Deep RL For Starcraft II

Andrew G. Chang
agchang1@stanford.edu

Abstract

Games have proven to be a challenging yet fruitful domain for reinforcement learning. One of the main areas that AI agents have surpassed human abilities are in board games, such as Go, where much of the difficulty lines in the exponentially large state space. Indeed, great progress has been made in building agents that can defeat the world's best in these games, such as AlphaGo. However, with highly-skilled competitive video games, the bar has risen even higher. DeepMind's recent release of the Starcraft II Learning Environment (SCLE) and their accompanying paper [5] poses the grand challenge of building an agent that can defeat best human pros. This project takes a introductory approach at building an agent that can play Starcraft.

1 Introduction

Starcraft II has been posed as the "grand new challenge" in reinforcement learning by DeepMind. It is not difficult to see why. The game is a real-time strategy (RTS) game that has captivated the minds of many casual and professional players due to its nuance and incredibly high skill cap. The game requires a combination of long-term planning and real-time mechanical skill. In the professional scene, the common format is a one-versus-one, however team games can be played. Therefore the game is multi-agent on a global scale since players need to coordinate, but also multi-agent on a local scale since the agent must control many individual units (workers, soldiers, etc). The game is also partially observable since the agent can only see what is in their camera's view and must actively explore to gather information about the map and enemies. One of the primary reasons the game is so difficult is because it requires constant multi-tasking, which requires mechanical skill and constant focus. The player is constantly in balancing act of building their economy, base and army, as well as controlling their current army to explore, defend, or attack. Given the difficulty of this game, SCLE provides a way to provide simpler subtasks of the game which are called "mini-games". These mini-games are meant to represent a subset of what an agent may need to learn to perform well at the full game. In my project, I chose the "DefeatRoaches" mini-game, which is supposed to test an agent's ability in a difficult combat scenario. The initial state starts with the agent controlling 9 marines against 5 roaches and the agent must kill as many roaches in two minutes. Each roach that is killed results in +10 reward whereas each marine killed results in -1 reward. If all roaches are killed before the time limit, five more are spawned with the agent also receiving five more marines. Note roaches are much more powerful units than marines, so the agent must devise some effective strategy to be successful. More formally, the mini-game can be modeled as a finite-horizon MDP with discrete state and actions and deterministic transitions. However, the state spaces and action spaces are very large: there are 101938719 possible actions and the states can effectively be treated as continuous since some features take on scalar values. My project trains an agent using the Asynchronous Advantage Actor Critic (A3C) method with a convolutional neural net to model the policy function.

2 Related Work

In DeepMind's paper [5] that introduces the problem, they present three baseline models, all using A3C but varying the network architecture. The first model is the same used in their Atari paper [2].

The second model uses a fully convolutional model which outputs spatial policies directly from the output of resolution-preserving convolutional layers, i.e. there is no fully-connected layer before the spatial policy. This is intended to preserve the spatial structure of the input. The third model is the same as the second, except with a LSTM module. They also present a supervised learning approach by using replay-data to fit a value function to predict values for each state, which can then be used directly as a policy. Previous environments have also been built for the original Starcraft, with accompanying models [4], however they differ in a fundamental way since their agent interface is more programmatic rather than human-centric. SCLE models actions similar to how a user would play, for example instead of sending individual commands to units directly, the agent has to select the unit first by drawing a rectangle around it (or using special actions to select idle units), and then issue commands to the selected unit. This makes the action space more complicated, but more akin to how a human interacts with the game.

3 Environment & Features

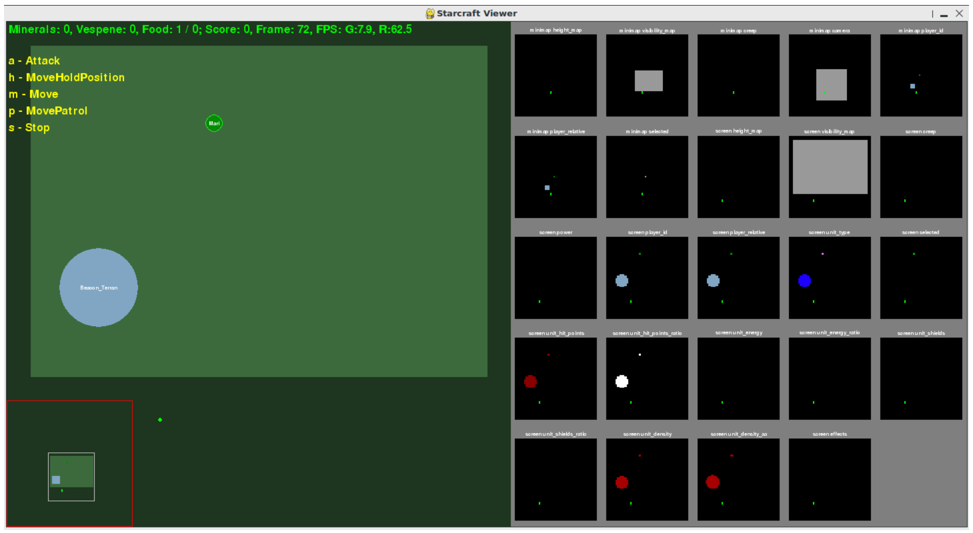


Figure 1: SCLE Feature Layers and pygame UI

The environment provides a run-loop in which the agent can perform an action in lockstep with the game simulation, so the agent can take as long as it likes to compute which action to take. The environment can be configured so the agent can act every n game steps, in my project I use $n = 8$ which corresponds to roughly 180 actions-per-minute similar to the rate of actions performed by a skilled human. If the agent were to act faster, for example every game step, then this would give an unfair advantage to the agent since it can play the game at a speed faster than what is humanly possible, although the benefits probably aren't too great. As mentioned before, the action space is modeled to be similar to a human interface. More specifically, the format of the action is a compound action consisting of a base action a_0 and then a variable number of arguments $a_1, ..., a_l$ each of which have variable dimension. An example action that moves a selected unit is "Move_screen", which takes a boolean argument "queued [2]" which determines if this command should be enqueued instead of performed instantly, and "screen [84,84]" which takes two numbers that represent a point on the discretized screen. At each state the number of available actions varies, but is presented to the agent in its state so it will know which actions are valid. The features consist of spatial features and non-spatial features. The spatial features consist of two sets of feature layers, minimap and screen. Minimap corresponds to a small minimap portion of the screen that is a downsized view of the entire map and screen is the primary view of some subsection of the map, depending on where the camera is. The feature layers represent pertinent information to an agent that is on the screen, for example the IDs of units on the screen. The screen feature is a $[n, n, 13]$ tensor where n is configurable and the minimap is $[64, 64, 7]$: in this project $n = 64$ so it matches the minimap. There are a number of non-spatial features, but for this project I only chose a subset of them: general player information $[13, 1]$, multi-select $[50, 7]$, and available actions $[524]$. General player information includes things like resource counts, multi-select contains unit information (health, shields) in the current selection, and available actions is essentially a one-hot vector of actions available to the agent at that state. This means the flattened state vector is 82807 dimensional. The UI that displays the screen and feature layers is shown in Figure 1 (note on Linux the full game graphics aren't available, only a simplified UI). Preprocessing is done on the features by normalizing scalar feature layers and log transforming non-spatial scalar features since features may have varying scales.

4 Methods

The approach I took in my project is a policy gradient method, specifically Asynchronous Advantage Actor Critic (A3C). This is the same approach as the DeepMind paper and it is one of the standard approaches for training RL agents due to its flexibility, especially with large action spaces. A3C is a on-policy method that tries to directly optimize the policy function $\pi_\theta(s)$ by deriving an estimator for the gradient. Our objective is to maximize the discounted sum of expected rewards: $J(\theta) = \mathbb{E}[\sum_{t=0}^{T} \gamma^t R(s_t, u_t) | \pi_\theta(s)] = \sum_\tau P(\tau; \theta) R(\tau)$ where $\tau = s_0, u_0, s_1, u_{1]}, ..., s_T, u_T$ is an entire trajectory in an episode (where u is the action taken), R is the sum of discounted rewards and P is the probability of the trajectory. It can be shown that the gradient of this objective function does not depend on the model dynamics and only on the term $\nabla_\theta \log \pi(\tau^{(i)}) R(\tau^{(i)})$. [1]. Therefore we can perform gradient ascent updates to maximize expected rewards using the log of our policy function and empirical rewards. In practice, to lower variance, the rewards are passed through an advantage function which intuitively subtracts a baseline to measure the value of the chosen action compared to the expected value of the state. Since our network now depends on a value function $V(s)$, we also add an output node from our network to estimate the value function by regressing it against the discounted sum of rewards received from that state R_t. The final term in the gradient update is an entropy regularization which penalizes the policy for a low entropy policy, i.e. there isn't much randomness. This is to encourage exploration in the policy so it doesn't greedily converge at a suboptimal strategy. Therefore, the full gradient update is:

$$A(s, a) \nabla_\theta \log \pi(s) + \beta \nabla_\theta (V(s) - R_t)^2 + \eta \sum_a \pi(a|s) \log \pi(a|s) \tag{1}$$

where β and η are hyperparameters to control the contribution of those terms to the gradient. For this project, I used $\beta = 0.5$ and $\eta = 0.001$. The advantage function I used is the generalized advantage estimation [3] which uses exponentially weighted samples of TD residuals: $\gamma V(s_{t+1}) - V(s_t)$ as an estimator of the advantage.

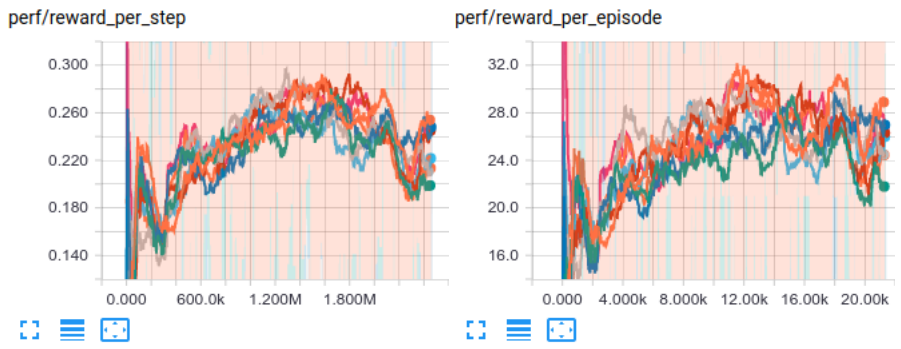
4.1 Modeling Action Policies

As mentioned previously, SCLE provides an action space where a base action is chosen, a_0, and based on this choice, a variable number of arguments with a variable number of dimensions are chosen: $a_1, ..., a_l$. If a single action policy is chosen with the action space flattened, this would over a hundred million actions in the single policy, and it would extremely sparse. Instead, for simplicity, policies are modeled independently, namely the base action, and each argument dimension. This means for a spatial argument, the x and y policies are modeled independently. This is a rather large assumption, since arguments policies aren't independent from the chosen base action nor even from each other, but in practice it seems to work decently. Therefore when selecting an action from the policy, the agent samples from the base action policy $\pi_\theta(a_0|s)$, masking out unavailable actions, and then depending on the base action chosen, samples from each argument policy dimension independently.

4.2 Network Architecture

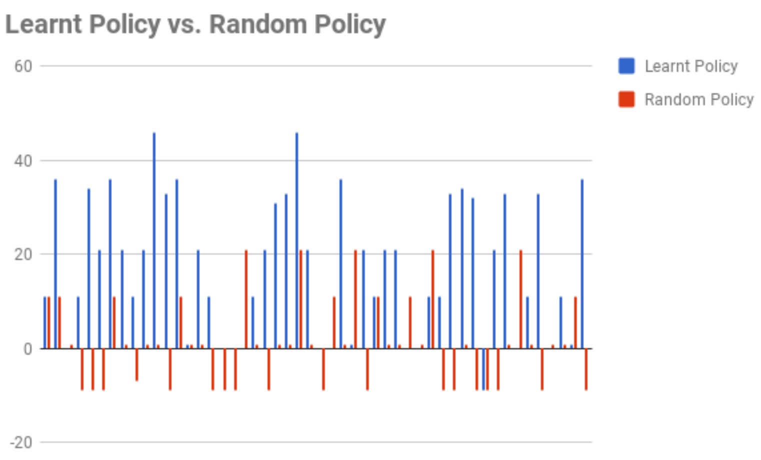
The network architecture chosen is similar to the Atarinet model used in DeepMind's Atari paper [2]. Namely the spatial features are input into two convolutional layers, with 16 filters of size 5x5 and 32 filters of size 3x3, respectively. The nonspatial features are passed through a fully connected layer with 32 units followed by a tanh non-linearity. The two spatial tensors are then flattened and concatenated with the nonspatial fully-connected output and then passed through a fully-connected layer with 512 units and ReLu nonlinearity. This vector is then passed through layers corresponding to each action and argument policy, using a softmax to output probability distributions.

4.3 Training Algorithm



One critical part of A3C is the use of multiple actors updating a shared global network. This plays the role of diversifying experiences the agent encounters and helps stabilize training issues due to non-stationary distributions and correlation of the states being sampled, as states within an episode are highly correlated. This is done in practice by running multiple actor threads which each have a local network and a shared global network. The actors perform gradient updates to the global network and synchronize their local networks to the global network at the beginning of each episode. This is somewhat analogous to the target network used in Deep Q-Networks and using multiple actors at once is similar to experience replay by decorrelating gradient updates with non-sequential experiences. The training algorithm begins by receiving an initial state s_0. The agent extracts spatial and nonspatial features from the state and feeds it into the network to receive policies over actions and arguments, as well as a value score for the state. The agent then samples these policies and performs action a_t and receives state s_{t+1} and reward r_t. It stores chosen actions, arguments, rewards, states, and values into a buffer, which will be used for backprop. After 40 iterations or the end of an episode, the agent performs backprop using it's state buffers, computed advantages using rewards and values, and computed expected rewards for encountered states to regress the value function. The agent receives the gradients and applies it to an optimizer to the global network. The agent then copies the global network weights to its local network and repeats. My project used 8 actor threads, since 16 slows the simulation down significantly on my computer, and trained for around 2 million steps with a learning rate sampled from the range $(1e-5, 1e-3)$. Two performance graphs are shown above, the rewards per episode and per step: we can see the learning plateaus at around a million game steps.

5 Results & Discussion



The learnt policy is compared with a random policy baseline over 100 trials: the learnt policy averages 17.64 with a max of 46 whereas the random policy has an average of 1.04 with 24 max. Human game play is omitted from these trials since the Linux release doesn't have the full game and has a different UI, but the DeepMind paper reports an average of 40 for an experienced player, whereas a professional player can get an average of 200. There is a wide skill gap in human policies, even for a simple minigame, which is a true testament to the difficulty of the game in its full setting. By analyzing the learnt policy, I observed the agent found a simple strategy, namely choosing to attach the roaches from the top or bottom. Since the marines and roaches both start in a vertical line formation, flanking the top or bottom of the line allows the marines to focus fire a single roach while the other roaches need to move to attack. This swings the advantage in the marines favor, since a head on fight would result in the marines losing, given they are much weaker units. Although a decent strategy, the agent failed to learn very nuanced strategies which would get it to surpass a human level, for example "microing" the units by moving damaged units away from focus and then re-engaging them after, a common and effective high-level human strategy. I suspect these emergent behaviors require a lot more experimentation with hyperparameters, since it is easy for the policy to converge on a simple strategy if it can get a steady state of rewards and doesn't explore the action space for more nuanced strategies. I would note in the DeepMind paper, they report running 100 experiments over randomly sampled hyperparameters and training over 500M game steps, which is a large amount of computing resources and time. During my different training attempts, I also observed difficulty in stabilizing training as small changes in hyperparameters could cause the training to converge at a bad strategy, or even diverge to get 0 score per episode. Future work could explore using different RL training algorithms or techniques to stabilize training and make it less prone to instability due to hyperparameter changes.

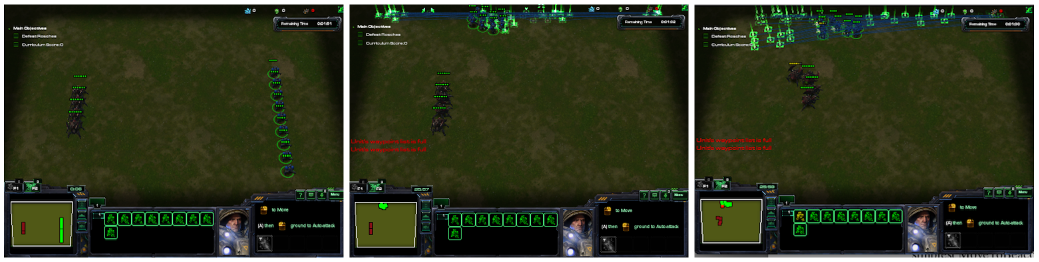


Figure 2: Marines spawn in vertical line and group and flank on top

5.1 Error Analysis

Unfortunately, the episodes for which agent received 0 score, and dropped the average substantially, are episodes where the initial state causes degenerate values in the network causing NaNs in the network. Some initial debugging suggests this could be due to some neurons having extremely large weights so a state which triggered those neurons would underflow the softmax (I believe the default tensorflow softmax implementation, i.e. tf.nn.softmax implements subtracting the largest logit). Hopefully with more experimentation I can pinpoint the issue, e.g. more trials with different learning rates. As for improving the policy to get out of this locally optimal strategy, I could experiment with hyperparameter search over the entropy regularization term. This is the classic exploration-exploitation tradeoff since the agent needs to explore different actions to learn good strategies and is akin to changing ϵ in ϵ-greedy. However, I suspect this will take a lot longer to train since some of these more advanced strategies require more nuanced actions that can have much higher delay in rewards rather than taking a locally optimal action: for example having a damaged marine run away from fire instead of fighting to the death to kill a low-health roach, so it can re-engage later.

References

[1] Deep rl bootcamp policy gradient notes. `https://drive.google.com/file/d/0BxXI_RttTZAhY216RTMtanBpUnc/view`.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. *ArXiv e-prints*, December 2013.

[3] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *ArXiv e-prints*, June 2015.

[4] Y. Tian, Q. Gong, W. Shang, Y. Wu, and C. L. Zitnick. ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games. *ArXiv e-prints*, July 2017.

[5] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. Sasha Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing. StarCraft II: A New Challenge for Reinforcement Learning. *ArXiv e-prints*, August 2017.