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 StarCraft II also opens a door to unparalleled opportunities to many challenging frontiers in AI. This course project explores and replicates DeepMind's work of SC2LE Vinyals et al. (2017) by applying different asynchronous algorithms described in Mnih et al. (2016) to start my journey in making game agents. I will start with modelling the game environment by MDP, then look into the details of a classic policy gradient reinforcement learning algorithm A3C, and define a fully connected CNN used by A3C to regress on two targets: a policy function π_θ and a value function $V_{\theta v}$. Another topic explored in the project is Deep Q-learning Network (Mnih et al. (2015), Mnih et al. (2013)), some comparisons between the policy based algorithm(A3C) and the value based algorithm (DQN) is made for the purpose of study. The project also covers the synchronous version of Advantage Actor-Critic(A2C) in order to compare the performance.

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 decades. 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 and can be played in a variety of settings such as 1 vs 1 similar to zero-sum game, n vs n a combination of zero-sum(between groups) and search problem(inside a group), and also more complex settings, 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 solved. The most recent progress in this field was Deepmind's AlphaStar, playing against Pro StarCraft II champion Grzegorz "MaNa", and was defeated at the last round. This is a huge leap of progress, yet there are still a lot of interesting and hard problems 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 from DeepMind, by representing and processing game frames with a fully connected Convolutional Neural Network (CNN), I will use deep reinforcement learning algorithm to play 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 with DQN [Mnih et al. (2015), Mnih et al. (2013)], a value based RL which has been proved very successful in Atari games; and to it's sister variance A2C which is an synchronous version of the Advantage Actor-Critic.

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 library. The StarCraft II binary although provided in Linux, the StarCraft II API (`https://github.com/Blizzard/s2client-proto`) supports both Linux and Windows. The StarCraft II API is an interface that provides full external control of StarCraft II and exposes functionality for developing software for a variety of applications. 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 reinforcement learning 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 AI in 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 idea of DQN implementation in this project is described in Mnih et al. (2015), which defined what is called 'AtariNet' structure with CNN, and use a cross-entropy like target Q value function using L2-norm loss to learn the optimal policy.

The Asychronous version deep reinforcement learning algorithm Asychronous Advantage Actor-Critic (A3C) and Asynchronous Deep Q-learning 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(Advantage Actor-Critic). A multi-threaded asynchronous structure is applied in A3C, and depends on different reinforcement learning algorithms used, various variants of asynchronous reinforcement learning 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 reinforcement learning algorithms that can train deep neural network policies reliably without large resource requirements.

3 Methods

3.1 Deep Reinforcement Learning with A3C, DQN and CNN

A typical reinforcement learning algorithm is the algorithm that tries to learn the optimal policy of an MDP by iteratively by applying the newly learned policy during to improve itself in 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., using ϵ-greedy) or policy based (such as Policy Gradient, Actor-Critic, etc.) algorithms.

In this course project, deep reinforcement learning with A3C and convolutional nerual 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.

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

Both the A3C and DQN algorithm are asynchronous and are implemented using multiple threads structure, each thread plays a separate agent, as proposed in Mnih et al. (2016); the FullyConv network is defined as proposed in Vinyals et al. (2017) and shared by all agents in both DQN and

A3C. 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 the 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 makes it easier to step forward to solve full StarCraft II game. 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 DQN. To evaluate the result of the trained model, I use the average score of the random agent provided by DeepMind as the baseline, DeepMind's FullyConv agent baseline result as the Oracle to compare with. An analysis of the agent behavior together with some game videos played by different trained models are also given in the paper. Also for the purpose of comparison I also include some results obtained from an A2C implementation.

4 Model, Algorithms and FullyConv

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. 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:

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

The discounted cumulative reward in an episode get by the agent is defined as:

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

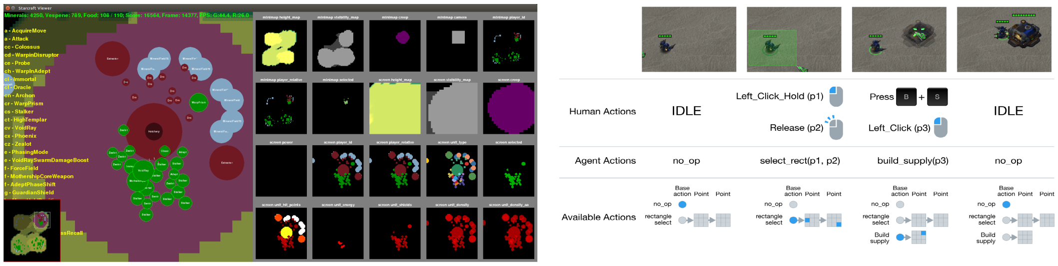

Figure 1: Model state and action. Left: screen and information feature on the left part of the image, minimap feature on the right side of the image; Right: examples of game actions.

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

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

4.1.1 States:

A typical StarCraft II game lasts for many thousands of frames and actions. The MDP state of it is composed of a frame observation(include screen feature and minimap feature) for every 8 frames, which can be represented as $M \times N$ 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 in `https://github.com/deepmind/pysc2/blob/master/docs/environment.md`.

- Screen. A detailed view of a subsection of the world. It is the on-screen view where players can see (figure 1 left side of the left pic), and where most actions are executed. In this task, the size of screen is a 64×64 matrix.
- Minimap. A coarse representation of the state of the entire world (figure 1 right side of the left pic). For example, terrain height, fog-of-war, creep, camera location, and player identity, etc. As shown in the top row of feature layers on figure 1, also a 64×64 matrix.
- Info. Text information in screen. 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 of game steps arrived.

4.1.2 Actions:

The environment's action space is a close mimic of human player's interaction with keyboard and mouse. figure 1 right side 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, 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; the first argument select_add is a binary, indicates whether it is 'select'(True) or 'add'(False); the remaining arguments are pairs of integers that define the start and end coordinates.

4.1.3 Rewards:

There are two different rewards structures described in Vinyals et al. (2017): ternary 1 (win) / 0 (tie) / -1 (loss) is received at the end of a game (with all-zero rewards during the game), and the Blizzard score which is used to guide the reinforcement learning 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

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 ← 1
repeat
    Reset gradients: dθ ← 0 and dθ_v ← 0.
    Synchronize thread-specific parameters θ' = θ and θ'_v = θ_v
    t_start = t
    Get state s_t
    repeat
        Perform a_t according to policy π(a_t|s_t; θ')
        Receive reward r_t and new state s_{t+1}
        t ← t + 1
        T ← T + 1
    until terminal s_t or t − t_start == t_max
    R = { 0               for terminal s_t
        { V(s_t, θ'_v)    for non-terminal s_t// Bootstrap from last state
    for i ∈ {t − 1, . . . , t_start} do
        R ← r_i + γR
        Accumulate gradients wrt θ': dθ ← dθ + ∇_θ' log π(a_i|s_i; θ')(R − V(s_i; θ'_v))
        Accumulate gradients wrt θ'_v: dθ_v ← dθ_v + ∂ (R − V(s_i; θ'_v))^2 /∂θ'_v
    end for
    Perform asynchronous update of θ using dθ and of θ_v using dθ_v.
until T > T_max
```

Figure 2: Pseudo code of an A3C implementation. Both asynchronous A3C and DQN can be found in Mnih et al. (2016)

calculated by adding up different components. In this project, the goal of the learning is to improve the accumulated score in each game episode.

4.2 Algorithm:

4.2.1 Policy Gradient and Actor-Critic

The goal of reinforcement learning is to find an optimal policy for the agent to obtain maximum rewards. Traditional Value based algorithms such as SARSA, TD and Q-learning are way too expensive computationally in the continuous space and suffering from the curse of dimensionality. Policy gradient approach generally have better convergence properties, lower variance, and very effective in high-dimentional or continuous action spaces.

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'. 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: $\Delta w = \beta(R^a_s + \gamma Q_w(s', a') - Q_w(s, a)) \nabla Q_w(s, a)$

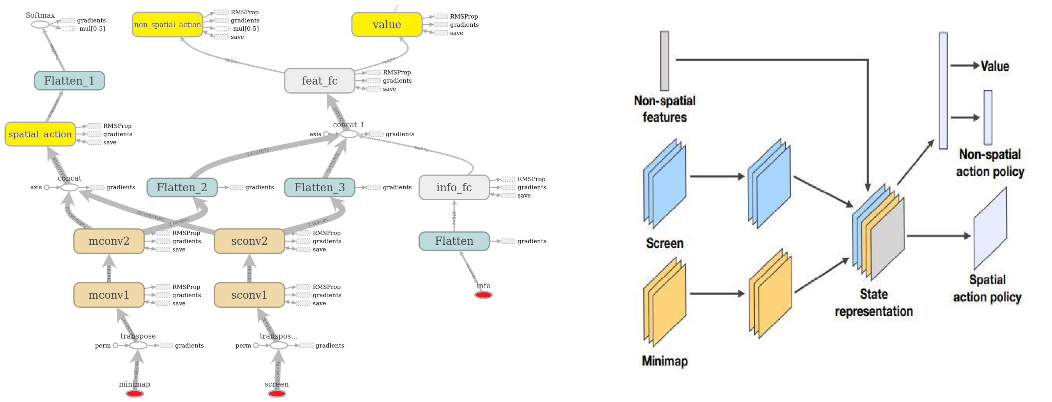

Figure 3: Left: An implementation of FullyConv generated by tensorboard; Right: FullyConv network proposed by SC2LE.

4.2.2 Asynchronous Advantage 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 significantly 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 learn just one set of critic parameters.

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 (also called n-step asynchronous algorithm). The update performed by the algorithm can be written 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 shown in figure 3.

4.2.3 Advantage 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. Because of asynchronous update, A3C has the problem of playing with policies of older 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 during training. Experiments on A2C is also attempted to compare the difference of the two algorithms.

4.2.4 Deep Q-Learning (DQN)

In order to compare the effectiveness of A3C over DQN in StarCraft II game, I also implemented an Asynchronous Deep Q-learning algorithm by combining ideas from Mnih et al. (2016) and Mnih et al. (2015). The regression target $Q(s,a)$ is simply a sum over cross-entropy of the two separate policies(spatial action and non-spatial action), uses the same FullyConv network as it is in A3C as the predictor for actions, and mean square error as the loss function. The algorithm doesn't have an 'Actor' role, hence has smaller size of parameters. As can be seen in the experiment results, DQN is also proved to be a successful algorithm for StartCraft II mini games, however it has a much higher variance compared to A3C.

4.3 FullyConv Network

In AtariNet each layer of the convolutional nerual network is generated from a spatial resolution reduced input and then fully connected to form the game state, which means the spatial information

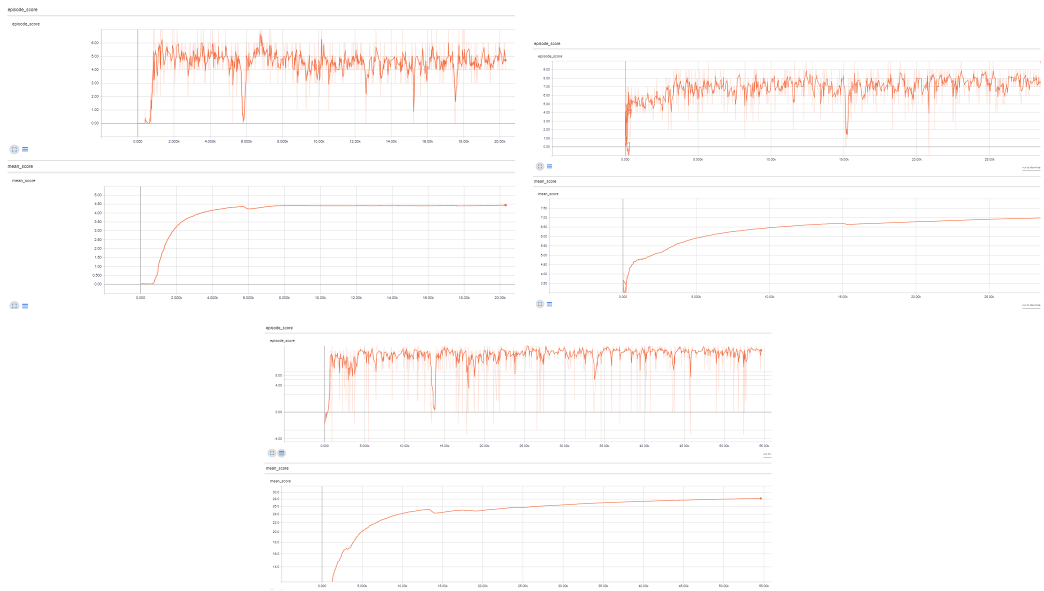

Figure 4: Episode score and Average score of A3C, MoveToBeacon on top left, FindAndDefeatZerglings on top right, DefeatRoaches at the bottom. Short trajectories makes a much smaller average score compared to evaluation score.

are abstracted away before actions are sampled. In StarCraft a major challenge is to infer spatial actions such as clicking on the screen and minimap, abstracting away spatial information is probably detrimental as these actions are within the same screen/minimap area.

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 right side of figure 3.

To compute the baseline and policies over non-spatial actions, the state representation is first passed through a fully-connected layer with 256 units and ReLU activation, 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 side is the one implemented in this experiment, 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, which may be problematic because spatial policy is not independent on non-spatial actions. On the right side the SC2LE proposed network uses all three features in inferring spatial policies.

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 non-spatial policy exploration, and 0.2 probability of random spatial policy 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. I also attempted using full episodes in training, but unable to finish the training because of the lack of computation resource.

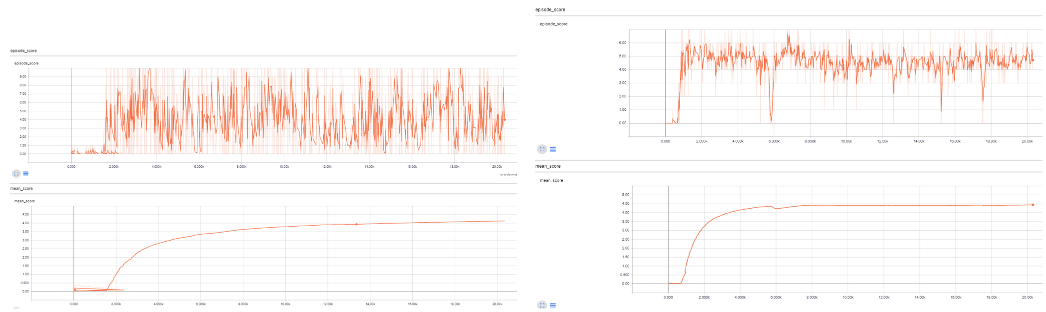

Figure 5: MoveToBeacon training episode/average score plot, DQN on left, A3C on right. From the plot it is obvious that DQN has a much higher variance compared to A3C.

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	-	-

As can be seen from the Figure, the average training score is much smaller than the evaluation score due to short trajectory. This turns out to be not much of a problem since the purpose of training is to learn optimal policies, and the resulting policies for all three mini games are improved significantly. One result of not choosing a proper 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 `https://youtu.be/i_OyJsAGEAs`, 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 a lot 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 table 1. 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 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 as expected by itself. This can be seen in the linked video `https://youtu.be/JOvCAtLtIHI`.

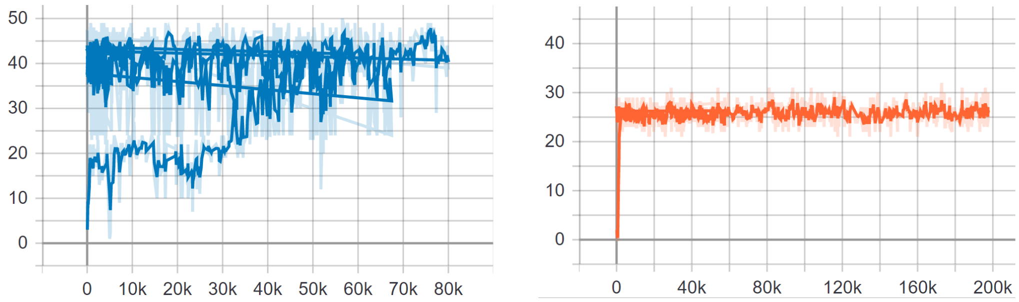

Figure 6: A2C training score (the plotting noise is caused by intermittent training)

5.3 Training with A2C

5.4 Algorithm:

The above table also includes some experiment results A2C, the synchronous version Advantage Actor-Critic, using implementation in `https://github.com/simonmeister/pysc2-rl-agents`. The training score is plotted in figure 6. The result is collected at an early stage of this experiment with a much longer training time, and hence gives a better result. 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 such as number of threads or a more compact CNN. 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, which 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 with a longer training period. Attempts also have been done to use full episode with 8 working threads in Asynchronous A3C, however the training couldn't be finished and it is likely because of memory exhaustion.

A2C has been shown by OpenAI 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, 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 MDP, 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 time compared to the DeepMind Baseline, the trend of the score plot is already enough to show that the agent is working.

Future work can be focused on the agent ability of understanding things like micro-actions in mini games. A human player would control separate marines in DefeatRoaches, and in Find and Defeat Zerglings game the marines even failed to explore the whole map, which is obviously counter-intuitive for a human player (a human player would constantly explore all the map to look for ermines). The large state-action space also makes the state exploration as hard as looking for a needle in a haystack, more advances algorithms are to be explored in future to tackle the 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 as well as providing advice to the project. 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.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

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`