

Artificial Intelligence Agents for StarCraft 2

Carlos Cardenas-Ruiz, Emilio Tonix-Gleason, and Julia Rodriguez-Abud

Artificial Intelligence

Abstract

This document describes a series of projects developed for the 2021 Cinvestav Guadalajara Machine Learning class given by Dr. Andres Mendez Vazquez. The course is comprised of three main parts: searching algorithms, bayesian networks, and multilayer perceptrons. The aim was to develop a StarCraft AI agent using the PySC2 Python framework. In total 5 agents were developed, 3 of which were developed for minimaps (maps with clear and small objectives) and the rest for the simple 64 map (a small map that actually emulates a normal game with all its components). Most agents had good results although they were not perfect implementations, as our time was quite limited and the learning curve for PySC2 was steep. Nevertheless, hopefully our learning process will help others going through the same path by using our documentation and github code as reference material.

Index Terms

CONTENTS

I	Introduction	4
I-A	Installation	4
I-B	About PySC2	4
I-C	Workflow	4
I-D	Roadmap	5
II	Uninformed Search	6
II-A	Beacon Agent	6
II-B	Mesh	6
II-C	Iterative Deepening Search (IDS)	7
II-D	Bellman Ford Implicit (BFI)	7
II-E	Conclusion	7
III	Informed Search	9
III-A	CollectMineralShards	9
III-B	Brush	9
III-C	A*	10
III-D	HillClimbing and Simulated Annealing	10
III-E	Conclusion	11
IV	Min Max and Probabilistic	12
IV-A	FindAndDefeatZerglings	12
	IV-A1 Description	12
IV-B	Alpha-beta Pruning / Minmax	12
IV-C	Cost function with Heuristic	13
IV-D	Conclusion	14
V	Bayesian Networks	15
V-A	Map Description	15
V-B	Map and characters overview	16
V-C	Bayesian newtork scouting	17
V-D	Reduce overfiting	18
V-E	Results	18
V-F	Conlusions	19
VI	Reinforcement learning and pytorch	20
VI-A	Map Description	20
VI-B	Defining Net	21
VI-C	Defining EPOCHs	22
VI-D	Results and Runs	23
VI-E	Conclusions	23

VII	Final release	23
VII-A	Map description	23
VII-B	Introduction	24
VII-C	Against Random agent	25
VII-D	Against Blizzard AI	27
VIII	Conclusions	29
	References	30
	Biographies	30
	Carlos Cardenas-Ruiz	30
	Emilio Tonix-Gleason	30
	Julia Rodriguez-Abud	30

I. INTRODUCTION

A. Installation

The easiest way to get PySC2 is through a pip install command:

python 2.7

```
1 $ pip install pysc2
```

python 3.X

```
1 $ pip3 install pysc2
```

For more info, templates and documentation on PySC2, visit the [website](#)

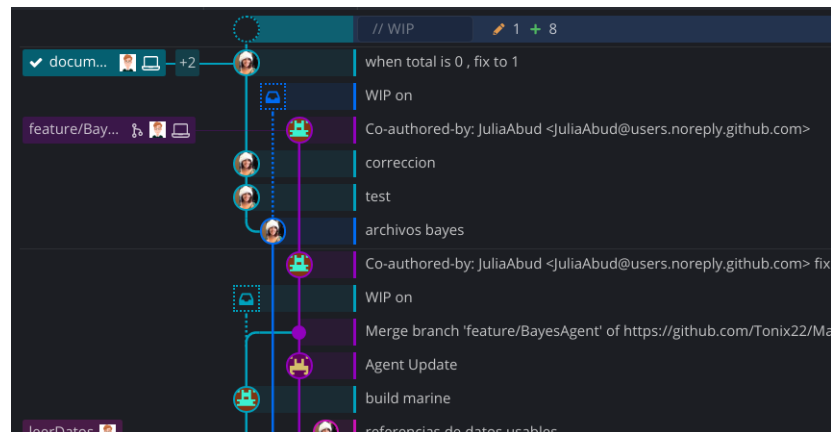
B. About PySC2

PySC2 [1] is a non-official Deepmind product commonly used for machine learning competitive programming. It is a multiplatform python framework that allows reading and writing of StarCraft II data. This allows the implementation of automatic strategies based on learning algorithms.

C. Workflow

To develop and produce our project we used [gitflow strategy](#). This allowed us to split tasks in batches, which are treated as features in the code implementation. In Git we used branching and merging to do parallel work and code review. When each task was completed, a branching merge was done.

Fig. 1. GitFlow



D. Roadmap

We used the DevOps methodology, which focuses on bringing the operations lifecycle into the same agile experience as the development teams. When adopting the DevOps philosophy the team remains responsible for the release during the entire lifecycle of the product. Organization and planning were done using a Kanban board. And the roadmap used for the project was done in an iterative way with the following steps:

- 1) Choosing StartCraftII map (or minigame)
- 2) Understanding the problem
- 3) Planning a solution with the course tools.
- 4) Designating tasks and features
- 5) Coding and doing of individual tasks
- 6) Merging and code reviewing
- 7) Testing
- 8) Releasing git code
- 9) Showing the product and the results for final review and feedback to the teacher

Fig. 2. Kanban board



II. UNINFORMED SEARCH

A. Beacon Agent

A map with 1 Marine and 1 Beacon. Rewards are earned by moving the marine to the beacon. Whenever the Marine earns a reward for reaching the Beacon, the Beacon is teleported to a random location (at least 5 units away from Marine).

Initial State

1 Marine at random location (unselected) 1 Beacon at random location (at least 4 units away from Marine)

Rewards

Marine reaches Beacon: +1

End Condition

Time elapsed

Time Limit

120 seconds

B. Mesh

The map is shown in a 64x64 array, giving a total of more than 4K data elements to process by any recursive algorithm. This would significantly underperform the user experience. A quantization of this data was done with the purpose of doing less iterations. For example, if we originally had a grid of 64x64, we could then transform it to a 8x8 grid.

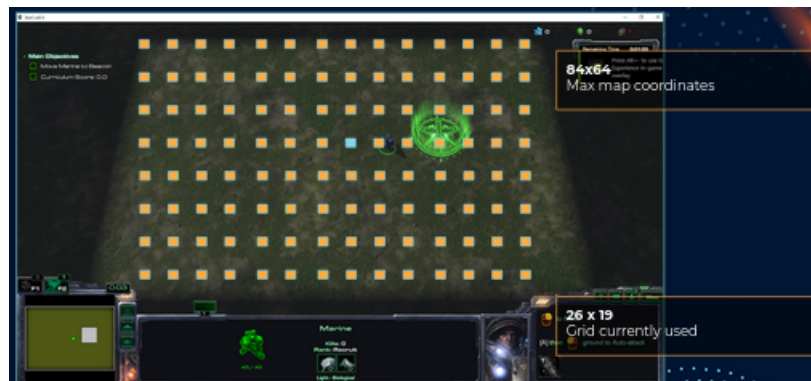


Fig. 3. Grid on the map

C. Iterative Deepening Search (IDS)

The node exploration is limited to the expansion of each individual node to its immediate neighbors in a straightforward way. The diagram below shows how the expansion of the nodes is done. The algorithm checks all the possible neighbors and connections only during the first run of the map. The idea being to use dynamic programming to avoid the repetition of expanding nodes that are already known, thus having a faster solution. In other words, the generated mesh helps to reduce the work because it removes the need for checking each pixel of the map.

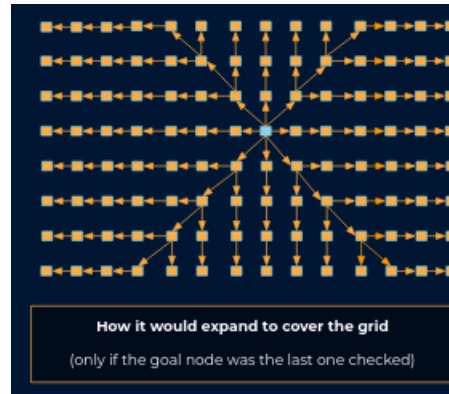


Fig. 4. Mesh exploration

D. Bellman Ford Implicit (BFI)

This algorithm uses a close/open logic to explore the nodes in a similar fashion to A*. The difference being that this method does not use an heuristic, it only makes use of the euclidean distance to choose the shortest path. It is a dynamic programming solution that cuts when the goal is reached.

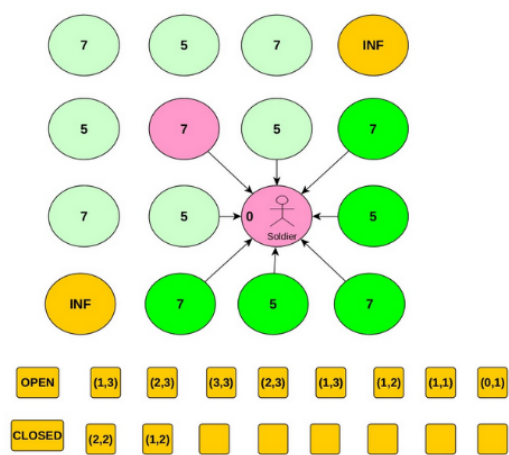


Fig. 5. Mesh Belman ford

E. Conclusion

Both methods were fast enough to keep up a good user experience. There was almost no delay perceived and the game was completely functional.

In this first stage of the project our team had to deal with dipping our toes in PySC2, so there was some time invested in this task besides the implementation of the algorithms. The main idea was to get familiarized with the framework.

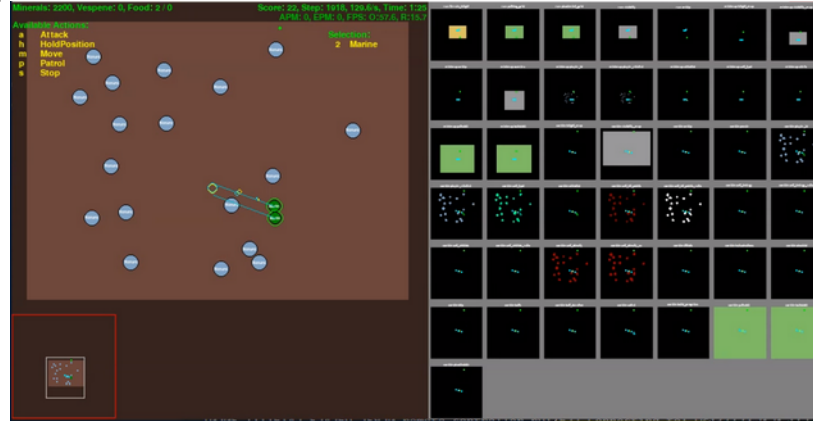
III. INFORMED SEARCH

A. *CollectMineralShards*

Description

A map with 2 Marines and an endless supply of Mineral Shards. Rewards are earned by moving the Marines to collect the Mineral Shards, with optimal collection requiring both Marine units to be split up and moved independently. Whenever all 20 Mineral Shards have been collected, a new set of 20 Mineral Shards are spawned at random locations (at least 2 units away from all Marines).

Fig. 6. Collect Minerals Map

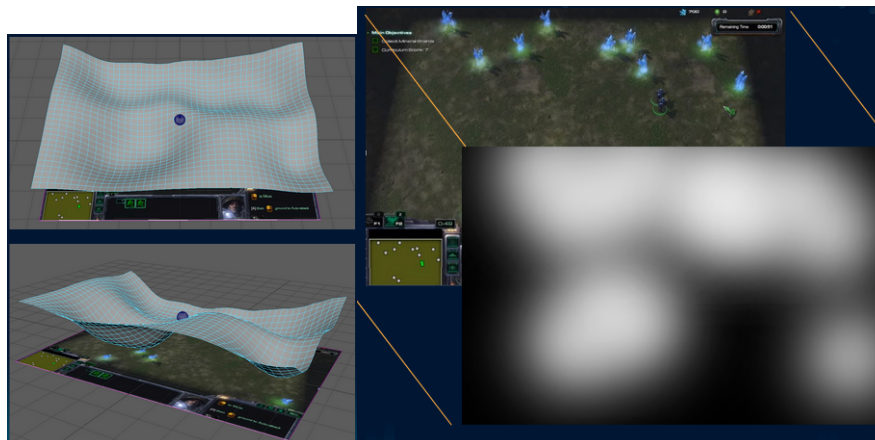


B. *Brush*

The brush is a squared matrix generated with a Gaussian distribution, the output is called a “heightmap”. The idea of the heightmap is to simulate a 3d plane with depth to be used by HillClimbing algorithms, where the higher the point on the map is proportionally direct to a higher density of minerals in that area.

- Map - Matrix with the size of our game screen (initialized with 0s)
- We can stamp out the brush in the map (it adds the values of our brush over a coord in the matrix)
- If we stamp several times, then we get concentrations points in our map to visualize the density of minerals

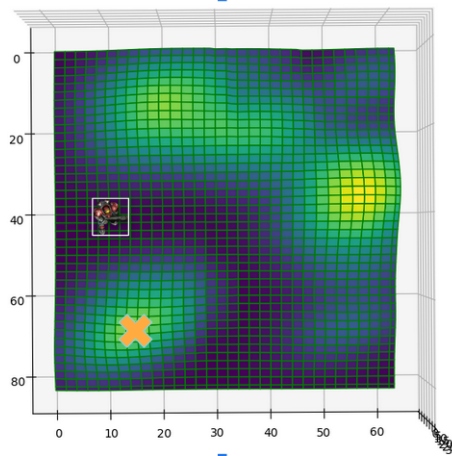
Fig. 7. Height map



C. A*

This version of the A* tries to find a next first-maximum-local by only using an heuristic that is based on the values of the mesh coordinates. The real problem is that it needs a lot of calculus when the map is almost empty because there is a lot of “flat ground” and that makes it difficult to find the maximum.

Fig. 8. Upper view of the heights used in A*



D. HillClimbing and Simulated Annealing

Using the height map, a max-height-point (best option to gather minerals) can be found. This is a Hill Climbing algorithm that could get stuck in a local maxima, so the strategy is to implement a chance given by a probability distribution. In this case the temperature of the boltzmann distribution $e^{\frac{\Delta E}{T}}$ $> rand(0,1)$ is used. The temperature decreases by an alpha factor inside a loop. The following equations were made for tuning the alpha factor. $T_0 =$ *inital temperature* , $a =$ *decreasing factor* , $n =$ *iterations*, $T_f =$ *final temperature*

$$T_0 * a^n = T_f$$

$$a = e^{\frac{\ln(\frac{T_f}{T_0})}{n}}$$

An application of the Simulated Annealing is shown below. The chance section is designed with the equation shown above.

Algorithm 1 Simulated Annealing

```

1 Algo(simulated annealing):
2   for Temp=Tmax to Tmin:
3       #*****CURRENT*****#
4       Energy_current = E(Current) # Function cost at given point
5       #*****NEXT*****#
6       N = Next(c) # go for next neighbour
7       Energy_Flanders = E(Next) # Function cost at Flanders
8       #*****DELTA*****#
9       Delta_Energy = Energy_Flanders - Energy_current
10      #*****UPDATE*****#
11      if(Delta_Energy >0): # if positive
12          Current = Next
13      #*****CHANCE*****#
14      #Delta was negative, lets give another chance and
15      #throw a probabilistic shot, maybe we update current
16      else if (e^(Delta_Energy/Temp) > rand(0,1)):
17          Current = Next
  
```

E. Conclusion

In this stage of the project, it was found that the Hill Climbing was faster than the A*. This could be because the height brush was helpful in reducing the iterations of the search and because getting the local optimal point in the Hill Climbing was cheaper than comparing all the nodes in the A*. The A* is more accurate but the trade off is noticeable in the time performance.

IV. MIN MAX AND PROBABILISTIC

A. FindAndDefeatZerglings

1) *Description*: A map with 3 Marines and an endless supply of stationary Zerglings. Rewards are earned by using the Marines to defeat Zerglings, with the optimal strategy requiring a combination of efficient exploration and combat. Whenever all 25 Zerglings have been defeated, a new set of 25 Zerglings are spawned at random locations (at least 9 units away from all Marines and at least 5 units away from all other Zerglings).

Initial State

- 3 Marines at map center (preselected)
- 2 Zerglings spawned at random locations inside player's vision range (between 7.5 and 9.5 units away from map center and at least 5 units away from all other Zerglings)
- 23 Zerglings spawned at random locations outside player's vision range (at least 10.5 units away from map center and at least 5 units away from all other Zerglings)

Rewards

- Zergling defeated: +1 Marine defeated: -1
- End Conditions Time elapsed All Marines defeated

Time Limit

- 180 seconds

Additional Notes

- Fog of War enabled
- Camera movement required (map is larger than single-screen)

B. Alpha-beta Prunning / Minmax

It was reused the children matrix of the map (Figure 4). But this time it is analysed the cost of each point depending of the not fogged view. Estimating the chance of win or lose, it is weigh with 1 win(Zerling death) , 0 tie(Zerling and Marine death) and -1 lose(Marine death). Knowing this information it was built an alpha beta prunning algorithm.

```

import sys
from params import *

class MinMax:
    def __init__(self):
        self.marines = 3
        self.mapa = None

    def set_mapa(self, mapa):
        self.mapa = mapa

    def minimax(self, depth, alpha, beta, maximizingPlayer, coord):
        hijos = self.mapa.expand(coord)
        if depth == 0 or len(hijos) == 0:
            return self.mapa.chanceMatrix[coord[1]][coord[0]], coord

        if maximizingPlayer:
            maxEval = (-1)*infinity
            for child in hijos:
                eval, fromCoord = self.minimax(depth - 1, alpha, beta, False, child)
                maxEval = max(maxEval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
            return maxEval, fromCoord

        else:
            minEval = infinity
            for child in hijos:
                eval, fromCoord = self.minimax(depth - 1, alpha, beta, True, child)
                minEval = min(minEval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    break
            return minEval, fromCoord

```

Fig. 9. Alpha beta pruning code

C. Cost function with Heuristic

Given the minimap and the explored areas, there is a shadowed grayscale look of the minimap. The info is provided in the following way : {0: Not seen before, 1: Seen before, but not visible, 2: Visible}. With this 3 states, it was calculated the enemies in state 1 and 2. Knowing this is calculated a density of enemies in relation with each shaded area, in this way it is given weights to each array of coordinates in state 1 and 2. Finally the distance of unexplored areas is taken and added to heuristic in an euclidan way.

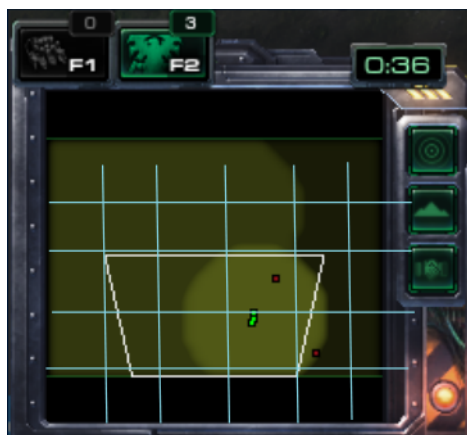


Fig. 10. Heuristic shaded map

D. Conclusion

In this particular case, alpha beta pruning didn't work because it is a real time game, so the response of the system is not quite accurate, system responds very slow to attacks, and just in some seconds all marines were death. In the other hand the heuristic one, performs better than the other, but its main problem was that the marines kill themselves when they almost explored the map. This could be because the set of points of the unexplored area where to close, and they attack themselves.

V. BAYESIAN NETWORKS

A. Map Description

This map is consist in a map medium map size with four camps that contains minerals and vespene gas.

This map is for two players that try to destroy each other. They are two agents against each other.

Initial State

- 1 command center
- 12 SCV

Rewards

- Win and keep your live

End Conditions

- Destroy the enemy

Time Limit

- No

Additional Notes

- We manually activated fog of war And visualize features

B. Map and characters overview

This map is a little more complex than others, because it could have more types of characters. SCV, are used to gather minerals, with this minerals it could be build supply depots, then this ones are used to build barracks, with the barracks the marines are built, and finally with marines attack is possible.

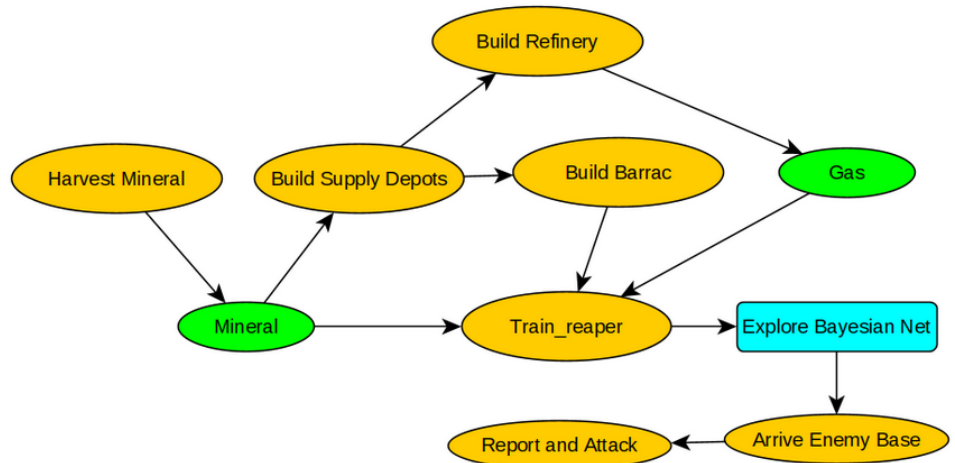
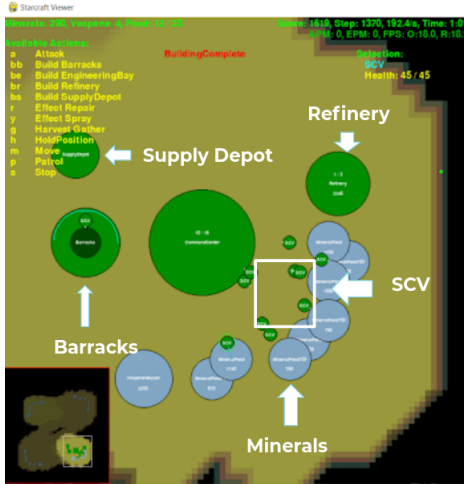


Fig. 11. Map overview simple 64

The main idea is to generate more marines as possible, as soon as one of them explore the map and then the others attack the enemy base. However the reaper is an ideal character to explore because could fly and skip mountains. So this character can avoid go around the mountains and hills, so he do a faster work.

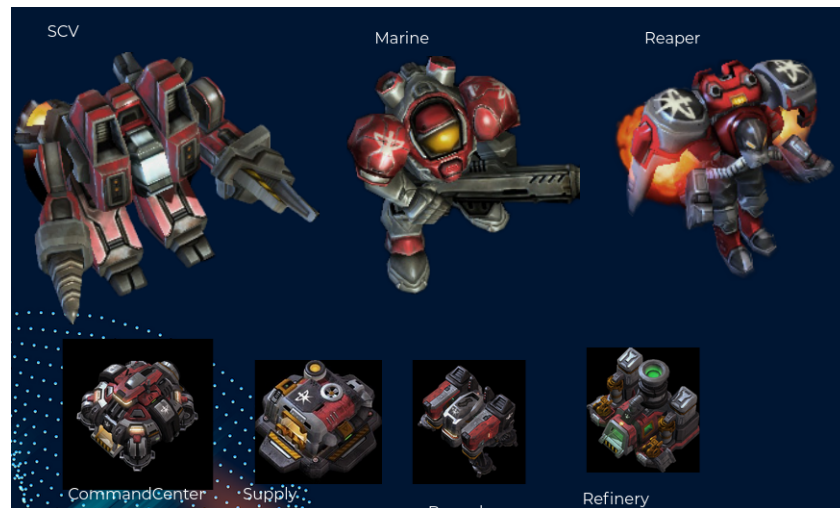


Fig. 12. Characters in simple 64

C. Bayesian newtork scouting

The pourpose of the bayesian network is to control the scouting, in others words, explore and find enemy base. This is a strategy that comes from micromanagment as a branch of RST AI workflow. Based on the manhatan distance a scout character will take a desition in only 4 possible ways. To do things simpler this task uses a reaper , that can skip mountains and hills.

Firstly, we emulate our bayesian network in a GUI (**unbbayes-4.22.18**). Secondly we pass our net to the **Pgmpy** library that is in python. Both softwares allow you to do belief propagation depending on the inputs given to the bayesian net. Finally it was done the integration with the simple64 commands and the **Pgmpy**.

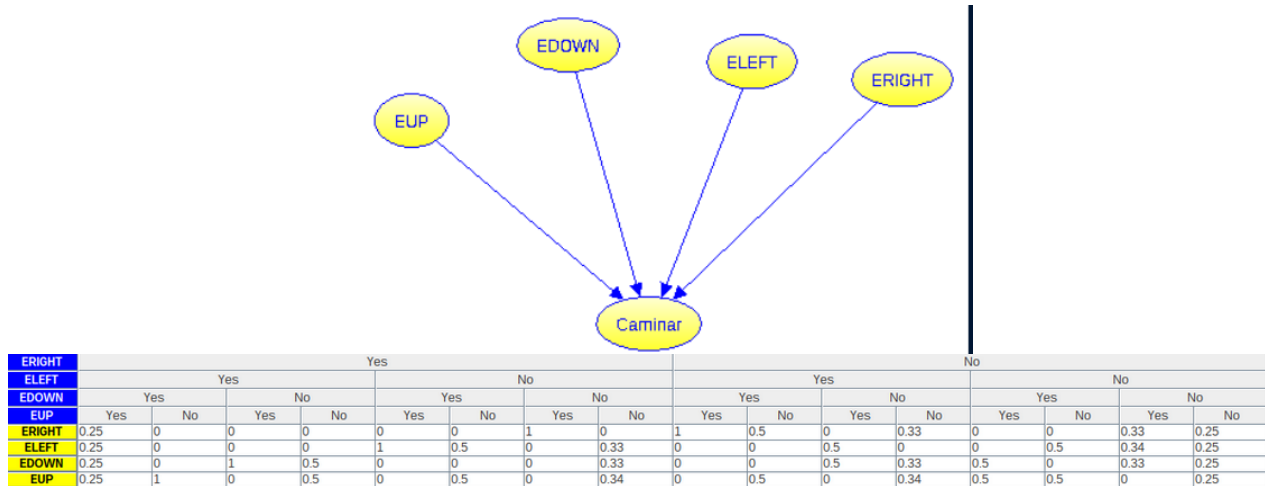


Fig. 13. Scouting bayesian network dag and its probabilities

```

class Explorer():
    def __init__(self):
        self.G = BayesianModel([
            ('exploredUp', 'walk'),
            ('exploredDown', 'walk'),
            ('exploredLeft', 'walk'),
            ('exploredRight', 'walk')])

        self.walk_cpd = TabularCPD(variable='walk',
                                    variable_card=4,
                                    #explored Right | y | n | y | n | y | n | y | n | y | n | y | n | y | n | y | n
                                    #explored Left | y | n | y | n | y | n | y | n | y | n | y | n | y | n | y | n
                                    #explored Down | y | n | y | n | y | n | y | n | y | n | y | n | y | n | y | n
                                    #explored Up | y | n | y | n | y | n | y | n | y | n | y | n | y | n | y | n
                                    values=[[0.25, 0.33, 0.33, 0.5, 0.33, 0.5, 0, 1, 0, 0, 0.5, 0, 0.5, 0.5, 0, 0.25], #walk Right
                                             [0.25, 0.33, 0.33, 0.5, 0, 0, 0.33, 0, 0.33, 0.5, 0, 1, 0.5, 0, 0, 0.25], #walk Left
                                             [0.25, 0.34, 0, 0, 0.33, 0.5, 0.33, 0, 0.33, 0.5, 0, 0, 0, 0.5, 0, 0.25], #walk Down
                                             [0.25, 0, 0.34, 0, 0.34, 0, 0.34, 0, 0.34, 0, 0.5, 0, 0, 0, 1, 0.25]], #walk Up
                                             evidence=[('exploredRight', 'exploredLeft', 'exploredDown', 'exploredUp'),
                                                         evidence_card=[2, 2, 2, 2])

        self.eUp_cpd = TabularCPD(variable='exploredUp',
                                    variable_card=2,
                                    values=[[0.5], [0.5]])
        self.eDown_cpd = TabularCPD(variable='exploredDown',
                                    variable_card=2,
                                    values=[[0.5], [0.5]])
        self.eLeft_cpd = TabularCPD(variable='exploredLeft',
                                    variable_card=2,
                                    values=[[0.5], [0.5]])
        self.eRight_cpd = TabularCPD(variable='exploredRight',
                                    variable_card=2,
                                    values=[[0.5], [0.5]])

        self.G.add_cpds(self.eUp_cpd,
                        self.eDown_cpd,
                        self.eLeft_cpd,
                        self.eRight_cpd,
                        self.walk_cpd)

```

Pgmpy
library

Fig. 14. Pgmpy python code

D. Reduce overfitting

Map is represented as a minimap in the pycsc2 framework, we use a feature of the minimap called pathable. That is the map that characters could used to move around, dark areas are 0 and 1 white ones, white areas are where characters is allow to walk (pathable). However data is presented in a fashion of 64x64 array, so to reduce the calculations it was a subsampled data with a grid, that reduces data size, encoding data in a shorter array of 8X8. This allow us to do less belief prograpagtions in the bayesian network.



Fig. 15. Grid bayesian network

E. Results

After grid processing and bayesian network integration, the scouter finally reach the desired area.



Fig. 16. Bayesian working

F. Conclusions

Designing Bayesian Networks can be a hard job. Assigning the probabilities is not so intuitive for humans and training is required to help the Bayesian Network to work optimally. Bayesian Networks could be mixed with other strategies, like A* and Alpha Beta pruning, but before you may have a good software planing to do.

VI. REINFORCEMENT LEARNING AND PYTORCH

A. Map Description

A map with 9 Marines on the opposite side from a group of 6 Zerglings and 4 Banelings. Rewards are earned by using the Marines to defeat Zerglings and Banelings. Whenever all Zerglings and Banelings have been defeated, a new group of 6 Zerglings and 4 Banelings is spawned and the player is awarded 4 additional Marines at full health, with all other surviving Marines retaining their existing health (no restore). Whenever new units are spawned, all unit positions are reset to opposite sides of the map.

Initial State

- 9 Marines in a vertical line at a random side of the map (preselected)
- 6 Zerglings and 4 Banelings in a group at the opposite side of the map from the Marines

Rewards

- Zergling defeated: +5
- Baneling defeated: +5
- Marine defeated: -1

End Conditions

- Time elapsed
- All Marines defeated

Time Limit

- 120 seconds

Additional Notes

- Fog of War disabled
- No camera movement required (single-screen)
- This map and DefeatRoaches are currently the only maps in the set that can include an automatic, mid-episode state change for player-controlled units. The Marine units are automatically moved back to a neutral position (at a random side of the map opposite the Roaches) when new units are spawned, which occurs whenever the current set of Zerglings and Banelings is defeated. This is done in order to guarantee that new units do not spawn within combat range of one another.



Fig. 17. Banelings and zerglings

B. Defining Net

For the net is used pytorch as the main framework. During the game we will train a neuronal net of 3 layers. Input with 6 Neurons, hidden with 64 and output with 2. The input neruron came from the state that RAW data of minimap gives. The input layer is defined as follows : {len (self.marines), marines_hp, len (self.zergling), zergling_hp, len (self.baneling), baneling_hp}}, and output layer is : {0 (zerling) and 1(banneling)}, this ones are two type of enemies that has different attack properties.

```
class LinearDeepQNetwork(nn.Module):
    def __init__(self, lr, n_actions, input_dims):
        super(LinearDeepQNetwork, self).__init__()

        self.fc1 = nn.Linear(input_dims, 64)
        self.fc2 = nn.Linear(64, n_actions)

        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.device = 'cpu'

        self.to(self.device)
    def forward(self, state):
        layer1 = F.relu(self.fc1(state))
        actions = self.fc2(layer1)

        return actions
```

Fig. 18. Python NN code

C. Defining EPOCHS

Usually the marines will be defeat because zerlings and banelings respawn, the idea is to survive such as make the high score. Once marines are defeat the system get the score and pass it to the net in order to correct the weights and do better decitions. In the image below it is shown, how the net is query to attack certain enemies, note that there is a time increment in the attacking cycle, this ones one provides sufficient respond time to the control system.

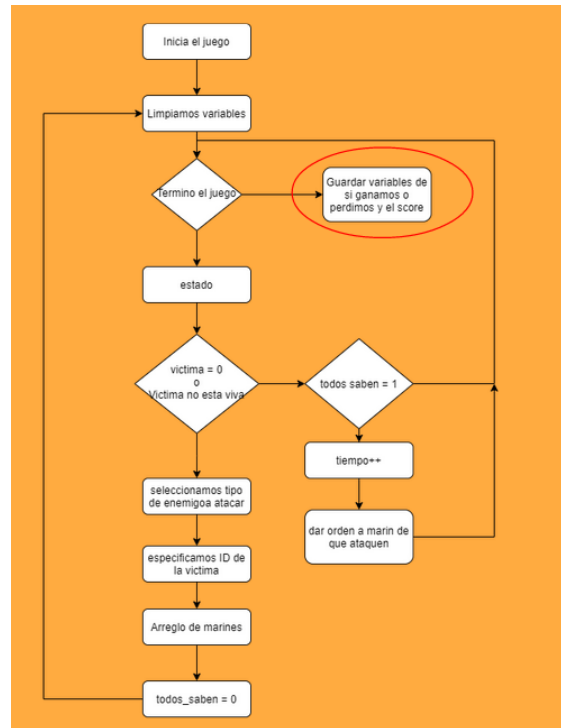


Fig. 19. Epochs cycling

D. Results and Runs

The more epochs have passed, the better the score is. It was tried with different learning rates. And it was found that $lr=0.05$ was the smoother.

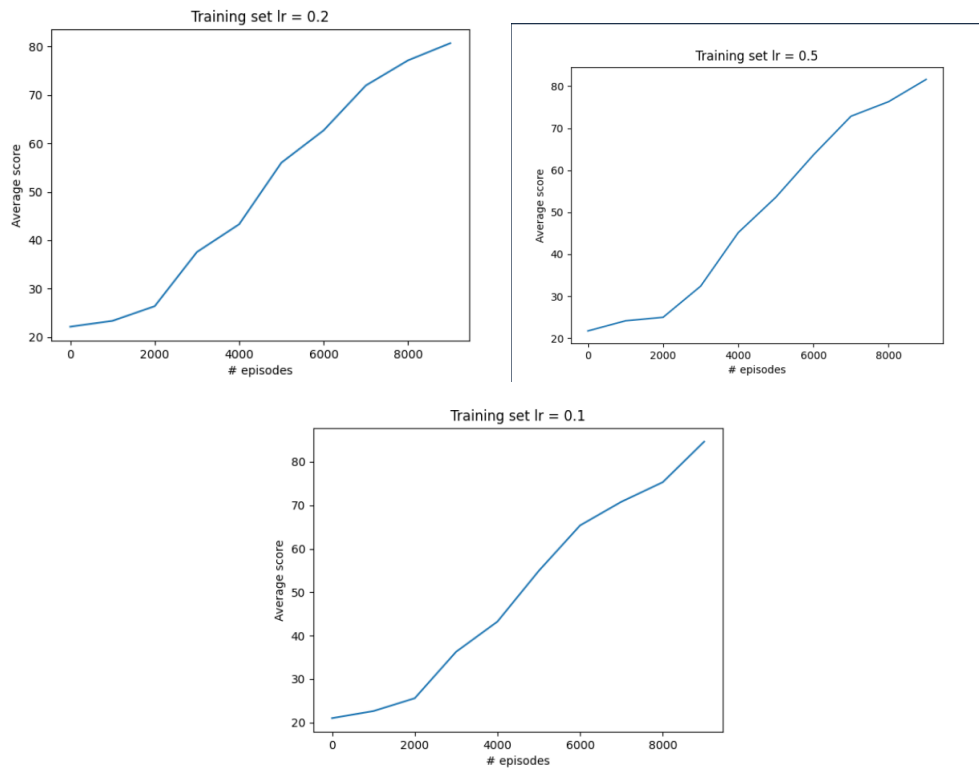


Fig. 20. Learning rate curves

E. Conclusions

The pytorch framework seems to be friendly, as you don't have to worry about some details and is also customizable. The implementation of the network works correctly, and the target for the next release is to integrate this one in the simple64

VII. FINAL RELEASE

A. Map description

This map is consist in a map medium map size with four camps that contains minerals and vespene gas. This map is for two players that try to destroy each other. They are two agents against each other.

Initial State

- 1 command center
- 12 SCV

Rewards

- Win and keep your live

End Conditions

- Destroy the enemy

Time Limit

- No

Additional Notes

- We manually activated fog of war and visualize features

B. Introduction

For this stage we keep using reinforcement learning using pytorch, but now the network should be in a more complex enviroment, where it has to deal with exploring the map, fog of war, allies items, and enemy known items. The input network data is given by the pysc2 framework. It has the agent status of each unit and visible field, the ouput take actions to do task related with certain dependencies. For example build marines depends on food, and build supply depots depends on minerals. To evaluate the perfomance of our NN, it was tracked an score data at the end of each game. This data was not used for the reward, because in the strategy it was considered other parameters for traing the NN.

0	total score
1	idle_production_time
2	idle_worker_time
3	total value of units
4	total value structures
5	killed value units
6	killed value structures
7	collected minerals
8	collected vespene
9	collection rate minerals
10	collection rate vespene
11	spent minerals
12	spent vespene

TABLE I
SCORES FOR MATCH

In general it was used the RELU function as activation function, for each layer, in total there are implemented 3 layers Input,Hidden and Output. Input vary on the implementation of agent, hidden is 64 long and finally output depends on implementation. The random factor epsilon is .1, and will decrease over time. The learing rate was constant at .33. Finally we save the trained weights every 100 matchs.


```

class LinearDeepQNetwork(nn.Module):
    def __init__(self, lr, n_actions, input_dims):
        super(LinearDeepQNetwork, self).__init__()

        self.fc1 = nn.Linear(input_dims, 64)
        self.fc2 = nn.Linear(64, n_actions)

        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')

        self.to(self.device)

    def forward(self, state):
        layer1 = F.relu(self.fc1(state))
        actions = self.fc2(layer1)

        return actions

class nnq():
    def __init__(self, input_dims, n_actions, lr, gamma=0.80, epsilon=.1, eps_dec=1e-5, eps_min=0.01):
        self.lr=lr
        self.input_dims = input_dims
        self.n_actions = n_actions
        self.gamma = gamma
        self.epsilon = epsilon
        self.eps_dec = eps_dec
        self.eps_min = eps_min
        self.action_space = [i for i in range(self.n_actions)]
        self.Q = LinearDeepQNetwork(self.lr, self.n_actions, self.input_dims)

```

Fig. 21. Generic Class implementation

C. Against Random agent

The neuron inputs and output are shown in the table below, note that all data is normalized considering max possible size for item. The NN is 21 input neurons that are read from the pyc2 framework, then 6 output neurons which represent the actions taken by the army.

```

class NNAgent(Agent):
    def __init__(self):
        super(NNAgent, self).__init__()
        self.NN_net = nnq(21,6,0.33) # 21 data in , 6 actions
        self.new_game()
    #data, normalized
    return (len(command_centers)/COMANDCENTERS, #0
            len(scvs)/SCVS_NUM, #1
            len(idle_scvs)/SCVS_NUM, #2
            len(supply_depots)/BUILDINGS, #3
            len(completed_supply_depots)/BUILDINGS, #4
            len(barracks)/BUILDINGS, #5
            len(completed_barracks)/BUILDINGS, #6
            len(marines)/MARINES_NUM, #7
            queued_marines/MARINES_NUM, #8
            free_supply/SUPPLY, #9
            can_afford_supply_depot, #10
            can_afford_barracks, #11
            can_afford_marine, #12
            len(enemy_command_centers)/COMANDCENTERS, #13
            len(enemy_scvs)/SCVS_NUM, #14
            len(enemy_idle_scvs)/SCVS_NUM, #15
            len(enemy_supply_depots)/BUILDINGS, #16
            len(enemy_completed_supply_depots)/BUILDINGS, #17
            len(enemy_barracks)/BUILDINGS, #18
            len(enemy_completed_barracks)/BUILDINGS, #19
            len(enemy_marines)/MARINES_NUM) #Se regresan todos nuestros valores necesarios. #20

class Agent(base_agent.BaseAgent):
    actions = ("do_nothing",
              "harvest_minerals",
              "build_supply_depot",
              "build_barracks",
              "train_marine",
              "attack") #generación

```

Fig. 22. Inputs and outputs NN

The neuronal network will fight against a marine team with the same possible actions as it, but the difference is that the enemy is a random agent, at first sight it must get over it, but it takes 1000 trainings to almost win every match. At 4000 it changes its strategy, it waits to produce more marines and send them in groups to upgrade performance, once it reached that point the agent stuck in the same winning point and strategy.

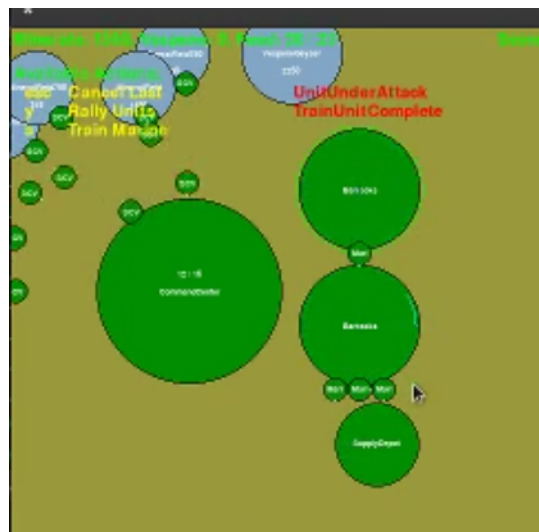


Fig. 23. Marines guarding the base

The reward was given by data input gather from pysc2 framework, every time machine has to decide, the feedback was given by weighted criteria, it was compared the observations states, actual state versus previous one. If enemy increases their units or items rewards is [-1], if all stills the same[0], and if the marines kill enemies or have more resources[1].

```
def update_reward(self, prev_obs, current_obs):

    #NEGATIVE REWARD
    if current_obs[ITEM.MARINES_LEN] < prev_obs[ITEM.MARINES_LEN] :
        self.ai_reward -= 1
    if current_obs[ITEM.SCVS_LEN] < prev_obs[ITEM.SCVS_LEN] :
        self.ai_reward -= 1
    if current_obs[ITEM.ENEMY_MARINES_LEN] > prev_obs[ITEM.ENEMY_MARINES_LEN]:
        self.ai_reward -= 1
    if current_obs[ITEM.BARRACKSES_LEN] < prev_obs[ITEM.BARRACKSES_LEN]:
        self.ai_reward -= 1
    if current_obs[ITEM.ENEMY_COMPLETED_SUPPLY_DEPOTS_LEN] > prev_obs[ITEM.ENEMY_COMPLETED_SUPPLY_DEPOTS_LEN]:
        self.ai_reward -= 1
    if current_obs[ITEM.IDLE_SCVS_LEN] > prev_obs[ITEM.IDLE_SCVS_LEN]:
        self.ai_reward -= 1

    #POSITIVE REWARDS

    if current_obs[ITEM.CAN_AFFORD_MARINE] or current_obs[ITEM.QUEUED_MARINES]:
        self.ai_reward += 1
    if current_obs[ITEM.SUPPLY_DEPOTS_LEN] > prev_obs[ITEM.SUPPLY_DEPOTS_LEN]:
        self.ai_reward += 1
    if current_obs[ITEM.BARRACKSES_LEN] > prev_obs[ITEM.BARRACKSES_LEN]:
        self.ai_reward += 1
    if current_obs[ITEM.MARINES_LEN] > prev_obs[ITEM.MARINES_LEN]:
        self.ai_reward += 1
    if current_obs[ITEM.ENEMY_MARINES_LEN] < prev_obs[ITEM.ENEMY_MARINES_LEN] :
        self.ai_reward += 1
    if current_obs[ITEM.ENEMY_BARRACKSES_LEN] < prev_obs[ITEM.ENEMY_BARRACKSES_LEN] :
        self.ai_reward += 1
    if current_obs[ITEM.ENEMY_SCVS_LEN] < prev_obs[ITEM.ENEMY_SCVS_LEN]:
        self.ai_reward += 1
```

Fig. 24. Reward

D. Against Blizzard AI

Blizzard AI is pretty good playing, it has different levels. However medium is really hard, so it was thought to firstly analyse game strategy in easy, it was observed that game uses other resources and items that we was not using, so we decided to level up the complexity of the network.

```

class Difficulty(enum.IntEnum):
    """Bot difficulties."""
    very_easy = sc_pb.VeryEasy
    easy = sc_pb.Easy
    medium = sc_pb.Medium
    medium_hard = sc_pb.MediumHard
    hard = sc_pb.Hard
    harder = sc_pb.Harder
    very_hard = sc_pb.VeryHard
    cheat_vision = sc_pb.CheatVision
    cheat_money = sc_pb.CheatMoney
    cheat_insane = sc_pb.CheatInsane

```

Fig. 25. Blizzard AI

It was added to NN other factors, such as Vespene gas collection, and marauders characters, which are stronger and bigger. In the image below is shown the new inputs and outputs added

```

class Agent(base_agent.BaseAgent):
    actions = ("do_nothing",
               "harvest_minerals",
               "build_supply_depot",
               "build_barracks",
               "train_marine",
               "attack",
               "build_refinery",
               "harvest_gas",
               "build_lab",
               "train_marauder",
               "attack_marauder")

    len(enemy_minerals)/100,
    len(enemy_marauder)/20,
    len(enemy_TechLab)/10,
    len(marauders)/10,
    len(TechLabs)/3) #Se re

```

Fig. 26. Bigger NN

As final result our agent didn't won vs easy, it does against very_easy, however between some matches it does a higher score.

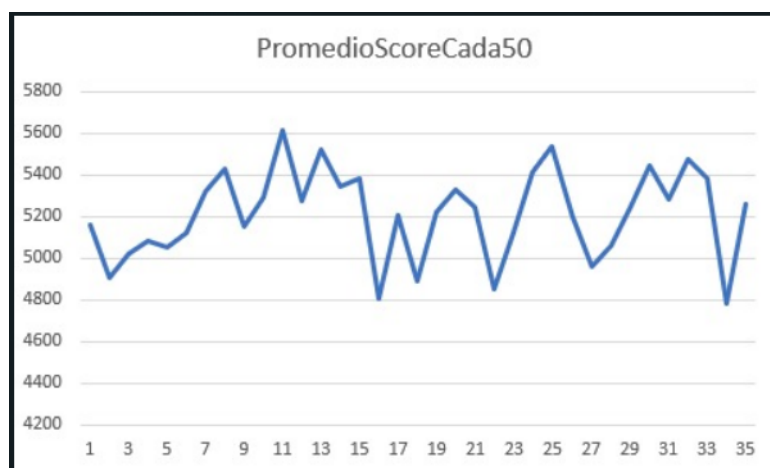


Fig. 27. Score history

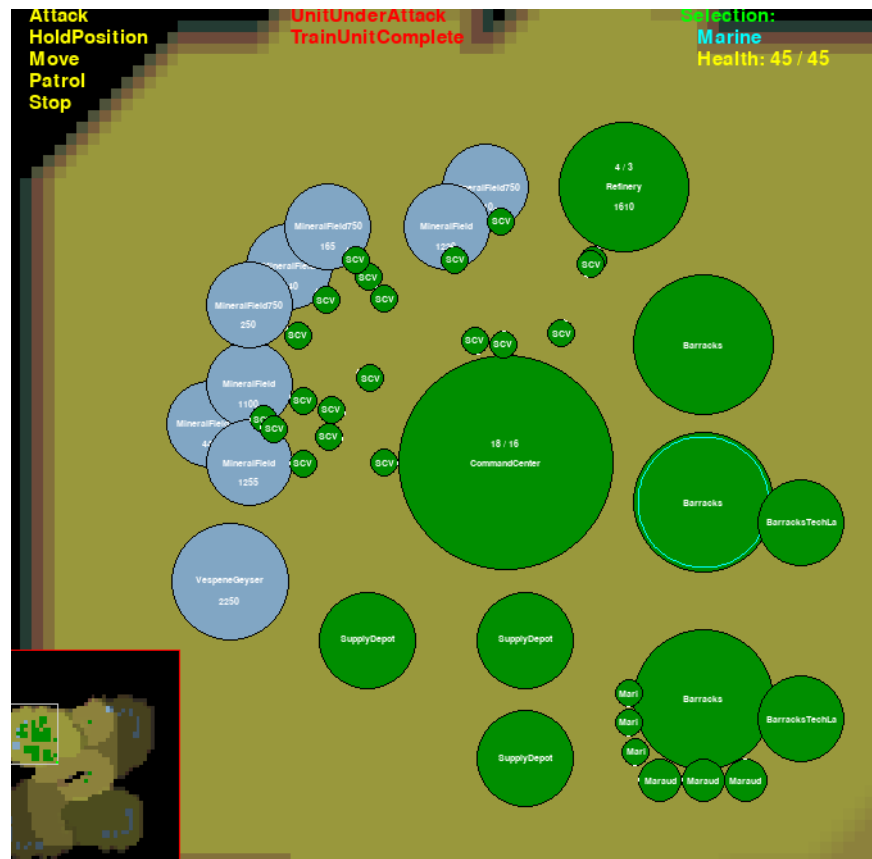


Fig. 28. More units more complex

VIII. CONCLUSIONS

There were two general training approaches following the implementation of the Reinforcement Learning algorithm in pytorch:

- An agent was generated with the ability to learn how to compete against a random-agent (an agent that randomly selects what to do from a more limited range of actions).
- The agent was then put to the test against StarCraft's AI in easy mode.

There were several factors and points that can be concluded from what was done in this project:

- The full decision tree with all the possible actions would be needed for the agent to be able to beat the AI from the game. In this project the actions were limited to some of those of the Terran race.
- The more actions available, the slower the decision making is, thus lowering the efficiency of the agent.
- When the list of actions is expanded, the quantity of possible strategies grows exponentially.
- The match tends to last longer when the agent makes good decisions and by consequence the training also takes more time.

The training time is an important factor because sometimes it is impossible to recognize if our agent is learning as expected until several hundred episodes have been played. This means that we need to take into consideration the amount of time consumed by the training, as we have limited time and resources to develop this project.

ACKNOWLEDGMENTS

We thank Dr. Andres Mendez Vazquez for being our mentor for the full length of the development of this project. We also thank M.Sc. Luis Mario Ramírez Solis for helping us visualize the framework to use at the beginning of this project.

REFERENCES

- [1] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "Starcraft ii: A new challenge for reinforcement learning," 2017.

AUTHORS

Carlos Cardenas-Ruiz is a Computer Systems Engineer. He has worked as a software developer and currently is pursuing his M.Sc. at Cinvestav Guadalajara.

Emilio Tonix-Gleason is a Computer Systems Engineer, He has worked as a software developer and will soon be pursuing his M.Sc. at Cinvestav Guadalajara.

Julia Rodriguez-Abud has a Digital Arts major. She has worked as a software developer and currently is pursuing her M.Sc. at Cinvestav Guadalajara.