PROJECT PROPOSAL: HIERARCHICAL MULTI-AGENT REINFORCEMENT LEARNING

Instructor: Josh McCoy

Course: ECS 170, Spring 2018

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 an 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 by even state-of-art learning methods. Finally, the goal of the game, which might appear to be 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. 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 a 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 focus 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 the 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 rewards. 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.

1.2. 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.

1.3. Solution

A possible approach to address the first drawback is to apply Hierarchical Deep Reinforcement Learning to the agent, which provides the agent a more intermediate reward for achieving its sub-goals. Second, The hierarchy of centralize policy and decentralized policy mechanism could be helpful for game agents to control actions of multiple units in game. Hence, we decided to use parameter sharing multi-agent gradient-descent Sarsa (PS-MAGDS) algorithm to for policy training. We apply a sharing policy for all the in-game units, while each of these units can update their own policies based on their own actions and rewards.

That being said, we will iterate on these ideas as we learn more and try them out. However, our general approach will not change. Overall, the structure of our project will follow this methodology: We will take a mini-game (either existing or new), and get baseline results on this mini-game using an off-the-shelf reinforcement learning algorithm. Then we will try some innovation with reinforcement learning. This innovation might be applying an approach that hasn't been applied to Starcraft 2, combining two existing approaches, or some idea of our own. We will compare our innovation against baseline results and interpret why it worked well or why it didn't.

1.4. Contribution

Since there has not been an published progress in 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 decentralized policy and Hierarchical learning will bring to the StarCraft AI agent. A noticeable improvement will indicate that these approach can be generalized to deal with different components of the game.

2. DESIGN AND APPROACHES

2.1. Hierarchical Multi-agent Reinforcement Learning

Temporal abstraction from the hierarchical deep Q network (H-DQN) helped the agent defined sub-goals and allows for flexible exploration while handling the sparse rewards issue through intrinsic rewards from the Critic-Controller mechanism. The parameter-sharing mechanism purposed in the PS-MAGDS algorithm allows cooperation among in-game units while encourage decentralized actions. Hereby, we propose an architecture that integrates parameter-sharing and H-DQN for building our StarCraft agent.

Showed in Figure (1), the agent received observation from external environment and produces actions. The agent has the two main component, Meta-Controller and controller, each has separate Deep Q neural Network.

2.1.1. Meta Controller

Meta Controller received sensory observations from environment, an instance of the observation could be the combination of features from time $t-1$, features from time t and that last joint-agent action. Then, it will produce a policy over goals. for all the goal in G in the Meta-Controller, the policy will choose one and passed it to the controller that maximize the Q function $Q_2(s_t, g_t; \theta_2)$, where θ_2 is the parameter of DQN of the meta-controller.

2.1.2. Controller

The controller consist of the Critic, which assigns intrinsic rewards to the agent if sub-goals received from the meta-controller has been reach; the Controller policy network, which has a parameter θ_1 that is shared by all the in-game units, and gets updates from all units simultaneously; a set of in-game units, each of which has its individual policy neural network but share the same parameter θ_1 as the central policy network.

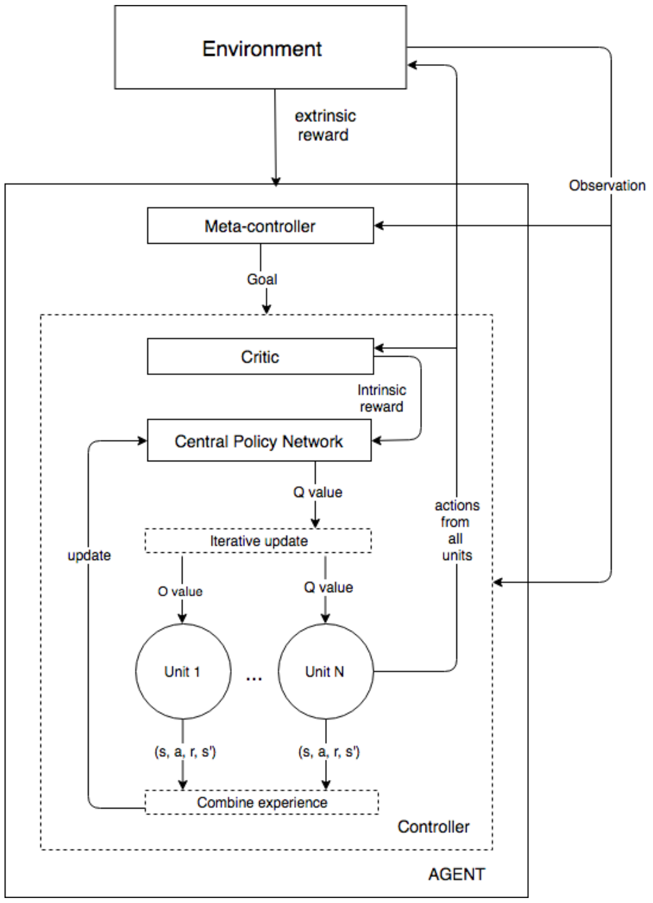

Figure (1). Architecture

2.1.3. Learning

For each episode, after initialization, the meta-controller will produce a sub-goal g using its Q function Q_2 and $\epsilon - GREEDY$ algorithm with observation as input. After passing the goal to the controller, training for the central policy network will begin at a initial state s_0. Noted the current total extrinsic reward F is still 0.

Central policy network will kick start a Q-value for the training. While g is not reached, based on the Q value, each of the in-game unit i, depended on their current states, they choose an actions based on $\epsilon - GREEDY$ algorithm, produced an action, receives an extrinsic reward from the external environment and intrinsic reward r_i from the Critic. Then stored the transition $(\{s_i, g\}, a_i, r_i, \{s'_i, g\})$ to replay D_1. After finished iterating over all units, the algorithm should update the parameter θ_1 with a random batch from replay D_1 using gradient descent method. θ_2 should also be updated in similar method with sampled replays from D_2, which is empty at the beginning of the training. Then update the total extrinsic reward F with extrinsic reward f, and move to the next state for all in-game-units.

When g is reached, store (s_0, g, F, s') to replay D_2, noted D_2 is the replay used to train meta-controller network. Then, using Q_2 with $\epsilon - GREEDY$ to produce another sub-goal, start central policy training again.

Overall, the meta-controller will provides a more intermediate rewards that compensate the sparsity and delay of the reward. Second, iterative training using output from multiple units should well connect the central policy and the individual policy.

2.2. 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.

3. SCOPE AND TIMELINE

Week of 4/30:

1. Get an off-the-shelf reinforcement learning (RL) algorithm working with PySC2 environment on Google Cloud
2. Learn about details relevant to our RL approach through papers, videos, articles, tutorials, etc.
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.
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

Week of 5/14:

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

Week of 5/21:

1. Finish implementation of our RL algorithms(s)
2. Conduct experiments based on our experiments design
3. Start working on final deliverables

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.

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 AND DELIEVERABLE

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