PROJECT REPORT: HIERARCHICAL MULTI-AGENT REINFORCEMENT LEARNING

Instructor: Josh McCoy, Artificial Intelligence, Spring 2018

{*} denoted sections that have been changed since original proposal

GROUP MEMBER

In alphabetical order:
Aidan Bean, ajuengli@ucdavis.edu, 997487516
Alexander Soong, asoong@ucdavis.edu, 998857734
Keshav Tirumurti, kctirumurti@ucdavis.edu, 999290785
Sam Cheng, samcheng@ucdavis.edu, 914377162
Terry Yang, yimyang@ucdavis.edu, 913248564

1. INTRODUCTION

StarCraft II has several significant obstacles for developing a game-playing agent that make it a difficult challenge in artificial intelligence. Firstly, the action space is large which results in a combinatorial explosion of possible game states. Secondly, the state space is high-dimensional given there are hundreds of elements in each game frame. Thirdly, game actions might result in long-term consequences that cannot be well captured even by state-of-art learning methods. Finally, the goal of the game, which might appear obvious to human players, is obscure for many learning methods to achieve.

Due to the these significant difficulties for any current artificial intelligence agent to play and win the full game, DeepMind has proposed several mini-games that each represent essential elements from the full game, e.g. combat, search and planning. These mini-games will serve as smaller steps for developing a more advanced game agent. Hence, a problem worthy to solve is to develop an agent that outperforms the baselines provided by DeepMind in one of the mini-games. And we choose the mini-game named BuildMarines.

1.1. Changes from the original proposal*

We stuck to the problem we described in the original proposal: Implement an artificial intelligent agent capable of playing full game, and achieve better results in infrastructure construction in BuildMarines than DeepMind.

1.2. Problem

We choose BuildMarines for the following reasons. Firstly, as described in the paper published by DeepMind and Blizzard, BuildMarines is the most strategical among all the mini-games. Second, the game agents from DeepMind achieves less than 10% of the human player performance, which is lower than any other mini-game. And by observing the replay, it is not difficult to see that the learning agent was inefficient in multi-unit control. The score shows that there is a large space for improvement. Thirdly, the ability to build infrastructures and armies quickly in the early stage is important in winning the full game, so an improvement in this mini-game skill can contribute to building a more efficient full game-playing StarCraft agent.

Most of the recent papers on StarCraft had been focused on developing learning models for mini-games entirely in group combat, while learning for infrastructure construction has not been done yet.

BuildMarines is a good representation of infrastructure construction in StarCraft II. It involves micromanagement, specifically in manipulation of multiple units; and resource planning, specifically for achieving a maximum score within a given time. From this technical aspect of artificial intelligence, progress in multi-unit control and resource planning can be easily generalized into real-world applications besides gaming.

As described in the paper, the best performing agent, *FullyConv*, composed of a multi-layer convolution network and has a objective to maximize a sparse and delayed reward. Yet, there is space for improvements. The training curve is very unstable. Moreover, in the replay, the units controlled by the agent are wandering. These issues are caused by two weaknesses. First, the agent is unclear in what objective it should accomplish even after large amount of training due to the sparse and delayed reward. Second, the agent used a centralized policies for all of the in-game units, and sometimes failed to produce meaningful cooperation among these units, therefore resulting in inefficient actions, which prevents itself from getting higher rewards.

1.3. Possible AI Approaches

A few papers had proposed possible solutions regarding these two possible drawbacks found in the baseline agent. For example, Hierarchical Deep Reinforcement Learning showed in a few seemingly abstract games a way to give RL game agents intrinsic motivation for exploration by developing mechanisms to let each agent assign sub-goals to itself. For the second drawback, researchers developed Monotonic Value Function Factorization that can train decentralized policies in the Deep Multi-agent reinforcement learning model and guarantee consistency between decentralized policies and a centralized policy over all the in-game units.

1.4. Solution*

After in-depth reading into the paper about the Hierarchical Deep Reinforcement Learning, we realized that the H-DQN will require an artificial objective being inputted to the Meta-Controller, which is different from what our idea previously: a proactive agent capable of discovering and defining objectives independently. This is seemingly no different than defining a different reward function for the agent.

Since the complicated-structured H-DQN doesn't seem to offer much, we decided to move on to the Dueling-DQN, a DQN alternative with generally much better performance than the baseline DQN on Atari games.

For detailed changes, please refer to the *design and approaches* section.

1.5. Contribution*

Since there has not been published progress, to our knowledge, of building artificial intelligence agent focusing on infrastructure construction in the game, our approach to this problem could be the first to examine what possible changes that the Dueling DQN will bring to the StarCraft AI agent. A noticeable improvement will indicate that these approaches can be generalized to deal with different components of the game.

2. DESIGN AND APPROACHES*

In this section, we described the reason behind changing the agent's reinforcement learning model architecture, and the new agents' model.

2.1. Old Model*

The original proposal described a multi-agent H-DQN architecture. We use a H-DQN to control multiple units in the game by assigning them different sub-goals and rewards. All the units will update and follow a central policy while also using their own sub-network to produce action based on their own states.

2.1.1. *Difficulties*

However, as mentioned in the *Introduction* section, H-DQN still required an artificial sub-goal for the agent to achieve, which is indifferent from other DQN alternatives. Hence, H-DQN lost its main attraction to us. Further, H-DQN is by far one of the most complicated structured DQN, even if it will produce the state-of-art result, the unknown amount of time we will spend on it will put us on the risk of not being able to produce measurable outcome for this in-class credit project.

We imagine to build a multi-agent network, where each agent produce based on both a central policy and their own policy based on each of their states. However, pysc2 doesn't support this mechanism, and we had to turn to the client supplied by Blizzard, called *sc2client-proto* for more specific unit control. *sc2client-proto* would take lots of time to get used to and offer us a more difficult environment to develop learning algorithm because it's written in multi-language. Hence, we decide to move forward from it.

2.2. New Models*

Class materials on reinforcement learning is not as much as we expected, for the reason of better practices, we decided to implement simple agents first then gradually increase the complexity level. In the beginning, we implemented scripted agent and Q table agent that are only capable of playing BuildMarines. At the later stage, we implement the Dueling DQN agent that is capable of playing the full game with no restriction on state space or action space.

We choose Dueling DQN is because Dueling DQN performs generally better than baseline DQN in Atari games. It required less computation power compare to parallel A3C. Further, in RTS games, we often need to consider the advantage of chosen actions, namely how much better an action is compared to another. Dueling DQN encoded this mechanism. And the BuildMarines minigame, which didn't offer much state changes but valued much on choosing the correct action is the perfect playground to test Dueling DQN.

2.2.1. *Scripted agent*

We made a scripted agent for minigame BuildMarines to get familiar with the pysc2 environment and observed the possible outcomes of the minigame. For its architecture, it's quite simple. The action space were explicitly defined. The coordinate for spatial actions (choosing location on the map, etc) is set. In every episode, the agent select exactly one worker and build exactly one depot, then exactly one barrack, finally that barrack starting building marines.

2.2.2. *Q table agent*

We explicitly defined a small action space and state space for the agent. The action space includes select a worker, build marines, do nothing or a few more. The state space is a few parameters included number of depot, and number of barracks. At each state, the agent will follow the traditional Q learning algorithm to choose the best available action produce the next state, update reward and start learning algorithm.

2.2.3. *DQN agent*

This was a very large jump from the previous agents. To implementing a working version, we learnt from online tutorials and coding examples. This agent utilized a DQN that follows the exact implementation of *FullyConv* agent from the DeepMind's paper. It is able to play any StarCraft II games.

First, the action space is split into spatial action (SA) and non-spatial actions (NSA). The spatial action space is the combination of the pixel coordinates of the map, namely, spatial action determines where in the map you will perform your non-spatial action. The non-spatial action space is a list of all possible actions in the action list defined by pysc2.

Second, the state space is a concatenation of all 32 feature screen layers, for the detailed list, please refer to the pysc2 documentation.

The network structure will be similar but with slight difference than the *FullyConv* net in the paper.

2.2.4. *Dueling DQN agent*

The only difference from the DQN agent is the Q value for non-spatial actions is calculated using a Dueling DQN.

2.3. DQN*

2.3.1. *Input prepossessing*

The numerical features are concatenated directly after taken the exponential log to reduce extreme skewness. Each categorical feature layer was broken down into multiple one-hot layers and being concatenated together. For example, given a layer with only two categories represented with integral value 1 and 2. This layer would be broken down in a layer where all 2 were turn to 0s while the 1s remained, and a layer where all 1 were turn to 0s while the 2s remained, and two layers will be concatenated together.

For action features, we created a one-hot vector with 1s on available action index slots.

2.3.2. *Network structure*

The first convolution layer has 16 filters with kernel size 5, and stride of 1 in each dimension. The activation function is *relu*. The second convolution layer has 32 filter with kernel size 3, and stride of 1 in each dimension. The activation function is *relu*. The output of the second convolution layer is then feed into a linear convolution layer with 1 filter of kernel size 1, which produced an output layer with the same dimension as a single layer screen input. After applied softmax, this output layer becomes the Q values for all spatial actions.

We didn't applied pooling is because the screen size is small and pooling might affect the agent's decision making when "seeing" less informative feature. However, we use drop-out for the final output to help prevent overfitting.

The action input is feed into a fully connected layer with 256 output units and *tanh* as activation function. Then the output is concatenated and feed into another fully connected layer with output dimension same as the total number of actions. After applied softmax function, the output became the Q value for all non-spatial actions.

The overall joint V value (for the Advantage calculation) is calculated by feeding the output of the fully connected layers with 256 units into a linear fully connected layer with 1 output unit.

We separately produced the target value y and current value $Q(s, a; \theta)$ from two networks with same configuration. The network to produce current value will be called $evaluationnet$, the other is the $targetnet$, a replacement of parameters on $targetnet$ will be performed after certain times of learning.

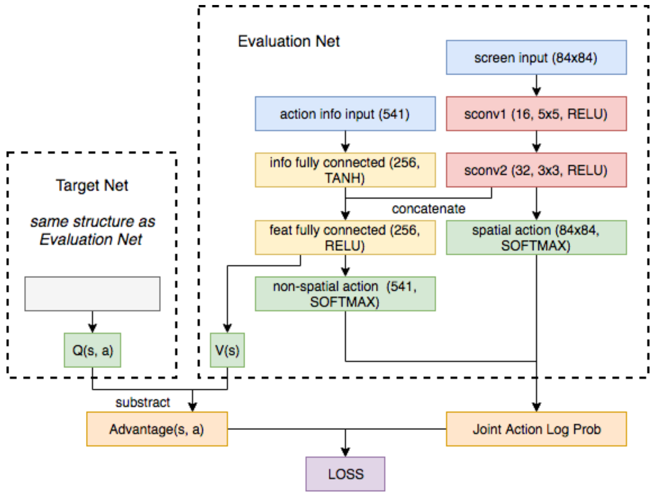

Figure (1). Architecture for DQN

2.3.3. *Loss function*

Although we didn't use the full power of A3C by running parallel agents and updated gradient using their loss simultaneously because the lack of access to large computation power, we still keep the Advantage mechanism in the loss function.

2.3.4. *Reward function*

We focused on improving the performance in the BuildMarines, yet the build-in reward from observation from pysc2 package has no score related to marines but some very sparse reward. We define our own reward function for BuildMarines. This function assign rewards based on the number of depot, barrack and marines built, ratio is [0.1, 0.1, 1]. We define it as such because the pysc2 cannot (at least we dont know) return the number of different units. Instead, We managed to get the number of pixel occupied by certain types of units. The size of depot and barrack is much larger than the marine.

This provided an idea for abandoning the sparse reward from pysc2, and instead, defining new reward functions when the agent built has different objective.

The Advantage A encodes how much better the action a is compared to other actions. And our loss function is defined as following, where the first half encoded policy loss, and the second half encoded value loss. We will perform gradient descent on this loss.

$$Loss = -MEAN(lP(J) \cdot A) - MEAN(Q \cdot A) \quad (1)$$

The $lP(J)$ is the exponential log of probability of the joint action J composed of selected non-spatial action (NSA) and selected spatial action (SA) from the network. And it is calculated as following:

$$lP(J) = VSA \cdot \ln \Sigma SA \cdot SAS + \ln \frac{\Sigma NSA \cdot NSAS}{\Sigma NSA \cdot VNSA} \quad (2)$$

VSA is the valid spatial action, SAS is the selected spatial action, NSA is all of the non-spatial actions. $NSAS$ is the selected non-spatial actions. $VNSA$ is the valid non-spatial actions. All operations are operated in multi-dimensional matrices.

We use RMSprop optimizer to reduce the gradient of the loss because RMSprop generally has a faster performance and can handle sparse data well.

2.4. Dueling DQN*

Besides the above structure and definition, the Q value for non-spatial action is calculated as $Q(s, a) = V(s) + A(s, a)$, so that the action can be generated more independently from the states.

$$A = Q(s, a) - V(s) \quad (3)$$

2.5. Experiment setup*

We added dropout rate of 0.2 to the final fully connected layers when predicting Q value for non-spatial actions and general V value. The screen layer size is set to 84. The learning rate is 0.001, reward decay is 0.9. We replaced the parameters of $targetnet$ after 200 times of learning. The experience size is 1000, and the batch size is 32. We sample random batch each time. For the RMSprop Optimizer, we set the decay to 0.99 and $\epsilon = 1e - 10$.

2.6. Technical Stack

Our technology stack will be centered around python. PySC2 is a python environment and OpenAI baseline RL implementations are also in python. Furthermore, there are a number of useful python libraries and frameworks for machine learning and data processing such as tensorflow, scikit-learn, numpy and pandas. Most of our development work will be done on a Google Cloud Ubuntu instance. The reason for using Google Cloud is to use a standardized platform between all members (versus using Windows, Mac, etc. and other environment differences on our personal devices). Google Cloud also has GPU support for fast training our RL algorithms which is important to complete the project on time. As for IDEs, we will use PyCharm. Github will be used for version control and also hosting all our deliverables. We will use git features like branching as we deem necessary. As for quality assurance, we will be using various unit tests and integration tests. Additionally, we will use PyCharm debugging features. Checking our results using visualization of elements like learning curves and agent actions will also be useful. Finally, we will review each others code to spot anomalies.

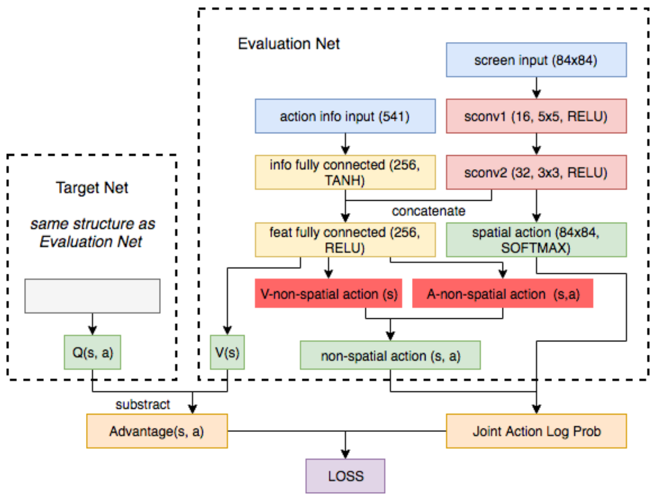

Figure (1). Architecture for Dueling DQN

3. SCOPE AND TIMELINE

Changes are in bold font.

Week of 4/30:

1. Get an off-the-shelf reinforcement learning (RL) algorithm working with PySC2 environment on Google Cloud **(Deepmind did not release their code for the SC2 baselines results in their paper)**

2. Learn about details relevant to our RL approach through papers, videos, articles, tutorials, etc. **(Compilation of resources will be shared on Github at projects end)**

3. Become familiar with Starcraft 2

Week of 5/7:

1. Design experiments to test for different aspects of our RL innovation in a systematic manner. Should

have a few different experiments that test for the effects of different variables such as hyperparameters, mini-game variations, network implementation details, etc. **(Changed approach from doing H-DQN to implementing Dueling DQN. Focus shifted from the research aspects to learning from implementation. Discussed design of DQN for SC2 game instead)**

2. Begin implementation of our RL approach in PySC2. We can start with simple changes first and work our way up to more complicated implementations **(We cannot implement multi-agent network in pysc2, so for the simplicity, the focus changed to learning tensorflow to implement basic DQN in PySC2 for a start)**

Week of 5/14:

1. Refine and finalize the design of our experiments based on feedback **(Experiments changed)**
2. Continue implementation of our RL algorithm(s) **(Done)**

Week of 5/21:

1. Finish implementation of our RL algorithms(s) **(Continued into the last week)**
2. Conduct experiments based on our experiments design **(Continued into the last week)**
3. Start working on final deliverables **(Continued into the last week)**

Week of 5/28:

1. Interpret results of experiments
2. Finish all final deliverables

4. DOCUMENTATION AND ACCESS

We will be using a Github repository for all documentation and commented code. We will include markup pages that contain all the info needed to run our code to reproduce our experiments as well as pdf documents containing all other project deliverables. For team communication, we will use Slack and Facebook. Finally, we will share documents and collaborate on Google Drive.

5. EVALUATION*

To begin with, we will establish baseline results with off-the-shelf reinforcement learning algorithms. We might use existing baselines such as the ones reported in the Deepmind paper or we might establish our own. The goal is for our RL agent to beat the baseline results. We plan to analyze the performance of our RL agent both qualitatively and quantitatively. Qualitatively, we will analyze replays of our agent as it learns to identify what its doing well and what its doing poorly. Quantitatively, we can look at various metrics such as the learning curve and the rate of convergence for the performance of our agent.

5.1. Results*

6. PLAN FOR DELIVERABLES

All final deliverables will be on our Github repository. We hope to share our work on major Starcraft AI forums to gain visibility and get feedback. Depending on the end quality of our work, we might refine it and submit it to a conference. We also plan to use our work for the UC Davis Starcraft 2 team.

7. SEPARATION OF TASKS FOR TEAM*

The original task separation became less restricted. For detailed difference, please compare to the original separation.

1. environment set up: all members
2. paper reading and research: sam
3. algorithm design: sam
4. algorithm implementation: sam
5. discovering pysc2: sam
6. pysc2 agent integration: sam
7. Google cloud operation:
8. experiment design: sam
9. running and recording experiments:
10. website design:
11. project report draft: sam
12. code testing and examination: sam

7.1. Original separation

Basic Tasks: All members

1. Learn about reinforcement learning (videos, papers, tutorials, etc.); focus on application to SC2
2. Learn about PySC2 environment (create agents, run agents, minigame/map creation); i.e. learn the framework
3. Learn about Starcraft 2 in general (e.g. macro, micro, strategies, etc.)

4. Deliverable: A solid understanding of what we can achieve in a reasonable time frame with reinforcement learning using PySC2; inform our understanding of all other tasks below

Google Cloud Environment Setup, Configuration, and Maintenance: Alex, Aidan

1. Google Cloud: Setup SC2 Game on Ubuntu, PySC2, ensure GPU compatibility

2. Deliverable: Allow all team members to ssh into the server and run basic agents with PySC2; help address any server problems that arise during development

Experiments Design: Terry, Sam, Aidan (with feedback from all members)

1. Look at existing literature; think about how to build off them/compare against them

2. Minigame/Task design and map creation if necessary

3. Establish baselines and compare Evaluation metrics (quantitative and qualitative)

4. General methodology

5. Deliverable: (1) draft a document detailing the above (2) get input from other members (3) refine document based on feedback for final version

Reinforcement Learning Algorithm Implementation : Keshav, Aidan, Alex, Sam

1. Look at existing implementations/tutorials to start with; then implement our own innovations

2. Make iterative changes and test them properly (work with QA)

3. Try implementing a few approaches ranging from easy to hard

4. Deliverable: All code for carrying out the experiments

Conducting Experiments: Sam, Keshav

1. Run the experiments as specified by our plan and record detailed results (work with QA)

2. Deliverable: Experiments results ready for analysis

Evaluating Results: Aidan, Keshav (with input from all members)

1. Compare against baselines

2. Qualitative and quantitative evaluation

3. Deliverable: (1) draft a document with the analysis (2) get input from all members (3) refine document based on feedback

Testing/QA: Terry; every member should also be at least lightly testing their own code while working

1. Think about possible sources of error and design ways to check for proper execution.

2. code review, unit testing modules, visualizing outputs, integration testing, etc.

3. Deliverable: Confidence that our results cant be invalidated by bugs; testing at each stage to ensure that problems dont propagate down our experimental pipeline

Inline Documentation: All members

Deliverable Reports: Terry [Two members] (every member proof reads it once)

1. Deliverable: Code tour of solution: maybe jupyter notebook demo (TBD); Project report with all the results from experiments above; all deliverables on our Github repository with commented code, markdown pages and other documentation