Playing Startcraft II with Reinforcement Learning

Haiyuan Mei
Stanford University
hmei0411@stanford.edu

Abstract

Real Time Strategy (RTS) games such as StarCraft 2 has long been a major challenge in AI research. The study of AI in SC2 also opens a door to unparalleled opportunities to many challenging frontiers, both in AI research and in real life. This paper explores and replicates DeepMind's work of SC2LE (StarCraft II: A New Challenge for Reinforcement Learning) [1]. In this paper I will firstly introduces the model for the game environment, including specifications of observations, actions and rewards; then look into the details of an important RL policy gradient algorithm: Asynchronous Advanced Actor-Critic(A3C); and a fully connected convolutional neural network agent(CNN) which is used by A3C in making decisions for the next move. Lastly I would like to also explore more on deep q learning and make some comparisons between the policy based algorithm(A3C) and the value based algorithm (DQN). In order to train deep reinforcement learning models, TensorFlow is used for both A3C and DQN.

1 Introduction

Blizzard's Startcraft series game is one of the most challenging Real Time Strategy game, and has gained immense commercial and cultural success over the past 20 years. However, unlike its popularity, the study of Artificial Intelligence in tackling the full StarCraft game progresses somewhat slow and seems relatively far from finished. StarCraft is a multi-agent game can can be played in a variety of settings such as two parts 1 vs 1 similar to zero-sum game, n vs n a combination of zero-sum(between groups) and search(inside a group), and also more complex settings such as multiple parts, such as 1 vs 1 vs 1, or n vs n vs n, etc. More over, the nature of imperfect information, large state/action space and delayed credit assignment which requires long-term strategies over thousands of steps etc, are all hard problems that need to be studied and tackled. The most recent in this field was about Deepmind's AlphaStar in Jan/2019, which was defeated at the last round by Pro StarCraft II champion Grzegorz "MaNa". It is a huge leap of progress in this field, yet there are still a lot more interesting and hard problems yet to be solved, which provides an unparalleled opportunity to explore many challenging new frontiers in AI.

In this paper, by using the PySC2 [Vinyals et al. (2017)] Library by DeepMind, by representing game states "images" using a fully connected Convolutional Neural Network (CNN), I will explore using A reinforcement algorithm to produce the optimal actions for a subset of pre-defined mini games provided by PySC2. Apart from exploring PySC2's proposed A3C and FullyConv implementation, attempts also made in the experiment to compare A3C to it's sister variance A2C which is an synchronous version of the Advanced Actor-Critic, and the previously very successful Deep Q-learning Network(DQN) in Atari games [Mnih et al. (2013)].

2 Related work

There have been a number of prior attempts at tackling computer games with reinforcement learning in recent years. The best example of games driving reinforcement learning research is the Arcade

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

Learning Environment ALE [Bellemare et al. (2012)], which allows easy and replicable experiments with Atari video games, and has been an incredible boon to recently reinforcement learning. There are also attempts to tackle previous version StarCraft(BroodWar) game, an overview can be obtained by the surveys by Ontañón et al. (2013) and Robertson and Watson (2014). The standard API before StartCraft II API has been The Brood War API and it's related wrappers [Synnaeve et al. (2016)].

The main work to base this project off is DeepMind's paper [Vinyals et al. (2017)]. It's main contribution is the release of SC2LE, which exposes StarCraft II as a research environment. There are three sub components in this release: a Linux StarCraft II binary, the StarCraft II API, and PySC2. The StarCraft II binary although provided in Linux, the training/evaluation as can be seen in the experiment later, can also be done in Windows. The StarCraft II API (`https://github.com/Blizzard/s2client-proto`) is an interface that provides full external control of StarCraft II, it exposes functionality for developing software for: 1, Scripted bots. 2, Machine-learning based bots. 3, Replay analysis. 4, Tool assisted human play. PySC2 is to some extent a python wrapper to the StarCraft II API which provides a python library for accessing Blizzard's StarCraft II API. It makes RL in StarCraft more straightforward: observations and actions are defined in terms of low resolution grids of features; rewards are based on the Blizzard score from the StarCraft II engine; several simplified mini-games are provided which is perfect first step to tackle StartCraft II. After the success of Atari games [Mnih et al. (2013)] which use Deep Q learning plus Atari Network, SC2LE proposed using asynchronous methods for deep reinforcement learning in StarCraft II (A3C), and proposed a fully connected convolutional neural network to improve the learning performance for the more complex frame image.

The Asychronous version deep reinforcement learning algorithm Asychronous Advanced Actor-Critic (A3C) used in this project is presented in Mnih et al. (2016). The name of A3C comes from the initials of the algorithm 'AAAC', short for A3C, as opposed to the synchronous version A2C(Advanced Actor-Critic). A multi-threaded asynchronous structure is applied in A3C, and depends on different reinforcement learning algorithms used, various variants of asynchronous RL can be generated, such as one-step Sarsa, one-step Q-learning, n-step Q-learning, and advantage actor-critic. The intention of asynchonous structure is to find RL algorithms that can train deep neural network policies reliably without large resource requirements.

3 Methods

3.1 Deep Reinforcement Learning with A3C and CNN

A typical RL algorithm is the algorithm that tries to learn the optimal policy of the MDP by iteratively applying the newly learned policy during training. The goodness of a policy can be evaluated by value iteration or policy iteration; and the policy improvement in reinforcement learning can be done by either value based algorithms (such as SARSA, Q-Learning, etc.) or policy based (such as Policy Gradient, Actor-Critic, etc.) algorithms.

In this course project, deep reinforcement learning with A3C and Convolutional network are used to tackle a subset of mini games. No replay data is used in this task, the learning is purely dependent on the agent starting aimlessly and randomly explore in each game, guided by the blizzard score in each episode to gradually improve it's optimal policy.

Compared to AtariNet which reduces spatial resolution of the input with each layer and ultimately finish with a fully connected layer that discards it completely, StarCraft II uses a fully connect network to tackle the challenge in inferring spatial actions (clicking on the screen and minimap). It might be detrimental to discard the spatial structure of the input if use similar network structure to AtariNet.

The A3C algorithm is implemented using multiple threads structure, each thread plays a separate agent, and a tensorflow network is defined as proposed in Vinyals et al. (2017) and shared by all agents. By utilizing TensorFlow's thread-safty of tf.Session, each agent steps through it's own trajectory and updates both it's policy and value parameters of a same tensorflow network.

3.2 Mini-Games Task Description

To study different elements of the game in isolation, and to provide further fine-grained steps towards playing the full game, I started on a series of mini-games that are constructed with the purpose of

testing a subset of actions or game mechanics with a clear reward structure. This proves to be a effective way in developing and evaluating learning algorithms, and opens a door to solve full game problems. The mini-games chosen for the task are:

- MoveToBeacon: The agent has a single marine that gets +1 each time it reaches a beacon, it will learn a greedy strategy (exploitation).
- FindAndDefeatZerglings: The agent starts with 3 marines and must explore a map to find and defeat individual Zerglings. This requires moving the camera and efficient exploration.
- DefeatRoaches: The agent starts with 9 marines and must defeat 4 roaches. Every time it defeats all of the roaches it gets 5 more marines as reinforcements and 4 new roaches spawn. The reward is +10 per roach killed and -1 per marine killed. The more marines it can keep alive, the more roaches it can defeat.

3.3 Evaluation, Baseline and Oracle

Different models are trained based on the chosen mini games as described above using both A3C and A2C. To evaluate the result of the trained model, I will choose the average score of the random agent provided by DeepMind as the baseline, and DeepMind's FullyConv agent baseline result as the Oracle to compare with. An analysis of the agent behavior and different structures of A3C and A2C is also included in the end.

4 Model, Algorithm and Network

4.1 Model

An finite-horizon Markov Decision Process (MDP) is used to model the problem. By default, each episode will be either terminated after a max number of steps (default to 60), or the game ends. The max number can be configured by parameters. Consider an Markov Decision Process (MDP), formalized as following:

- S: the set of states
- $s_0 \in$ **States** : starting state
- $a = A(s)$: possible actions from state s
- $\pi = T(s, a, s')$: probability of s0 if take action a in state s
- $R(s, a, s')$: reward for the transition (s, a, s')
- $IsEnd(s)$: whether at end of game
- $0 \leq \gamma \leq 1$: discount factor (such as 0.99)

Policy π is a mapping or distribution form S to A. Assuming one agent, using a policy π, starts from state s_0, chooses an action a_0, gains a reward $r_0 = R(s_0; a_0)$, then transforms to next state s_1 according to distribution $T(s_0, a_0, s_1)$ and repeats this process. This will generate an episode with trajectory τ as following:

$$\tau = s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2 \ldots$$

For finite-horizon problem, the end state is either when time step exceeds T, or the end of the trajectory; the exploration is also over. The discounted cumulative reward in an episode get by the agent is defined as:

$$G = r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$$

where G is called return of the episode. An Reinforcement Learning algorithm aims to find an optimal policy which maximizes the expected return.

$$\pi_{opt} = \arg\max \mathbb{E}[\sum_{t=0}^{T} \gamma^t R(s_t, a_t)]$$

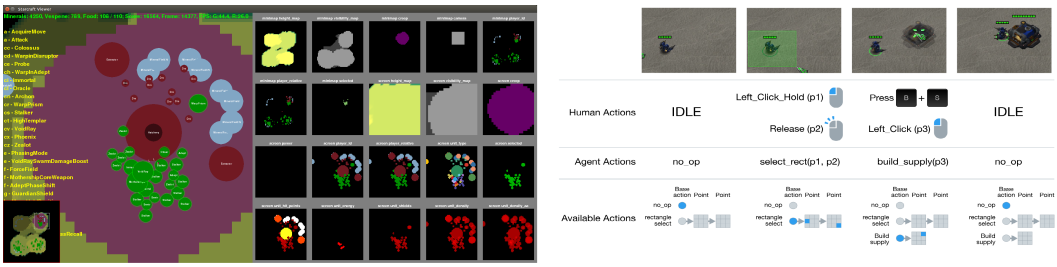

Figure 1: Model state and action. Left: screen and information feature on the left part of the image, minimap feature on the right side of the image; Right: examples of game actions.

4.1.1 States:

A typical StarCraft II game last for many thousands of frames and actions, a finite-horizon MDP is used to model the game. The MDP state is composed of a frame observation(include screen feature and minimap feature) for every 8 frames, which can be represented as MxN matrices, and some non-spatial information. Frame matrices represent the abstracted RGB images seen during human play which consist of a set of "feature layers" maintaining the core spatial and graphical concepts of StarCraft II, described briefly as following [Environment]:

- Screen. A detailed view of a subsection of the world. It is the on-screen view where players can see, and where most actions are executed. In this task, the size of screen is also a 64x64 matrix.
- Minimap. A coarse representation of the state of the entire world. For example, terrain height, fog-of-war, creep, camera location, and player identity, etc. are shown in the top row of feature layers on Figure 1.

As shown on top of the screen image, the non-spatial information contains information such as gas and minerals collected, unit built etc., and a set of valid actions currently available depending on current game state.

The initial state s_0 is obtained whenever the agent calls env.reset(), it will restart the game with the given environment configuration.

The last state s_{end} is either when the game ends, or when the maximum number(configurable) of steps arrived.

4.1.2 Actions:

The environment's action space is a close mimic of human player interaction with keyboard and mouse. Figure 1 shows an example of a sequence of actions to build a supply. The format of an action a is composed of a function ID and a list of arguments. For instance, consider selecting multiple units by drawing a rectangle. The action can be written as select_rect(select_add, (x1, y1), (x2, y2)), where 'select_rect' is the function ID in PySC2 lib; the first argument select_add is binary, indicates whether it is 'select'(True) or 'add'(False); the remaining arguments are pairs of integers that define start and end coordinates. At every game state, there are only a small subset of available actions out of approximately 300 action functions.

4.1.3 Rewards:

There are two different rewards structures that are described in Vinyals et al. (2017): ternary 1 (win) / 0 (tie) / -1 (loss) received at the end of a game (with all-zero rewards during the game), and the Blizzard score which is used to guide the RL in this task. The Blizzard score is what the players see on the screen at each step of the game during interacting with SC2LE environment. It is calculated by adding up different components: the current resources, the upgrades researched and units and buildings currently alive and being built. A player's cumulative rewards accumulates and decreases as these 3 components accumulates and descreases.

4.2 Algorithm:

4.2.1 Policy Gradient and Actor-Critic

The goal of reinforcement learning is to find an optimal behavior strategy for the agent to obtain optimal rewards. Traditional Value based algorithms such as SARSA, TD and Q-learning, are way too expensive computationally in the continuous space. Take policy/value iteration for example, the policy improvement step $\arg\max_{a \in A} Q_\pi(s, a)$ requires a full scan of the action space, and suffering from the curse of dimensionality. Policy gradient approach generally have better convergence properties, and very effective in high-dimentional or continuous action spaces[David Silver's RL course].

Policy Gradient Theorem. For any differentiable policy $\pi_\theta(s, a)$, for any of the policy objective functions $J = J1$ (episodic environments we can use the start value), J_{avR} (Or the average reward per time-step), $\frac{1}{1-\gamma} J_{avV}$ (continuing environments we can use the average value [Sutton and Barto (2018)]), the policy gradient is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) Q_{\pi_\theta}(s, a)]$$

In vanilla policy gradient algorithm, the return v_t at the end of each episode is used as an unbiased sample of $Q_{\pi_\theta}(s, a)$, and the increment of parameter θ is $\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$. Actor-Critic is introduced as a way to reduce variance of the vanilla policy gradient algorithm, and replaces the episode return with an estimated action-value function parameterized by w, which is just what is called a 'critic'. And two different parameters are to be updated in learning:

- Actor: $\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) Q_w(s, a)$
- Critic: $\Delta w = \beta(R_s^a + \gamma Q_w(s', a') - Q_w(s, a)) \nabla Q_w(s, a)$

4.2.2 Asynchronous Advanced Actor-Critic (A3C)

Asynchronous Advantage Actor-Critic Mnih et al. (2016), short for A3C, is a classic policy gradient method with the special focus on parallel training. In A3C, the advantage function $A(s, a) = Q_w(s, a) - V_{\theta v}(s)$ is used as the critic to signicantly reduce variance of policy gradient. Note that this introduces another set of parameters for value function, so in reality the advantage function is approximated by one step TD error, and the update increment of value function parameter is $\Delta\theta_v = \beta(r + \gamma V_{\theta v}(s') - V_{\theta v}(s)) \nabla V_{\theta v}(s)$ instead, in order to only learn only one set of critic parameters.

In conjunction with the actor part, we can see that A3C maintains both a policy function $\pi(a_t|s_t, \theta)$ and an estimate of the value function $V(s_t; \theta_v)$. The policy and the value function are updated after every t_{max} actions or when a terminal state is reached. The update performed by the algorithm can be seen as $\nabla_{\theta'} \log \pi(a_t|s_t; \theta') A(s_t, a_t; \theta, \theta_v)$ where $A(s_t, a_t; \theta, \theta_v)$ is an estimate of the advantage function given by $\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$, where k can vary from state to state and is upper-bounded by t_{max}. The pseudocode for the algorithm is presented in Supplementary Algorithm S3.

4.2.3 Advanced Actor-Critic (A2C)

A2C is a synchronous, deterministic version of A3C. It has been shown by OpenAI that it is able to utilize GPUs more efficiently while achieve the same or better performance than A3C. The drawback is it's more resource demanding, mainly refers to GPU, but on a lot cases lack the ability of utilizing multiple CPU architecture. Compared to A3C, because of asynchronous update, A3C has the problem of playing with policies of different versions and therefore the update would not be optimal, while A2C simply waits for all the parallel actors to finish their work before updating the parameters and is able to always play with the latest updated policy. Experiments on A2C is also attempted to compare the difference of the two algorithm.

4.2.4 Deep Q-Learning (DQN)

In order to compare the effectiveness of A3C over DQN in StarCraft II game, I also tried implementing an Asynchronous Deep Q-learning algorithm as described in Mnih et al. (2016), but was not able to

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

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

Figure 2: Pseudo code of an A3C implementation, from [Mnih et al. (2016)]

complete testing and evaluation of the algorithm. The algorithm doesn't have an 'Actor' role, hence has smaller size of parameters. State-action value is used as the regression target in the training, with mean square error as the loss function.

4.3 FullyConv Network

In AtariNet each layer of the convolutional network is generated from a spatial resolution reduced input and then fully connected to form the game state, which means the spatial information are abstracted away before actions are smapled. However in StarCraft , a major challenge is to infer spatial actions such as clicking on the screen and minimap. Abstract away spatial information is probably detrimental as these actions are within the same screen/minimap area. Here we propose a fully convolutional network agent, which predicts spatial actions directly through a sequence of resolution-preserving convolutional layers.

FullyConv is the fully connected network proposed by Vinyals et al. (2017) which has no stride and uses padding at every layer, both screen and minimap inputs are passed through separate 2-layer convolutional networks with 16, 32 filters of size 5 x 5, 3 x 3 respectively. The game state is then formed by the concatenation of the screen and minimap network outputs, as well as non-spatial information broadcasted along the channel dimension, as shown in Figure 3 right.

To compute the baseline and policies over categorical (non-spatial) actions, the state representation is first passed through a fully-connected layer with 256 units and ReLU activations, followed by fully-connected linear layers. Finally, a policy over spatial actions is obtained using 1 x 1 convolution of the state representation with a single output channel.

Figure 3 shows two networks, on the left is the one used in this task, on the right is the one proposed by SC2LE. The discrepancy between the two is that in the left implementation, the spatial policy depends only on the two spatial features, while on the right side SC2LE proposed uses all three features in inferring spatial policies.

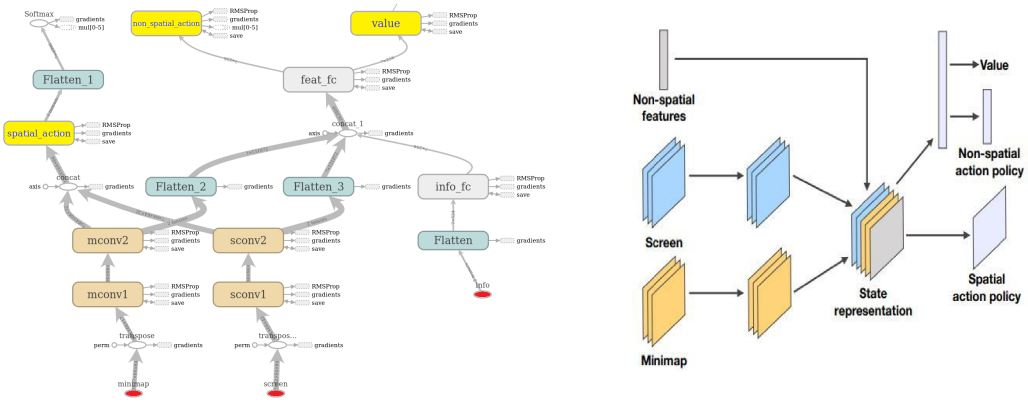

Figure 3: Left: An implementation of FullyConv generated by tensorboard; Right: FullyConv network proposed by SC2LE.

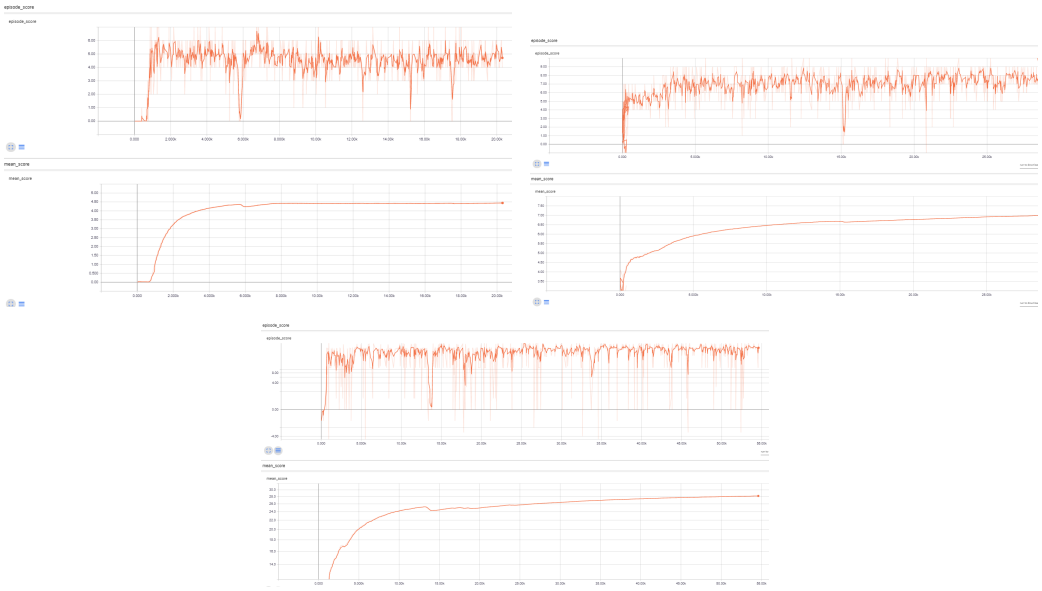

Figure 4: A3C training score plot, MoveToBeacon on top left, FindAndDefeatZerglings on top right, DefeatRoaches at the bottom. Short trajectories makes a much smaller average score compared to evaluation score.

5 Experiments

5.1 Training with A3C

As has been explained in Methods section, the task is a pure deep reinforcement learning experiment, no replay data is used in the training. Instead, 3 mini games are chosen in the experiment, each starts with 0.05 probability of random policy exploration, and 0.2 probability of spatial policy random exploration. The maximum game steps is chosen either 60 or 100 to shorten the length of trajectory of each game episode, with 8 concurrent working threads running on a Windows 10 with an 8G RAM nVidia GPU, and 2 concurrent working threads running on a 2 CPU Google cloud Debian system with an 8G RAM Tesla GPU.

As can be seen from the Figure, the score for training experiment is much smaller than the evaluation experiment. This is because the in training experiments has short trajectory, only 60 states in an episode, much shorter than in evaluation, which uses the end of game only from the game environment. This does not matter since the purpose of training is to output an optimal policy, and the resulting policies for all three mini games are learned as expected. However a problem of not choosing a proper

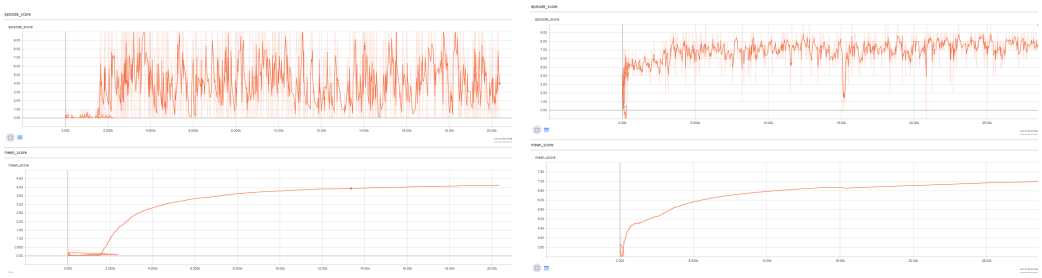

Figure 5: DQN training score plot, MoveToBeacon on top left, DefeatRoaches on top right. From the plot it is obvious that DQN has a much higher variance compared to A3C.

Table 1: Evaluation results. A3C and DQN results use evaluation score, trained for a short period of time(only less than 24 hours each), they have much smaller average score due to lack of training and a max game steps of 60 for DefeatRoaches and FindAndDefeatZerglings, max game state 100 for MoveToBeacon in A3C, and max game steps for MoveToBeacon in DQN. Row A2C uses training score with about 3 days learning using @simonmeister's implementation.

Experiments	MoveToBeacon	FindAndDefeatZergling	DefeatRoaches
DeepMind Random	1	4	1
DeepMind Baseline	26	45	100
A2C*	25.9	40.9	-
Deep A3C(short training)	22.25	18.27	53.58
DQN(short training)	21.0	-	-

trajectory length is that, if the training doesn't last long enough, the agent can end up with an optimal policy not making a single move! This can be seen in the linked videos, in the 'DefeatRoaches' mini game, when the distance between marines and roaches is far, the marines simply stay where they are until the end of an episode.

The following table shows the comparison between models trained in the experiment (evaluation result), and the output of the baseline (Random Agent) and the Oracle (DeepMind's FullyConv baseline). From the table it is clear that the learning has successfully learned policies that achieve a certain closeness to DeepMind's FullyConv baseline, and far better than the random policy.

5.2 Training with DQN

The experiment result of DQN is also given in the result table and the score plot in Figure 5. Use a maximum trajectory game step of 120 and trained for a short period of time(Only 12 hours for MoveToBeacon with DQN), the resulting policy improvement is also successful consider the short training time. Comparing the episode score between DQN and A3C we can see that DQN has a problem of very high variance, which justifies the argument of Actor-Critic being able to reduce the variance in the model section. Another interesting problem with DQN experiment is that, probably because of the short training trajectory and the insufficient training time, the resulting model seems to be unable to make the first selection action, and I have to manually select the marine and then the model starts playing by itself. This can be seen in the linked video .

5.3 Training with A2C

The above table also includes some experiment results A2C, the synchronous version Advanced Actor-Critic, using implementation in `https://github.com/simonmeister/pysc2-rl-agents`. The result is collected at an early stage of this project with a much longer training time, and hence gives a better result. The biggest advantage of the synchronous is when there is a powerful GPU, the batch policy gradient runs multiple game environments synchronously, the update of gradient are all done in the same time and there is no more concerns learning with an old version of parameters. This makes A2C more resource demanding, with only 4 synchronous agent running, the GPU memory usage is almost 8G already. On the contrary the A3C version can work even without a GPU, and

running on a GPU with 8 threads concurrently with 60 max episode steps for MoveToBeacon only takes about 4G of the GPU memory.

6 Discussion

Short trajectories in A3C algorithm in this experiment is to be replaced by full episodes in future during training, with a tuned hyper-parameters. If the training doesn't last long enough, short episode will likely cause the problem of the agent not being able to make the first move, as has been shown in `https://youtu.be/i_OyJsAGEAs` for A3C and `https://youtu.be/J0vCAtLtIHI` for DQN, the marines sometimes simply stays put till the end of the game; This problem can also be fixed by a more sufficient training. Attempts also have been done to use full episode with 8 working threads in Asynchronous A3C, however the training couldn't be finished and which is likely because of memory exhaustion. In order to observe effect of full episode learning, the number of thread need to be reduced in order to alleviate the exhaustion of resource.

A2C has been shown to be able to utilize GPUs more efficiently and work better with large batch sizes while achieving same or better performance than A3C. Both A2C and A3C are considered more powerful than DQN, in that as a Value based algorithms, DQN is way too expensive computationally in the continuous space, policy/value iteration for policy improvement step $\arg\max_{a \in A} Q_\pi(s, a)$ requires a full scan of the action space, and suffers from the curse of dimensionality. More over, as can be seen from the result of the experiments, DQN has a much higher variance as compared to A3C.

7 Conclusion and Future Work

This project covers a wide range of topics, including, and Supervised Learning, Value based Deep Q-learning, policy based Actor-Critic, A3C and A2C, as well as Convolutional Neural Network. All three agents (A3C,DQN and A2C) in the experiments successfully learned to play the selected mini games far better than the DeepMind random agent. Although the average score for A3C is still to be improved due to lack of training compared to the DeepMind Baseline, the trend of the score plot is already enough to show that the agent is working.

However it lacks the ability of understanding things like micro-actions in mini games such as DefeatRoaches; In 'Find and Defeat Zerglings' the marines even failed to explore the whole map, which is obviously conter-intuitive for a human player (a human player would constantly explore all the map to look for ermines). The large state-action space makes the state exploration as hard as looking for a needle in a haystack, more advances algorithms are needed in tackling full StarCraft II games.

There is yet another important part of SC2LE that needs to be studied in the future: the use of replay data. It will be very likely that by applying supervised learning against the replay data, the exploration of game states will be more effective and hence the learning time will be shortened dramatically.

8 Acknowledgements

Thanks to project mentor Benjamin Petit for reviewing and grading my work, and thanks to all Stanford CS221(2019 Spring) teaching staff for delivering an amazing AI course.

Code: https://github.com/haroldmei/pysc2-study

References

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2012). The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. (2013). A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4):293–311.

Robertson, G. and Watson, I. D. (2014). A review of real-time strategy game ai. *AI Magazine*, 35:75–104.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

Synnaeve, G., Nardelli, N., Auvolat, A., Chintala, S., Lacroix, T., Lin, Z., Richoux, F., and Usunier, N. (2016). Torchcraft: a library for machine learning research on real-time strategy games. *CoRR*, abs/1611.00625.

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T. P., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J., and Tsing, R. (2017). Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782.

The StarCraft II API: `https://github.com/Blizzard/s2client-proto`

Environment: `https://github.com/deepmind/pysc2/blob/master/docs/environment.md`

Lilian W: `https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html`

David Silver: `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf`