Machine Learning Engineer Nanodegree

Capstone Project

Ji Ma
August 23st, 2017

I. Definition

Project Overview

Recently, OpenAI's Artificial Intelligence beat the best player in popular strategy game on 1v1 match, stands for a oustanding milestone after the legend of Deepmind's Alpha Go. It stands for a new era of the AI powered bot in the game way more complex than Go. Also, what interested me so much is that Deepmind also released a brand new Reinforcement Learning Environment for the game StarCraft II, which is one of my favorate game. But the reality is I'm not very good at competing with human players in StarCraft II. Since I am a big fan of Deep Learning and Neural Network, I decided to use the Deep RL to implement the AI to achieve some RL tasks in this brand new handy environment by myself :)

About the sc2le paper:

Because a lot of my work here will be based on the research result discussed in this paper(https://deepmind.com/documents/110/sc2le.pdf), I will elaborate their work a little bit. They introduced the sc2 learning environment in details, talked about RL baseline agents and discussed supervised learning from replays.

Problem Statement

As declared in the paper along with the Learning Environment, the current best result worked by the most professional researchers from Deepmind could not beat a simple bot in Easy level. It takes so much work and computing powers to achieve the goals like that considering the complexity of the StarCraft II. So my goal should be realistic, tackle the Mini Game like MoveToBeacon and FindAndDefeatZerglings should be a good practice to get started.
Among all those mini games, I will specificly tackle **BuildMarines**, which described below according to pysc2 documents:

Description

A map with 12 SCVs, 1 Command Center, and 8 Mineral Fields. Rewards are earned by building Marines. This is accomplished by using SCVs to collect minerals, which are used to build Supply Depots and Barracks, which can then build Marines.

Initial State

12 SCVs beside the Command Center (unselected) 1 Command Center at a fixed location 8 Mineral Fields at fixed locations Player Resources: 50 Minerals, 0 Vespene, 12/15 Supply

Rewards

Reward total is equal to the total number of Marines built

End Condition

Time elapsed

Time Limit

900 seconds

Additional Notes

- Fog of War disabled
- No camera movement required (single-screen)
- This is the only map in the set that explicitly limits the available actions of the units to disallow actions which are not pertinent to the goal of the map. Actions that are not required for building Marines have been removed.

Metrics

Reward total is equal to the total number of Marines built

II. Analysis

Data Exploration

The data I will first explore around is pysc2 itself.

According to it's source code:

Action space The action space is huge which is 524 actions for the regular game.

We could get valid actions dynamicly in each step

Status Space Depends on unit type, the status space could be enormous

1. screen*xy = 84 (the screen size will be screen*xy* screen_xy) we could change it later
2. unit*type*size = 1850 (The unit type size)
3. so, the status space would be screen*xy^2 * unit*type_size

In each Step in Deep RL, pysc2 provided a handy `TimeStep` class which will provide 4 major features for Deep RL:

- step_type: First, Mid, Last(useful to find the step progress in the episode)
- Reward
- Discount: the discound initialized in env
- Obervation: infomation contains
- screen(different type), unit_type screen will be used
- minimap
- etc

In order to know the performance of Random Agent and in the mean while get a sense of data generated in following aspects:

- screenList: the numpy array, will help me narrow down the unit type for minigame
- actionList: the numpy array, will help me narrow down the action space for minigame
- rList: the overall reward(performance matric)

Each of the Build Marine mini game has 15000 frames limit, and I used step_mul=8 as default(8 frames per action).
So, 15000 frames means 1875 steps per game.
I want 4000 games/episode sample data here, so I use 1875000 steps on 4 threads to collect data The result is similar to what has been described in the sc2le paper. Average 1 reward for random policy. Pretty bad.

Agent	Metric	MoveToBeacon	CollectMineralShards	FindAndDefeatZerglings	DefeatRoaches	DefeatZerglingsAndBanelings	CollectMineralsAndGas	BuildMarines
Random policy	mean	1	17	4	1	23	12	< 1
	max	6	35	19	46	118	750	5
Random search	mean	25	32	21	51	55	2318	8
	max	29	57	33	241	159	3940	46
DeepMind human player	mean	26	133	46	41	729	6880	138
	max	28	142	49	81	757	6952	142
StarCraft GrandMaster	mean	28	177	61	215	727	7566	133
	max	28	179	61	363	848	7566	133
Atari-net	best mean	25	96	49	101	81	3356	< 1
	max	33	131	59	351	352	3505	20
FullyConv	best mean	26	103	45	100	62	3978	3
	max	45	134	56	355	251	4130	42
FullyConv LSTM	best mean	26	104	44	98	96	3351	6
	max	35	137	57	373	444	3995	62

Some of the common action would be used in the Build Marines mini games are:

0~13 :

```python
 Function.ui_func(0, "no_op", no_op),
    Function.ui_func(1, "move_camera", move_camera),
    Function.ui_func(2, "select_point", select_point),
    Function.ui_func(3, "select_rect", select_rect),
    Function.ui_func(4, "select_control_group", control_group),
    Function.ui_func(5, "select_unit", select_unit,
                     lambda obs: obs.ui_data.HasField("multi")),
    Function.ui_func(6, "select_idle_worker", select_idle_worker,
                     lambda obs: obs.player_common.idle_worker_count > 0),
    Function.ui_func(7, "select_army", select_army,
                     lambda obs: obs.player_common.army_count > 0),
    Function.ui_func(8, "select_warp_gates", select_warp_gates,
                     lambda obs: obs.player_common.warp_gate_count > 0),
    Function.ui_func(9, "select_larva", select_larva,
                     lambda obs: obs.player_common.larva_count > 0),
    Function.ui_func(10, "unload", unload,
                     lambda obs: obs.ui_data.HasField("cargo")),
    Function.ui_func(11, "build_queue", build_queue,
                     lambda obs: obs.ui_data.HasField("production")),
    # Everything below here is generated with gen_actions.py
    Function.ability(12, "Attack_screen", cmd_screen, 3674),
    Function.ability(13, "Attack_minimap", cmd_minimap, 3674),
```

42,

```python
Function.ability(42, "Build_Barracks_screen", cmd_screen, 321),
```

91,

```python
Function.ability(91, "Build_SupplyDepot_screen", cmd_screen, 319),
```

264,
269,
274,
331,
332,
333,
334,
343,
344,
451,
452,
453,

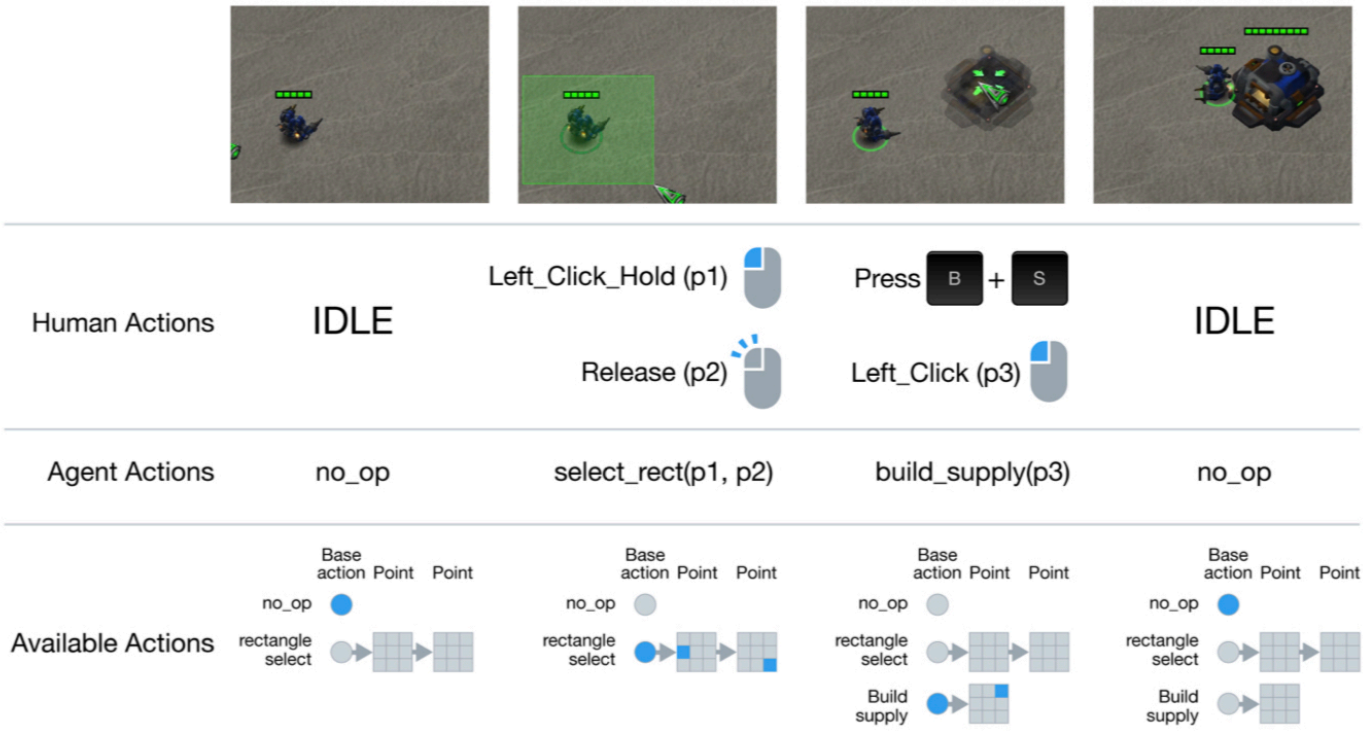
477,
490

You could look up what other action id from the link below.

For the detailed explanation of action id's correspondence, please refer to https://github.com/deepmind/pysc2/blob/master/pysc2/lib/actions.py

Exploratory Visualization

The input screen will be 64*64. Each entry will have a unit represenataion described above.

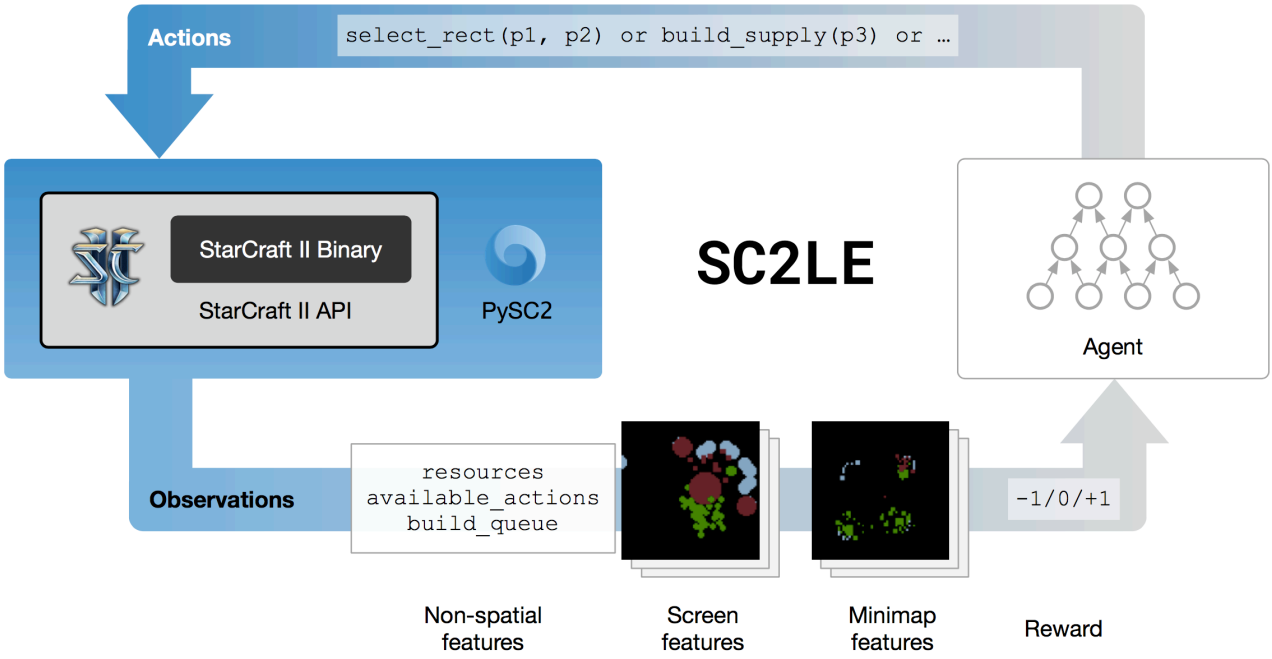


Here is an example of how a action will be performed by the learning agent setup in the sc2 environment. As you can see in the diagram, the human actions has been wrapped up as functions provided for agent.

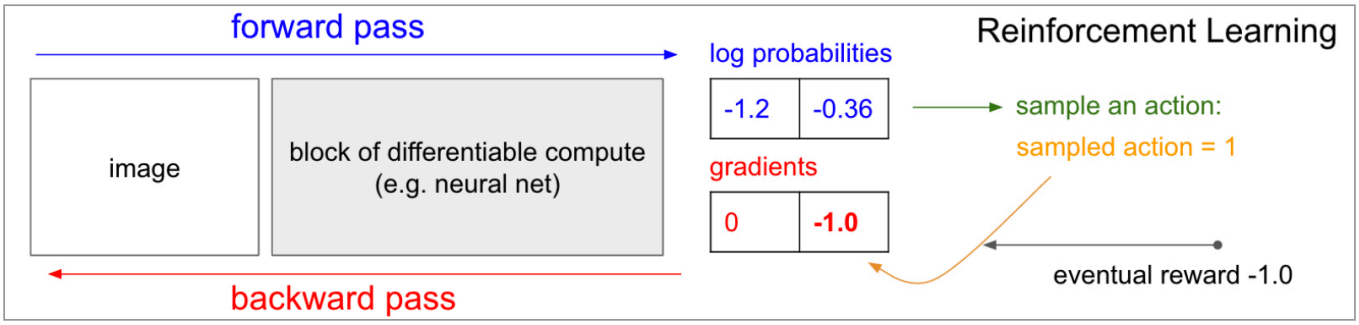
The agent will first based on the base action(the action feed to the step loop in the environment) to get what actions available. Then it will decide to choose one of those available actions. And then, after we choose the action, rectangle select in this example, we could pass in the arguments(denoted by two point in the abstracted 3*3 screen) to select the unit. Then after that, we feed this base action to the environment, to enable the action to build the supply, and pass the location/point argument in it.

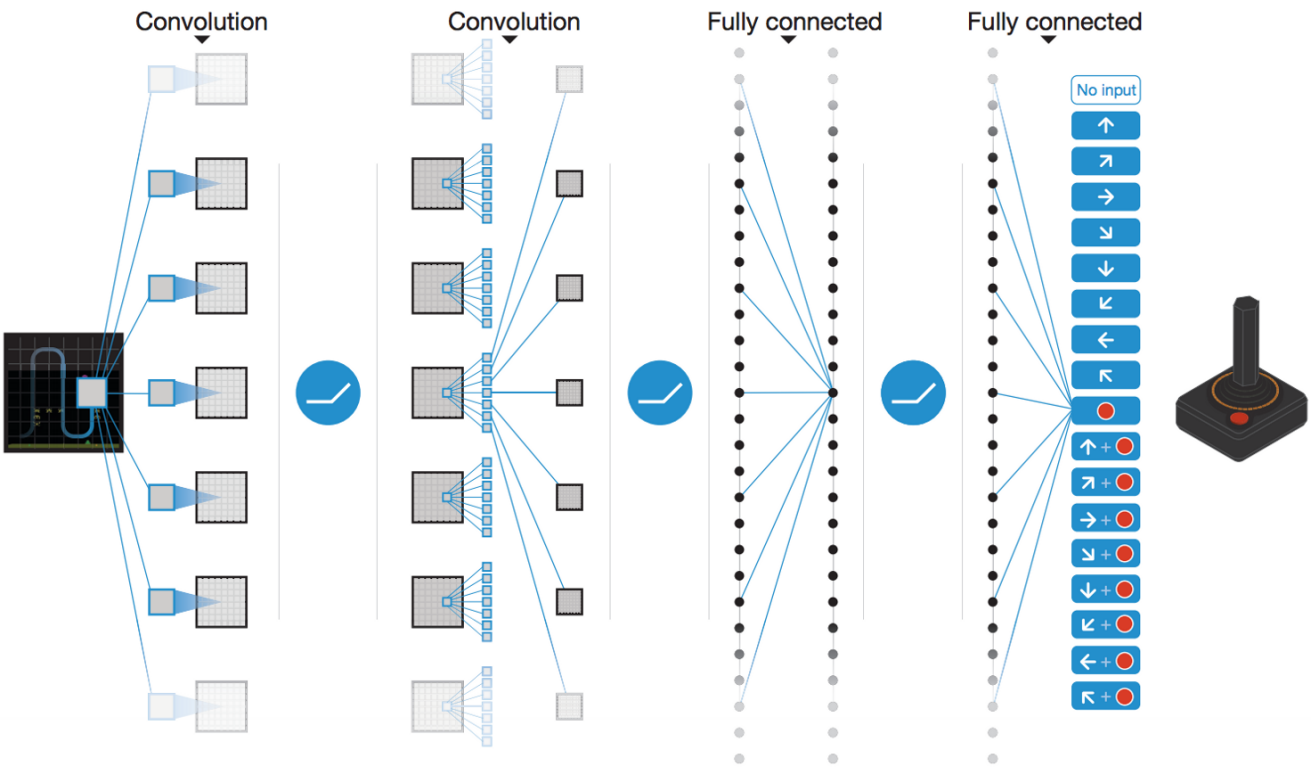
This is basically how the agent action would be executed. action -> environment -> avail actions(observation) -> choose action -> repeat

Algorithms and Techniques



Deep Q Network





Before we get started, what is Q learning? According to Wikipedia:

> **Q-learning** is a model-free reinforcement learning technique. Specifically, *Q*-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. A policy is a rule that the agent follows in selecting actions, given the state it is in. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state. One of the strengths of *Q*-learning is that it is able to compare the expected utility of the available actions without requiring a model of the environment. Additionally, *Q*-learning can handle problems with stochastic transitions and rewards, without requiring any adaptations. It has been proven that for any finite MDP, *Q*-learning eventually finds an optimal policy, in the sense that the expected value of the total reward return over all successive steps, starting from the current state, is the maximum achievable.https://en.wikipedia.org/wiki/Q-learning#cite_note-1

However, for our game environment, the action space and feature space could be enormous for traditional Q-Learning to get all possible Q Learning Table for it(It is fairly impossible for the Q table to collect all trial and error information from this type of game.) So, we need to use Deep Q Network to solve this problem

from Q learning to DQN:

1. Going from a single-layer network to a multi-layer convolutional network.
2. Implementing Experience Replay, which will allow our network to train itself using stored memories from it's experience.

3. Utilizing a second “target” network, which we will use to compute target Q-values during our updates.

Bellman Equation:

states that the expected long-term reward for a given action is equal to the immediate reward from the current action combined with the expected reward from the best future action taken at the following state.

$Q(s,a) = r + \gamma(max(Q(s',a'))$

loss function

Our loss function will be sum-of-squares loss, where the difference between the current predicted Q-values, and the “target” value is computed and the gradients passed through the network.

$Loss = \sum(Q\text{-target} - Q)^2$

Double DQN

The goal of Dueling DQN is to have a network that separately computes the advantage and value functions, and combines them back into a single Q-function only at the final layer.

Instead of taking the max over Q-values when computing the target-Q value for our training step, we use our primary network to chose an action, and our target network to generate the target Q-value for that action. By decoupling the action choice from the target Q-value generation, we are able to substantially reduce the overestimation, and train faster and more reliably.

$Q\text{-Target} = r + \gamma Q(s',argmax(Q(s',a,\Theta),\Theta'))$

Supervised Learning: learn both a **value function** (i.e., predicting the winner of the game from game observations), and a **policy** (i.e., predicting the action taken from game observations).

Benchmark

Random Agents

According to sc2le paper, they suggested two main random agents.
I will particularly use **Random Policy**
Random Policy will uniformly pick random action among all valid actions. (Samples in action space)

III. Methodology

Data Preprocessing

In walkthrough ipython notebook.

I derived the overall valid unit_type input and action input.

Valid unit type for Build Marines:

```
#unit frequency
[[       0 31386888]
 [      18  1517332]
 [      19  1073447]
 [      21   338735]
 [      45   636247]
 [      48     4367]
 [     341  1769464]]
```

Valid action type with ambiguous arguments for Build Marines:

```
{0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 10,
 11,
 12,
 13,
 42,
 91,
 264,
 269,
 274,
 331,
 332,
 333,
 334,
 343,
 344,
 451,
 452,
 453,
 477,
 490}
```

Not every single unit types/actions will be used in the mini games like this, so I reduced the space for both of them to reduce the training time.

As for the screen prepossessing, there is no need for us to do it manually. The learning environment already done the job. For example, in this case our feature input is 64*64 2d array(screen), and each of the location will have a unit id to represent what object in the screen. So, we could just simply feed our model with this environment generated screen features. You could get a glimpse from the free from free form visualization section below to see what processed screen features look like.

Implementation

Deep Q-Learning Algorithm

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
    select an action a
        with probability ϵ select a random action
        otherwise select a = argmaxa'Q(s,a')
    carry out action a
    observe reward r and new state s'
    store experience <s, a, r, s'> in replay memory D

    sample random transitions <ss, aa, rr, ss'> from replay memory D
    calculate target for each minibatch transition
        if ss' is terminal state then tt = rr
        otherwise tt = rr + γmaxa'Q(ss', aa')
    train the Q network using (tt - Q(ss, aa))^2 as loss

    s = s'
until terminated
```

We need to implement the experience buffer first to provide the random sample history from the previous experience. To do this we could simply implement a array list to do that.

Then for each Q network, I use the following architecture

The conv2d layers are

- num*outputs=32, kernel*size=8, stride=4
- num*outputs=64, kernel*size=4, stride=2
- num*outputs=64, kernel*size=3, stride=1
- num*outputs=512, kernel*size=4, stride=1

for the deep q network's convolutional layers.

Then we take the output from the final convolutional layer and split it into separate advantage and value streams.

Then combine them together to get our final Q-values.

The final step is obtain the loss by taking the sum of squares difference between the target and prediction Q values and minimize the loss along the training process.

The next step is we combine our Deep Q Learning process with the sc2 learning environment.

for each step, we got a new observation(In this case, the variable timestep will hold the observations).

Then, we extract the information s(screen feature before action), r(reward), a(action performed to get the s), s'(screen feature after we perform action) and d(the episode is end or not)to store them in the experience buffer(similar to deep q table)

After a certain amount of random experiencing, we are going to perform the learning procedure described above.

- Get a random batch of experiences.
- Perform the Double-DQN update to the target Q-values.
- Update the network with our target values.
- Update the target network toward the primary network.

```
sample random transitions <ss, aa, rr, ss'> from replay memory D
   calculate target for each minibatch transition
       if ss' is terminal state then tt = rr
       otherwise tt = rr + γmaxa'Q(ss', aa')
   train the Q network using (tt - Q(ss, aa))^2 as loss
```

Refinement

I found that if we just use random agent as the experience buffer, it is very difficult to get the agent to get a reward higher than 1. So the learning process is going to be extremely long if we only utilized the random agent for experience buffer.

So, I gave a little customized push for the random agent:

if the barrack(the building that we need to build the marine) is available:

- if the barrack is selected in the screen: Build marines :)

- if barrack is not selected yet: Select all barracks

I initially tried to use kernal size 7 for last convolutional layer. However, the result is really problematic. So I choose to use kernal_size 4 for my Q learning model.

Another problem is the buffer size for our experience buffer, I found that it initially could not learn any thing from its experience, so what I did is to make it proportional to the max episode length by a factor of 20. So, it could at least remember 20 episodes game memory and remember from that.

IV. Results

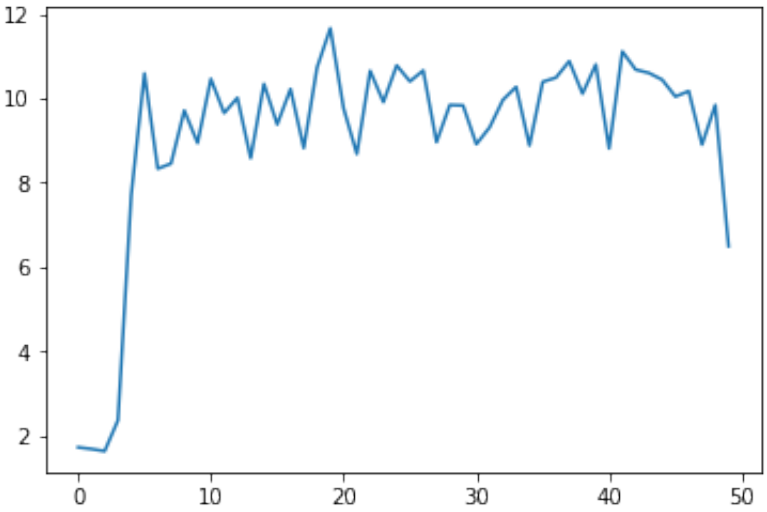
Model Evaluation and Validation

Since the major input for this game environment is the screen pixels input, a model with convolutional networks would be fit for dealing with the visual input like this. The deep q learning technique combines the beauty of q learning which is a classic reinforcement learning solution and ideas of the convolutional network, so Deep Q Learning is a good fit for the mini game like this.

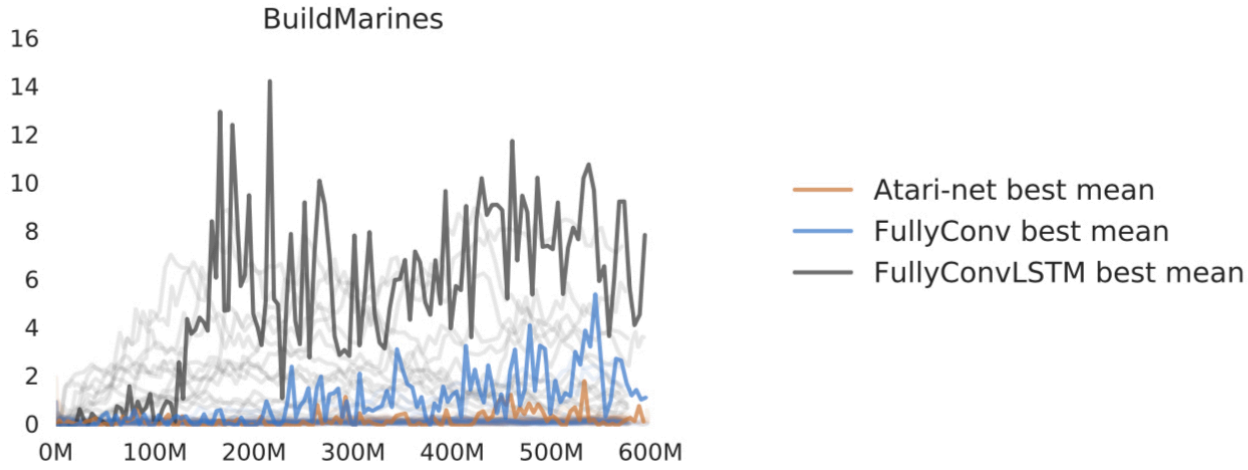
We achieved average 10 reward is a huge improvement from the random agent. More over, if we try to use the trained model to play the game(set load_model=True), the result of the RL agent could achieve around 10 marines output, which is pretty nice. Feel free to train the model first, and then load it for testing. However, it seems like it could achieve around average 10 reward from 1k epsisode, so I might adjust the training episodes later to reduce the training time and avoid over fit.

Justification

The final result is definitely beating the previous random agent which only got around 1 average reward over time. The Deep Q Network has increased it’s reward from nearly 0 to around average 10 reward per episode. This is far beyond the result from our previous random agent benchmark which only has average 1 reward.



Plus, when we compare our approach to this problem to the solution provided by sc2le paper, our result is pretty close to their performance and even much more stable:



V. Conclusion

Free-Form Visualization

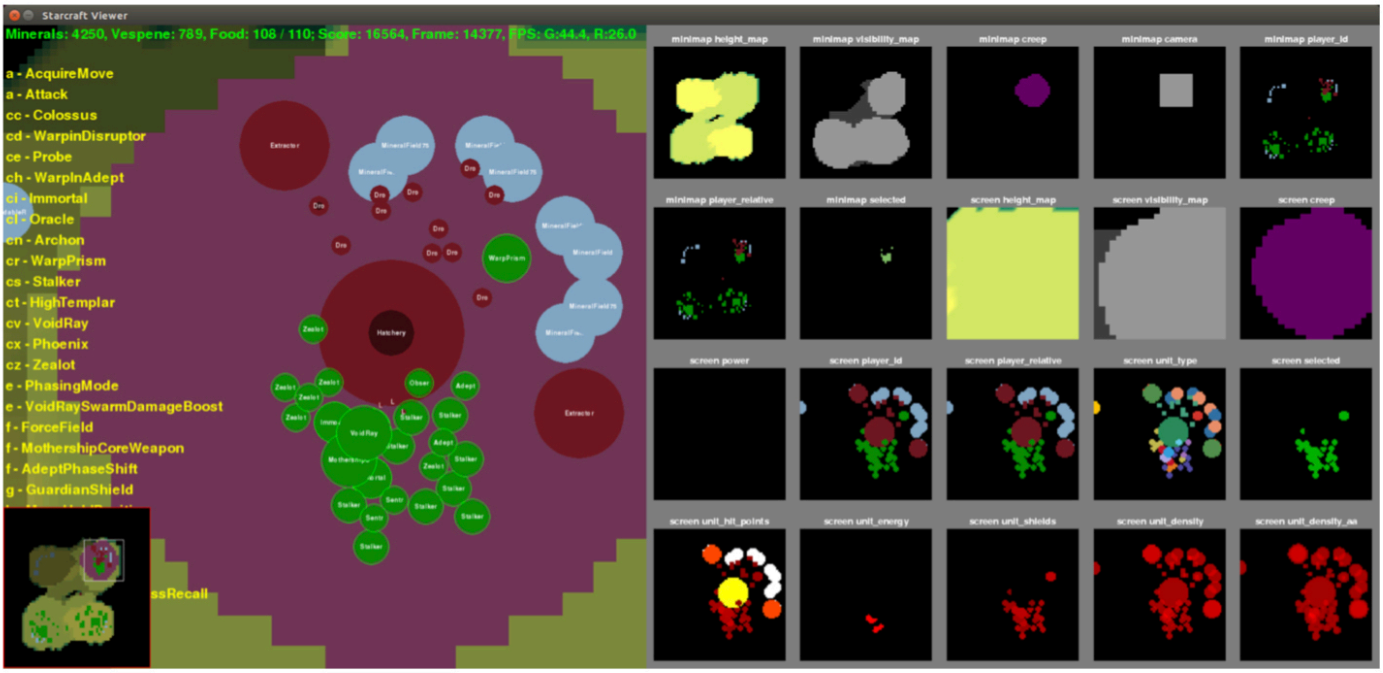


Figure 2: The PySC2 viewer shows a human interpretable view of the game on the left, and coloured versions of the feature layers on the right. For example, terrain height, fog-of-war, creep, camera location, and player identity, are shown in the top row of feature layers. A video can be found at `https://youtu.be/-fKUyT14G-8`.

Reflection

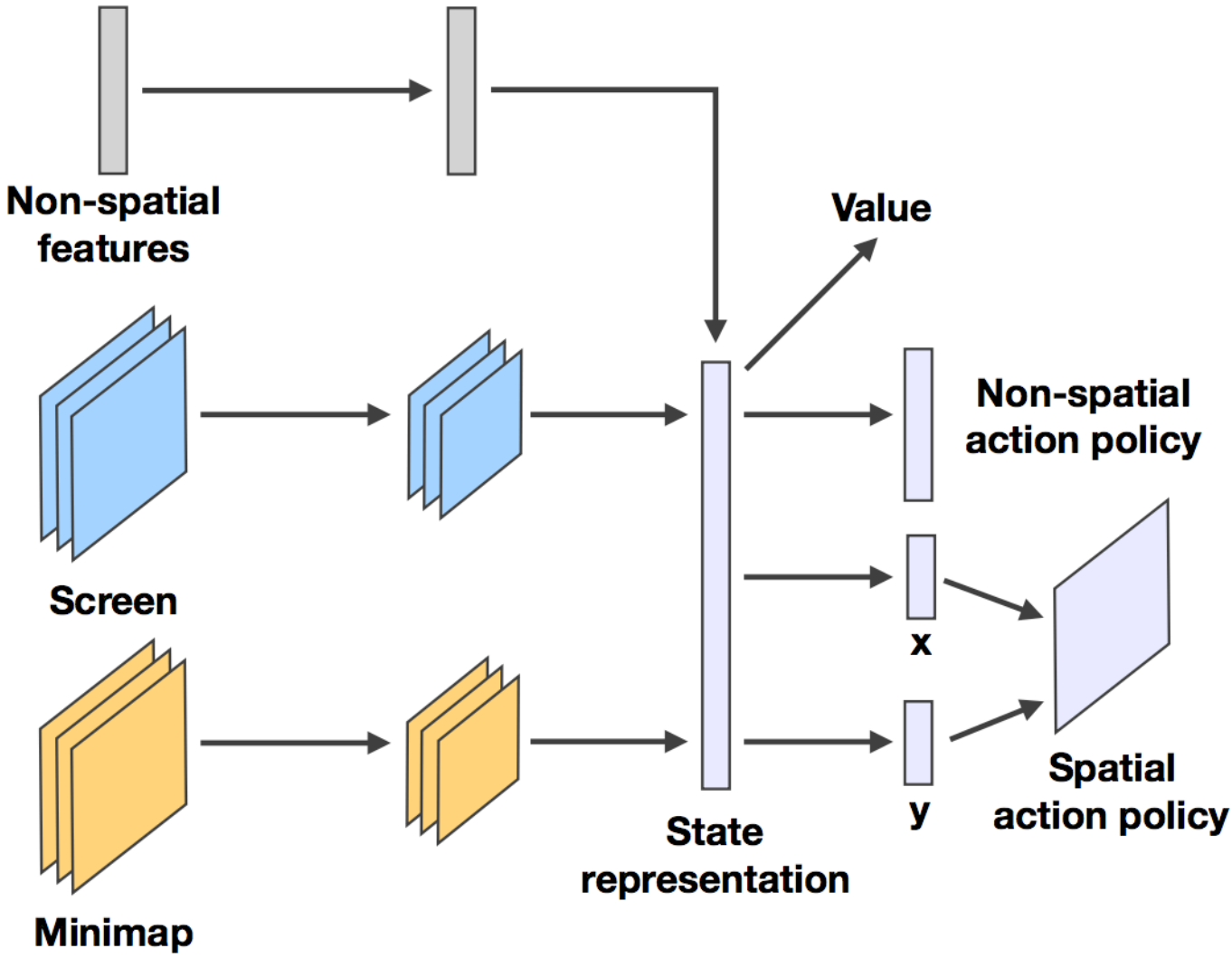
Doing this project is not a easy job at all. This Starcraft reinforcement learning environment just released in around 2 months. When I started to work on this project 2 months ago, I though it is going to be easy, however I was WRONG. DeepMind did NOT provide a well written API documentation at all for the learning environment. So, what I need to do is dive into the source code and figure out what exactly happened and make annotation by myself. And some of the discovery process(majorly debugging) process is painful and time consuming. I literally got stuck on a function related to the unit selection function just because the return value from this function are having a y,x order instead of x,y... Gosh, hope somebody write down a document for this learning environment.

Trying the new “toy” out there is hard, I tried to reached out Udacity mentor but they neither have any experience with this before. And the tutorial online is very limited. But finally, I made it. :)

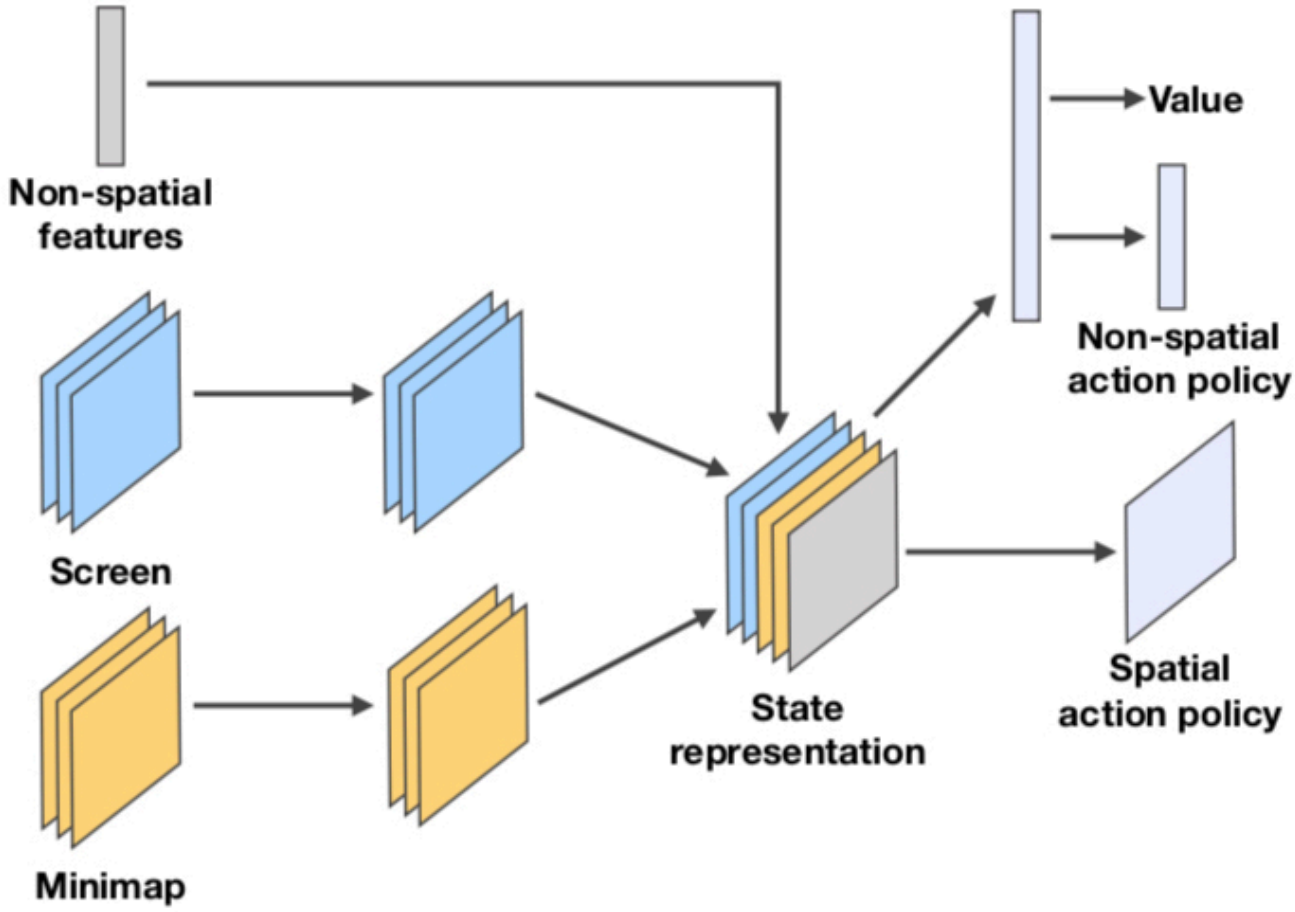
Improvement

I may try to improve the performance by adding the batch_normalization for each of the conv2d layer. Also, it would be beneficial to feed the network with more inputs like more feature layers(minimaps, height map, etc) and nonspacial information tensors(health, gas, money) provided in sc2. In sc2le paper, they implemented this instead of only one screen layer I used in my project. I assume a proper tweak from this implementation will generate a better result.

Atari-Net and Fully Conv



(a) Atari-net



(b) FullyConv

References

github.com/Blizzard/s2client-proto
github.com/deepmind/pysc2

Simple Reinforcement Learning with Tensorflow series by Arthur Juliani
Deep Reinforcement Learning: Pong from Pixels by Andrej Karpathy http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/

https://hackernoon.com/deep-learning-cnns-in-tensorflow-with-gpus-cba6efe0acc2

https://chatbotslife.com/building-a-basic-pysc2-agent-b109cde1477c