Starcraft Intelligent Agents

Zeynep Gürler
dept. of Computer Engineering
Istanbul Technical University
Istanbul, Turkey
gurler17@itu.edu.tr
150170031

Serra Uysal
dept. of Computer Engineering
Istanbul Technical University
Istanbul, Turkey
uysals17@itu.edu.tr
150170117

Medina Zaganjori
dept. of Computer Engineering
Istanbul Technical University
Istanbul, Turkey
zaganjori16@itu.edu.tr
150160908

Joana Hoxhaj
dept. of Computer Engineering
Istanbul Technical University
Istanbul, Turkey
hoxhaj16@itu.edu.tr
150160909

***Abstract*—Reinforcement Learning is one of the most important tools in logical framework of games. As technology has been improved, even components capabilities has reached a high complexity in the intelligent framework they represent. During last years researchers have provided successful benchmarks at implementing intelligent agents that give optimal performance during training. In this research work we investigate different frameworks over two environments or mini games: MoveToBeacon, CollectMineralShards. We tried our baseline methods and train the agent in different machines. We believe that future work should analyze the limitations of the current research.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Machine Learning in video-games has been improved its goal-oriented intelligent behavior in order to present the best features during training agent. Researchers had worked on real strategy games that involve a well-trained agent and human professional player. AlphaStar is the first Artificial Intelligent agent that reached best result toward previous agents like Terran, Zerg. The agent was trained over a fully connected neural network by using semi-supervised learning. Reinforcement Learning is connected to Meta-Learning optimization where observations are used in order to learn the network to perform better in the next actions. Wide range of saved experiences is called meta-data. Combination of learning system and neural network leads to maximization of winning logits probabilities. The work was inspired from the performance of other agents like: AlphaGo and AlphaZero that outperformed human capabilities and improved modern game areas.
Starcraft represents a very complex environment. Exploration and Eploitation can be hard to be adapted in our system considering that training time is very huge and the training machines environment is not a complex architecture. In order to improve the performance of agents studies focus their importance on creating a policy condition for the agent's actions. The involvement of policy constraint improved the performance of the players for more than 80 percent of previous players.
We investigated different approaches of our architectures in our environment and gave some details for the techniques of reinforcement learning even though our research work has some limitations related to yielding efficient results due to the lack of machine complexity or high-performance computers. A. Reinforcement Learning something something for theMoveToBeacon map: Joana Hoxhaj, B. Reinforcement Learning - Epsilon Greedy Approach : Medina Zaganjori, C. Spawning Pool Building Task with Reinforcement Learning: Serra Uysal and D. Collecting mineral shards with Reinforcement Learning agent: Zeynep. Link of our video:

A. *Reinforcement Learning for the MoveToBeacon map-Joana*

a) Reinforcement Learning: is better for you to discover what actions and decisions can give you the biggest reward long term. It helps you to take your decisions sequentially, unlike in other machine learning methods, such as supervised learning, where the decisions are made based on the input at the beginning. It functions on the basis of interacting with the environment, and is great for AI where human interaction is common, such as games, and in our case, StarCraft 2. However, it is not suitable for all areas, as it needs a lot of computing-power and is very time-consuming, especially when the action space is large. When enough data is available to solve the problem with a different learning method, we can avoid reinforcement learning.

b) : The map that I have chosen is MoveToBeacon. The agent is trained to go towards a position in an intelligent manner for its goal to be accomplished

B. Reinforcement Learning Approach - Epsilon Greedy Approach-Medina

a) Reinforcement Learning Agent: In the researching papers that we have found so far it is very hard to implement models that belong to the first creators of Reinforcement Learning Agents. Although environmental, complexity, graphical card impediments I focused my work on identifying some differences of optimizer usages and how a buffer memory helps the algorithm to perform better. I tried variants of epsilon greedy algorithms that follow the problem solving by choosing locally an optimal choice in each stage. Reinforcement learning has served different concepts that use policy learning while deciding to perform an optimal action. This algorithm takes into consideration the usage of exploration and exploitation. The map in Fig.2 and Fig.3 that I have chosen is also MoveToBeacon where the agent is trained to move to a specific position in a smart way in order to reach its goal.

b) Q-Learning: We use a Q-table approach that is filled with state-action tuples. When the agent reaches a goal it takes a reward for its performance. The algorithm is performed in a set of time steps and we use learning parameters alpha and gamma. These will be referring to (epsilon min) and (epsilon max). These parameters will be used as agent hyper parameters during training. The next action will be chosen based on epsilon greedy strategy. We set the learning parameter or alpha in order to learn itself during performance and perform an optimal new action. Another hyper parameter worth to mention is a discount factor that predefined the future reward of the agent. An action will be based on the Q-values that Q-table stores. This hyper parameter will help us to try different actions during training. By trying different goal positions the agent learns its way of performing better in the next stage [2]. We tried different epsilon values and we found out that numbers that are closer to zero seem to give better performance than larger values as in Fig.1.

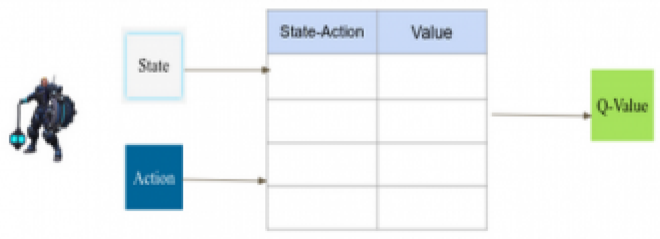


Fig. 1. Q Table Decisions

Q learning is a reinforcement learning approach that finds the best action based on a current state. Even though we have a policy constraint algorithm tries different and random actions in order to maximize agent's reward. But in order to see good performance we need to train an agent. In our case we followed a Deep Q Reinforcement Learning Agent. Q-Table will refer to the action with the highest Q value and the evaluation of the value will define the efficiency of the action. In Fig.4 memory buffer will help us to store all the actions and states so training will be performed over batches of data [3].

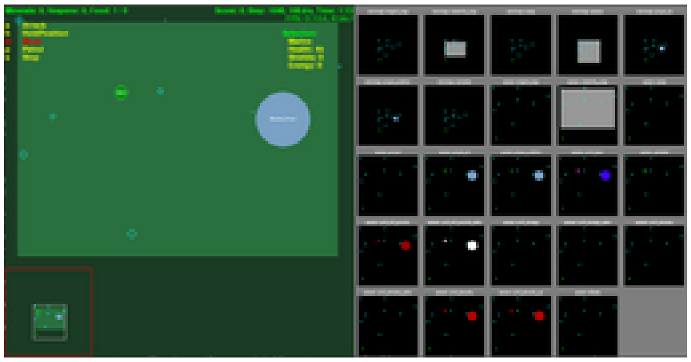


Fig. 2. MoveToBeacon Environment

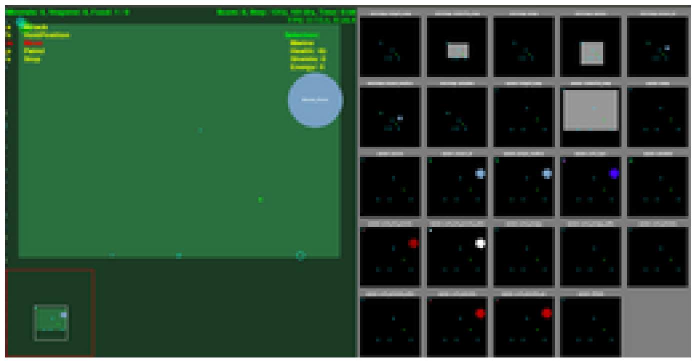


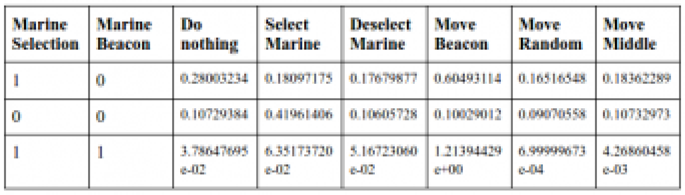
Fig. 3. MoveToBeacon Environment

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma max_a Q(S^l, a) - Q(S,A)] \quad (1)$$

c) Fully Connected Neural Network: Input data is very important in machine learning. Precept data will be referring to received precepts and will be prepared to set on a deep neural network in order to have a state representation. State helps the coder to inform policy constraints in order to select action that is best adapted with the environment. Next model state gets improved from the reward that agents get. By using pysc2 libraries we can iteratively extract the feature screen as input data in shape of A x B pixels. We use a convolution neural network to train the algorithm. We set the filter dimensions, filter attributes, depth of the layer. Fully connected layer will be used to aggregate information from previous layers. The set of hyper parameters and parameters that will be used during training are: epsilon min, epsilon max, discount factor, training frequency, target update frequency [4]. A full schematic illustration is given in Fig.6.

C. Spawning Pool Building Task with Reinforcement Learning - Serra

This part focuses on how to make an AI agent for building Spawning Pool in the StarCraft II game. The code imple-



Marine Selection	Marine Beacon	Do nothing	Select Marine	Deselect Marine	Move Beacon	Move Random	Move Middle
1	0	0.28003234	0.18097175	0.17679877	0.60493114	0.16516548	0.18362289
0	0	0.10729384	0.41961406	0.10605728	0.10029012	0.09070558	0.10732973
1	1	3.78647695 e-02	6.35173720 e-02	5.16723060 e-02	1.21394429 e+00	6.99999673 e-04	4.26860458 e-03

Fig. 4. Q Table MoveToBeacon Environment

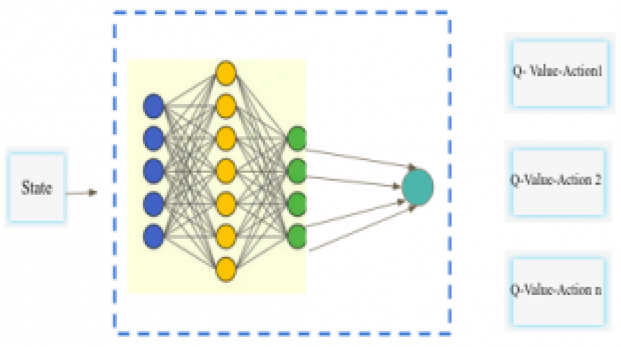
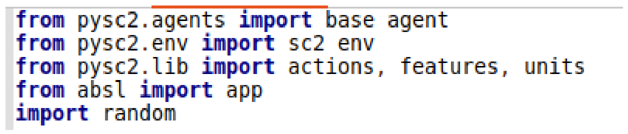


Fig. 5. Schematic Illustration of CNN

mented for this part is called agent.py and Tensorflow GPU is used in order to train the model.

a) Introduction to the environment: In the StarCraft II game there are lot's of mini games that can be chosen as a target and do the implementations upon it. Spawning Pool is one them and it uses Zerg agent as a base. The main idea in this mini game is after collecting enough minerals, agent builds a Spawning Pool if there is not another one already found in the environment. To do that pysc2 version 2.0 is used for implementation of the code. Pysc2 is DeepMind's Python component of the StarCraft II Learning Environment. Different from the previous version (pysc2 version 1.2) with the new editions, units can be reached with an ease that is why this version is preferred in this part.

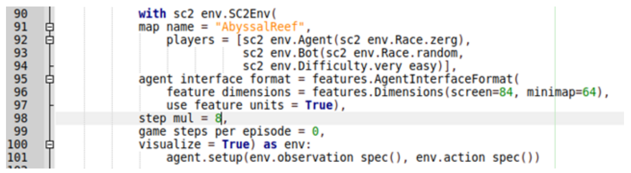


```
from pysc2.agents import base agent
from pysc2.env import sc2 env
from pysc2.lib import actions, features, units
from absl import app
import random
```

Fig. 6. Libraries for agent.py

The dependencies in order to run the code are shown in the **Fig.** 6. base_agent used in order to deal with agent properties, env is used for setting up the environment and actions, features and units are for determining the available actions after observation of the screen ad choosing the desired units. The fourth line in this figure (from absl import app) is also a difference from the previous version of pysc2 that is used to open the game. Random is used in order to randomize some operations that will be explained later.

In order to configure the properties of the agent, a class named Agent is defined. It has functions like determining and setting the unit type like a general base agent has for this game and the step function is defined in a way that steps will be cover our desired operation, building a spawning pool. If the requirements that have been defined before are satisfied, the agent builds a spawning pool in the coordinate points that are chosen randomly. Later on the main function is defined where the agent is constructed and the environment is observed.



```
90          with sc2 env.SC2Env(
91          map name = "AbyssalReef",
92              players = [sc2 env.Agent(sc2 env.Race.zerg),
93                         sc2 env.Bot(sc2 env.Race.random,
94                         sc2 env.Difficulty.very easy)],
95          agent interface format = features.AgentInterfaceFormat(
96              feature dimensions = features.Dimensions(screen=84, minimap=64),
97              use feature units = True),
98          step mul = 8,
99          game steps per episode = 0,
100         visualize = True) as env:
101             agent.setup(env.observation spec(), env.action spec())
```

Fig. 7. Environment setup part from agent.py

Fig. 7 shows the description of the environment and as it can be seen from the line 91 the map that is used here is called "AbyssalReef". This is a mini map that is provided by DeepMind [1] and after downloading the maps from there, they should be placed in the Maps folder that is found inside the StarCraft folder. The most commonly used map is "Simple64" which is also tested with this agent while implementing it and it works properly with that map too. map_name can be changed into any other map name that is found under the Maps folder and the game can be run with that map if it is wanted. After the main description is done, the game is started with the main function and map is seen on the screen with agent taking actions.

b) Reinforcement Learning for StarCraft II: In Artificial Intelligence, Reinforcement Learning is widely preferred in gaming area. Different from Supervised Learning algorithms, in this method there will not be any input that contains the proper actions for the agent but instead it will learn through it's actions by playing the game. To do so it needs to have a memory to keep track of previous actions and it should have a grading system that compares the results of the actions and if that action provides a better score than it receives a point. After implementing the agent as explained above and running it with randomized location for building, I observed that when Spawning Pool is built between the Hatchery and enemy Zerg agents where they come to attack the Hatchery, enemy agents are targeting the Spawning Pool first which gives time to Hatchery to gain more score. An example scenario is shown in the **Fig.** 8 below:

Since I have not defined an attack action for agent, in every scenario it looses to the enemy however my main objective here was score points that are gained through game time. So while defining the the grading system I needed to focus on the score rather then win/loose (1/0) situation that is considered as the case in previously defined methods.

c) Results of the part: Due to complexity of the environment I have spent most of my time to figure out a way that agents work for selected task. StarCraft II includes a lot

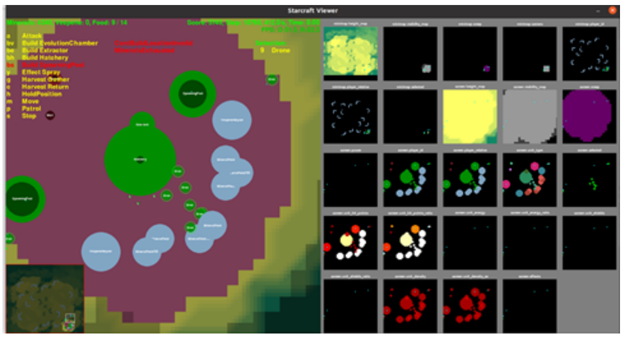


Fig. 8. Screenshot of the game

of mini games and agents have lot's of action option. After trying bunch of different games I finally decided on building Spawning Pool since I have observed that in this task the action performance can be done intelligently and this would increase the score for the agent. I have tried to add a learning method that would reward the agent and and keep the results in the memory and after training the agent enough it would perform a human-like performance. However I could not managed to make it run properly and since it was causing some errors I had to delete those parts in the source code (agent.py) and the remaining code only runs the agent with a randomized method that chooses the coordinates of the Spawning Pool randomly.

D. *Collecting mineral shards with Reinforcement Learning agent - Zeynep*

The environment that we chose was pysc2 [1], a python module for Starcraft 2 learning environment to work on Reinforcement Learning (RL) agents. pysc2 provides several minigames. Each minigame focuses on a different subproblem that we encounter while creating an Artificial Intelligence (AI) agent for Starcraft 2. Therefore, this module helps us approach each subproblem individually to achieve a more agent later.

After environment selection, I searched for some tutorials online to learn how to use the environment and how to build an RL model. Also, I searched for other methods to create a smart agent and searched for how to play the game to gain some strategy. Since there were no supplementary resources about what I mentioned above from the course itself, I had to dive deep into the internet for all and we actually needed much more time for even just learning. Therewithal, there were a few sources on the internet about the topic.

a) Selection of the minigame: In order to decrease the training running time, I selected one of the simplest minigames in the API that is CollectingMineralShards. In this game, the aim is to collect minerals as many as possible with only two marines. In the beginning, 20 minerals and 2 marines randomly spawn on the map. If the marines achieve to collect all minerals under 120 seconds 20 more minerals randomly spawn. The game stops after 120 seconds.

b) Construction of the environment: The minigame is played by only one player. I chose Terran to be the player

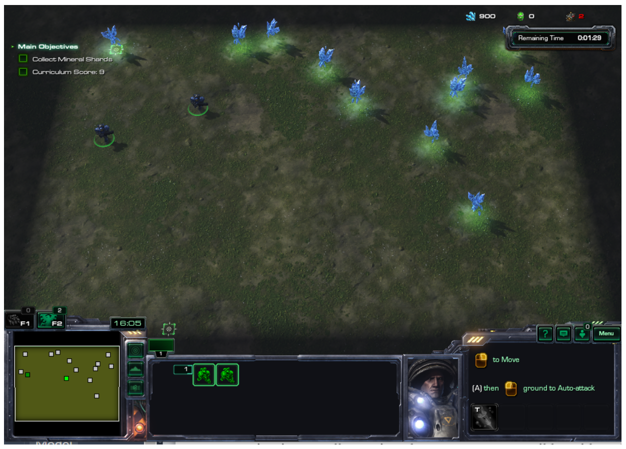


Fig. 9. Environment of the minigame CollectingMineralShards

out of Terran, Zerg, and Protoss. I built the environment, set the screen size, etc.

I searched for all action functions that are provided for CollectingMineralShards minigame by pysc2. There are lots of actions in pysc2.lib.actions but not all of them can be used for this minigame. Therefore, I chose select_army, select_point, Move_screen, and no_op functions to be performed by my agent. Select_army selects both 2 marines, select_point selects a specific point on the map, Move_screen moves the selected army or a marine if it is selected by using select_point at the previous step to a specific point on the map. Lastly, no_op is for doing nothing and it is used to see if the model might learn when to do nothing and when to take action. Because at the beginning of the training agent takes an action and then it takes another action before seeing the result of the previous such as it moves to another point before getting to the previously called point.

c) A fully connected Reinforcement Learning network with Policy Gradient: I built an RL network that uses Policy Gradient. This loss function is used for RL models and is similar to maximum likelihood but it uses rewards by multiplying actions with their rewards to compute which actions benefit us more. Below equation shows the policy gradient function.

$$\theta_{t+1} = \theta_t + \alpha \hat{A}(s, \alpha) \nabla_\theta \log \pi_\theta(s|\alpha) \tag{2}$$

I built the network as one input layer, two hidden layers, and one output layer. The input layer has a size of screen $height \times screenwidth \times channels$. I set the screen size as 84×84 and the number of channels is 3 since there are three channels in an image (RGB). The first hidden layer has 512 neurons and the second hidden layer has 256 neurons. Lastly, the output layer has 4 neurons each outputting the corresponding action's probability of being the best action to take and it also outputs coordinates of a point that it thinks a mineral is high-possibly there. **Fig.** 10 displays the network structure of my model.

d) Make action function: So the model returns the best possible action's id out of "select army", "select point", "move screen", and "no operation" at each step with some

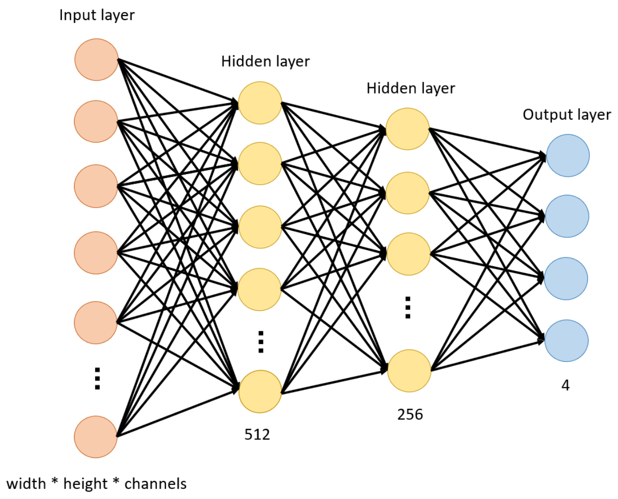
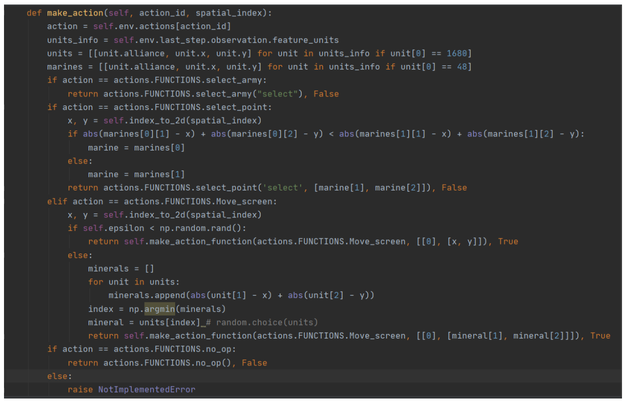


Fig. 10. Network structure

coordinates on the map. If the chosen action is no operation, previously selected unit(selected marine by ”select point”) or units(selected by ”select army”) do nothing. If the chosen action is select army, both two marines are selected and it means that they will make the same action at the next step. If the chosen action is select point, the agent will select the point where the one marine, which is the closest to the returned x and y coordinates, lays. The strategy behind this function is to be able to make the two marines act differently, go to different areas on the map to increase the chance of finding a mineral. Finally, if the chosen action is move screen, the agent will either move the screen to the x and y coordinates that are returned by the RL model or it will move the screen to the coordinates of the unit that is closest to the returned coordinates out of all units in the map. The rate of which these two actions occur is decided according to the epsilon value. The below **Fig.** 11 shows the make action function.



```
def make_action(self, action_id, spatial_index):
    action = self.env.actions[action_id]
    units_info = self.env.last_step.observation.feature_units
    units = [[unit.alliance, unit.x, unit.y] for unit in units_info if unit[0] == 1680]
    marines = [[unit.alliance, unit.x, unit.y] for unit in units_info if unit[0] == 48]
    if action == actions.FUNCTIONS.select_army:
        return actions.FUNCTIONS.select_army("select"), False
    if action == actions.FUNCTIONS.select_point:
        x, y = self.index_to_2d(spatial_index)
        if abs(marines[0][1] - x) + abs(marines[0][2] - y) < abs(marines[1][1] - x) + abs(marines[1][2] - y):
            marine = marines[0]
        else:
            marine = marines[1]
        return actions.FUNCTIONS.select_point('select', [marine[1], marine[2]]), False
    elif action == actions.FUNCTIONS.Move_screen:
        x, y = self.index_to_2d(spatial_index)
        if self.epsilon < np.random.rand():
            return self.make_action_function(actions.FUNCTIONS.Move_screen, [[0], [x, y]]), True
        else:
            minerals = []
            for unit in units:
                minerals.append(abs(unit[1] - x) + abs(unit[2] - y))
            index = np.argmin(minerals)
            mineral = units[index] # random.choice(units)
            return self.make_action_function(actions.FUNCTIONS.Move_screen, [[0], [mineral[1], mineral[2]]]), True
    if action == actions.FUNCTIONS.no_op:
        return actions.FUNCTIONS.no_op(), False
    else:
        raise NotImplementedError
```

Fig. 11. Make action function

e) Results of the part: I trained the RL agent for 18 episodes (1 episode = 20000 runs). I compared two trainings, first one (RL without CU) was not moving the screen to the unit that is closest to the coordinates returned by the model. I added that strategy to the second agent (RL with CU). As we see on **Tab.** I, moving to the unit that is closest to the returned coordinates boosted the mean and the max number of collections.

TABLE I
RESULTS OF THE RL AGENT TRAININGS

Method	Mean collection	Max collection
RL agent with CU	51.9	60
RL agent without CU	32.3	41

CONCLUSION

To conclude, Reinforcement Learning is acutely common in implementing an AI agent for gaming purposes. That was our motivation to use it in our project as well. Reinforcement Learning provides the agent to learn from it’s actions by rewarding them positively or negatively depending on the results of the action. We used Reinforcement Learning approaches in order to create an agent that functions in the StarCraft II environment, which is a highly complex environment that consists of union of plenty different mini games. Since it is really complex, instead of trying to solve it as it is, we have divided it into some mini games together with our group mates, for each of us to create an AI based agent that functions in the chosen mini game. The details of each part is explained in the related parts of the report and and source codes are added together with the project file.

REFERENCES

[1] R. Ring, “PySC2 - StarCraft II Learning Environment,”, DeepMind, 2019. [Online]. Available: https://github.com/deepmind/pysc2/blob/master/README.md .[Accessed: 10-Feb-2021]
[2] Zh. Pang, R. Liu, Zh. Meng, Y. Zhang, Y. Yut, T. Lu “On Reinforcement Learning for Full-length Game of StarCraft,” February 2019.
[3] N. Justesen and S. Risi. “Learning macromanagement in starcraft from replays using deeplearning”. pages 162–169, 2017.
[4] R. Sutton and A. Barto. “Introduction to reinforcement learning”, volume 135. MIT pressCambridge, 1998.