Q learning and Monte Carlo Q Learning for Flappy Bird game

Nikola Zubić

Artificial intelligence Laboratory, Department of Computer Science, Faculty of Technical Sciences, University of Novi Sad

ABSTRACT

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. Along with supervised and unsupervised learning, it is one of three basic machine learning paradigms.

Q-learning is a model-free reinforcement learning algorithm. A model-free algorithm is an algorithm which does not use the transition probability distribution (and the reward function) associated with the Markov decision process, which in Reinforcement learning, represents the problem to be solved. The goal of Q-learning is to learn policy, which tells an agent what action to take under what circumstances.

The main idea of Monte Carlo Search (MCS) is to make some look ahead probes from a non-terminal state to the end of the game by selecting random moves for the players to estimate the value of that state.
Variant of Q learning combined with MCS is Monte Carlo Q learning.

It performs certain number of look ahead in search of good moves. The more simulations it has, the better the action it finds will be.

CONTACT

Nikola Zubić
Faculty of Technical Sciences, Novi Sad
Email: nikolazubic97@yahoo.com
Website: github.com/NikolaZubic

INTRODUCTION

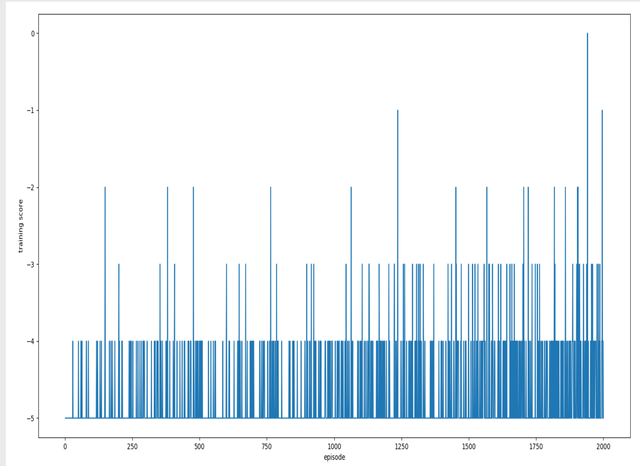
In this project we use Q learning and Monte Carlo Q Learning in order to train bird agent in Flappy Bird game, so it can maximize its score. The main goal of agent is to surpass through gaps between pipes.

We compare performance od these two agents. In this project we use PyGame Learning Environment (PLE) for mimicking the Arcade Learning Environment interface, allowing a quick start to Reinforcement Learning in Python.

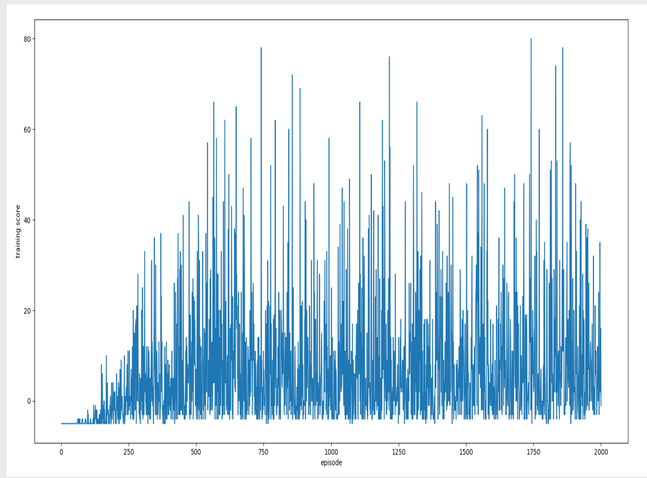
Game is restricted to 30 frames per second (30 fps). In action set we have two actions: “up” which causes bird to accelerate upwards, and “none” action. If the bird makes contact with the ground, pipes or goes above the top of the screen the game is over.

INITIAL RESULTS AFTER 2000 EPISODES

Q learning training for 2000 episodes lasted about 2 minutes, and Monte Carlo Q learning training for 2000 episodes lasted about 9 minutes. Here are the results.



Graph 1. Q learning training results after 2000 episodes. Best training score on average was about 0.



Graph 2. Monte Carlo Q learning training after 2000 episodes. Best training score on average was about 80.

Game difficulty	Q learning	Monte Carlo Q learning
Easy	5 (avg. 0.85)	infinity
Medium	4 (avg. 0.55)	infinity
Hard	2 (avg 0.3)	766 (avg. 431.1)
Speed of light	2 (avg 0.5)	286 (avg. 164.25)

Table 1. Best and average scores in game after 2000 episodes of training and 20 agent plays

Game difficulty	Q learning	Monte Carlo Q learning
Easy	1261 (avg. 456.3)	infinity
Medium	365 (avg. 198.45)	infinity
Hard	281 (avg. 108.9)	300 (avg. 135.55)
Speed of light	116 (avg. 32.5)	286 (avg. 164.25)

Table 2. Best and average scores in game after 150 000 episodes of training and 20 agent plays

APPROACHES

Q-learning: An off-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

Monte Carlo tree search will selectively try moves based on how good it thinks they are, thereby focusing its effort on moves that are most likely to happen.

1. Create a new Monte Carlo tree;
   The root node describes current game state. The state will then be used later to generate some successor states, and to evaluate them.
2. Simulation says how many expansions to run;
   Current node is expanded by trying every action and constructing a corresponding child node for each action.
3. Calculate the value of every child node;
   The game in the child node is rolled out by randomly taking moves from the child state until a terminal state. For every node, we store: W which is accumulated value and the number of times rollouts have been run at or below that node N.
4. Propagate information from the child nodes back up the tree in order to increase parent's W and visit count;
   Its accumulated value W is then set to the total accumulated value of its children.
5. From leaf nodes we select node for next state that has biggest upper confidence tree (UTC) score;
6. Expand selected leaf node and propagate values back up;
   W reflects whether that state can lead to a loss or not.

The tree is gradually expanded and we (hopefully) explore the possible moves, identifying the best move to take. The bot then actually makes a move in the original, real game by picking the first child with the highest number of visits.

CONCLUSION

These experiments show us that with time, Q learning is getting better. It improves Q-values and becomes efficient enough for this problem, like an experienced human player.
Monte Carlo Q Learning is simulating 60 times possible future incomes (every iteration) and his table is filled quickly with “good actions”. He converges very good after even 2000 episodes of training (about 9 minutes). So, at very beginning Monte Carlo Q learning is converging very fast. After 2000 episodes he has done 60 simulations per each which is 2000 * 60 = 120 000.
In every training phase, he is always better than basic Q learning agent.
But at “hard” game difficulty Monte Carlo Q learning shows poorer results after 150 000 iterations than after 2000 iterations because of large number of simulations. Large number of simulations over-train our agent, so when he encounters slightly different phase (pixel y position) he does not know exactly how to behave, performs random action, and often dies.