School of Computing, Media, and the Arts
Teesside University
Middlesbrough TS1 3BA

# Can a game agent in a Real-time Strategy game learn to control units with Machine Learning?

An analysis, Design, and implementation Report

Submitted in partial requirements for the degree of MSc Computer Science

Date: 29/1/2021

Callum Powley (S6083592)

Supervisor: Chunyan Mu

# Table of Contents

# Introduction

Machine learning is the study of computer algorithm that allow artificial intelligence the ability to automatically learn and improve from experience without being programmed. The algorithms that machine learning use fall under three different types, these are supervised learning, unsupervised learning, and reinforcement learning. This is also a very big field of study with loads of information and researching being done on a day-to-day basis. However, when it comes to apply these to a Real-Time-Strategy scenario only a handful has been done, DeepMind managed to get an AI to be within the top 1% of the player base when it came to StarCraft2. This bring me onto the Aim of what this research and report is all about.

## Research Aim

The aim of the paper is test and evaluate different types of learning algorithms when it comes to a Real-Time-Strategy scenario. This will mainly be looking at controlling one unit when it comes into a combat with the AI. Since this will also be my first time doing this sort of project, I will need to research into machine learning algorithms and the best way to implement them.

## Research Question

Can a game agent in a Real-Time-Strategy game learn to control units with Machine Learning?

## Research Objectives

The objectives of this research are:

- To investigate previous research that has been done based on Learning in an RTS
- To Implement Machine Learning algorithms into my scenario
- To explore the possibilities of using different algorithms to help further my outcome
- To critically evaluate the algorithms that I have used and see if there is any way to improve them

# Literature Review

While looking for algorithms and research that has been done in this sector of the machine learning there has been a few papers that have used the same algorithms with differences. These are Q learning, SARSA(State-Action-Reward-State-Action) and Bayesian Model all these papers have changed the algorithms slightly to optimize it more.

## Algorithms

### Q-learning

Q-Learning was one of the first algorithms that I came across while reading through the papers. This algorithm is a model free learning algorithm that means it does not need the transition function to learn the optimal policy. This means that the agent uses the environment by testing possible results of an action in a certain state. The main idea of Q-Learning is to learn the utility of executing an action within a certain state.

The Q-Value is given as **Q (A, S)** which contains the utility of executing action A on the state S. Due to the agents not knowing the transition function, the agent will learn how to act through checking the possible results of performing an action within a given state. Utility of a state can be described as the highest reward that is possible to obtain in a state. The utility of a state is written as **U(S) = maxaQ (a, s).**

Q-learning requires a table where data and information about the learning can be stored. This is not that big of a deal when it comes small data sets and simple problems, but when it comes to applying this to a Real-Time-Strategy game the table and data get and start to get out of control.

FirstPaper developed an approach to solve this that compresses the number of entries in a Q-value table which uses a deep auto-encoder. With doing so they also updated the Q-learning algorithm by removing the transition function that gives the following equation:

$$\mathbf{Q(a, s) = Q(a, s)+\alpha(R(s)+\gamma maxa0Q(a\ 0\ , s\ 0\ )-Q(a, s)).}$$

Key:

α = Learning rate – Ranges from 0 and 1, 0 = nothing learned and 1 = learned value is fully considered.

γ = discount factor – Ranges from 0 and 1, 0 = future rewards are irrelevant, and 1 = fully considered.

They also apply the Q-learning to each unit separately where they only have 5 possible action for their given states. Thus, at the end of their training they share they experience using their Q-values which are normalizing to make a brand new set.

$$Q(s,a) = \frac{\sum_{i=0}^{agents} Q_i(s,a) * frequency(Q_i(s,a))}{frequency(Q(s,a))}$$

*Figure 1 Normalization Function*

Where Q(s,a) is the new Q value for all the units for the state s and action a. Qi(s,a) is the Q value of a unit I for the given state s and action a.

This is a three-step algorithm that consist of assigning a Q-table to a unit based on its given type. Then composes an action for the state, using one action for each unit, then lastly in the terminal state merges the Q-tables of the units with the same type. The pseudo code for this can be seen in figure 2.

**Algorithm 2:** Unit Q-learning pseudo code.

```
Input: s, The actual game state.
Output: An action Action.
Persistent: QTables, a set containing one Q-table to
 each unit type ;
   U, set of player units ;
   s, the previous state, initially null;
Action = ∅ ;
for unit u ∈ U do
    if u.QTable = ∅ then
        u.QTable := QTables(u.type) ;
    end
    Action.add(QLearning(u,s)) ;
end
if isTerminal(s) then
    for type ∈ U do
        Q := mergeTables(type) ;
        QTables(type) := Q ;
    end
end
return Action
```

*Figure 2 Unit Q-learning pseudo cod*

The second Paper instead of doing an encoder that compresses the Q-Value state down they decided to use the Q-learning algorithm in their simpler one-step versions along side a variation of these that uses eligibility traces that allows offset of delayed rewards.

The one-step versions equation is written as shown:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t)+\alpha[r_{t+1}+\gamma \max_a Q(s_{t+1},a_t)-Q(s_t,a_t)]$$

Second paper also used eligibility traces which are a basic mechanism for reinforcement learning, this method applies temporal credit. Meaning that it is not only the values for the most recently visited state that is updated but also the values that have been visited within a certain time frame. Cutting off eligibility traces becomes necessary when using Q-Learning since it is an off-policy algorithm.

Second Paper uses Watkins's Q(λ) which is shown in figure 3

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } (s,a) \neq (s_t,a_t); \\ 0 & \text{if } Q_{t-1}(s_t,a_t) \neq max Q_{t-1}(s_t,a_t) \\ \gamma\lambda e_{t-1}(s)+1 & \text{if } (s,a)=(s_t,a_t) \\ & \text{and } Q_{t-1}(s_t,a_t) = max Q_{t-1}(s_t,a_t). \end{cases}$$

*Figure 3 Watkins's equation*

### SARSA (State-Action-Reward-State-Action)

SARSA is a variation of Q-learning that has a small change to the update rule. This learns actions values relative to the policy it is given to follow. Unlike Q-Learning that does it relative to the exploration policy. This also means that the action with the biggest reward is not used for the next state, but the action chosen according to the same policy that led to the present state.

The basic SARSA algorithm is written as shown:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t)+\alpha[r_{t+1}+\gamma Q(s_{t+1},a_{t+1})-Q(s_t,a_t)]$$

Since SARSA is based on Q-learning most of the algorithm is the same but the part that makes SARSA is the quintuple that is (st,at,rt+1,st+1,at+1).

First paper applied their update rule at the end of each quintuple, while the second paper used a one-step method for this as well.

### Artificial Neural Networks (ANN)

Artificial Neural Networks are a group of models that are loosely inspired on the human brain that is designed to recognize patters, ANN's are composed of multiple nodes organized in layer. The nodes combine different inputs from data normally a table of sorts with a set of coefficients that amplifies out suppress an input.

An ANN contains three types of layers. This consist of an input layer that receives data that will feed into the other layers. Next, we have hidden layer that collects the data from the input layer. Lastly, we have the output layer that is responsible for receiving the signal from the hidden layer. Every layer that is between the input and output layers is considered a hidden layer.

### Deep auto-encoder

A Deep auto-encoder is a type of neural network that can convert inputs to a more compact version of itself. It composed of two connect symmetrical neural networks, the first encodes the input into an internal representation and the second decodes the internal representation back to the original input.

## Model

Since the environment they used to test their agent in is very complex they had to use some very simple states and actions that the agent will be able to use.

### States

The state that was chosen to get the best result for the agent were weapon cooldown, distance to closest enemy, number of enemy units in range and lastly, health of the unit. Weapon cooldown is to see if the weapon that the unit us using is on cooldown, the distance to the closest enemy is done as a percentage of the possible movement distance of the unit within the near future. The distances are grouped into four difference group these are ranging from <= 25% and >120%.

For the last two number of enemy units in range is the number that are within range of the unit's weapon since each weapon has a different range level. Lastly, health which is just the

remaining health of the unit the agent is controlling these have also been classing into four difference classes split into 25% each.

### Actions

There are only two actions that the agent can pick from and these are fight or retreat. The fight action is just a simple combat manager that determines all the enemy units within the agent's units attack range and selects the opponent with the lowest health that can be killed the quickest. If there is not any when the action is triggered, then nothing will happen until the next action has been chosen.

The other action retreat is to get away from danger. This works by using a weighted vector that is computed that leads away from all the enemy units that are within range and would be able to travel to the unit within the next time frame.

### Rewards

The reward is based on the difference in the health of both the enemy's and the player's units between two states. The reward is computed as the difference in health in the enemy's units the damage done by the agent minus the difference in the RL agent's unit. This can be seen below in figure 3.

$$reward_{t+1} = \sum_{i=1}^{m} enemy\_unit\_health_{i_t} - enemy\_unit\_health_{i_{t+1}} - (agent\_unit\_health_t - agent\_unit\_health_{t+1})$$

*Figure 4 Reward Calculation*

### Environments

When looking for what environments to use for my project there quite a few different environments I could have picked to use. These are PySC2 (Starcraft 2 Learning Environment), BWAPI (Brood War API), microRTS and ELF.

#### *PySC2*

PySC2 was developed by Deepmind and is used a lot for machine learning this is because the environment exposes StarCraft 2 as a research environment. It is built upon the StarCraft 2

API. This environment is open sourced and is optimised for RL agents. The main language that is used with PySC2 is Python so in sense this is a Python wrapper, so I will need to use Python if I would like to use this one. Since StarCraft is a massive game getting an agent to learn just simple things would be tricky and hard to do. So, to help this PySC2 can run on mini games that you can make yourself this incense makes it a lot easier to train for a specific part of the game.

### *MicroRTS*

MicroRTS is a simple version of a Real-Time-Strategy game that was made with the purpose of AI research. Since MircroRTS is a much simpler version of a tradition Real-Time-Strategy it becomes a very good tool to test and refine algorithms before putting them into a full and complex game like StarCraft. Java is the main language that is used for this environment. MicroRTS still consist of two players trying to eliminate each other with different type of units. The units that are in MicroRTS are workers, light, ranged and heavy and the structures that can be found are a base, barracks and minerals.
Unlike PySC2 there is not anything like minigames where you can test your agent to do a certain thing so you will need to set up the environment fully to start them tasks.

### *ELF*

ELF (Extensive, Lightweight and Flexible) is a research-oriented platform that can offer games with properties, efficient simulation, and highly customizable environment settings. ELF allows for both game parameters to change as well as new game additions. Training for the reinforced learning is deeply build into the environment and is very flexible. ELF is also capable of running games that are written in C/C++ and uses a C++/Python framework.

## Development Log

For the first week of developing my work I dedicated to get all the software setup and get used to the API that I will be using. Since this will be my first-time using python since a very long time, I needed to have a quick refresher on the language.

I decided to use Anaconda for my environment since it was one of the best machines learning platforms and has most of the packages already installed. I then used Visual Code for my IDE.
I then remembered about DeepMind and that they were able to create an AI that can play full games. They also released the API that they used to create this, so I decided to move other to this API since its more aimed towards learning.
API Link: https://github.com/deepmind/pysc2
With this I made a very simple learning AI that uses Q learning to determine what action to undertake. These options are attack, build marines and do nothing, since this is my first bot, I made it very simple just to get used to the API. The q learning table is based of MorvanZhou table. https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow

DeepMind has supplied a few minigames that can be used to train AI in without needing to play a full game. And one of them is a group of units trying to kill another unit to see how well they can "kite". This works by the AI starting with about 4 marines and need to kill 3 enemy units and if they kill the unit, they respawn vising more and more enemies, but the health of the marines does not refresh. Thus, providing the agent with a problem of trying to kill as many as it can without losing all its units.
I have decided to use this as a base for my minigame and changing it, so the AI only controls one unit and the enemy only has one unit as well. This will help testing the algorithms so quicker so I will not need to create a testing bed from scratch, and I am able to pick which units are being used so the Ai will be able to get smarter since some have cooldown abilities as well.


This is a screenshot of what the minigame will be for now, once I have got enough testing with the algorithms, I will be adding in some more units that the AI can control maybe even a mix of them.

Next week I will be working on one of the most common algorithms and that will be Q-Learning.

The first step to getting Q-learning into my AI was thinking what states and actions I would like the use. To make this simple for myself to understand the agent will only have 2 states to pick from and these are attack and retreat. Attack will just be a simple attack command that will attack click towards the enemy. While as the retreat command will click behind the agent's unit so it runs away from the enemy. In sense this should allow the agent to learn how to kite the enemy in which they attack them right away click behind them self before they get into their own attack range.

The code for the attack state is written like so:

```
    def attack(self, obs):

        if actions.FUNCTIONS.Attack_screen.id in obs.observation.available_act
ions:
            player_relative = obs.observation.feature_screen.player_relative
            targets = _xy_locs(player_relative == features.PlayerRelative.ENEM
Y)

            if not targets:
                return actions.FUNCTIONS.no_op()

            target = targets[np.argmax(np.array(targets)[:, 1])]
            return actions.FUNCTIONS.Attack_screen("now", target)

        if actions.FUNCTIONS.select_army.id in obs.observation.available_actio
ns:
            return actions.FUNCTIONS.select_army("select")
```

```
        return actions.FUNCTIONS.no_op()
```

The code for this can be a little bit confusing at first since the API is not the best to understand at first. All it does is use observers which are layers within the game's API. It then finds out which units the player has, and which are the enemies by using the in-game layers. If it cannot find any targets the function returns "no_op" which tells the algorithm to skip this step and try to find a new action to use. The retreat function works the same as the attack apart from it moves the unit away at a random position using randint. This is not the best way of doing this but the quickest way at the time so I will be able to get something to work.

When it comes to picking which state to use three different values are returned from the function and these are the health reminding of the agent's unit, how many units are left, and the number of enemies left. I will need to see how well just using three lots of data are when it comes to the effectivity of the algorithm.

# My Experiment

## The Environment

The agent is being tested on an edited version of the DeepMind minigame. This is where instead of using marines to kill roaches, I am using Reapers to kill Zerglings. This is because the Reapers have a better movement speed than the Zerglings and they are also use ranged attacks to the importance of kiting is greatly needed here. The agent will always spawn on the left and the enemies on the right this was to make the actions easier on myself to code.


*Figure 5 The Test Environment*

## Model

### Actions

The agent will be having two actions only this is to keep it as simplistic as I can while getting the best results, these are attack and retreat. The attack action is a simple one where it gets the location of the enemy and does an attack command to the friendly units. Retreat checks to see if the enemy is too close then runs away in the other direction.

The attack action works as so, it first gets the location of the player and the enemy units, then checks to see if there are any enemies alive if not skip the action. I check the distance between the two units and if they are too close, I skip the action this is so the agents knows

that it can not be too close to be in harms way. Then if it is not too close to the enemy it attacks.

Retreat starts by getting the units positions as well, then once again checks to make sure that there are some enemies alive. Then a value of two is set for a distance check this value can be changed when I see fit, this is then checked with the hypotenuse of the enemy and agents' positions. If they are too close the agent, then move away from the enemy.

### States

There are seven states that are being fed into the Q-learning algorithm and they are length of reaper, length of enemies, reapers health, enemies health, reaper weapon cooldown, reapers radius and enemies radius.

Length of reaper / enemies – all this does is return the number of units.

Reapers / enemies health – returns the health of the units.

Reaper weapon cooldown – returns how long the reapers attack timer is left.

Reapers / enemies radius – returns the attack range of the units.

When picks what states I was needing I took some inspiration from "**S. Wender and I. Watson"** since they also added I the weapon cooldown as a state. This is so the unit is not trying to attack when it not able to and thus allowing the agent to move away instead. This is quite critical to my results since my main point is to show that an agent can learn to move away from the enemy unit when it needs to. Most of the other states are things you look out for yourself if you were playing yourself, so they seemed most fitting to add in. You will need to know how many units and enemies there is on the battlefield and providing this information to the agent is not the necessary at the time being since there is only one of each but when the training starts to ramp up and we add a few more in this will be valuable information. Each unit in the game has a different attack range in which they can issue an attack and hit the other units; this is where the radius state comes in since we need to know the attack range of the units.

I made sure when picking the states that this is information that a player would also be able to obtain themselves with enough experience in the game. Since I did not want to make an agent that is using cheating ways to learn to beat the enemy.

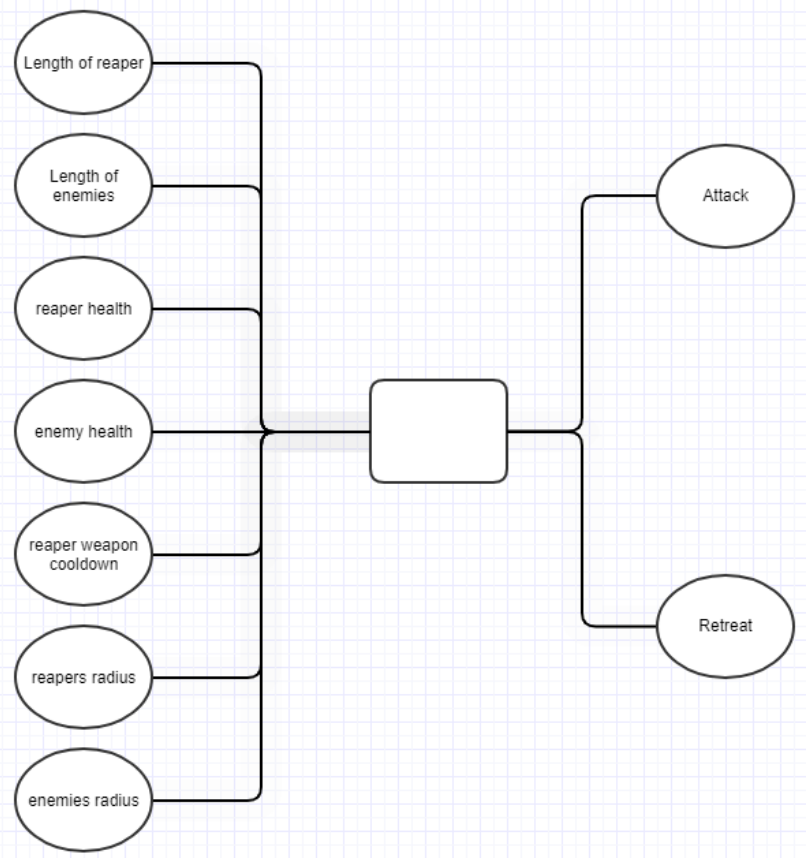


*Figure 6 Diagram of the states and action*

## Reward

The agent will get rewards base on the number of enemies he has killed in one life and how much life they have the end of each episode. Getting the rewards right are crucial for Q learning to work correctly so the health reward is broken down into 3 stages. Since the Reapers has a base health of 60 the agent will get 3 point if they health is greater or equal to 25, 2 if it is greater than 10 then 1 if less. The values and brackets will need to be find tuned but for now I am using what I can achieve when playing the minigame.

The agent will also lose a point if they are killed or if the timer runs out. This is due to when it was training it would sometimes learn to kill one unit then go hide in the corner till the timer ran out. Doing so lead to some different results then kept doing that for the rest of the training.

## Algorithm

Q-Learning I the algorithm I picked to use. When it comes to what action the agent should pick, we use a greedy parameter which tells the algorithm to pick the best action 90% of the time and the other 10 to pick a random one. To pick the best action it receives the values for each action then picks the highest values and the action allocated to that.

## Results and Evaluation

The training was done over the course of 1000 episodes this was due to hardware limitations, and it takes four hours per training session. As expected, the training started out rough in the beginning but then started to pick up in later episodes. When it came to around about the 800 episodes the agent just stopped learning things and either just sat in a corner or just did not improve on what it was doing. I think this was down to the learning part of the algorithm or the data I used for it to pick the best action. This did take a good couple of changes to what states were being used to achieve this. Below is a chart on the value of each action and how it was different from the other ones.

| Index | retreat | attack |
|---|---|---|
| (1, 1, 60, 35, 0, 1, 0) | 3.55454e-08 | 1.45463e-07 |
| (1, 1, 60, 31, 17, 1, 0) | 3.55454e-08 | 1.54582e-07 |
| (1, 1, 60, 31, 9, 1, 0) | 3.6268e-08 | 1.54582e-07 |
| (1, 1, 60, 31, 1, 1, 0) | 1.9501e-07 | 1.54582e-07 |
| (1, 1, 60, 23, 12, 1, 0) | 1.9501e-07 | 1.54582e-07 |
| (1, 1, 60, 23, 4, 1, 0) | 6.10762e-07 | 1.54582e-07 |
| (1, 1, 55, 15, 17, 1, 0) | 6.10775e-07 | 0.000285478 |
| (1, 1, 55, 15, 9, 1, 0) | 6.403e-07 | 0.000285478 |
| (1, 1, 50, 15, 1, 1, 0) | 6.403e-07 | 0.000288354 |
| (1, 1, 45, 8, 12, 1, 0) | 3.19458e-05 | 0.000288384 |

*Figure 7 Chart of which action is taken*

Overall, I think the results that was obtained were quite good. The agent is learning on how to kill a unit by kiting backwards out of the enemies' attack range, but I will need to apply this into a full fledge game. The agent did show some good results with the rewards that it has gotten back, it kept an average score of about 6 which it killed three enemies in one life span. This can be improved with the amount of data and maybe a bigger map. Which leads me to the things that went wrong with this project. Since this was my first-time using machine learning and Q learning I needed to learn about the algorithm and how it can be implemented into my work. When it comes to improvements for the agent, I would have liked to use a much bigger map, so the agent has a much bigger room to move around.

Changing up what data is passed into the algorithm could also give us a much better outcome. Getting another algorithm in was one of the other objectives for my work but since I was having a lot of trouble with just getting Q-Learning in I decided to just focus on that one instead.

## Further Work

When it comes to further work related to my project, I would like to get the algorithm to work a bit better and get the agent to fully kite. To achieve this, I will need to make a bigger map, so the agent is not compound to a very tiny map, so it has a bigger player area. Refined what data is being sent to the algorithm since I do not think that I need all the data I am sending to it already. Test with multiple agents and enemies, and make a agent with a different algorithm so I can compare and analyse them.

# References

**Amado, Leonardo. “Reinforcement learning applied to RTS games.” (2017).**

**S. Wender and I. Watson, "Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar," 2012 IEEE Conference on Computational Intelligence and Games (CIG), Granada, 2012, pp. 402-408, doi: 10.1109/CIG.2012.6374183. – second paper**

**StarCraft II: A New Challenge for Reinforcement Learning. arXiv 2017**

**Liu, Tianlin et al. “A Hierarchical Model for StarCraft II Mini-Game.” *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)* (2019): 222-227.**

**DeepMind, “Pysc2,” https://github.com/deepmind/pysc2/, 2017**

**Pang, Zhen-Jia et al. “On Reinforcement Learning for Full-length Game of StarCraft.” *AAAI* (2019).**

**Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and Larry Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. arXiv preprint arXiv:1707.01067, 2017.**

**MicroRTS https://github.com/santiontanon/microrts**