AI Agent for the Atari Game Phoenix

Sizhu Cheng {scheng72} , Richard Hsieh {rhsieh91} , Brian Chan {bchan17}

AI Agent for the Atari Game Phoenix

Sizhu Cheng {scheng72} , Richard Hsieh {rhsieh91} , Brian Chan {bchan17}

Abstract

AI agents for games showcase a capability to discern optimal actions resulting in larger rewards in a given game. Previous work on AI Agents for the Phoenix Atari game include reinforcement learning algorithms featuring Q Learning with Function Approximation and Deep Q Networks (DQN) utilizing the Atari emulator's 128-byte RAM state rather than visual input [2]. We build upon this work with the RAM state by exploring the reinforcement learning algorithms of Monte Carlo Tree Search and DQN with experience replay to reduce the probability of the network converging on a local minimum. In our results we show that both approaches perform better than random baseline agents but still are far from human level play.

Game Description

The Visual representation of Phoenix

Gameplay:

1. The agent controls the spaceship at the bottom of the screen, whose objective is to zap the enemies flying above while avoiding contact with the enemies' projectiles and the enemies themselves.
2. The agent receives points when they zap an enemy
3. Or zap all enemies on a level and progress to the next one; there, different types of monsters can behave differently (i.e. dive-bombing the player).
4. Human expert-level play can regularly achieve scores in the tens of thousands.

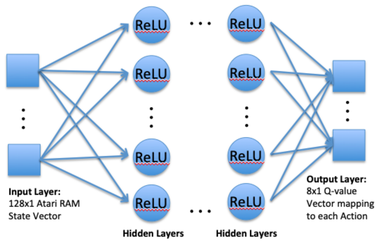
Agent Problem and Actions:

1. At any given moment, the agent is able to take 8 actions. All the information of the game states. are encoded in the compact 128-byte RAM vector.
2. Difficulties of this game for the agent include the very large state space and lack of explicit information provided in the encoded game state.

Models

Baseline

- Discretize the continuous state space by 8
- Digitize the sum of indices into 8 bins
- Adopt action in corresponding bin with probability 0.5
- Take random action in the rest of the cases
- No learning



DQN Architecture

Deep Q-learning Network (DQN)

- Neural network (NN) for function approximation in Q-learning

$$w \leftarrow w - \eta[Q(s_t, a_t; w) - (r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; w))]\theta(s_t, a_t)$$

where w are weight matrices and θ are feature vectors

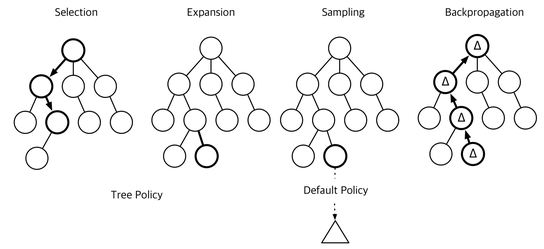
- NN performs feature extraction and weight updates for predicting Q-values of future states
- Non-linear ReLU activation function used in hidden layers
- Game observations stored in replay memory and randomly sampled to train model

Monte-Carlo Tree Search (MCTS)

- Involving selection, expansion, sampling and backpropagation
- Simulate using random policy and remember the one with largest UCB value
- UCB is defined as

$$\frac{w_i}{n_i} + C\sqrt{\frac{\log N_i}{n_i}}$$

where w_i is the number of wins, n_i is the number of visits for this node, N_i is the number of visits for its parent node. Here we set $C = 1$.



Hyperparameter Tuning

MCTS

1. **C**:Empirical constant using in node selection
2. **Loops**: Number of games played. For a specific set of parameters, scores are moving averages across all loops
3. **Playouts**: Number of samples drawn every Monte-Carlo Simulation
4. **Maximum Depth**: Maximum depth to search in the tree. In the Phoenix game setting, this is equivalent to the number of actions taken sequentially

DQN

1. **Exploration Rate, ϵ**: How often agent chooses random action
2. **Exploration Decay**: Decay over time to balance exploration vs exploitation
3. **Learning Rate, η**: How quickly weights are adjusted every time model is refitted
4. **Number of Layers/Number of Nodes per Layer** : eural network feature extraction
5. **Batch Size**: Samples to draw from memory at each model refitting
6. **Memory Size**: Max capacity of replay memory

Results

Loops	Playouts	Max Depth	Moving-Average Scores	Max Scores
100	50	2000	857.40	3230.00
100	50	5000	770.40	2740.00
100	50	8000	746.40	2790.00
100	50	10000	904.40	3780.00
170	500	5000	1019.40	3510.00
60	1000	2000	835.83	2770.00

Table 1: Moving Average Scores and Max Scores using MCTS

Layers/ Nodes	$\epsilon_{max}/\epsilon_{min}$	ϵ_{decay}	Refit Batch Size	Refit Freq.	Average Score
2/24	0.2/0.2	1.0	500	1k steps	325.23
3/24	0.2/0.2	1.0	500	1k steps	590.86
3/24	1.0/0.53	0.9992	500	500 steps	325.23
3/128	1.0/0.05	.99999	32	1 step	577.28
4/24	0.2/0.2	1.0	5000	1 step	546.20
4/24	1.0/0.1	0.995	500	1 step	590.86

Table 2: Average Scores using DQN

Discussion

- For MCTS, the larger the maximum depth, the larger the scores tend to be. Increase the loops and playouts also make MCTS more accurate as well
- Overall, performance achieved using both approaches are far lower than even average human players.
- The best performance was attained by MCTS

Future Work

- Try a weighted MCTS instead of our current version. It is believed that weighted MCTS is similar to model-free gradient method, which performs more efficient with less depth needed for search [1]
- Implement prioritized experience replay for DQN to filter out poor training examples
- Continue tuning hyperparameters, particularly for batch/memory size alongside traditional parameters.

References

[1] Justin Fu and Irving Hsu. "Model-Based Reinforcement Learning for Playing Atari Games". In: (2016).

[2] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv:1312.5602* (2013).