Learning in Games via Opponent Strategy Estimation and Policy Search

Yavar Naddaf
Department of Computer Science
University of British Columbia
Vancouver, BC
yavar@naddaf.name

Nando de Freitas (Supervisor)
Department of Computer Science
University of British Columbia
Vancouver, BC
nando@cs.ubc.ca

Abstract

In this paper, we model the problem of learning to play a game as a POMDP. The agent's hidden state is the opponent's current strategy and the agent's action is to choose a corresponding strategy. We apply particle filters to estimate the opponent's strategy and use policy search to find a good strategy for the AI agent.

1 Introduction

Despite the advances in the artificial intelligence and machine learning fields, the game-AI in the video game industry is still mainly implemented by creating large FSMs [3]. The time and labour required to implement and fine-tune these FSMs grows exceedingly as the games become more complicated. Also, since FSMs are by nature deterministic, the AI actions usually become predictable for the players, which results in a poor gaming experience [5]. Here, we demonstrate how a combination of particle filtering and reinforcement learning methods can be used so that the game-AI estimates the player's strategy and computes its own strategy accordingly. Instead of coding a large FSM for all players, the idea is to learn a different FSM for each individual player which is fine-tuned to her strategy.

2 Model Specification

2.1 Game Features

Our assumption is that in each instance of the game, the human player chooses her actions based on some features of the game environment. For some game environment X, we define the feature vector $F(X) = [f_1(X), f_2(X), ..., f_m(X)]$ where each feature $f_j(X)$ captures some aspect of the game. These features vary within different games and are picked by human experts. For instance, in a strategy game the features can capture the number of opponent's attack units, the amount of resources left in the game, or the time passed since the opponent's last attack.

2.2 Modeling players as classifiers

Given the game features $F(X)$, we can model the human player as a classifier, where at time t the probability of action y_t is given by

$$p(y_t|F(X_t), \theta_{op}) = \phi(F(X_t), \theta_{op})$$

ϕ is a parametric classification function, such as a neural network or ridge regression, with parameters θ_{op}. The main idea is that by estimating θ_{op}, we can predict the opponent's actions in different game settings. Furthermore, the AI player too can be modeled as a classifier with parameters θ_{ai}. In this setting, learning to play a game against an opponent is reduced to finding a 'good' set of θ_{ai} given our estimate of θ_{op}.

3 Opponent strategy estimation via particle filtering

3.1 General approach

To estimate the opponent's strategy, we employ a particle filtering approach suggested by Hojen-Sorensen, de Freitas and Fog for on-line binary classification [6]. This method is well suited to our problem because it is able to compute the class memberships as the classes themselves evolve over time. This will allow us to estimate a non-stationary strategy which varies through the game[1].

We adopt a Hidden Markov Model in which at time t, the opponent's classification parameters θ_t is the hidden state and the opponent's action y_t is the observation. We further assume that the classification parameters follow a random walk $\theta_t = \theta_{t-1} + \mu_t$ where $\mu_t \sim \aleph(0, \delta^2 I)$. In a generic important sampling algorithm, we start with N prior samples $\theta_0^{(i)}$. On each time step t, we predict N new samples $\hat{\theta}_t^{(i)} \sim p(\theta_t|\theta_{t-1})$, and set the importance weight of each sample to its likelihood $\omega_t^{(i)} = p(y_t|\hat{\theta}_t^{(i)})$. We then eliminate the samples with low importance weights (small likelihood) and multiply samples with high importance weights (large likelihood) to obtain N random samples $\tilde{\theta}_t^{(i)}$. To avoid a skewed importance weights distribution, in which many particles have no children and some have a large number of children, we also introduce an MCMC step. (See [4] and [2] for a detailed introduction to particle filtering and MCMC methods).

3.2 Binomial likelihood

Let us first consider a game with only two possible actions $\{0, 1\}$. We choose $\phi(F(X_t), \theta_t) = P(1|F(X_t))$ i.e. given the game features $F(X_t)$, our classification function returns the probability that the opponent will carry out action 1. Similarly, $P(0|F(X_t)) = 1 - \phi(F(X_t), \theta_t)$. The likelihood of opponent's action y_t is given by a binomial distribution:

$$P(y_t|F(X_t), \theta_t) = (\phi(F(X_t), \theta_t))^{y_t} (1 - \phi(F(X_t), \theta_t))^{1-y_t}$$

3.3 Multinomial likelihood

In a game with more than two actions $\{0, 1, \ldots, K\}$, a possible approach is to use a set of $\theta_t^{(1:K)}$ and let the probability that the opponent will play action i be $P(y_t = i|F(X_t)) = \phi(F(X_t), \theta_t^{(i)})$. However, this approach does not ensure that the probability of all actions

[1] There is a significantly more efficient alternative to what we use here, introduced by Andrieu, de Freitas and Doucet, in which instead of directly sampling θ, an augmented variable of much smaller dimension is sampled. See [1].

will sum to 1. To make the action probabilities sum to 1, we use a *softmax function*, so that the probability of action i becomes:

$$P(y_t = i|F(X_t), \theta_t) = \frac{\phi(F(X_t), \theta_t^{(i)})}{\sum_{j=1}^{K} \phi(F(X_t), \theta_t^{(j)})}$$

3.4 Particle Filtering algorithm

Now that we can compute the importance weight $\omega_t = p(y_t|F(X_t), \hat{\theta}_t)$ for each particle, we can use the particle filtering algorithm presented in Figure 1 to estimate the opponent's classification parameters.

- Start with N prior samples: $\theta_0^{(i)}$ $(i = 1, \ldots, N)$
- On each time step t of the game:
  1. Observe game features $F(X_t)$ and opponent action y_t
  2. Importance sampling step
     - For $i = 1, \ldots, N$ sample $\hat{\theta}_t^{(i)} = \theta_{t-1}^{(i)} + u_t$
     - For $i = 1, \ldots, N$, set $\omega_t^{(i)} = p(y_t|F(X_t), \hat{\theta}_t^{(i)})$
     - For $i = 1, \ldots, N$, normalize the importance weights:

$$\tilde{\omega}_t^{(i)} = \omega_t^{(i)} \left[\sum_{j=1}^{N} \omega_t^{(j)} \right]^{-1}$$

  3. Selection Step
     - Eliminate samples with low importance weights and multiply samples with high importance weights to obtain N random samples $\tilde{\theta}_t^{(i)}$
  4. MCM Step
     - For $i = 1, \ldots, N$
       - Sample $v \sim \mu_{[0,1]}$
       - Sample the proposal candidate $\theta_t^{*(i)} = \theta_{t-1}^{(i)} + u_t$
       - If $v \leq \min \left\{ 1, \frac{p(y_t|F(X_t), \theta_t^{*(i)})}{p(y_t|F(X_t), \tilde{\theta}_t^{(i)})} \right\}$
         - then accept move: $\theta_t^{(i)} = \theta_t^{*(i)}$
         - else reject move: $\theta_t^{(i)} = \tilde{\theta}_t^{(i)}$

Figure 1: Particle filtering algorithm for estimating opponent's classification parameters

4 Learning to play as a POMDP problem

4.1 Modeling the problem as a POMDP

If we model players as classifiers (as discussed in section 2.2), learning to play a game can be achieved in two steps:

- First we estimate the opponent's current classification parameters θ_{op} by looking at the history of the game features and the corresponding opponent actions.
- Once we have an estimate of the opponent's classification parameters (which enables us to predict the opponent's actions in different game settings), we try to come up with classification parameters for the AI player that maximizes its expected reward.

These two steps can be modeled by a Partially Observable Markov Decision Process, in which the agent's current (hidden) state is the opponent's classification parameters, the agent's observations are the game features plus the opponent's action, and the agent's action is choosing its own classification parameters.

4.2 Policy Search

Policy Search is a method to find an optimal policy in a POMDP by finding:

$$\pi^* = \arg\max_{\pi \in \prod}(V^\pi(s_0))$$

where Π is a set of policy functions $\pi^{(i)}$ each mapping the set of states S to the set of possible actions A. $V^\pi(s_0)$ is the expected reward of starting at state s_0 and following the policy π. In essence, what we are trying to do is to search in a set of policy functions for the policy that maximizes the expected reward. Of course, we do not know what the expected reward of each policy is, but we can approximate it using a Monte Carlo approach. Assuming that we know the initial state s_0 and we have a finite horizon of N steps, we will run M simulations to approximate $V^\pi(s_0)$. On each simulation i, we start from s_0, choose an action $a_0^{(i)}$ based on the policy π and end up in state $s_1^{(i)}$ based on the state-transition model $p(s'|s,a)$, receiving the reward $R(s_1^{(i)})$. Then, we choose another action $a_1^{(i)}$ based on π and transit to $s_2^{(i)}$, receiving the reward $R(s_2^{(i)})$. We will continue this until we reach the final state $s_N^{(i)}$. After M simulations, the Monte Carlo estimate of the expected reward of π will be:

$$\hat{V}^\pi(s_0) = \frac{1}{M}\left[\sum_{i=1}^{M}\sum_{t=0}^{N}\gamma^t R(s_t^{(i)})\right]$$

where γ is the discount factor. Now that we can evaluate the expected reward of each policy, finding $\pi^* \in \Pi$ becomes an optimization problem which can be solved by any standard optimization method, such as gradient descent or simulated annealing.

A problem with the above approach is that in order to estimate the expected reward of each policy, we have to sample a set of random numbers to simulate the state transitions[2]. Using different random numbers to estimate each policy will result in a high variance in $\hat{V}^\pi(s_0)$ and a poor π^* will be found by the algorithm. The solution is to sample M random seeds at the beginning of the algorithm, and use the *same* set of random seeds to estimate the expected reward of each policy. (This approach is based on Andrew Ng's *Pegasus* algorithm. See his thesis [7] for a very comprehensive introduction to POMDP, policy search methods, and in particular Pegasus).

[2]If the rewards are stochastic, as is the case in our example in section 5, finding $R(s_t^{(i)})$ will also require sampling random numbers.

Table 1: Possible actions in Sand-Wars and their outcomes

ACTION	OUTCOME
Upgrade the castle wall	Probability of being hit is reduced by w_1 in the current turn and by w_2 in the following turn
Upgrade the catapult	Probability of hitting the opponent is increased by c
Throw the sand bag	The opponent's castle is hit by probability *P(hit) = f(Player's Catapult Upgrade, Opponent's Castle Wall Upgrade)*

5 Experiment: Sand-Wars

5.1 Game description

To test our POMDP player, we developed a simple strategy game, called: Sand-Wars. There are two players in the game, each controlling a castle and a catapult. On every turn, each player receives a bag of sand and is given three choices on what to do with it. The player can either upgrade her castle walls, upgrade her catapult, or through the bag toward the other castle. Table 1 describes the outcome of each action[3]. Once both players have chosen their actions, both actions are carried out simultaneously. If a player hit the other's castle, she receives one point. At the end of N turns, the player with the higher score wins the game.

5.2 Implementation

To apply the algorithms described in sections 3 and 4 on the Sand-Wars game, we chose the game features to be a vector of four elements: *F(X) = [Player's Wall upgrade, Player's Catapult upgrade, Opponent's Wall Upgrade, Opponent's Catapult Upgrade]* (all values are normalized to be between 0 and 1). We used a logistic function as our parametric classification function, so that:

$$\phi(F(X), \theta) = \frac{1}{1 + e^{-\theta \cdot F(X)}}$$

Finally, we chose simulated annealing as our optimization method for the policy search algorithm.

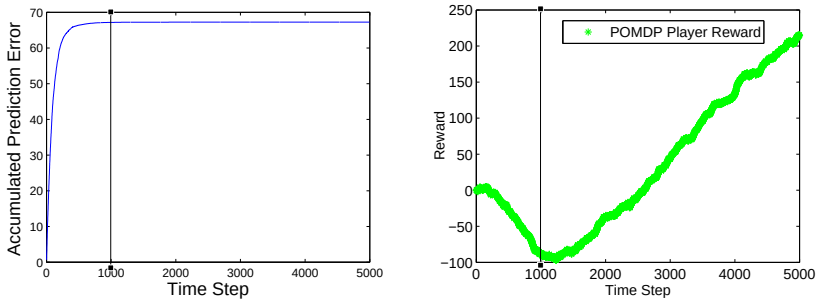
5.3 Results

We played our POMDP player against a number of hand-coded opponents for a few thousand rounds. In general, the algorithm was able to predict the opponent's actions with small variance within a few hundred rounds. Also, the player's policy usually improved significantly. However, policy serach did not always find the optimum policy by the end of the game. A big challenge was finding good parameters for the simulated annealing algorithm (e.g. the initial temperature, the cooling rate, number of optimization steps, etc.). Improved results may be possible by either finding better parameters for the simulated annealing algorithm, or using a different optimization method.

Figure 2-(a) shows the particle filter's accumulated error in predicting the opponent's actions, when playing against a 'conservative player' for 5000 rounds. A 'conservative

[3]Note that a wall upgrade is only effective for the current and the following turn. This time limit also applies to upgrading the catapult, which is only effective for the very next turn.

player' is a hard-coded player who always upgrades her castle walls if there is no wall upgrades, and throws the sand bag otherwise. Figure 2-(b) shows the obtained reward in the same game. We started the policy-search algorithm at the 1000_{th} round and re-ran it every 50 rounds.



(a) Accumulated prediction error (b) POMDP player's reward

Figure 2: Playing vs. the conservative player

In the game plotted in Figure 2, the POMDP player learned to repeatedly upgrade the catapult and then throw the sand bag, which seemed like a reasonable strategy given our values of w_1, w_2, and c. In later experiments, we changed w_1, w_2, and c so that the catapult upgrade was less beneficial. The POMDP player then gave up on the catapult upgrades and learned to simply throw the sand bag on all turns.

6 Conclusion

In this paper we modeled the problem of learning to play a game as a POMDP, and applied particle filtering and policy search to try to solve it. We demonstrated our approach on a simple strategy game with some promising results.

References

[1] Andrieu, C., de Freitas, N., Doucet, A. (2001). RaoBlackwellised particle filtering via data augmentation. *Advances in Neural Information Processing Systems*, 13.

[2] Andrieu, C., de Freitas, N., Doucet, A., Jordan, M. (2001). An introduction to MCMC for Machine Learning. *Machine Learning*, 50, 5–43.

[3] Buckland, M. (2002). *AI Techniques for Game Programming*. Cincinnati: Thomson Course Technology.

[4] Doucet, A., de Freitas, N., Gordon, N. (2001). An Introduction to Sequential Monte Carlo Methods. *Sequential Monte Carlo Methods in Practice*, 3-14.

[5] Fairclough, C., Fagan, M., Namee, B., Cunningham, P. (2001). Research directions for AI in computer games. *Proceedings of the Twelfth Irish Conference on Artificial Intelligence and Cognitive Science* , 333-344.

[6] Hojen-Sorensen, P., de Freitas, N., Fog, T. (2000). On-line probabilistic classification with particle filters. *Neural Networks for Signal Processing X, 2000. Proceedings of the 2000 IEEE Signal Processing Society Workshop*, 1, 386–395.

[7] Ng, A. Y. (2003). *Shaping and Policy Search in Reinforcement Learning*. PhD thesis, EECS, University of California, Berkeley.