved.

4.1.3 Hyperparameters and Specification Files (Specs)

Before starting an experiment, several parameters have to be adjusted by the human operator. These are mainly hyperparameters (like optimizer learning rate, discount factor, ...) and framework relevant parameters (saving replay, using different action spaces and reward functions, ...). For our framework, experiment adjustments are defined through dedicated specification files called specs (in CSV format). All relevant parameters can be adapted in a clear overview instead of scrolling through code. This makes hyperparameter tuning simple and fast. The hyperparameters used for the learning tasks are split up into environment and agent specific parameters (agent specs and env specs). They can be adapted independently from each other, which makes it simple to interchange agent configurations for training and testing on other environments. Additionally, the specs offer a simple interface for further framework adaptions (like a graphical user interface, where specifications can be saved and loaded). An example of how the specs are saved in the spec_summary.csv file can be found in the Appendix. (Figure 25).

4.1.4 Environment

In the scope of this work, a wrapper class was written for the PYSC2 environments, that works similar to environments from the gym library. An environment base class forms an abstract foundation that provides general functions. Depending on each PYSC2 map, a new class can derive from this base class, to adapt individual functionality. For the minigame maps used in this work, the observation space has been reduced to a single 2-dimensional layer and additional information (e.g. distances to beacon, positions of units...) is calculated that can be used for supervised learning.

4.1.5 DQN Base Agent

Similar to the environment base class, a DQN base agent class has been designed that is able to handle simple DQN-based algorithms. The main task of the agent is to use its underlying policy module to select an action for every time step. The DQN base agent class therefore translate the observation information from the environment into a compatible state information tensor. Additionally, the agent keeps track of counters and flags, that were defined by the agent specs. The deep learning functionality of the policy is fully encapsulated in the policy module and not the agent class itself.

4.1.6 Exponential Epsilon Greedy Decay

The agent chooses its actions according to an epsilon greedy policy, i.e. there is a certain probability that the best action with the best Q-value is taken, otherwise a random action is selected. The agent starts with a high value in epsilon, i.e. the probability to execute random actions is higher in the early stages of learning. This favors an explorative behaviour above exploitation in the beginning of the learning process. The epsilon decreases exponentially in order to shift the behavior from explorative to exploitative. In an ideal setting, the agent learns everything during its exploration phase and then slowly transits to a pure exploitative behavior in order to find the optimal policy. The equations below show how the current epsilon is decreasing (n is the total amount of taken time steps).

$$\text{Probabilities} = \begin{cases} \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) \cdot e^{-\frac{n}{\epsilon_{decay}}} & \text{take random action} \\ 1 - \left[\epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) \cdot e^{-\frac{n}{\epsilon_{decay}}}\right] & \text{take best action} \end{cases}$$

4.1.7 DQN-Module

The actual agent policy is separated from the agent class and implemented with RL policy modules. This approach offers multiple benefits:

- It makes it easy to quickly replace a policy module with another learning algorithm (e.g. actor critic, policy gradient based) while keeping the overall structure of the agent the same.
- Policy modules could also be written with other deep learning libraries. (This keeps the framework flexible to developers who rather like to use e.g. TensorFlow or Keras).
- A policy module can be used as an intra-skill policy and offers all the wrapper methods for accessing the underlying neural network. Implementing hierarchical architectures with policy modules as pre-learned skills is easier since the policy modules offer powerful interfaces.

The DQN base agent incorporates a simple DQN module that acts like a wrapper class that can access the underlying neural network with many useful deep learning methods like:

- Calculation of the forward pass.
- Storing transitions in the replay buffer.
- Sampling a mini-batch from the replay buffer.
- Finding the best action values for a batch.
- Calculating the TD-target.
- Optimizing the network weights on a sampled batch.

4.1.8 Neural Network Architectures

The neural network architecture for the DQN module is interchangeable too. For the minigames an architecture with 3 convolutional and 3 feed forward layers was used (Figure 26). The first convolutional layer uses a kernel size of 13, stride of 1 and contains of 32 channels. The second convolutional layer consists of a kernel with size 5, stride 2 and 64 channels. The third convolutional layer uses a kernel with size 3, stride 1 and 128 channels. No padding was used.

All convolutional layers use a ReLU activation function and the outputs of these layers go through a 2×2 MaxPooling layer. The output of the last convolutional layer gets flattened and is propagated through three fully connected layers (all except the last one use ReLU activation and a dropout of 0.5). The first fully connected layer uses 256 neurons and the second 512. The last layer (output layer) uses a softmax activation function and has a size equal to the used action space (e.g. 5376 for a full grid agent or 4 for the compass agent). An overview of the network architecture can be found in the appendix in Figure 26.

4.2 Reducing Action Space Complexity

Huge action spaces pose a problem to RL. The complete action space of the original PYSC2 framework consist of around 400 actions. Additionally, some actions require a spatial component consisting of an x and y coordinate. Not only is the resulting action space large, it changes over time too, depending on selected units or built structures. The simplest way, to handle large action spaces is to reduce the number of meaningful actions in every time step. This technique was also used by DeepMinds AlphaStar, which reduces the vast action space of the original game to a smaller action space of "approximately 10 to the 26 legal actions at every time step" [4].
For the simplest minigame, Move2Beacon, the marine must be selected in the first time step (PYSC2 action: SelectArmy). To the end of the episode the agent must select a new position to which the marine moves to at every time step (PYSC2 action: MoveScreen). By predefining these actions, the action space can be reduced to the spatial map size which is a 84x64 grid. This grid size still results in 5376 possible actions that can be selected at every time step which makes simple DQN implementations unfeasible. Therefore two approaches where used to reduce the action space complexity for the DQN learning algorithm.

4.2.1 Grid Agent

A simple, yet powerful solution is a limitation of the action space to a coarser grid size. For the minigame Move2Beacon, it makes no difference if the marine travels exactly to the center of a beacon or selecting a grid point which is sufficiently close to the center.

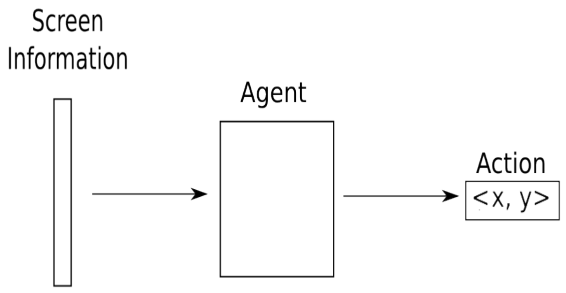


Figure 6: Grid Agent: gets the screen information as input and generates an action that selects a specific coordinate on the map.

Even though the grid agent already reduces the action space drastically, it still can be fairly large. For a coarse grid of 20x20 actions, the action space is still of size 400. Also, travel time of the marine varies depending on the initial and chosen end position.

4.2.2 Compass Agent

The second approach is an agent that only uses step wise actions in compass direction (up, right, down, left) and therefore has an action space dimensionality of size 4. In comparison to the grid agent with grid factor 20, the compass agent reduces the action space by a factor of 100, which makes it more sample-efficient and faster in learning. Another advantage of the compass agent is that the travel distance of the marine is fixed. When performing an action, it is assured that the agent reaches his destination in a defined time.
The compass agent has the disadvantage that it cannot move diagonally. In the worst case, where marine and beacon lie on a diagonal line, the marine needs $\sqrt{2}$ more time steps to reach the beacon.[5]

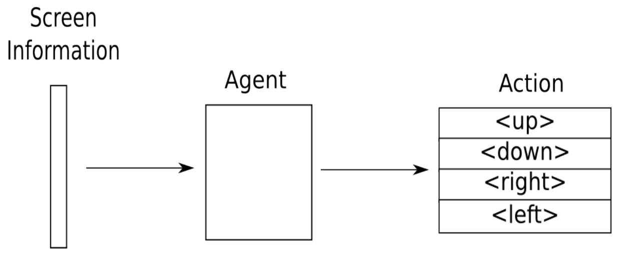


Figure 7: Compass Agent: gets the screen information as input and generates an action that moves the marine a specified amount of steps up, right, down or left relative to the current position of the marine.

4.3 Enhancing Reward Structure

Sparse rewards are problematic for reinforcement learning. Successful transitions that yield any kind of information (either good or bad) are rarely encountered and thus slow down the learning process. The easiest way to improve a sparse environment is by introducing additional rewards that punish unwanted behavior or add a bonus on desirable behavior. For our implementation, two additional reward functions have been implemented.

- The distance reward adds a normalized negative bonus on the distance between the marine and the beacon. This reward function was mainly used for the grid agent.
- The diff reward adds a positive reward equal to the distance traveled in every time step (difference between s_t and s_{t+1}). This reward function was mainly used for the compass agent since absolute changes were low in every time step, but differential changes were well defined. (e.g. moving directly in beacon direction results in $+1$ reward, moving away results in a -1 reward.)

Other possibilities to enhance the reward structure, that were considered, yet not implemented are negative step rewards (preventing the agent from moving around aimlessly) or curiosity rewards (new states are rewarded/ another way to force exploration).

[5]According to Pythagoras theorem

4.4 HDRLN Agent

4.4.1 Architecture

The basic HDRLN concept is compatible with the general RL experiment loop. An HDRLN agent module was designed instead of setting up a whole new framework. The basic functionality is similar to the DQN base agent, but it is augmented by 4 different modules:

Controller The policy of the HDRLN agent is setup in two stages. The controller is the first stage and the central component of the HDRLN module. It embodies the skill policy and learns when to use skills and when to use primitive actions. In general, the controller could be implemented with hardcoded, table driven or deep RL methods. For our framework we implemented the controller as just another RL module. Depending on the deep skill module type, the controller either selects the intra-skill action policies from the deep skill network array or the distilled policy network.

Deep Skill Network Array (DSN Array) The intra-skill action policies of the skills are represented by deep skill networks. If the deep skill module is selected as DSN array, the pre-trained RL modules from the minigames can simply be added as new entries in the DSN array. In general these skills do not have to use deep learning to be compatible with the framework. It could be possible to add hard-coded skills that offer the necessary interface methods. This technique was also used in [12] to selectively train skill modules, while keeping other skill modules constant with hard-coded policies.

Distilled Policy Network For a lifelong learning approach, it is necessary to distill the skills of the deep skill network array into a single network with one output layer for the primitive actions and N additional output layers for N distilled skills. This network can be implemented as another RL module, where the number of output layers has to be defined before starting the policy distillation routine. The policy distillation process shows, how important it is to use neural network libraries that can use dynamic computational graphs (like PyTorch). When adding a skill to a distilled policy network, an additional layer must be added to the network dynamically.

Skill Experience Replay Buffer (SERB) The SERB inherits its basic functionality from the standard replay buffer used in the DQN agent module. When storing a skill transition which took k steps, $s_{\tau+k}$ has to be stored instead of $s_{\tau+1}$.
Similarly, the skill reward (sum of discounted rewards encountered while following the skill policy) $R^{\sigma}_{s_{\tau+k}} = r_{\tau+1} + \gamma r_{\tau+2} + ... + \gamma^{k-1} r_{\tau+k}$ has to be stored instead of the one-step reward $r_{\tau+1}$.

4.5 Toyproblems

Because of it's large action space, the StarCraft II minigames are fairly hard to learn. Even for the reduced grid action space and adapted reward function, the agents had issues with finding a stable policy. It was not possible to extract any meaningful skills that could be used with a hierarchical framework. Therefore the architecture and DQN algorithm were tested on less complex tasks from the gym library, namely Cartpole, Pendulum and Acrobot. For a detailed description of the toy problems and the used neural network architecture, refer to the appendix section.

5 Results

Our framework was tested on six different environments including three of the SC2LE minigames (Move2Beacon, CollectMineralShards, DefeatRoaches) and three toy environments from the gym library (Pendulum-v0, Acrobot-v0, CartPole-v1). The used hyperparameter sets and the experiment routines can be found in the appendix.

5.1 Toy Problems

5.1.1 Cartpole

Figure 8 shows the learning progress of the DQN agent trained on the Cartpole environment. Within 1500 episodes the agent was able to learn a policy that always achieved the maximum score of 1500. An interesting observation is, that the model of epoch 1800 was achieving poor results but the agent later fixed its policy to reach maximum rewards again.

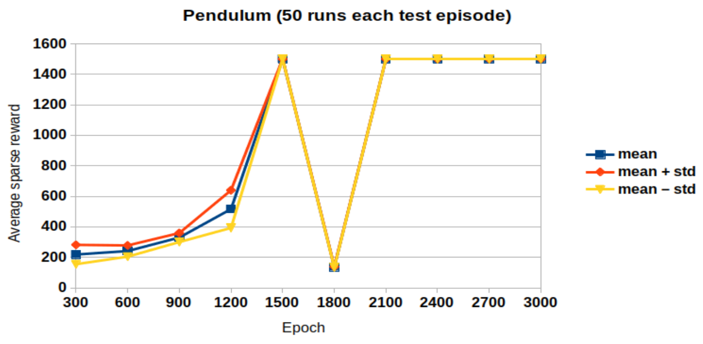


Figure 8: Performance progression on Cartpole

Since every single of the 1000 test runs had a total reward of 1500, the reward histogram offers no information and is thus not included.

5.1.2 Pendulum

Figure 9 shows the learning progress of the DQN agent trained on the Pendulum environment. The agent quickly found a policy that was able to successfully swing up the pendulum for the majority of the test runs. A particular good model was found in episode 2700 where the agent received an overall higher reward and the standard deviation of the test runs were lower too, which means, the agent performed consistently well.

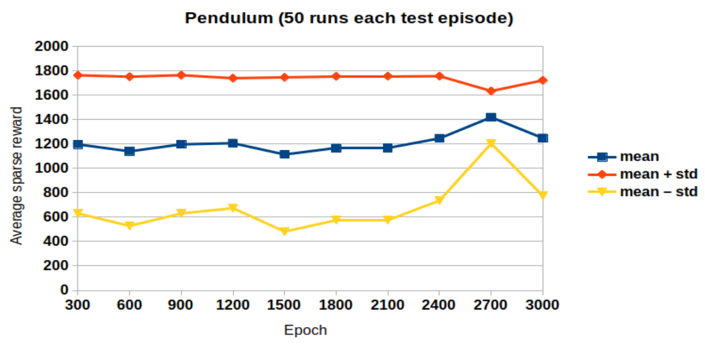


Figure 9: Performance progression on Pendulum

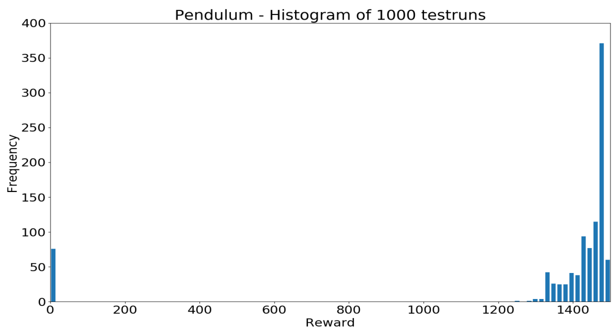


Figure 10: Reward histogram of Pendulum on 1000 test runs (after 2700 training episodes)

If the pendulum starts in a slightly diagonal position it was able to swing up and balance for the rest of the episode. However, the test runs show, that there are several outliers, where no reward at all was achieved. These results were observed, when the pendulum started in a straight-down state, where it had no momentum.
A possible problem for this behavior could be the small action space which only consist of three action (clockwise, counter-clockwise, idle) and therefore three different torques. The torques must be strong enough, to let the pendulum swing up, but still small enough to not push the pendulum out of the upright goal state. A more refined action space with 5 actions (strong clockwise, weak clockwise, strong counter-clockwise, weak counter-clockwise, idle) could help to achieve better results.

5.1.3 Acrobot

Figure 11 shows the learning progress of the DQN agent trained on the Acrobat environment. Within 700 episodes the agent was able to learn a policy that achieved the maximum score of around -70 for most of the test runs. It is interesting, that this almost optimal policy was refined in a time span of around 100 episodes, which results in a step-like characteristic of Figure 11.

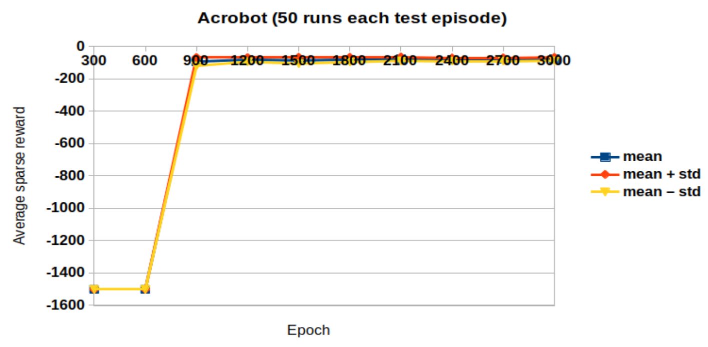


Figure 11: Performance progression on Acrobot

The long evaluation for the model 1500 confirmed these results (Figure 12). In almost all test episodes, the agent was able to swing above the goal line within 200 time steps. 10 out of 1000 test runs resulted in outliers. Only 2 test runs were observed as totally failed episode where the agent was not able to solve the task at all.

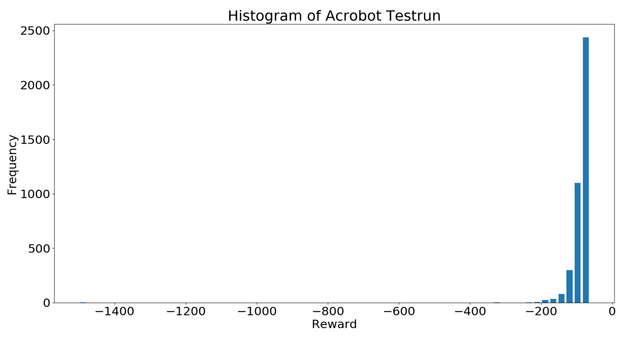


Figure 12: Reward histogram of Acrobot on 1000 test runs (after 3000 training episodes)

5.2 StarCraft 2 Minigames

5.2.1 Move2Beacon - Compass Agent

Figure 13 shows the learning progress of the DQN agent trained on the Move2Beacon environment with compass action space. The agent was trained for 2000 episodes. In episode 600 it achieved its peak performance with an average reward of around 21 collected beacons. Additionally, the standard deviation was only half as high in comparison to other training stages.

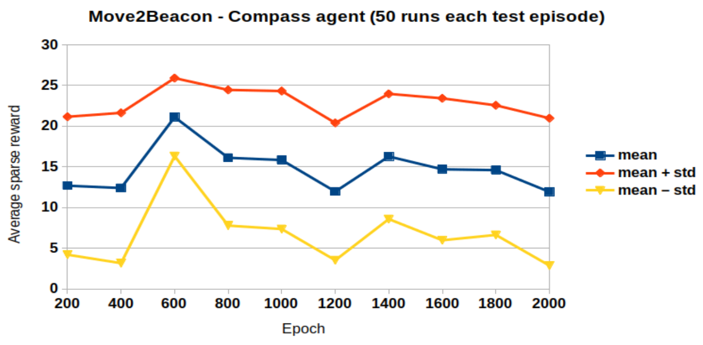


Figure 13: Performance progression of compass agent on Move2Beacon

The reward histogram (Figure 18) shows that the agent was scoring around 22 beacons in successful episodes. This is around 3 beacons less than the successful baselines presented in the original SC2LE paper.

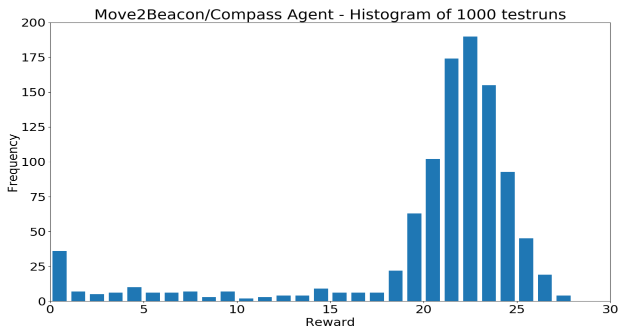


Figure 14: Reward histogram of Move2Beacon compass agent on 1000 test runs (after 600 training episodes)

The compass action space does not allow for diagonal movement, which lengthens marine travel distances. Also, fault states got the marine stuck at the edges of the map or jumping between two positions. The reward histogram shows that these faults can occur anytime during the tests. The peak at zero might be explained by episodes where the agent is initialized in fault states. It is still unclear how and why these states occur.

5.2.2 Move2Beacon - Grid Agent

Figure 15 shows the learning progress of the DQN agent trained on the Move2Beacon environment with a 84x64 grid action space. Over the course of 30000 training episodes the agent constantly improved its policy. It started with an average reward of 4 in episode 3000 and could improve its statistical results to an average reward of 8 beacons per episode. The standard deviation stayed the same for the whole training, which means the agent performed equally consistent at every stage of training.

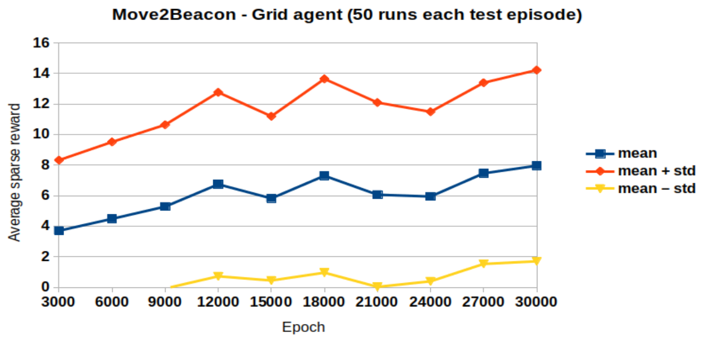


Figure 15: Performance progression of grid agent on Move2Beacon

The analysis of the simulation showed that the agent is actually learning the correct policy and selecting the right grid positions. This sometimes resulted in perfect episodes where the agent was able to collect all beacons. However, due to a faulty behavior in a subset of states, the agent was failing early in many test episodes. Figure 18 shows that the agent often encountered episodes with low reward.

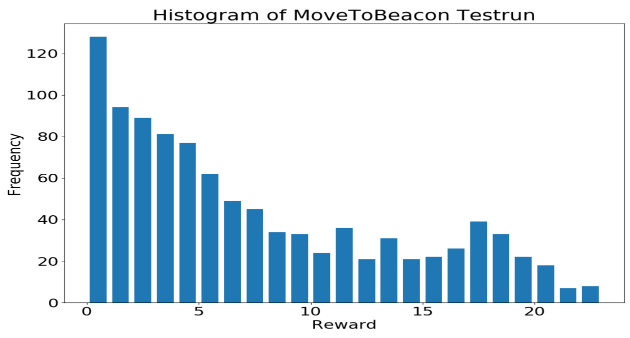


Figure 16: Reward histogram of Move2Beacon grid agent on 1000 test runs (after 30000 training episodes)

5.2.3 CollectMineralShards

Figure 17 shows the learning progress of the DQN agent trained on the CollectMineralShards environment with a 20x20 grid action space. The agent had a constant average performance of 8 collected shards along the 27000 training episodes[6]. The standard deviation stayed the same for the whole training, which means the agent performed equally consistent at every stage of training. In episode 21000 the agent performed the best.
The visual analysis showed that the agent is learning a policy that sometimes encourages the agent to travel to different regions of the map. However, the agent gets stuck at the edges of the map or in fault states (similar to the Move2Beacon environment) for almost all episodes and only collects minerals that are located on the way.

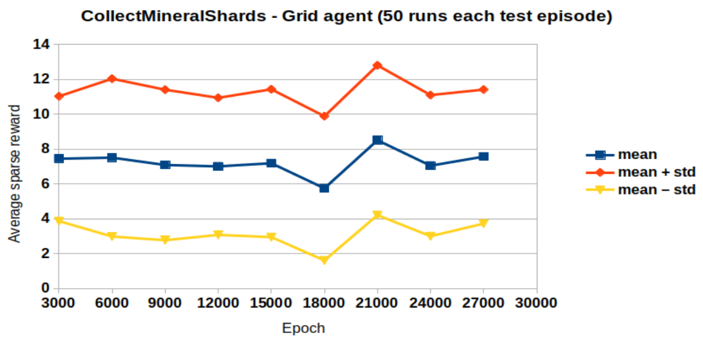


Figure 17: Performance progression of grid agent on CollectMineralShards

Compared to the baselines provided in the original SC2LE paper, our agent performs poorly. It is even beaten by the random agent, because minerals are spread around the maps and random actions might eventually generate some reward. However, the baseline agent was achieving good results after 50-100 million training episodes.

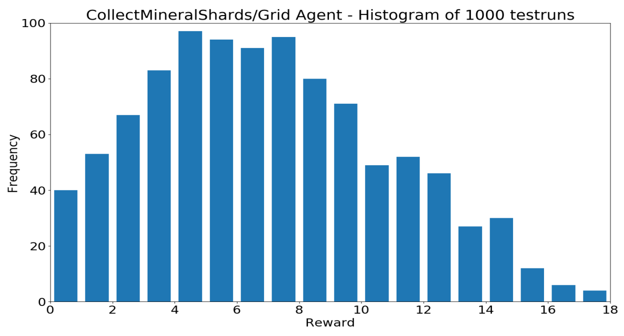


Figure 18: Reward histogram of CollectMineralShards grid agent on 1000 test runs (after 21000 training episodes)

[6]The agent was supposed to train for 30000 episodes but the training script was aborted because the computer rebooted unexpectedly.

5.2.4 DefeatRoaches

Figure 19 shows the learning progress of the DQN agent trained on the DefeatRoaches environment with a 20x20 grid action space. The agent had a constant average performance around 30 along the 30000 training episodes. The standard deviation stayed the same for the whole training, which means the agent performed equally consistent at every stage of training. In episode 30000 the agent performed the best.

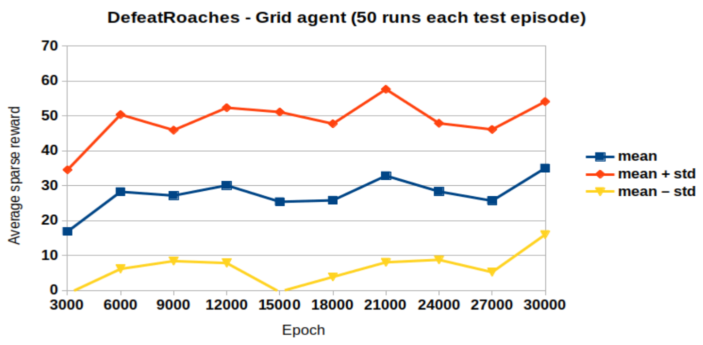


Figure 19: Performance progression of grid agent on DefeatRoaches

The visual analysis showed that the agent learned that attacking the enemies will generate positive rewards. The reward histogram shows, that in most cases the agent manages to kill a few of the Roaches before loosing all marines (<50). These results are close to the DeepMind human player performance from the SC2LE paper. However, the baseline agents highest score was 81, which our agent could surpass several times. In one episode the agent even reached a score of 250, which was also achieved by the random search baseline agent.

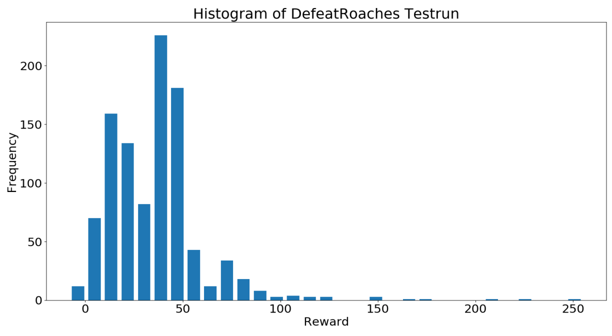


Figure 20: Reward histogram of DefeatRoaches grid agent on 1000 test runs (after 30000 training episodes)

Surprisingly, the agent finished some episodes with negative total reward. This issue arises because in StarCraft II it is possible to kill own units with friendly fire. If the agent rallies its marines through repeatedly attacking the same position, the marines will eventually end up killing each other.

6 Discussion

Achievements We implemented a framework, that can make use of modular gym-like environments and agent modules. The RL functionality is hereby encapsulated in modules, which can be implemented with any deep learning library, like PyTorch, TensorFlow or Keras. The framework features base classes for agents and environments, which can be used for inheritance. RL experiments are fully parameterized through specification files (specs) which are stored in CSV format. Results of training and testing runs are automatically stored as list and plot results in the respective experiment folders. Agent models are saved automatically in a cyclic fashion or when an experiment is aborted. For the SC2LE minigames, the replay buffer can store the data in a more efficient data type format which increases its capacity. A training, testing and supervised training mode has been added, which can guide the agent to select actions that fill its replay buffer with valuable transitions.
Additionally, a wrapper library was implemented for the SC2LE minigames that makes them accessible like gym environments. The state information was altered to a single layer, which correctly represents the positions of friendly, neutral and enemy units.[7] We implemented two simplified action spaces and two additional reward functions (for the Move2Beacon environment) that drastically reduce the complexity of the problem to simplify the learning process.
We created a simple and adaptable DQN agent from scratch. Furthermore, we designed and tried to implement an HDRLN agent module using hierarchical methods according to the Minecraft paper [9]. We started to set up the HDRLN agent and successfully tested that our general approach was working as intended. The pre-trained RL modules from the minigames could simply be added as skills into the framework. However, it was not possible to complete and successfully test the hierarchical agent due to time constraints. Also, the trained skills from the minigames showed sub-optimal behavior and therefore training a hierarchical agent with sub-optimal skills would have been expected to deliver poor results. Finally, the SC2LE framework did not offer any complex domain, that could make use of the minigame skills. It is in general possible to create new maps with a software provided by Blizzard, but we had no time to setup the complex domain, we had designed. (Figure 28)

Test results We tested our framework on six different environments with our vanilla DQN algorithm, based on PyTorch. The agent was finding successful policies for the three gym environments even though the hyperparameter sets were not fully optimized. Only the Pendulum environment showed some sub-optimal behavior when the agent started in a straight-down state.
For the StarCraft II environments, we found that the agent was able to partly learn meaningful behavior on all of the environments but it often happened that the agent bugged

[7]Some of the layers were not working correctly when using the original PYSC2 environment.

out for some states. We still have not found the reason for this issue, which massively worsens the test results for all tested environments and action spaces.
The compass agent was achieving scores close to the baseline agent on the Move2Beacon environment. Because of its inherent movement limitations it had to take some detours instead of travelling straight to the beacon. The grid agent also learned a successful policy and had several episodes were it was able to achieve baseline results. Its test performance got massively sabotaged by the bugged behavior mentioned above. The CollectMineralShards environment was overall performing poorly because the agents got stuck in certain positions. A possible solution would be to let the agent train longer or to optimize its hyperparameter set. Both solutions require massive computation time. Finally, the DQN agent was partly successful on the DefeatRoaches environment, acting slighly better than novice human players. The DefeatRoaches environment poses a challenge to humans, since it requires fast reaction time and micro-management, skills where a computer has its advantages over inexperienced humans.

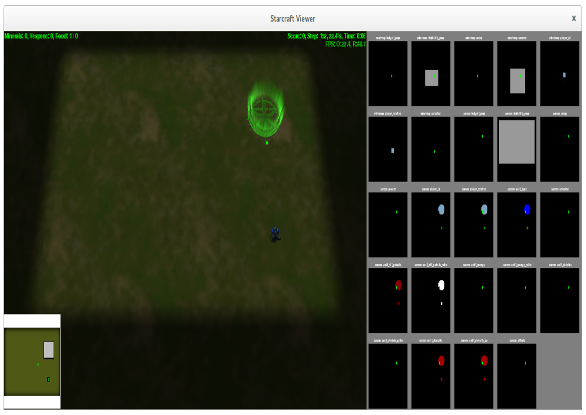
Future Work When implementing a hierarchical architecture with multiple levels, the most important step is to provide environments, where the agents can learn more elaborate skills in a curriculum learning way. That's why it is necessary to create new StarCraft II environments for learning new skills (based on already learned skills) and complex domains on which these skills can be tested properly. As soon as a suitable complex domain is available and the bug with the fault states is solved, it would be interesting to test the HDRLN agent.
More sophisticated implementation of RL bots in the field of video games have shown, that self play or evolutionary leagues can be used to efficiently find and improve agents with promising hyperparameter sets and policy. It would be an interesting task to set up these kind of leagues, where bots can compete against each other over many evolutionary cycles and to investigate if this procedure finds successful and robust agents faster than other approaches.
The more complex an environment becomes, the harder it is to define supervised mathematical modules, that generate expert data. Sometimes humans are the best way for an RL agent to train on early stages. It might be promising to implement new scripts, that make it possible to learn from human expert data, either through standard learning by filling the replay buffer with valuable transitions or with the help of policy distillation.

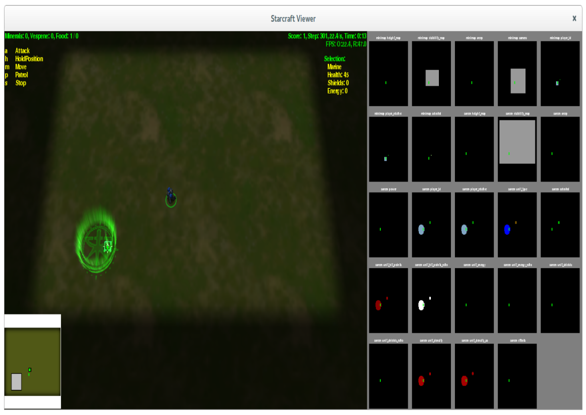
7 Appendix

7.1 Minigames

In this work, only 3 out of the 7 minigames were investigated:
MoveToBeacon,CollectMineralShards and DefeatRoaches (for a full description check Pysc2 Minigames).

7.1.1 MoveToBeacon

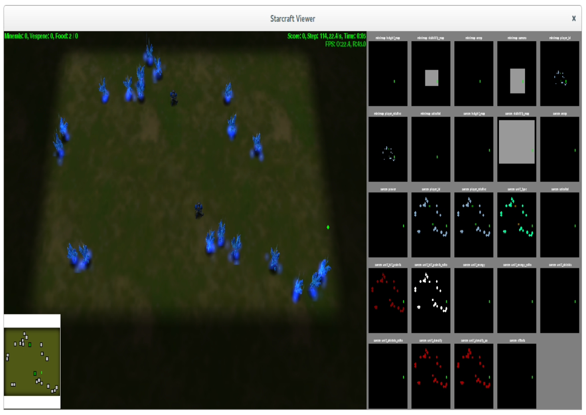
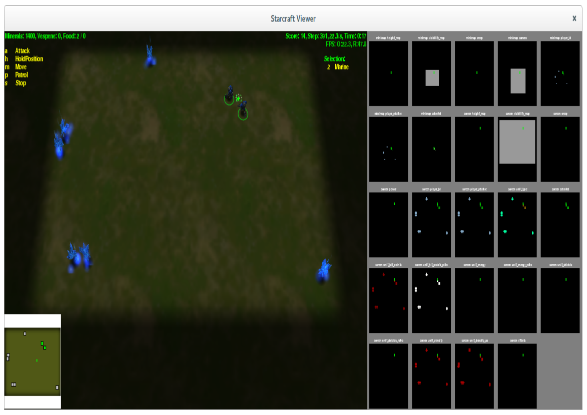




In this minigame, the agent controls one terran standard attack unit 'Marine' and the task at hand is to move the marine into the beacon, which results in a instantanious reward of $+1$. Whenever the marine enters a beacon, the beacon disappears and respawns at a random position on the map (at least five units away from the current position of the marine).
Overview:

- Initial State
  - 1 Marine at random location (not preselected)
  - 1 Beacon at random location
- Rewards
  - Entering the beacon: $+1$
- End Condition
  - Time step limit reached
- Time Step Limit
  - $\frac{1920}{step\_mul}$ timesteps
- Additional Notes
  - Fog of War disabled
  - No camera movement required

7.1.2 CollectMineralShards

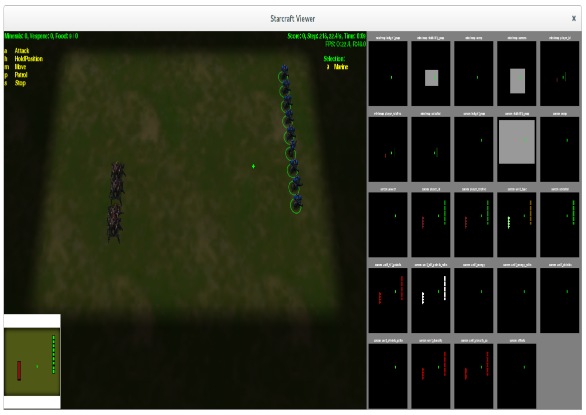



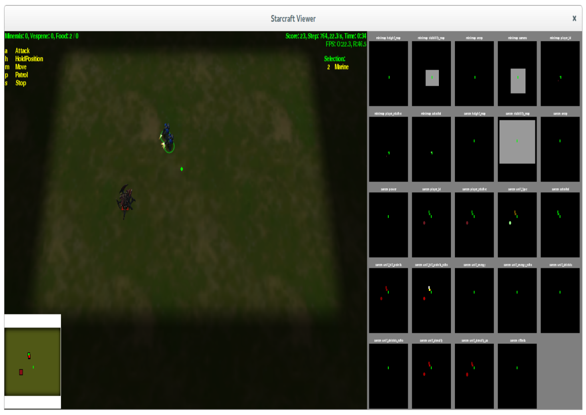
On this map, the agent controls 2 Marines and has to collect an endless supply of Mineral Shards. Rewards are earned by moving the marines over the shards. Whenever all 20 shards are collected, a new set of 20 shards is spawned at random locations but at least 2 units away from the marines.

Overview:

- Initial State
  - 2 Marines at random location (not preselected)
  - 20 Mineral Shards at random locations
- Rewards
  - Marine collects Mineral Shard: $+1$
- End Condition
  - Time step limit reached
- Time Step Limit
  - $\frac{1920}{step\_mul}$ timesteps
- Additional Notes
  - Fog of War disabled
  - No camera movement required
  - Needs the Wings of Liberty Campaign mod (was never an issue during this work)

7.1.3 DefeatRoaches





In this minigame, the agent starts with 9 marines and the opponent starts with 4 Roaches. Rewards are earned by killing the Roaches but a negative reward is gained whenever the agent looses one marine. The agent has to learn an optimal strategy to kill all roaches without loosing marines. One strategy would be to focus fire on one roach, kill it, focus fire the next and so on. Whenever the 4 roaches are defeated, the agent gets 5 additional marines at full health with all other surviving marines (they retain their old health status). Whenever new units are spawned, all unit positions are reset to opposite sides of the map.

Overview:

- Initial State
  - 9 Marines vertically aligned at random side of the map (preselected)
  - 4 Roaches vertically aligned on the opposite side of the map from the marines
- Rewards
  - Roach defeated: +10
  - Marine defeated: -1
- End Conditions
  - Time step limit reached
  - All Marines defeated
- Time Step Limit
  - $\frac{1920}{step\_mul}$ timesteps
- Additional Notes
  - Fog of War disabled
  - No camera movement required
  - Needs the Wings of Liberty Campaign mod (was never an issue during this work)
  - Units spawn out of combat range from each other

7.2 Gym Toyproblems

In this work, three different gym environments from the classic control family have been chosen because auf their overall similarity. The action space of all environments have been normalized, s.t. one action in the Acrobot environment would result in the same action in for example in the Cartpole environment. This had to be done to make the knowledge transfer possible.

7.2.1 Cartpole

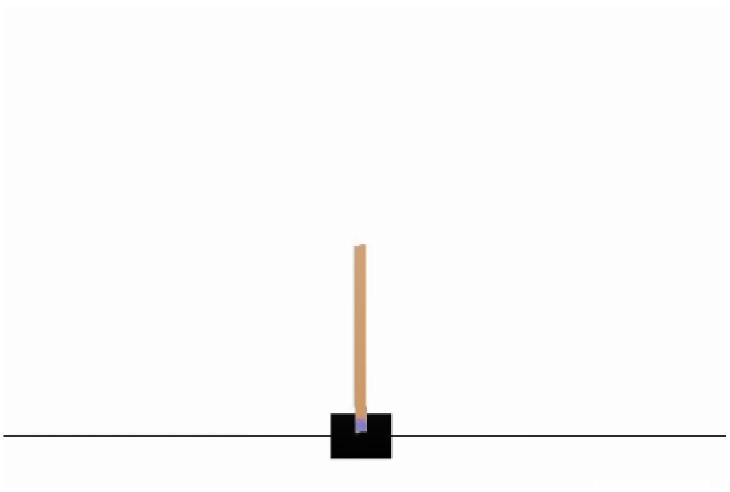


Figure 21: Cartpole gym environment

- Environment Id
  - Original: CartPole-v1
  - Custom (this work): gym-toy-cartpole-v0
- State space: [Cart Position, Cart Velocity, Pole Angle, Pole Velocity at Tip]
- Action space
  - Original: $[-1, 0, 1] \rightarrow$ [Push cart to left, NOP, Push cart to right]
  - Normalized: $[-1, 0, 1] \rightarrow$ [Push cart to left, NOP, Push cart to right]
- Goal: The cart should stay in a certain region and the pole should be prevented from falling over
- Reward: Reward is 1 for every step taken, including the termination step
- End conditions
  - Pole falls over
  - Cart leaves certain area
  - After 1500 timesteps

7.2.2 Acrobot

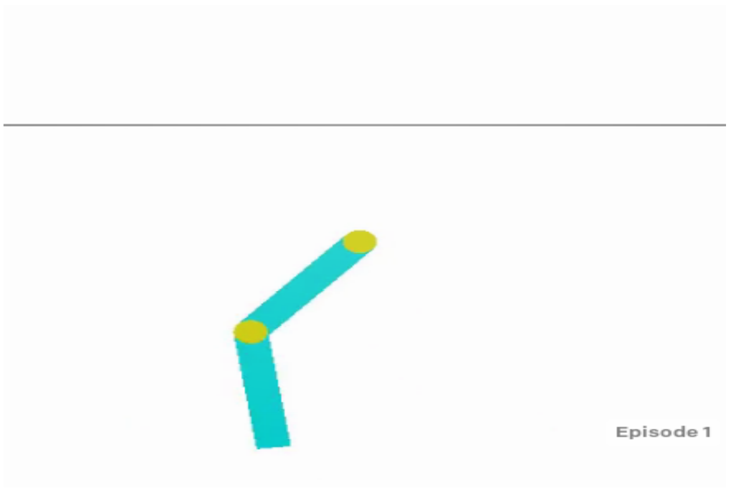


Figure 22: Acrobot gym environment

- Environment Id:
  - Original: Acrobot-v1
  - Custom (this work): gym-toy-acrobot-v0
- State space: $[\cos\theta_1, \sin\theta_1, \cos\theta_2, \sin\theta_2, \dot{\theta}_1, \dot{\theta}_2]$ where θ represents the angle of the joints and $\dot{\theta}$ represents the joint angular velocity.
- Action space
  - Original: $[-1,\ 0,\ 1] \rightarrow$ [Push middle joint to left, NOP, Push middle joint to right]
  - Normalized:
    $[-1,\ 0,\ 1] \rightarrow$ [Push middle joint to left, NOP, Push middle joint to right]
- Goal: swing the end-effector above the line by applying torque to on the middle joint
- Reward: -1 each step and 0 if state is terminal, i.e. end effector is above the line
- End conditions
  - After 1500 timesteps

7.2.3 Pendulum

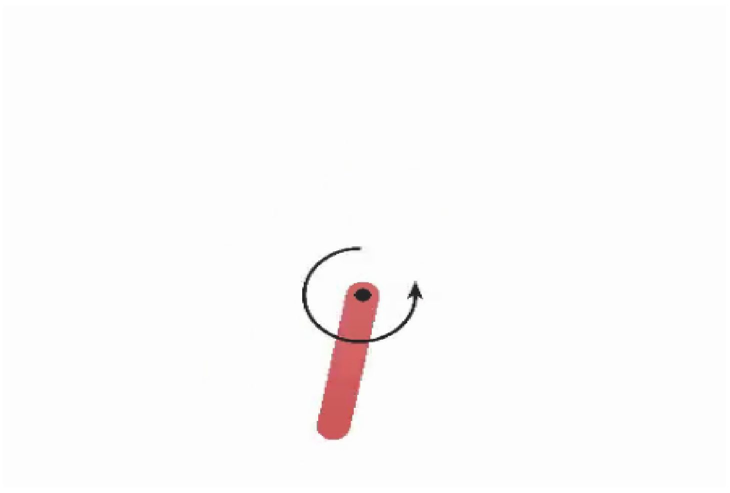
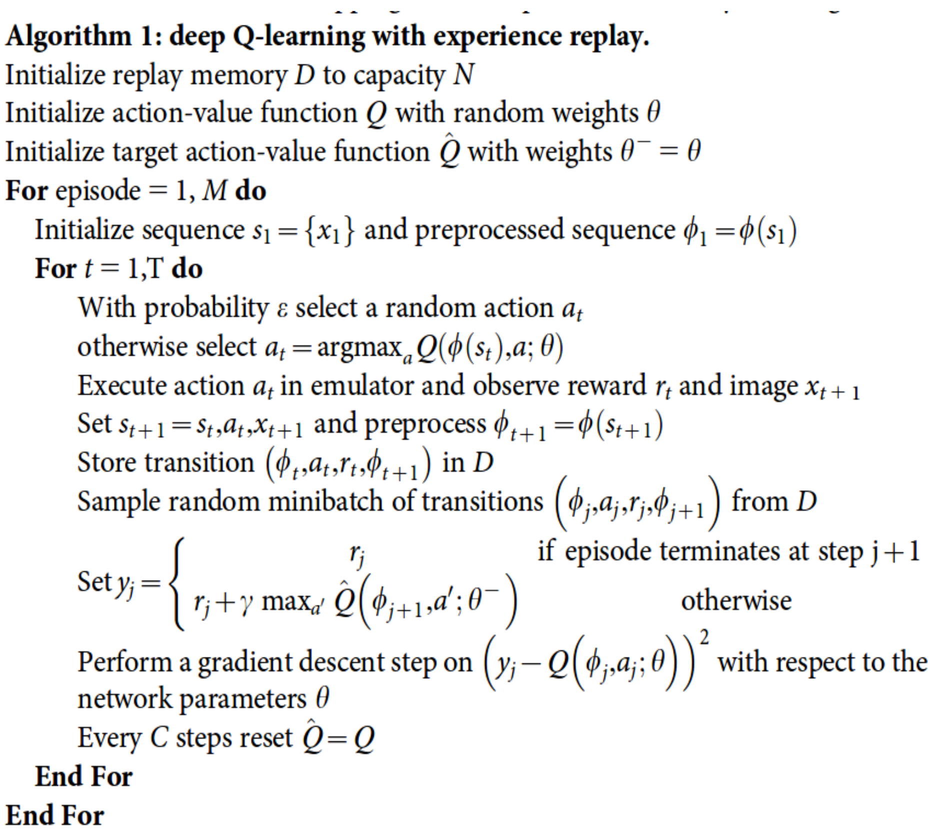


Figure 23: Pendulum gym environment

- Environment Id:
  - Original: Pendulum-v0
  - Custom (this work): gym-toy-pendulum-v0
- State space: $[\cos\theta, \sin\theta, \dot{\theta}]$ where θ represents the angle of the joint and $\dot{\theta}$ represents the joint angular velocity.
- Action space
  - Original: Continous action space
  - Normalized: $[-1,\ 0,\ 1] \rightarrow$ [apply torque CW, NOP, apply torque CCW]
- Goal: Swing the pendulum upwards and keep it upright by balancing
- Reward: $+1$ for every timestep the end-effector is in upright position
- End conditions
  - After 1500 timesteps

7.3 DQN-algorithm

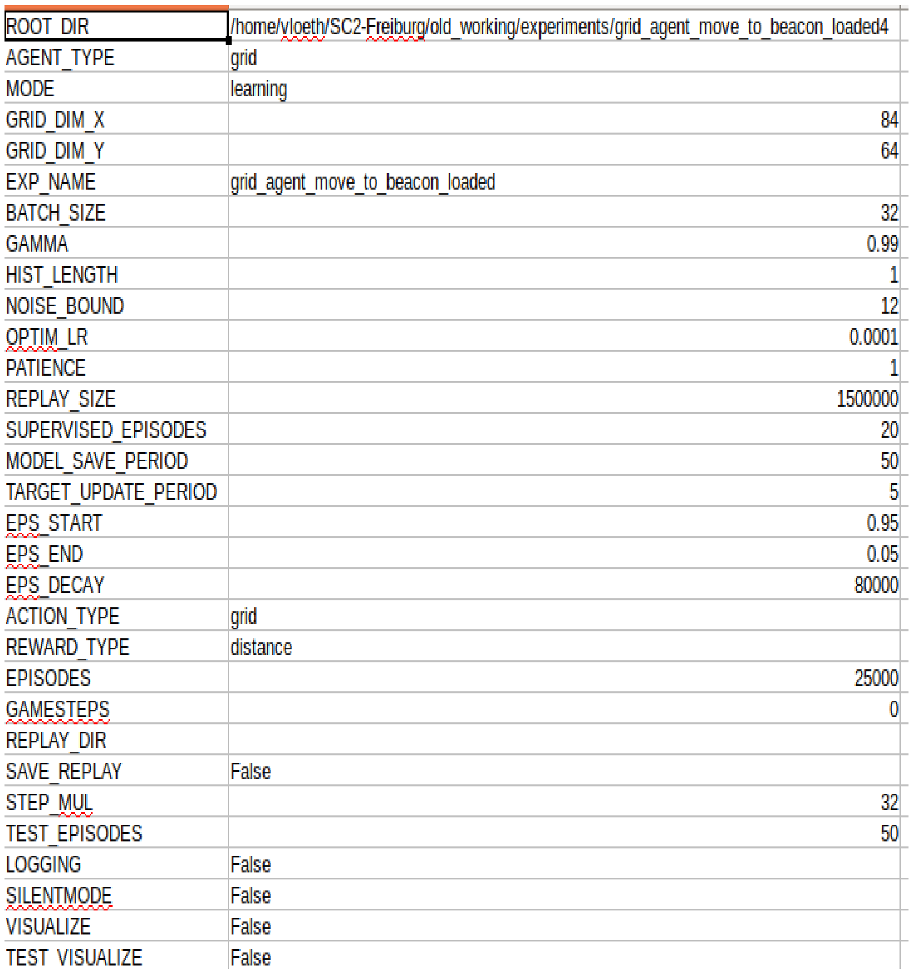


```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode $= 1, M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient 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 For
End For
```

Figure 24: Vanilla DQN from nature letter [13]

7.4 Spec Summary

The following figure shows the hyperparameter set of the spec summary (agent and enviroment parameters).



ROOT_DIR	/home/vloeth/SC2-Freiburg/old_working/experiments/grid_agent_move_to_beacon_loaded4
AGENT_TYPE	grid
MODE	learning
GRID_DIM_X	84
GRID_DIM_Y	64
EXP_NAME	grid_agent_move_to_beacon_loaded
BATCH_SIZE	32
GAMMA	0.99
HIST_LENGTH	1
NOISE_BOUND	12
OPTIM_LR	0.0001
PATIENCE	1
REPLAY_SIZE	1500000
SUPERVISED_EPISODES	20
MODEL_SAVE_PERIOD	50
TARGET_UPDATE_PERIOD	5
EPS_START	0.95
EPS_END	0.05
EPS_DECAY	80000
ACTION_TYPE	grid
REWARD_TYPE	distance
EPISODES	25000
GAMESTEPS	0
REPLAY_DIR	
SAVE_REPLAY	False
STEP_MUL	32
TEST_EPISODES	50
LOGGING	False
SILENTMODE	False
VISUALIZE	False
TEST_VISUALIZE	False

Figure 25: Example for spec summary and which hyper-parameters have to be set by the operator

7.5 Network architectures

The neural network architecture for the SC2LE minigames was chosen as follows:

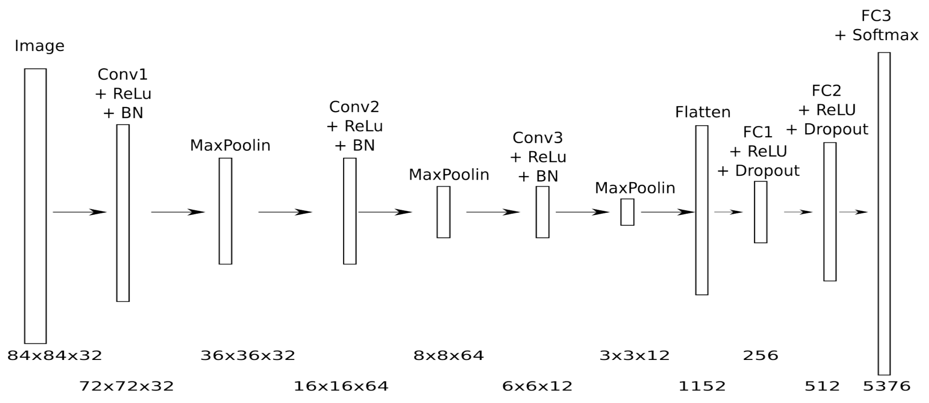


Figure 26: Network architecture for the SC2LE minigames (Full grid action space output layer)

The neural network architecture for the gym env toy problems was chosen as follows:

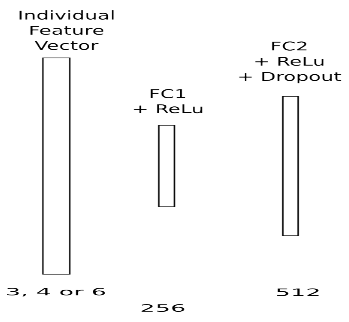


Figure 27: Network architecture for the gym toy problems

7.6 Test Routines and Hyperparameter Sets

7.6.1 Toy Problems

The experiment procedure was the same for every toy problem. The agent was trained on 3000 episodes. Every 300th epoch, the model of the agent was evaluated with a test run of 50 episodes. The performance progression of the model was investigated and the best model was used for a long evaluation test with 1000 episodes. A reward histogram was created to show, how the rewards of perfect runs, mediocre performances and outliers are distributed.
The same hyperparameter set has been used for all of the three toy problems:

discount factor	0.99
batch size	128
learning rate	0.0001
Epsilon start	0.95
Epsilon end	0.01
Epsilon decay	40000
Target update period	5
Model save period	50
Replay buffer size	2000000

Table 1: Hyperparameter set for the toy problems

7.6.2 Compass Agent

The agent was trained on 2000 episodes. Every 200th epoch, the model of the agent was evaluated with a test run of 50 episodes. The performance progression of the model was investigated and the best model was used for a long evaluation test with 1000 episodes. A reward histogram was created to show, how the rewards of perfect runs, mediocre performances and outliers are distributed. The following hyperparameter set has been used for the Move2Beacon compass agents:

discount factor	0.99
batch size	32
learning rate	0.0002
Epsilon start	0.95
Epsilon end	0.05
Epsilon decay	80000
Target update period	5
Model save period	50
Replay buffer size	1000000

Table 2: Hyperparameter set for the compass agent on Move2Beacon

7.6.3 SC2LE Grid Agent

The experiment procedure was similar for all the grid agents for the SC2LE environments. The agent was trained on 30000 episodes. Every 3000th epoch, the model of the agent was evaluated with a test run of 50 episodes. The performance progression of the model was investigated and the best model was used for a long evaluation test with 1000 episodes. A reward histogram was created to show, how the rewards of perfect runs, mediocre performances and outliers are distributed. The following hyperparameter sets have been used for the grid agents:

discount factor	0.99
batch size	32
learning rate	0.0001
Epsilon start	0.95
Epsilon end	0.05
Epsilon decay	80000
Target update period	5
Model save period	50
Replay buffer size	1500000

Table 3: Hyperparameter set for the grid agent on Move2Beacon

discount factor	0.99
batch size	128
learning rate	0.0001
Epsilon start	1
Epsilon end	0.1
Epsilon decay	1000000
Target update period	5
Model save period	500
Replay buffer size	1500000

Table 4: Hyperparameter set for the grid agent on CollectMineralShards

discount factor	0.99
batch size	32
learning rate	0.0001
Epsilon start	1
Epsilon end	0.1
Epsilon decay	80000
Target update period	5
Model save period	50
Replay buffer size	1500000

Table 5: Hyperparameter set for the grid agent on CollectMineralShards

7.6.4 Complex Domain

Our design for a complex domain based on the minigames is shown in the following figure. In the middle of the map an army of marines is spawned, similar to the DefeatRoaches domain. Around the spawn area, the 3 objectives (Beacon, Shards, Roaches) from the minigames are spawned at random locations. In one episode the beacon could be spawned somewhere in the top area, in another it could be spawned somewhere in another area. One of the four outer areas stays free to avoid abundant rewards if the agent just takes any action at all. If the agent successfully achieves all objectives, a new cycle is started.

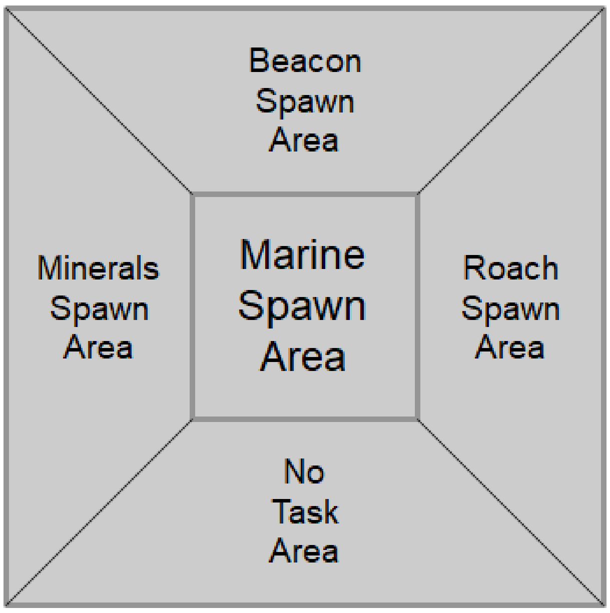
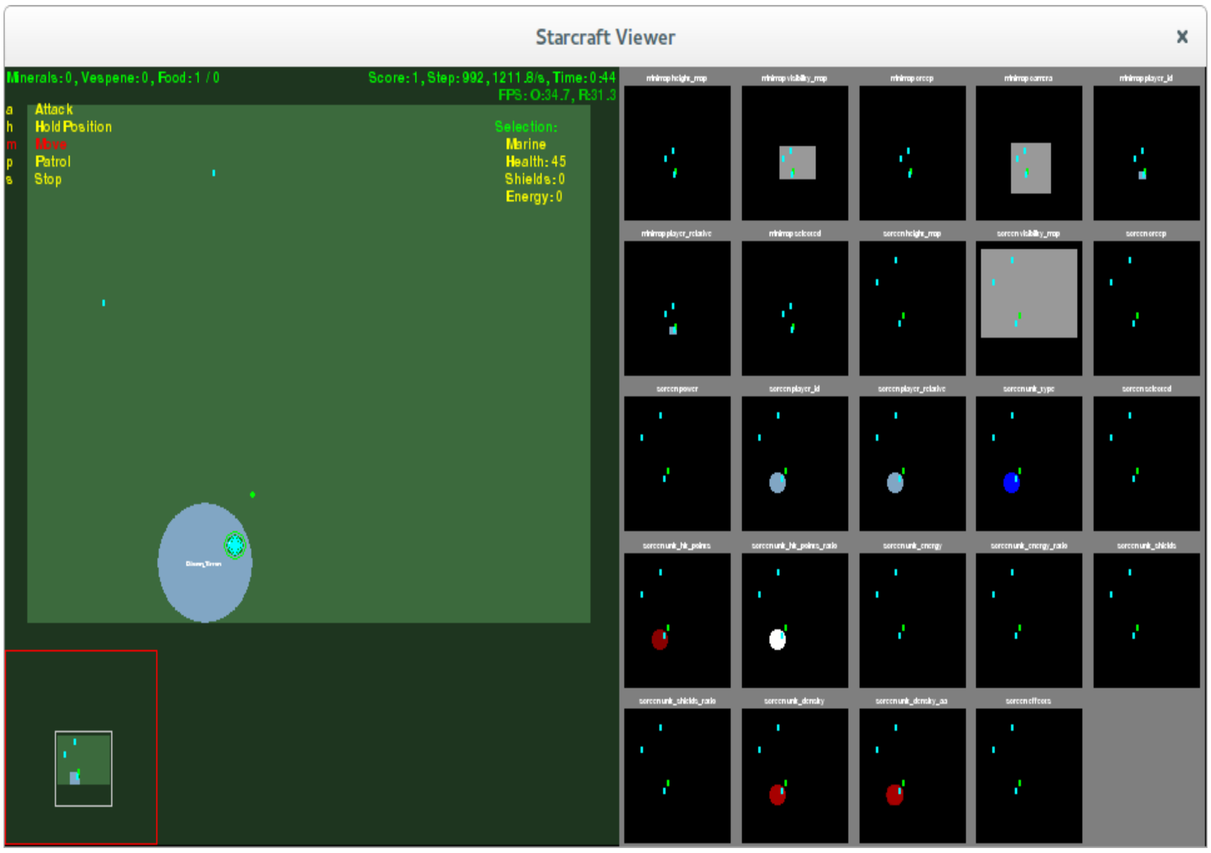


Figure 28: Design idea for the complex domain

A hierarchical agent can show its full potential in this environment, while making the use of skills non-trivial. We expect the agents optimal policy to first attack the Roaches and moving towards the mineral shards and the beacon while still attacking the Roaches. A hierarchical agent needs to learn a good balance between using the skills and primitive actions. Another possible solution would be that the agent splits up its army between the three tasks, which however could result in negative rewards, if one of the marine dies.

7.7 Bugs

The following figure shows that the agent sometimes was standing inside of the beacon but was not triggering the reward.



```
Minerals: 0, Vespene: 0, Food: 1 / 0          Score: 1, Step: 992, 1211.8/s, Time: 0:44
                                                              FPS: O:34.7, R:31.3
Selection:
  Marine
  Health: 45
  Shields: 0
  Energy: 0
```

Figure 29: Beacon bug

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[2] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[3] OpenAI. Openai five. `https://blog.openai.com/openai-five/`, 2018.

[4] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M. Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Yuhuai Wu, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[5] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[6] Roman Ring. Reaver: Modular deep reinforcement learning framework. `https://github.com/inoryy/reaver`, 2018.

[7] Zhen-Jia Pang, Ruo-Ze Liu, Zhou-Yu Meng, Yi Zhang, Yang Yu, and Tong Lu. On reinforcement learning for full-length game of starcraft. *CoRR*, abs/1809.09095, 2018.

[8] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, January 2003.

[9] Chen Tessler, Shahar Givony, Tom Zahavy, Daniel J. Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. *CoRR*, abs/1604.07255, 2016.

[10] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In Sven Koenig and Robert C. Holte, editors, *Abstraction, Reformulation, and Approximation*, pages 212–223, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[11] Daniel L. Silver, Qiang Yang, and Lianghao Li. Lifelong machine learning systems: Beyond learning algorithms. In *AAAI Spring Symposium: Lifelong Machine Learning*, volume SS-13-05 of *AAAI Technical Report*. AAAI, 2013.

[12] Dennis Lee, Haoran Tang, Jeffrey O. Zhang, Huazhe Xu, Trevor Darrell, and Pieter Abbeel. Modular architecture for starcraft II with deep reinforcement learning. *CoRR*, abs/1811.03555, 2018.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529 EP –, Feb 2015.