<megamind.ai>

Viresh Gupta (2016118) and Peeyush Kushwaha (2016254)

I. Problem Statement / Motivation

Working in adversarial settings is harder than playing single player games. What's even tougher going beyond the basic moves of the games and how to interact with arcade elements and training for tasks which usually require long-term decision making, planning and strategizing in presence of other agents.

We wanted to explore learning techniques for the above challenge. For this, we chose Pommerman - a NIPS Competition Track environment to train our agent on and trained several RL models which beats a random agent at 100% of the games, and our best agent beats the baseline agent (manually hard-coded) at 35% of the games.

Pommerman is a game similar to the classic bombmerman. In the game, multiple players are in a maze and have the ability to move and to plant bombs. The objective is to trap the enemy and kill them. The game also has some powerups which increase agent's capabilities.

We also submitted our second best agent (33.4%) to NIPS Competition to be held on 8th December.

II. Literature Review

We looked at two papers

- ”CLASS Q-L : A Q-Learning Algorithm for Adversarial Real-Time Strategy Games” by Jaidee and Avila
- ”Markov Games as a Framework for Multi-Agent Reinforcement Learning” by Michael L. Littman

The first describes a Q-learning variant for a realtime multiplayer strategy game Wargus (which is quite like the game ”age of empires”). For each race of characters available, the authors train a different Q-table. Since each race has unique abilities, they'll dominate other races at different strategies, thus having different quality values for states is warranted.

The second paper describes an extension to Q-learning for 2-player games inspired by the minimax algorithm. Rather than maintain a $Q[s, a]$ table, the authors propose maintaining a $Q[s, a, o]$ table where o is the action taken by the opponent. To use these Q-values for deciding a policy, we assume that the opponent will take an action which minimizes our Q-value, and we subsequently pick an action which maximizes our Q-value under opponent's optimal action.

Neither of these were directly applicable to our problem. For one, Pommerman does not have multiple races of characters, and it's a 4-player game rather than a 2-player game. This posed a challenge to us where we had to model these techniques for our solution.

We also used a standard TD Q-learning algorithms and compared our results with PPO and DQN approaches from a library called Tensorforce.

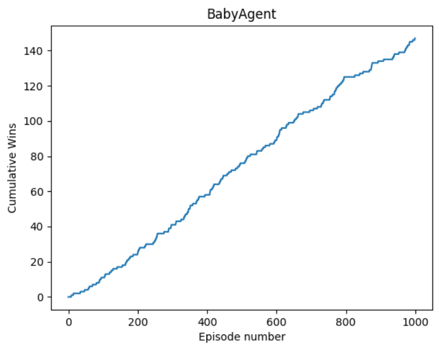
III. Database Details

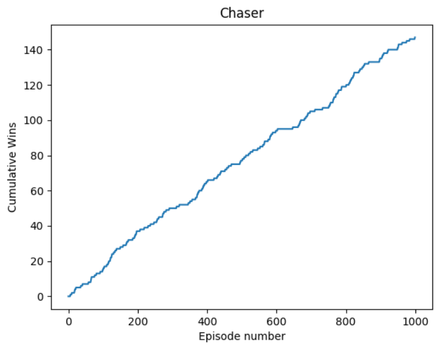
Not applicable

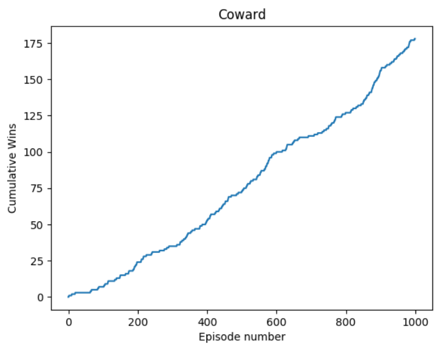
IV. Methodology Followed

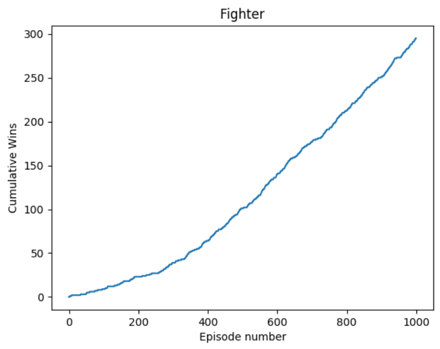
We trained our agents in a variety of ways. We modelled our state space as a Markov Model, initially starting with only 4 different states, gradually modelling the problem in a better way, including more possibilities as states in the subsequent versions of our Agents.

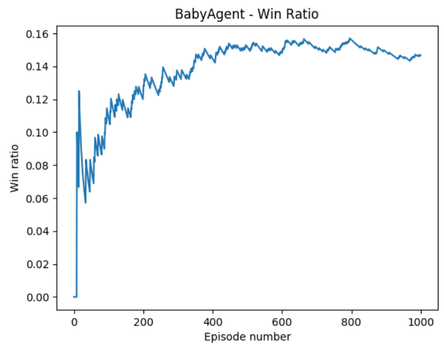
Our experiments yielded the following results:

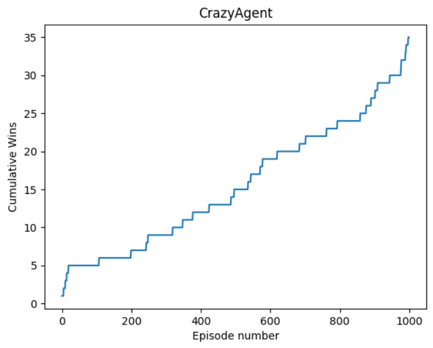


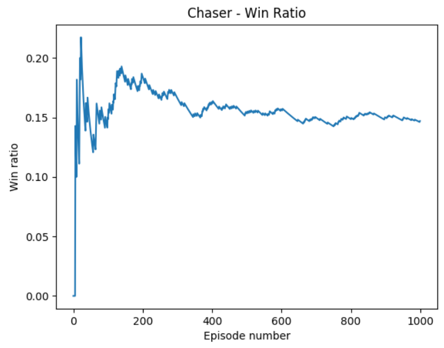


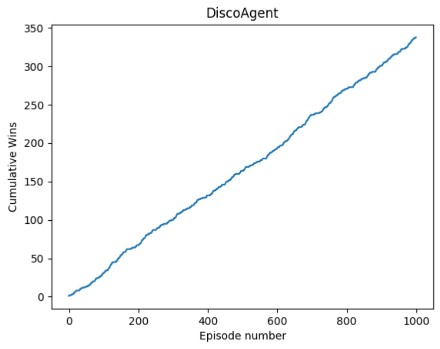


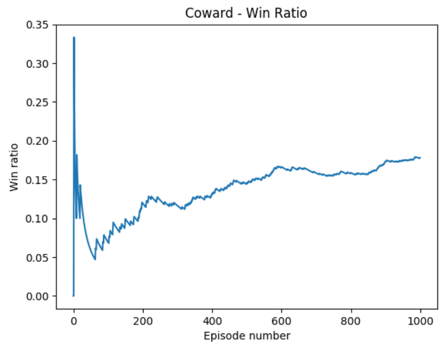


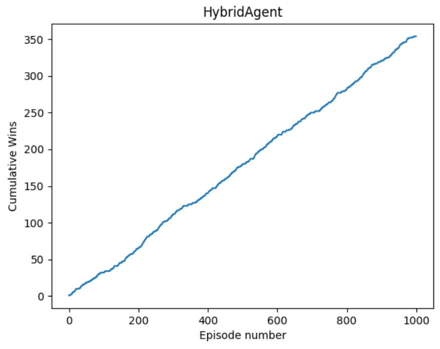


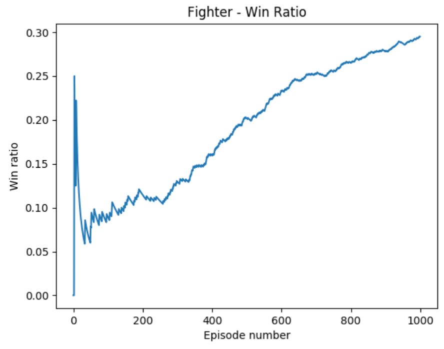




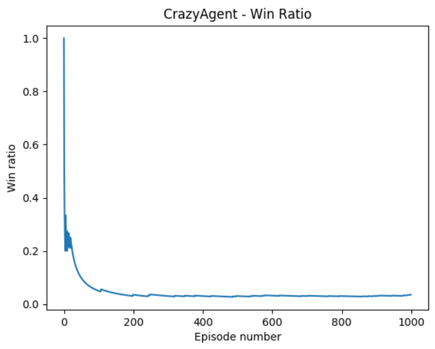


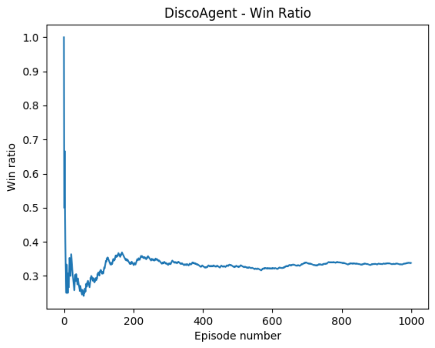


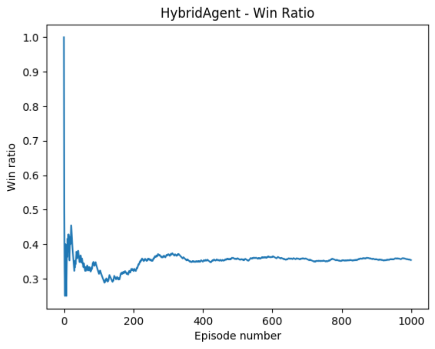




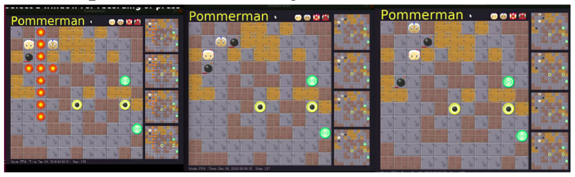
For a better analysis , we drew the win ratio graphs:







Also we got to observe that our agent could learn how to kick early in the state modelling (e.g Hybrid Agent could easily learn how to kick). E.g:



Red Agent Learns to kick

A. State Modelling

The game has both a partially observable and a fully observable mode. HybridAgent and DiscoAgent are functional on both, and BabyAgent and ClassQL agents require a fully observable environment.-

In Pommerman, each game is randomly generated. Therefore we had to model states in a way that they're independent of the random game level elements. We use a combination of the states below for different agents.

- Has a bomb in proximity: if there is a bomb in the immediate 8 squares surrounding the agent or if there is a bomb in 4x4 grid centered at our agent.
- Has an enemy in proximity : similar values as above
- Is Surrounded: The agent is vulnerable to attacks when surrounded on 3 sides, this is a boolean value indicating if the agent is currently surrounded
- Status of powerups: If our agent has powerups or not
- Number of enemies alive
- Direction (north, west, south, east) to the nearest enemy / powerup
- Status of enemy powerups (requires fully observable environment)

B. Training and Parameters

We tried a variety of training approaches

- Gradually increasing the difficulty of the agents trained against
- Using an exploration function to give an initial boost to state action pairs which are unexplored
- Using a decaying exploration parameter

For training our final agents, we used an exploration function with a cutoff of 100 visits before a state is considered explored, and we kept a very low non-decaying exploration parameter (0.01).

Our discount factor was high (0.99) because an episode consisted of visiting on an average about 150 states before the highest or the lowest rewards of winning and losing are awarded.

We trained the agents against the rival we measured them on (SimpleAgent, which was a hardcoded agent included in Pommerman) and against each other.

C. ClassQL

From BabyAgent as a base agent, we trained a CowardAgent, ChaserAgent, and a FigherAgent. Each of these use the same state space, have the same capabilities, but are trained for different playing styles (as the names suggest) by using different reward functions. Thus, each has its own Q-table.

D. Augmenting our Agents' Percepts and Actions

For BabyAgent, we experimented with providing it with richer information extracted from the environment (such as if the enemy has any ammo left or not) in a fully-observable environment. We also provided it with following ”virtual actions”:

- Chase the nearest enemy
- Get nearest powerup
- Get out of a boxed-in situation to a less constrained area

This was to test what happens when the RL agent learns to make the decisions as the situation requires it, but doesn't necessarily have to also learn how to carry out certain complex maneuvers.

We implemented breadth-first search for getting out from the boxed-in situation and all-pairs-shortest algorithm for the first two, which calculates paths in the beginning of the game and caches it as the maze does not change throughout the game.

V. Results

A. DiscoAgent and HybridAgent

These agents were developed primarily to work in a partially observable environment. DiscoAgent and HybridAgent beat SimpleAgent with 33% and 35% win rate. We submitted DiscoAgent to the NIPS Competition on 21st November. The difference between the two is that DiscoAgent has directions to the nearest enemy and powerup as a part of its model while the HybridAgent does not.

B. BabyAgent

FighterAgent was our fastest learning agent. Its aggressive strategy meant that it got to more win states earlier. When we trained BabyAgent, CowardAgent, and FighterAgent together from scratch, FighterAgent got 4x more wins than BabyAgent and 6x more than CowardAgent, by winning 1800 times.

C. PPO and DQN

We used Tensorforce's Proximal Policy Optimization and DQN implementations to compare our own agents. When given the same training time as our agents, PPO did not perform even against the random agents, and while DQN did achieve a win rate of 50% against SimpleAgent (as compared to our 35%), we did not see interesting behaviour (it learned to sit in place and wait for the other agent to kill itself).

VI. Takeaways

Even with an environment with observable internal state, the complexity of the state can be too high and careful modelling is required. For reinforcement learning, it's very important that the Markov assumption holds. We were careful about this in our own modelling of states and rewards.

ClassQL is a very good approach for games to learn and leverage different strategies of Gameplay.

VII. Failures

A. CrazyAgent

Initially, we started with a small state space (as little as 32 and 10 states) as we were worried about the iterations required with larger states, but the agent simply did not have enough information to make quality decisions at this level. We quickly learned that the agent could be reasonably trained to give decent performance on even up to 5000 states.

B. Minimax QL

Minimax QL is a great idea for 2-player games, but extending the Q-table to 3 adversaries would add a $|A|^3$ scaling factor to the initial Q-table's size. This was very large in our case (6^3 and 9^3), and we could not apply it to Pommerman effectively. We did implement Minimax QL on a simple game of Nim.

VIII. Conclusion

We also wish to add to our takeaways that non-deep reinforcement learning still gives good performance with careful modelling, especially with limited training.

We still think reinforcement learning has a long way to go in performing better in strategic environments.