Reinforcement Learning Applied to RTS games

Leonardo Rosa Amado
Felipe Meneguzzi
8 May, 2017
PUCRS
Introduction
Introduction

- Reinforcement learning focuses on maximizing the total reward of an agent through repeated interactions with an environment.
- In traditional approaches an agent must explore a substantial sample of the state-space before convergence
  - Thus, traditional approaches struggle to converge when faced with large state-spaces ($\geq 10^k$ states).
  - Most “real world” problems have much larger state-spaces. E.g. chess.
- Alternatively, we can generate hypotheses of state features and try to generalize the reward function
  - However, this depends on good features and a good function hypothesis
Introduction

- Intuition of behind our work:
  - Compress traditional state representations domain-independently
  - Use traditional reinforcement learning on the compressed state-space
  - Aggregate experience from multiple, similar agents

- Main challenges:
  - How do we create a faithful representation of the states?
  - How do we address combinatorial explosion of multiple, parallel agent actions?

- Technical approach:
  - Learn a compressed state representation using deep auto-encoders
  - Reduce combinatorial explosion of RTS games by learning for individual unit types (in lieu of solving a Dec-MDP)
Background
• Real Time Strategy (RTS) games are a very challenging gaming environment for AI control (branching factor on the order of $10^{50}$)

• MicroRTS is an abstract simulation environment with similar rules to fully fledged RTS games (e.g. StarCraft, Command and Conquer).
  • Much simpler to modify and test
  • Only 4 types of units and 2 structures
  • Open AI integration API
  • Used as testbed for AI planning and MCTS approaches for RTS control
Figure 1: MicroRTS game state.
Approach
We use two key techniques to converge to a policy in RTS games:

- Train a deep auto-encoder to mitigate the state-space size
- Unit Q-Learning to mitigate the branching factor
Deep auto-encoders

Figure 2: Deep auto-encoder.
Our approach to compressing the state-space consists of 3 steps:

1. design a binary representation for the state space, the *raw encoding*;
2. design an auto-encoder that takes as input the raw encoding and narrows it into 15 neurons (bits), creating a *canonical encoding*; and
3. train the network using state-action pairs from the MicroRTS game.
Deep auto-encoder

Assuming a trained encoder $E(s, a)$, we modify the Q-learning update so that the tables are mapped through $E(s, a)$:

$$Q(E(s, a)) \leftarrow Q(E(s, a)) + \alpha(R(s) + \gamma \max_a Q(E(s', a')) - Q(E(s, a)))$$

- We train the auto-encoder offline with a fixed dataset of AI MicroRTS matches;
- Since we encode all updates through $E(s, a)$, the Q-table consists exclusively of encoded pairs.
To train the auto-encoder, we first model all binary features of the MicroRTS game state, e.g.:
- the position of all units from the player and the enemy;
- health of the player and enemy bases; etc

We use two Random strategies available from MicroRTS to generate a training dataset for the auto-encoder
- These strategies execute random actions, generating multiple state-action pairs with each player's units scattered around the map

Finally, we train the auto-encoder using this dataset
Unit Q-Learning

- Each player action in a MicroRTS game state is the combination of actions for all units on the map
  - Resulting in more than $3^5$ actions turn
- To avoid dealing with this very large branching factor, we use independent learning, which analyzes the best action for each unit locally.
  - Each unit generates an independent Q-Learning update.
  - The overall player action then becomes the group of the best action of each unit.
- At the end of each learning episode:
  - Units of the same role share their experience, building a unified table for the role using the algorithm below
  \[
  Q(s, a) = \sum_{i=0}^{agents} \frac{Q_i(s, a) \times \text{frequency}(Q_i(s, a))}{\text{frequency}(Q(s, a))}
  \]
  - This table is used as the base for new episodes.
Unit Q-Learning

Figure 3: Unit Q-Learning process life cycle.
Experiments and Results
Experiment Setup

- All tests were made in the 8x8 grid scenario of MicroRTS.
- The computer used for the experiments has the following specifications:
  - Intel CPU I5 2.7ghz.
  - 8GB RAM.
  - Java VM 1024 GB.
  - 6M Cache.
- When matching against other strategies to evaluate our win rate, we trained using 200 games, and then played 20 games with learning disabled.
Separate Roles for Workers

- Workers in MicroRTS can be used for both harvesting resources and attacking other units.
- To avoid this problem when merging the tables, we separate workers in two types, the **harvesters** and the **attackers**.
- The difference is that the harvester workers are rewarded for gathering resources.
- Both are rewarded for attacking enemy units.
- Other units (heavy, light, ranged) are considered attackers.
Figure 4: Convergence of Attacker and Harvester.
Figure 5: Q-Table size of Attacker and Harvester.
## Comparison against other Strategies/Algorithms

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Wins</th>
<th>Draws</th>
<th>Losses</th>
<th>Win rate</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passive</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>+ 20</td>
</tr>
<tr>
<td>Random</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>+ 20</td>
</tr>
<tr>
<td>Random Biased</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>+ 20</td>
</tr>
<tr>
<td>Heavy Rush</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>+ 20</td>
</tr>
<tr>
<td>Light Rush</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>+ 20</td>
</tr>
<tr>
<td>Ranged Rush</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>+ 20</td>
</tr>
<tr>
<td>Worker Rush</td>
<td>9</td>
<td>4</td>
<td>7</td>
<td>45%</td>
<td>+ 2</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>17</td>
<td>3</td>
<td>0</td>
<td>85%</td>
<td>+ 17</td>
</tr>
<tr>
<td>NaiveMCTS</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>40%</td>
<td>- 2</td>
</tr>
</tbody>
</table>
Finally, we analyzed the time for each approach to execute 10 cycles, which is the shortest period for any action in MicroRTS.

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Average time (s)</th>
<th>Maximum time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passive</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>Random</td>
<td>$\sim$0s</td>
<td>$\sim$0s</td>
</tr>
<tr>
<td>Random Biased</td>
<td>$\sim$0s</td>
<td>$\sim$0s</td>
</tr>
<tr>
<td>Heavy Rush</td>
<td>0.001s</td>
<td>0.05s</td>
</tr>
<tr>
<td>Light Rush</td>
<td>0.001s</td>
<td>0.01s</td>
</tr>
<tr>
<td>Ranged Rush</td>
<td>0.001s</td>
<td>0.03s</td>
</tr>
<tr>
<td>Worker Rush</td>
<td>0.05s</td>
<td>0.1s</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>2.0s</td>
<td>2.303s</td>
</tr>
<tr>
<td>NaveMCTS</td>
<td>2.0s</td>
<td>2.545s</td>
</tr>
<tr>
<td>Our approach</td>
<td>0.3s</td>
<td>0.511s</td>
</tr>
</tbody>
</table>
Conclusion
We developed an approach to play RTS games using traditional Q-learning distributed over multiple units with compressed Q-tables:

- The combination of approaches obtained promising results in the MicroRTS;
- Converged to a policy analogous to the best fixed strategy

As future work:

- Evaluate the performance using other auto-encoders, such as the denoising stacked auto-encoder.
- Learn the reward function using inverse reinforcement learning on the already implemented strategies.
Thank You
Related Work
  
  • Nair presents a distributed RL architecture to play Atari games.
  • The state is the game image encoded by a deep neural network.
  • Multiple instances of the environment are used to accelerate the training.
Multi-agent Reinforcement Learning

  - Multiple agents acting in the same environment.
  - They learn independently.
  - Independent learning can not ensure convergence to an optimal policy.
  - Policy coordination is required to build a global policy.