Reinforcement Learning in StarCraft 2

Franz Papst

March 29, 2018

1 Introduction

Recent advances in deep reinforcement learning have successfully mastered the game of Go [5] and are able to achieve super-human performance in classic Atari video games [2]. After this accomplishments, artificial intelligence researchers are looking for a new grand challenge. The computer real-time strategy game StarCraft 2 seems to be a promising domain for research on reinforcement learning algorithms, since it provides a problem set that is more challenging that what was done in prior work. E.g. it is a multi-agent problem with multiple players interacting, in an environment that only gives imperfect information and has a large action space [1].

This report is about my work, which is dealing with a subdomain for the whole problem set: it is about building an agent that is able to harvest resources as efficiently as possible. To solve this task I implemented an agent using the asynchronous advantage actor critic (A3C) algorithm [3] with TensorFlow. This report will firstly introduce the StarCraft 2 problem domain, by giving a detailed introduction into its environment. Then I will describe my agent, its architecture, the A3C algorithm and how I implemented this algorithm. Finally I will analyse the performance of the agent as well as its behaviour.

2 Environment

The pysc2 framework[1] allows StarCraft 2 agents to be written in Python, by providing a Python interface for the StarCraft 2 API[2]. This interface was build with reinforcement learning in mind and provides informations about the state of the current game, so it can be easily used in reinforcement learning algorithms. In the following three subsections I will introduce which observations are provided by the framework, how they are connected to states and what actions the agent can perform.

The information in this section is take from [6] and [7] or from reading the source code myself.

[1] `https://github.com/deepmind/pysc2` verified 2018-03-29

[2] `https://github.com/Blizzard/s2client-api` verified 2018-03-29

2.1 Observations

The pysc2 framework provides the following observations:

- `available_actions`
- `build_queue`
- `cargo`
- `cargo_slots_available`
- `control_groups`
- `game_loop`
- `minimap`
- `multi_select`
- `player`
- `score_cumulative`
- `screen`
- `single_select`

Probably the most important observations are `screen` and `minimap`, because the contain information about the units on screen, respectively on the game map. Both consist of different feature layers, as shown by figure 1. All feature layers are two-dimensional arrays, containing the information similar to the pixels in an image.

2.1.1 minimap

The minimap is a low resolution of the whole game map, it gives an overview of what's going on in the game, but with less detail. Initially it shows the whole map, but only the terrain including information about where to find resources, the position of the opponent is not shown, it is hidden by the so-called "fog-of-war"[3]. The minimap is a $(7, x, y)$ tensor, where x and y are the x- and y-resolution of the minimap. The first dimension represents the following features:

0. `height_map:` Shows the terrain level. It takes values [0,255], which gradually denote the height of the map, 0 being the bottom layer and 255 being the highest elevation of the map.
1. `visibility:` Which parts of the map are hidden, have been seen or are currently visible. It takes values [0,1,2] denoting [hidden, have been seen, currently visible]
2. `creep:` Which parts of the map are covered with Zerg creep[4]. It takes values [0,1] denoting denoting [no creep, creep].

[3]Fog-of-war means that only those parts of the game world are visible where the player has a unit. If no unit is there, the visibility of that area slowly fades out so that only buildings remain visible, but no units. Changes (e.g. newly build buildings) will not be shown, until the player send a new unit to explore. This mechanism of the game encourages the player to explore the map.

[4]One of the three races in the game, the Zerg, can only build buildings if the ground is covered with "creep". Only their command centre and the gas-extractor do not require to be build on creep. Creep can be extended by building special structures or by performing

3. `camera:` Which part of the maps is visible by the screen layer. It takes values [0,1] denoting [not visible, visible]
4. `player_id:` Which player owns the units. It takes values [0,16], where values from 0 to 15 denote the absolute player_id and 16 denotes neutral units.
5. `player_relative:` Which status the units have relative to the player. It takes values [0, 1, 2, 3, 4] denoting [background, self, ally, neutral, hostile].
6. `selected:` Which units are selected. It takes values [0,1] denoting [not selected, selected].

2.1.2 screen

The screen represents the visual/spatial features of the current game, similar to the `minimap`, showing only a part of the game map, but with a higher resolution. So far the screen is only represented through feature layers, but for future releases it is planned to add an RGB Pixel representation [1]. Just like for the `minimap` also here some details are hidden by the fog-of-war, if there is no player unit present at the part of the map which is currently shown by `screen`. The screen is a $(17, x, y)$ tensor, where x and y are the x- and y-resolution of the game and the first dimension represents the following features:

0. `height_map:` The terrain level, like to `height_level` from `minimap`.
1. `visibility_map:` The visibility of the map, like `visibility` from `minimap`.
2. `creep:` Which parts of the screen are covered with Zerg creep, like `creep` from `minimap`.
3. `power:` Which parts of the screen are supplied with power.[5]
4. `player_id:` Which player owns the units on screen, like `player_id` from `minimap`.
5. `player_relative:` Which status the units on screen have relative to the player, like `player_relative` from `minimap`.
6. `unit_type:` The unit ids for all units on screen, the unit id is an integer from 0 to 894.[6]
7. `selected:` Which units on screen are selected. It takes values [0,1] denoting [selected, not selected].

a special action for expansion. Zerg units move faster and regenerate when on creep, if a Zerg building is not surrounded by creep, it will take damage over time (apart from those buildings which don't need creep in the first place). This game-mechanism emphasizes the organic/insect-like nature of the Zerg, as well as adding another strategical dimension to the game, when playing as Zerg, since it restricts where the player can build structures.

[5] Analogous to the Zerg's creep, Protoss structures need to be build on a part of the game map, that is supplied with power. Pylons supply the surrounding fields with power, meaning that other structures can be build there. If a pylon gets destroyed, the surrounding structures are shutting down, meaning being unable to operate, unless they are in reach of another pylon, which supplies them with energy. Like the Zerg's creep this limits the places, where Protoss can put structures (just like for the Zerg, this limitation does not apply to a command centre and a gas assimilator and also not to the pylons themself), making it more difficult to expand.

[6] A list of unit ids can be found here: `https://github.com/Blizzard/s2client-api/blob/master/include/sc2api/sc2_typeenums.h` line 25ff., verified 2017-11-29.

8. `unit_hit_points:` The absolute hit points the units on screen have. It takes values from 0, denoting no unit, to whatever value is the current amount of hit points for units on screen.
9. `unit_hit_points_ratio:` The relative amount of hit points of the units on screen with respect to their maximum amount. It takes values from 0 to 255, denoting no unit/unit destroyed or a unit with 100% hit points.
10. `unit_energy:` The absolute amount of energy points units on screen have.[7] It takes values from 0, denoting no unit/energy to whatever value is the current amount of energy for units on screen.
11. `unit_energy_ratio:` The relative amount of energy, similar to `unit_hit_points_ratio`. It takes values from 0 to 255, denoting no unit/unit has no energy or a unit with 100% energy points.
12. `unit_shields:` The absolute amount of shield points units on screen have.[8] It takes values for 0, denoting no unit/shield points to whatever value is the current value of shield points for units on screen.
13. `unit_shields_ratio:` The relative amount of energy, similar to `unit_hit_points_ratio`. It takes values from 0 to 255, denoting no unit/shield points or a unit with 100% shield points.
14. `unit_density:` How many units are in this pixel.
15. `unit_density_aa:` Like `unit_density`, but anti-aliased with a maximum of 16 per unit per pixel. It shows how much of a pixel is covered by a unit, if multiple units are on a pixel it each proportion will be summed up to a value of maximum 256.
16. `effects:` Effects are the visualisation of an ongoing special action (e.g. healing Terran units). It takes integer values from 0 to 3687.[9]

2.1.3 player

A (11) tensor showing general information about the player, giving the following informations:

0. `player_id:` The ID of the player, an integer from 0 to 15.
1. `minerals:` Current count of minerals.
2. `vespene:` Current count of vespene gas.
3. `food_used:` The current amount of supply used[10].
4. `food_cap:` The current maximum number of supply.
5. `food_army:` How much of the supply is used for army units.
6. `food_workers:` How much of the supply is used for worker units.

[7] Note that only some units have energy points, those are used to perform special actions.

[8] Note that only Protoss units have shields.

[9] A list of effect ids can be found here: `https://github.com/Blizzard/s2client-api/blob/master/include/sc2api/sc2_typeenums.h` line 388ff., verified 2018-03-29.

[10] Besides minerals and gas, food, also known as supply, is the third kind of resource. It limits how many units a player can have in total, if the current supply cap is reached no more new units can be produced. The maximum amount of supply is 200, but this does not mean, that every player can have maximum 200 units, since stronger units require more than 1 supply, the most supply one unit can use is 8. In order to raise this supply cap, the player has to build special buildings/units depending on which race he plays.

7. `idle_worker_count:` How many of the workers are currently idle.
8. `army_count:` Current number of all army units.
9. `warp_gate_count:` Current number of warp gates, only for Protoss.
10. `larva_count:` Current number of larva, only for Zerg.

2.1.4 single_select

A (7) tensor showing information about the selected unit:

0. `unit_type:` The type of the unit.
1. `player_relative:` Which status of the unit, relative to the player. It takes values [0, 1, 2, 3, 4] denoting [background, self, ally, neutral, hostile].
2. `health:` The current health points of the unit as absolute number.[11]
3. `shields:` The current shield points of the unit as absolute number.[12]
4. `energy:` The current amount of energy points of the unit as absolute number.[13]
5. `transport_slot:` If the unit is transported, the amount of transport slots taken.
6. `build_progress:` If the unit is being build, the percentage of how far the building process is.

2.1.5 multi_select

A $(n, 7)$ tensor, similar to single_select, but for n selected units.

2.1.6 build_queue

A $(n, 7)$ tensor, similar to `single_select`, but for all units that are in the build queue of a production building, where n is the number of units in the build queue.

2.1.7 cargo

A $(n, 7)$ tensor, similar to `single_select`, but for n units that are in a transporter.

2.1.8 control_groups

A $(10, 2)$ tensor showing the unit leader type and count for all 10 control groups. A control group of units is a group that is mapped to a certain hot-key (0-9) in order to quickly access them.

[11] Note that just from this value it is not evident, if the unit is damaged or not, this information can be derived from the `unit_hit_points_ratio` layer from the `screen` tensor.

[12] Note that similar to `health` it does not give information about the ration of the shield to its maximum value, this information can be derived from the `unit_shields_ratio` layer from the `screen` tensor.

[13] Note that similar to `health` it does not give information about the ration of the shield to its maximum value, this information can be derived from the `unit_energy_ratio` layer from the `screen` tensor.

2.1.9 available_actions

A (n) tensor listing all actions that are available at the time of this observation. The amount of total actions is quite high, but not all actions are available at any given point. The number n of actions that are available at a given point in time is actually quite low. Which actions are available depends on what (if a) unit is selected: Some units have a lot of actions, others less (e.g. buildings have in general less). But there are some general actions (like moving the screen), which are always available.

2.1.10 score_cumulative

A $(13,)$ tensor showing the "Blizzard score". The Blizzard score is usually presented to the player after the game, showing specific values of how he played, in order to judge how good it was:

0. `score:` The total Blizzard score for the current step, the higher the better.
1. `idle_production_time:` Sum of the time where the player's production is stuck, because of reaching the supply cap, the lower the better.[14]
2. `idle_worker_time:` Sum of the time the player's worker units spend doing nothing, the lower the better.
3. `total_value_units:` Total sum of the value of all units the player built (value does not get decreased if a unit gets destroyed), the higher the better.
4. `total_value_structures:` Total sum of the value of all structures the player built (value does not get decreased if a structure gets destroyed), the higher the better.
5. `killed_value_units:` Total sum of all destroyed enemy units[15], the higher the better.
6. `killed_value_structures:` Total sum of all destroyed enemy buildings[16], the higher the better.
7. `collected_minerals:` Total amount of collected minerals, the higher the better.
8. `collected_vespene:` Total amount of collected gas, the higher the better.
9. `collection_rate_minerals:` Current rate of mineral collection (minerals per minute), the higher the better.
10. `collection_rate_vespene:` Current rate of gas collection (gas per minute), the higher the better.
11. `spent_minerals:` The total amount of minerals spent by the player, here it really depends which is better, in general the higher the better, since it

[14] Note that for Zerg this value does not have any meaning, since the way it's calculated it's always increasing.

[15] Note that the value is not the same as for the `total_value_units` fields: `https://blizzard.github.io/s2client-api/sc2__score_8h_source.html`, line 109f., verified 2018-03-29.

[16] Note that the value is not the same as for the `total_value_structures` fields: `https://blizzard.github.io/s2client-api/sc2__score_8h_source.html`, line 109f., verified 2018-03-29.

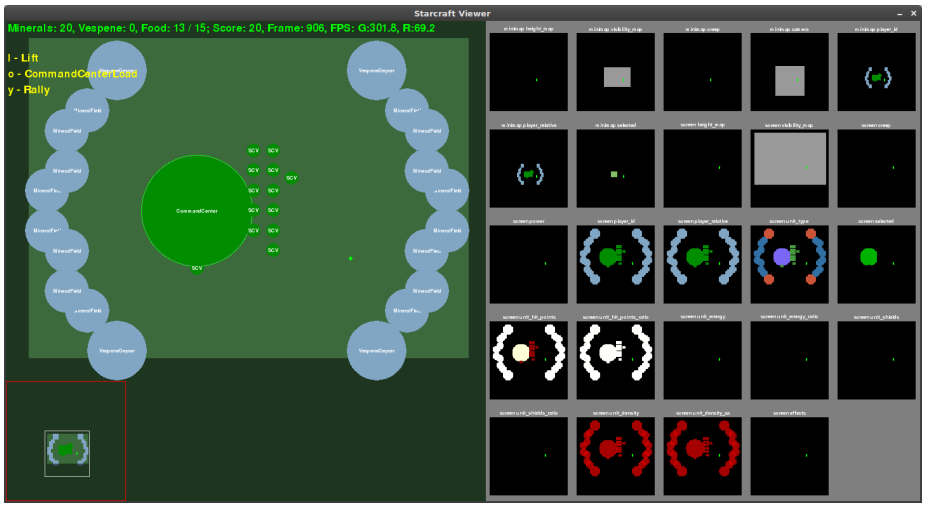

Figure 1: Graphical representation of the different feature layers in the pysc2 framework.

correlates with a bigger, more advanced army and more workers (better economy).

12. `spent_vespene:` The total amount of gas spent by the player, here it really depends which is better, in general the higher the better, since it correlates with a bigger, more advanced army.

2.1.11 game_loop

A scalar that gives the number of the current iteration in the game loop.

2.1.12 cargo_slots_available

A scalar that indicates how many cargo slots are available in a selected transporter.

2.2 States

As the exhaustive description of all observations during a single game step shows, the combination of possible states is huge. Therefore only the combination of a few features qualifies to become states for the reinforcement learning algorithm. What makes the whole situation even more difficult is the fact that the action space is in general continuous, a large number of actions needs x- and y-coordinates where it's performed.

To reduce the agent's complexity, I will only use one of the three races for the beginning. I chose the Terrans, the humans, since they are usually the race

used to teach the basics of StarCraft II. Unlike the other two races, they have no restrictions where they can build their structures.

I will use following:

- `minimap`
  - `visibility`
  - `camera`
- `screen`
  - `player_relative`
  - `unit_type`
- `player`
  - `minerals`
  - `vespene`
  - `food_used`
  - `food_cap`
  - `food_workers`
  - `idle_worker_count`
- `score_cumulative`
  - `score`
  - `idle_worker_time`
  - `collected_minerals`
  - `collected_vespene`
  - `collection_rate_minerals`
  - `collection_rate_vespene`
- `available_actions`

2.3 Actions

As mentioned above, the action space is huge, since it is continuous. But also the number of different actions is rather large (in total there are 524 different actions). The framework tackles this issue by only allowing the agent to execute valid actions, meaning limiting the agent to only execute actions that are currently available. The currently available actions can be taken from the `available_actions` tensor. To keep things easy in the beginning only the following 15 actions will be used:

`no_op`: Do nothing. It requires no target position.

`move_camera`: Moves the camera, so it is centred around the target position. It requires a target position on the minimap.

`select_point`: Selects what is at the target position (the action is also executed, if there is nothing at the target position, in that case nothing is selected). It requires a target position on the screen.

`select_idle_worker`: Selects an idle worker (more or less randomly). It requires no target position.

`Build_CommandCenter_screen":` Builds a command centre. It requires a target position on the screen.

`Build_Refinery_screen:` Builds a refinery for gas on the screen, note that a refinery can only by build on a vespene geyser. It requires a target position on the screen.

`Build_SupplyDepot_screen":` Builds a supply depot. It requires a target position on screen.

`Harvest_Gather_screen:` Sends a worker unit to collect resources. It requires a target position.

`Harvest_Return_quick:` Makes a resource collecting worker unit to return the currently collected resources to base immediately. It requires not target position.

`Morph_SupplyDepot_Lower_quick:` Lowers a supply depot, so it doesn't block units from passing the field it is build on. It requires no target position.

`Morph_SupplyDepot_Raise_quick:` Raises a previously lowered supply depot (when constructed supply depots are always raised), so it does prevent units from passing the field it is build on. It requires no target position.

`Move_screen:` Moves units to the given position on the screen. It requires a target position on the screen.

`Move_minimap:` Moves units to the given position on the minimap. It requires a target position on the minimap.

`Rally_Workers_screen:` Sets a rally point for workers on the screen. It requires a target position on screen.

`Rally_Workers_minimap:` Sets a rally point for workers on the minimap. It requires a target position on the minimap.

3 Agent

3.1 Architecture

Figure 2 gives a schematic overview of the agent's architecture. The features of screen and minimap are extracted using two convolutional neural networks each. As mentioned above three features from the screen and two from the minimap are used. The output of the last convolutional neural networks is concatenated with the the oupput of a fully connected neural network that was fed with the non-spatial inputs described above which gives the state representation. The state representation fed into a another convoluted neural network that gives the coordinates for a spatial action. The state representation is also fed into another fully connected neural network, which gives value of the current state, the same fully connected neural network is fed into another fully connected network that gives the non-spatial action.

The output of all neural networks is encoded in one-hot encoding, also the one that gives the coordinates for a spatial action. In this case the output layer of the neural network has a size of $x \cdot y$ and the coordinates for X and Y are calculated by getting the position of the highest output and performing an

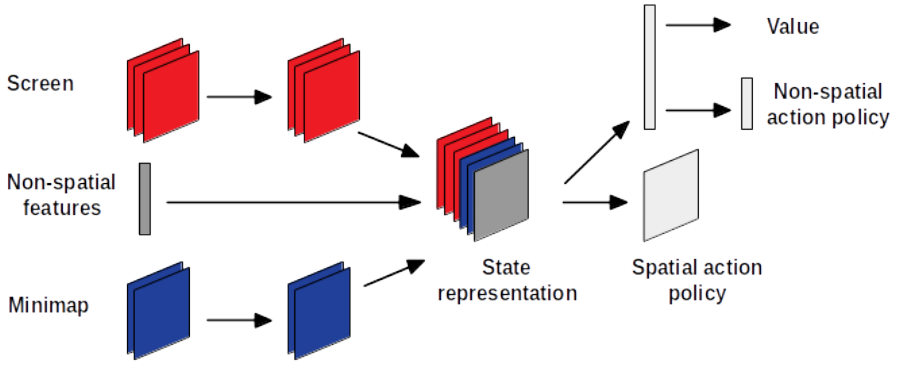

Figure 2: Schematic overview of the agent's architecture.

integer division by the size of y for the Y-coordinate and a modulo by the size of x for the X-coordinate.

The first convolutional neural network for the screen and the minimap has 16 output filters, a kernel size of 5 and a stride of 1, the second one has 32 output filters, a kernel size of 3 and a stride of 1. The fully connected neural network for the non-spatial features has 256 outputs with tanh as activation function , the fully connected neural network for the state representation has 256 outputs with relu as activation function, the fully connected neural network for the non-spatial actions hast the number of available actions as output[17] with softmax as activation function. The fully connected network for the value function has only one output and no activation function, since I am only interested in the value it returns.

The agent uses TensorFlow 1.5 as library for machine learning, the convolutional and fully connected neural networks are from the `contrib.layers` sub-module, `contrib` features volatile and experimental code, `contrib.layers` features more higher level operations for building neural networks.

In order to be able to run the algorithm on a regular laptop[18], a few tweaks and reductions in complexity of the environment had to be made. The size of the screen and the minimap was reduced to 32×32. The agent was run in 16 parallel instances.

[17] Which is 15, as described above.

[18] The agent was run on my laptop, which has an i7-7700HQ CPU, 16GB RAM, a GeForce GTX 1050 with 4GB of VRAM.

3.2 Algorithm

The asynchronous advantage actor critic algorithm (A3C) [3] was used as reinforcement learning algorithm. Its advantages are its simplicity, its efficiency as well as its performance. It is relative easy to implement and resource friendly, meaning that it doesn't require specialised hardware like GPUs to run efficiently[19], but it still outperforms deep Q-learning on the Atari-domain and also succeeds in various continuous motor control tasks [3]. Previously deep reinforcement learning algorithms had problems with stability, deep Q-learning solved this problem by introducing a replay buffer [2], replay memory has the drawback of being quite resource consuming per real interaction Asynchronous approaches like A3C solve the problem of stability by introducing a parallel approach, the parallel execution and weight updates of different agent instances keeps the system stable.

A3C maintains a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V(s_t;\theta_v)$. In terms of actor-critic the policy is the actor and the estimate of the value function is the critic. Both get updated after every t_max steps performed by the agent[20]. The update of the neural network's weights can be described as $\nabla\theta' = log\pi(a_t|s_t;\theta')A(s_t,a_t;\theta,\theta')$, where $A(s_t,a_t;\theta,\theta')$ is an estimate of the advantage function given by $\sum_{i=0}^{k-1}\gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta_v) - V(s_t;\theta)$, where k is the number of the current state [21] and is limited by t_{max}. The pseudocode for the algorithm is shown in algorithm 1.

3.3 Implementation of the Algorithm

3.3.1 Agent Step

The algorithm for updating the weights is described in section 3.2. In every step the agent is fed with the above mentioned features from the screen, minimap and non-spatial features, in addition to the non-spatial features from the environment also whether the current availability from the executable actions is fed as input. The neural networks return the action to perform and the location where it should be performed. The output gets filtered, so that only a valid action (an action that the agent is currently able to do gets returned) gets passed on to the environment. During training there is a certain probability that the performed action is random or that the position where it performed is random. This level of randomness decreases depending on how many steps the agent has performed. The probability if an action is can be computed using this formula:

$$P(a) = 1 - \frac{num\_steps + (1 - \epsilon_{explore}) \cdot T_{max}}{T_{max}} \tag{1}$$

Where $\epsilon_{explore} = 0.4$ and $T_{max} = 600 \cdot 10^6$ (the maximum amount of steps that the agent makes). In addition to this adaptive probability there exists

[19] When run on such specialised hardware it of course drastically speeds up the computation.
[20] In my implementation I perform an update after an episode finishes, so $t_{max} = 840$.
[21] In my case this is equal to the number of the current step.

Algorithm 1 Pseudocode for the asynchronous advantage actor-critic algorithm, taken from [3]

//Globally shared parameters: θ, θ_v and counter $T = 0$
//Thread specific parameters: θ' and θ'_v
Initialise thread step-counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta \leftarrow 0$.
  Synchronise thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t or $t - t_{start} == t_{max}$
  $$\mathrm{R} = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{//Bootstrap from last state} \end{cases}$$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients θ'_v: $d\theta_v + \frac{\partial (R - V(s_i;\theta'_v))^2}{\partial \theta'_v}$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_max$

a second chance for an action to be random: an ϵ-greedy exploration where $\epsilon = 0.05$. This was implemented so that the agent keeps exploring, even when the adaptive exploration probability reaches 0, which happens after $240 \cdot 10^6$ steps. The total probability of a random action is:

$$P(total) = P(a) + P(\epsilon) \tag{2}$$

In the beginning (when $num\_steps = 0$) this would be $0.4 + 0.05 = 0.45$. The possibility of an position being random is computed using the same formula.

All the performed actions (with their positions) are stored, as well as the state for each step of the agent. The saved actions and states will be used for the training of the neural network as well as for logging the results of an episode with the actions the agent performed in this episode. In order to keep the size of the log files reasonable, only the actions for the top 10 episodes for minerals and gas, as well as the last 10 episodes are kept for every instance of the agent.

3.3.2 Update of the weights

After an episode finished, the weights of the neural networks are updated. The updates are done asynchronously, where each agent instance is contributing to the weights of the neural networks.

As described in section 3.2, first the cumulated rewards are calculated. For that list of rewards is initialised with either 0, when the episode reached a terminal state or the reward is bootstrapped from the last state. From that on the lists of performed actions and states are reversed and the rewards get cumulated by adding the sum of the last rewards multiplied by the discount factor[22]. By doing so the earlier the action took place, the higher the reward it gets assigned, given that the agent keeps on doing actions that give rewards. If the agent performs actions that decrease the rewards, also the reward decreases. The reward function is explained in section 3.

The optimiser for updating the weights is the RMSPropOptimizer [4] with $\epsilon = 0.1$ and $decay = 0.99$, the learning rate is variable and decreases the more steps the agent has performed. It can be calculated using the following formula:

$$\eta_0 \cdot \left(1 - 0.9 \cdot \frac{num\_steps}{T_{max}}\right) \tag{3}$$

Where $\eta_0 = 10^{-3}$ and $T_{max} = 600 \cdot 10^6$. The learning rate is fed into TensorFlow like an input for a neural network.

The other inputs that are fed are the states for each single step (screen, minimap and non-spatial features), the rewards for each single step, whether the selected action is spatial, which position was selected, which non-spatial actions are valid for the current state (this depends on which units are currently

[22]The discount factor γ is set to 0.99, because I am interested in the long term results of an action.

selected), which non-spatial action is selected and the learning rate. The variables for whether an action is spatial and valid non-spatial actions are used to create a mask for the computations of the logarithmic probability of an action.

One episodes consists of 840 steps, in which actions are performed[23] In order to not need more memory for the updating of the weights than my GPU provides, I had to split the input data into 20 batches, where each includes number of 42 states, used as input.

After every 500 episodes, once the updating of the weights is done, the current weights are saved to hard disk.

3.3.3 Reward Function

The reward function for each step is:

$$r_i = minerals \ + \ gas \ \cdot \ 10 \ + \ minerals\_rate \ \cdot \ 10 \ + \ gas\_rate \ \cdot \ 100 \quad (4)$$

The variables *minerals* and *gas* donate the totally collected amount of minerals and gas, *minerals_rate* and *gas_rate* denote the current collection rates of minerals and gas.

I gave gas a higher value than minerals, because minerals are rather straight forward to collect (the agent only has to select a worker unit and give the harvest command on a mineral field), while collection gas is much more complicated (the agent has to select a worker unit and then give a command to build a refinery on a specific field where a geyser is, in order to increase the collection rate of gas the agent can send more worker units to the build refinery).

I also decided to multiply the collection rate of both resources by the factor 10 so that the collection rates are having significant influence on the total reward, also when the total collected amount of resources is already high.

4 Results

4.1 Rewards

Figure 3 shows all the gained rewards for every episode as well as the smoothed average of the rewards. Both plots show a very steep increase in the gained after around 15000 episodes. This lead to a plateau at a reward of around 150000 from around 17000 episodes to around 30000 episodes, where rewards were rather high in average with a few very high outliers. From 30000 episodes to 35000 episodes the average rewards were constantly decreasing, with two spikes, due to very high or rather high rewards. Around 36000 episodes there is another spike in the average rewards, that is around 100000. From that on the rewards reach another plateau at around 40000 till the end. Looking at the plot of all rewards shows that there is a high variance in gained rewards.

[23]The execution of a map stops after 6720 steps, so for the environment it is actually 6720 steps, but since the agent only executed every 8th step, it only performs 840 actions.

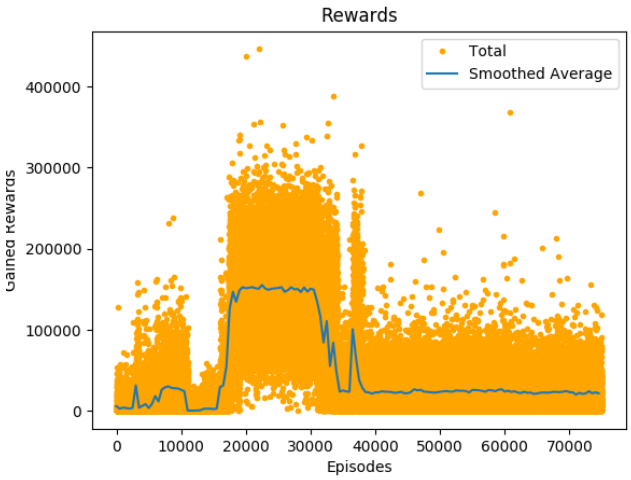

Figure 3: Gained rewards of the agent. The orange dots show the rewards of each single episode, the blue line shows the average of 500 episodes. The agent run for 75000 steps.

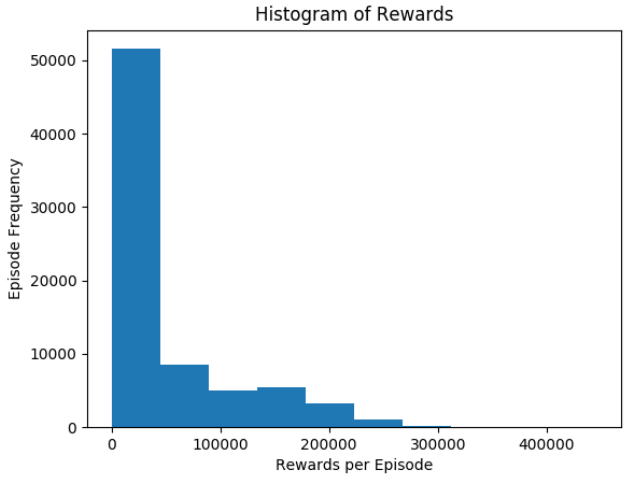

Figure 4: Histogram of rewards per episode, the histogram has 10 bins, with a size of 44606.4 each.

Episode	Reward	Minerals	Gas
22104	446064.0	2345	468
19972	437483.9	1775	556
33434	388482.7	2375	432
60775	368278.0	1410	644
22227	356823.1	1315	452
32624	355436.2	1545	500
21229	354093.9	2590	164
25682	352979.6	2975	0
19094	340669.9	2930	0
32586	339692.5	3095	0

Table 1: The top 10 episodes in terms of rewards.

Figure 4 shows that the vast majority of episodes is in the first bin, meaning that they had a reward from 0 to 44606.4, this also shows that the agent was stuck in a plateau where the reward was around 40000 for most of its runtime, more than 50000 episodes fall in this bin and only had a reward between 178425.6 and 223032 than between 133819.2 and 178425.6, which is not apparent from the plot of all rewards.

Table 1 shows the top 10 episodes in terms of rewards. The top 7 episodes are episodes which are a combination of relatively high amounts of collected gas and high amounts of collected minerals, while in the last three episodes only minerals had been collected. The reason why the episode with a higher mineral count is because for the log files I use the last reward and not a sum of all rewards in an episode. By doing so, it is also visible if an agent performed actions that led to a decrease of the reward. This happened e.g. in episode 32586, where the agent started collecting minerals quickly, but then performed actions that didn't contribute to the increase of the reward and thus leading to a lower reward even though the total amount of collected minerals is higher than for e.g. episode 19094. Note that the final reward is only used for logging, for training the agent all cumulated discounted rewards that the agent receives are used.

4.2 Minerals

Figure 5 looks very similar to figure 3, indicating that the collected minerals are mostly responsible for the gained rewards. Also around 15000 episodes there is a steep rise in the amount of collected minerals, to slightly less than 2500 collected minerals per episode. As for the gained rewards, the agent stays at this plateau until around 30000 episodes, until it steadily decreases to around 700 collected minerals per episode until the end, there is also a spike around 36000 episodes, where it goes up to around 1000 collected minerals per episode. Around 59000 episodes there is also a small spike up to 800 collected minerals per episode, which does not have much influence on the gained rewards. Note

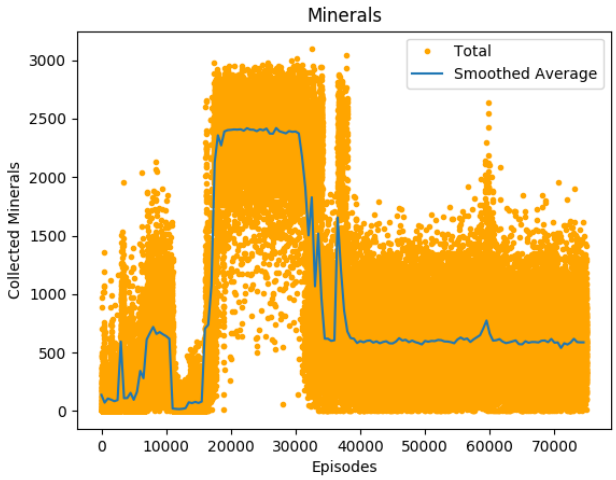

Figure 5: Collected minerals of the agent. The orange dots show the rewards of each single episode, the blue line shows the average of 500 episodes. The agent run for 75000 steps.

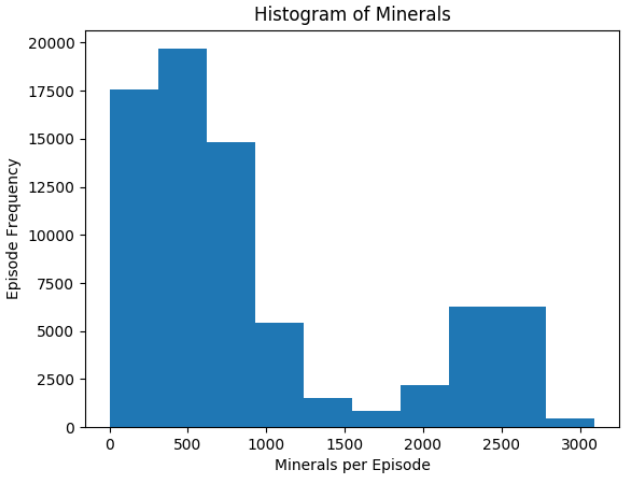

Figure 6: Histogram of collected minerals per episode, the histogram has 10 bins with a size of 309.5 each.

Episode	Reward	Minerals	Gas
32586	339692.5	3095	0
37906	327749.7	3045	0
30191	277834.5	3020	0
29342	337969.1	2995	0
17437	261376.8	2980	0
25682	352979.6	2975	0
30978	258900.1	2965	0
24885	256661.0	2960	0
30162	334053.1	2960	0
26006	316695.7	2955	0

Table 2: The top 10 episodes in terms of collected minerals.

that here the variance is also very high, but around the one high plateau, there are almost no very low outliers, meaning that during this plateau the agent practically collected minerals during every episode.

Figure 6 has also some similarities to figure 4, but there is one obvious difference: the bin with the most episodes in is from 309.5 to 619, so there are more episodes where the agent collected at least 309.5 minerals than where it collected no minerals. Also the plateau of where the agent collects between 2476 and 2785.5 minerals per episode for around 14000 is easily visible in the histogram. This influences the slight spike in the histogram of the rewards, although since the influence of collected minerals on the total reward is damped, it is not so visible in the rewards histogram.

Table 2 shows the top 10 episodes in terms of collected minerals. As mentioned above also here a higher amount of collected gas does not necessarily mean a higher amount of gained reward. It is notable that even though most of the top 10 episodes happened when the agent reached the highest plateau, the three episodes with the most collected minerals happened when the agent left this plateau and the are responsible for the spikes in the figure 5. Episode 17437 might be responsible for the steep increase in gained rewards and collected minerals that is visible in figure 3 and figure 5, together with episodes 19972 and 19094 as shown in table 1 because they gave very high rewards early in the agent's runtime.

4.3 Gas

Figure 7 shows how much gas an agent collected per episode, was well as a smoothed average. The variance is even higher than for rewards or collected minerals. The smoothed average is very low and usually around 0, but there are some very high outliers. The smoothed average is oscillating between 0 and 20 gas collected per episode. This indicates that the agent has not learned how to collect gas and that the higher amounts of collected gas are the product of the random exploration. Looking at figure 8, the histogram of collected gas,

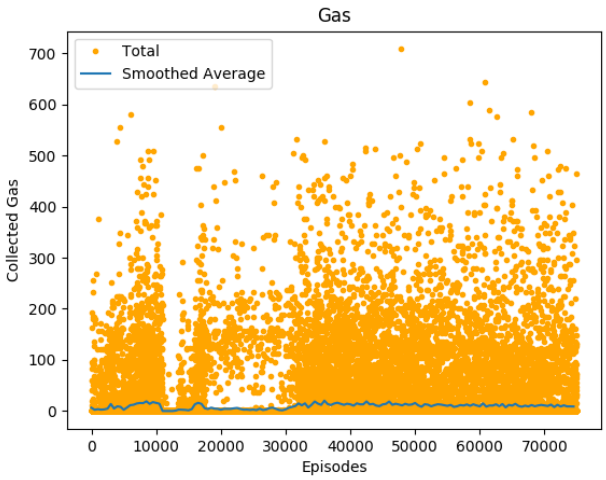

Figure 7: Collected gas of the agent. The orange dots show the rewards of each single episode, the blue line shows the average of 500 episodes. The agent run for 75000 steps.

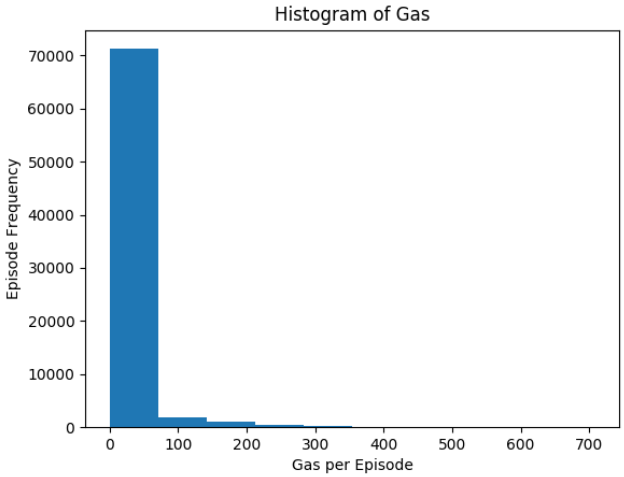

Figure 8: Histogram of collected gas per episode, the histogram has 10 bins with a size of 70.8 each.

Episode	Reward	Minerals	Gas
47860	152955.0	760	708
60775	368278.0	1410	644
18985	334472.5	1595	636
58515	244735.3	1100	604
61472	187292.0	665	588
67944	213293.6	915	584
6113	106560.4	480	580
62611	160610.8	355	576
19972	437483.9	1775	556
4370	127439.7	640	556

Table 3: The top 10 episodes in terms of collected gas.

strengthens this assumption: over 70000 episodes collected between 0 and 70.8 gas.

Table 3 shows the top 10 episodes in terms of collected gas. This table again highlights the rather random nature of the agent's gas collection. Even though collecting gas is higher valued than collecting minerals, the rewards of the top episodes for collected gas can not compete with the top episodes for collected minerals in terms of gained rewards, with the exception of episodes that also have a high amount of collected minerals, especially again episode 19972.

4.4 Analysis of Actions

In order to keep the file size of the logfiles reasonable only the actions for the top 10 episodes in terms of collected minerals and collected gas and the last 10 episodes where kept. For 16 agent instances this makes a maximum of 480 episodes. In my case 468 episodes were analysed with a total of 393120 actions.

Table 4 shows the frequency of all actions, table 5 shows how many of them were random and table 6 shows how many of them were not random. Especially table 6 shows that the agent learned the relation between performing the action "Harvester_Gather_screen" and getting a high reward and therefore tried to perform this action over 50% of the time. It also learned that sometimes doing nothing is also beneficial for achieving this goal, so "no_op" is the second most frequent performed action. The actions for selecting: "select_point" and "select_idle_worker" were also learned from the agent, but only performed in very few cases. Taking a look at the action logs shows that the agent needed to perform a random select action before it was able to send worker units harvesting. This means that the agent did not learn the rule that it has to first select worker units, before it can send them to harvest. It is notable that the action "select_idle_worker" gets less often chosen by the agent than the action "select_point", even though this action is easier to perform, since it requires no position. The agent was not able to learn the relation between building a refinery and collecting gas, since the action for building a refinery

Action	Occurrences
Harvest_Gather_screen	53.57%
no_op	18.073%
select_point	5.2032%
move_camera	5.1394%
select_idle_worker	3.6681%
Move_minimap	3.3873%
Move_screen	3.3578%
Build_Refinery_screen	2.2372%
Build_SupplyDepot_screen	1.8124%
Build_CommandCenter_screen	0.80255%
Rally_Workers_screen	0.75193%
Rally_Workers_minimap	0.74176%
Harvest_Return_quick	0.55428%
Morph_SupplyDepot_Lower_quick	0.36859%
Morph_SupplyDepot_Raise_quick	0.33171%

Table 4: Total frequency of Actions.

("Build_refinery_screen") only gets performed as random action.

The agent did not develop any particular strategy on how to maximise the amount of collected resources.

4.4.1 Running the Agent on a different Map

The trained agent was also run on different maps than the one it was trained, but without much success. Since the training map did not require any exploring of the map[24], the agent was not really able to perform much exploration of the map for finding new resource patches, so it was just performing the same actions as it was doing on the training map, so it lead to the same results.

4.5 Interpretation of the Results

As written above, the agent was able to learn the relation between giving the command "Harvester_gather_screen" and an increase in reward, but the agent was not really able to learn the rule that it has to select a worker unit before it can give the harvest command. One reason for that could be that episodes where the "select_idle_worker" was given early on only returned relatively small final rewards. For example the episodes 63074, 34261 and 64866 the "select_idle_worker" command was given quite early, but the total amount of collected minerals was rather low (except for episode 64866), also the final reward wasn't as high as one would expect. This is due to the agent ordering the worker units to harvest early on, but then gives other commands, which keeps them from harvesting. Another special case is episode 37906, it was the

[24]It was exactly as so big that everything fitted on one screen.

Action	Occurrences
move_camera	5.1196%
no_op	5.1023%
select_point	5.0814%
select_idle_worker	3.6396%
Harvest_Gather_screen	3.4297%
Move_minimap	3.3873%
Move_screen	3.3578%
Build_Refinery_screen	2.2372%
Build_SupplyDepot_screen	1.8124%
Build_CommandCenter_screen	0.80255%
Rally_Workers_screen	0.75193%
Rally_Workers_minimap	0.74176%
Harvest_Return_quick	0.55428%
Morph_SupplyDepot_Lower_quick	0.36859%
Morph_SupplyDepot_Raise_quick	0.33171%

Table 5: Frequency of random Actions.

Action	Occurrences
Harvest_Gather_screen	50.141%
no_op	12.971%
select_point	0.12185%
select_idle_worker	0.02849%
move_camera	0.019841%

Table 6: Frequency of non random Actions.

second best episode in terms of collected minerals and also the reward was quite high. In this episode the agent didn't give much "useless" commands, that kept the worker units from collecting minerals. The reason why I was only looking at the "select_idle_worker" command is, because there it is more likely that the agent actually selects a worker unit, for select point this can not be said, since the agent could also select a random point and so selects no unit.

The agent not learning the relation between selecting a worker unit before it can give the harvesting command also explains why the agent wasn't able to learn how to collect gas. In order to collect gas it had to also learn the relation between building a refinery at the right position, something it also didn't do.

The agent performing some select actions, and especially that those actions were performed rather early, indicate that the agent would have been able to learn this correlations, if given more time. After 75000 episodes the agent was stuck in a local minimum and the average rewards as well as the average collected minerals per episode were plateauing. It is difficult to say, how long this plateau would have lasted, but given that the agent reached a higher plateau before it is likely that the agent would have left that local minimum.

5 Conclusion

Running and observing the behaviour of the agent leads to the conclusion, that the state space of StarCraft II is so large, that it can not be efficiently explored by a reinforcement learning algorithm without specialised hardware, even though with the A3C algorithm a reinforcement learning algorithm that is known to have rather low resource demands was chosen.

The agent showed some "intelligent" behaviour, especially since it learned the relation between harvesting resources and an increase in reward. From the action logs of the agent however it is obvious that the agent did not learn that it had to select a worker unit before it can give the command to harvest minerals nor did it learn how to harvest gas. The data from the action logs however also suggests that the agent could learn the relation of selecting a worker unit before it can give the harvesting command, if it is trained for a longer time.

Two things that possibly could speed up this convergence process are using the `single_select` tensor as input as well as performing an update after n steps instead of after every episode. The former would give the agent a more direct feedback whether a select action was successful (so far this feedback is only given indirectly by whether more actions become available), while the latter would make improvements in the action policy faster available to the agent.

Another possibility would have been to give the agent more guidance in the learning process. It might would have made the agent faster in solving the task, but it would also required more intervention from my side and would override one big advantage of reinforcement learning: that the agent is able to find a solution with little to no prior knowledge build into it.

References

[1] Vinyals, Oriol, et al. "StarCraft II: A New Challenge for Reinforcement Learning." *arXiv preprint arXiv:1708.04782* (2017).

[2] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning", *arXiv preprint arXiv:1312.5602* (2013).

[3] Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." *International Conference on Machine Learning*. 2016.

[4] Hinton, G. et al. "Lecture 6a Overview of Mini-Batch Gradient Descent." *Lecture Notes Distributed in CSC321 of University of Toronto.* 2014. Available online: `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf` verified 2018-03-29.

[5] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *Nature* 529.7587 (2016): 484-489.

[6] DeepMind Technologies Limited. "PySC2 - StarCraft II Learning Environment" *GitHub* 2017. Available online: `https://github.com/deepmind/pysc2/blob/master/docs/environment.md` verified 2018-03-29.

[7] Blizzard Entertainment, Inc. "SC2API Documentation" *GitHub* 2017. Available online: `https://blizzard.github.io/s2client-api/`, verified 2018-03-29.