Reinforcement Learning in the Presence of Hidden States

Andrew Arnold Andrew Howard

{aoa5, ah679}@columbia.edu
Computer Science Department
Columbia University
500 West 120th Street, New York, NY 10027

May 7, 2002

Abstract

We present a system to train adaptive poker playing agents. These agents are trained using a variation on reinforcement learning. This variation uses EM-based clustering to map explicitly seen inputs to implicitly defined hidden states. It then chooses actions based on these hidden states and all subsequent rewards.

1 Introduction

The game of poker presents a unique combination of problems when viewed from a machine learning perspective. While many other games have already been studied and excellent results achieved using non-machine learning approaches, such as chess or even tic-tac-toe, these games are distinct from a game such as poker in that, in chess, for instance, at each move decision, a player has complete knowledge of the current game state: namely, the position of each piece on the board. Using this information, a simple approach would be to search possible subsequent opponent moves, using a technique like α-β pruning, and chose the move with the best prospects, as measured by some heuristic. In poker, however, this complete state is not entirely visible. For instance, one's opponents' hole cards are hidden throughout the game. In addition, the space of possible subsequent cards that might come out of the deck is quite large and, since it is entirely unpredictable, barring the player's knowledge of already played cards, represents a large unknown. While this problem seems quite daunting, some solace can be taken in the fact that human players seem to fare rather well.

Work has already been done in the past using statistics and basic probability theory [1], but this lacks the ability to learn to adapt to specific opponents. This is the second main reason poker provides such a unique challenge: because of all of the hidden information, a player can only be consistently successful by inferring information based on the actions and behaviors of one's opponents. With humans in a live game, this usually takes the form of watching for an opponent's tell, a behavioral cue, like a twitch, that consistently reveals some information about that opponent. This level of observation, at least for now, is beyond the state of the art for computer vision / gesture recognition systems. One can instead focus on the explicit betting history of each player during the course of the game to model an opponent.

2 Problem Domain

Texas Hold'em was chosen as our problem domain because it is the most strategically pure variation of poker. It limits the randomness of luck.

2.1 Rules of Texas Hold'em

In the game of Texas Hold'em up to 10 players are dealt two cards, face down. At this point, a round of betting takes place, which allows for up to four bets (an initial bet plus at most three raises). Three communal cards (the board) are then dealt face up, followed by another round of betting. Another card is dealt face up, and another round of betting transpires. Then a there is a final, fifth card is dealt, and a final round of betting. Each time a player is faced with a betting decision, he must select one of three discrete actions:

- Fold - Abandon one's hand and surrender all money already in pot
- Call - Match the current bet by placing that much money in the pot
- Raise - Match the current bet, and add an additional amount to it that must be matched by all subsequent players

A correlation between action and hand strength can be discerned by even the most novice poker player. This was one of the meager goals of this research, i.e., to allow an agent to recognize that an opponent who raises probably has a better hand than an opponent who calls.

2.2 Feature Representation

Once the domain of interest had been selected, and the problem constrained, a useful representation of the salient features of the problem space had to be developed. A distinction should be made clear here between raw input data and features. Raw data would be something like the character string "Ac9c3c4c5c" representing a five card hand of the Ace of clubs, 9 of clubs, 3 of clubs, 4 of clubs and 5 of clubs. While this does capture all the information of the game state, a more salient representation might be something like "flush, Ace high." By incorporating more of this *a priori* information into the feature representation, the agent should learn the domain that much more easily.

2.2.1 Cards

In terms of card representation, previous researchers [1], upon whose work much of the poker specific implementation of this agent was based, had constructed two such features, derived from the raw input data, that seemed to have the most significance. It was on these features, and not the raw data, that the agent learned.

The first of these was hand strength. This was a relative ranking of the player's current hand tabulated by searching over all possible opponent hands and counting the number of these possible hands the current player's hand beat, and how many it lost to.

Similarly, the second feature, hand potential, consisted of an exhaustive search over all possible subsequent cards that might be dealt. The best possible hand formed between the player's current hand and each projected hand was then determined, and this hand's rank again calculated. The hand potential, then, is a ratio between the number of times the current hand improved, positive

potential, and the number of times its strength decreased, negative potential. As an example, consider two hole hands: a pair of aces, and a 5 and 8 of clubs. While the pair of aces clearly has higher strength than the 5 and 8, at the beginning, it will likely only improve in 2 out of 50 cards, by drawing another ace, whereas the 5 and 8 of clubs can improve to a flush by drawing any 3 of the 11 remaining clubs. Thus this flush draw would have higher positive potential than the pair of aces.

2.2.2 Betting

Representing the betting history of the opponents is a subtler issue. The features of interest are not as readily apparent. For instance, would it be useful to record the betting action of every opponent for every hand played? Or would that type of input just confuse the agent and hinder learning? In the end, again, a compromise was struck between data richness and feature preprocessing. Different betting history representations were used for different experiments. The one that was found to be the best compromise was to incorporate how much the agent had invested in the pot, to help deter arbitrary folding, and also to see how much the other players had in, to learn attributes of opponents such as loose playing, frequent bluffing (observed as an opponent who frequently has large amounts in the pot but only seldom wins), etc. This kept the computational complexity low while providing valuable bias from which the agent could learn, both about its own strategy and that of its opponents.

3 Algorithms

The crucial problem is to learn to choose, with some probability distribution, an appropriate action given a certain input. If this were the entirety of it, a basic EM algorithm could be used to cluster into actions based on inputs. However, as shown graphically in Figure 1, there is a hidden state space blocking the actions and inputs.

Figure 1: A graphical model showing the dependency between inputs, x, states, s, and actions, a. Directed arrows represent influence.

To get around this problem, we take advantage of the independence between x and a, given s. In other words, we can marginalize the hidden state s and then use the fact that $p(a|x^n) = \Sigma_s p(a|s)p(x^n)$ to decompose the value of interest, $p(a|x^n)$, into its constituent components, which we can calculate. What follows is an algorithm that implements such a computation.

3.1 Reinforcement Learning and the Q-Algorithm

Reinforcement learning is the task of having an agent learn the appropriate behavior, given a certain state, that maximizes all future rewards. The classic examples of this kind of agent would be a robot navigating in an environment, and the computer intelligence behind a game. Formally, an agent first senses s_t the current state where $s_t \in S$ the set of all distinct states. It must then choose a_t where $a_t \in A$ the set of all distinct actions. The environment will then respond with a reward $r_t = r(s_t, a_t)$ and a new state $s_{t+1} = \delta(s_t, a_t)$. δ and r are part of the environment and are not known to the agent. They can also be nondeterministic functions. The goal is to learn a

policy, $\pi : S \rightarrow A$ that will maximize all future rewards $\sum_t^\infty$. In order to learn this policy we will create a data structure called the Q-table and an update rule called the Q-algorithm. The Q-table is a two dimensional matrix indexed by states and actions. This table will contain the average reward plus the discounted future rewards for each state and action pair. It is impossible to know all future rewards due to the nondeterministic nature of r and δ so a recursive approximation has been derived [4] to approximate Q, the Q-table.

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \gamma \max_{a'} \hat{Q}_{n-1}(s',a')] \tag{1}$$

$\hat{Q}_n$ has been shown to converge to Q_n as the iterations $n \rightarrow \infty$. $\alpha_n = 1/(1+visits_n(s,a))$ scales how much this new value affects the Q-table, and $0 < \gamma < 1$ is a discount on all future rewards. This algorithm works when the states s are explicitly known. However if they are not explicitly known a modification to this algorithm must be made.

3.2 Gaussian Clustering Using the Expectation Maximization Algorithm

A Guassian mixture model [3] is used to cluster from the input space, x, to the state space, s. The Expectation-Maximization algorithm (EM) is used to learn these clusters, given unlabelled input vectors as described in §2.2. Specifically, the mean, μ_i, and covariance, Σ_i, of each Guassian have to be learned. In addition, each Guassian is weighted by a mixing proportion π_i such that $\Sigma_i^M \pi_i = 1$. By varying the number of Guassians, M, used in the mixture model, the number of explicit states being modeled can be controlled.

Given these model parameters, Θ, the basic EM algorithm iterates between

- calculating the posterior probabilities, τ, given the current Θ
- recalculating the Θ according to these τ

This process, while guaranteed to converge to at least a local maxima of the log-likelihood function, does not incorporate any information about the action taken or reward received, all of which are dependent on the clustering of inputs to states, as shown graphically in Figure 1.

3.3 A New Approach to EM

A way around this limitation, as proposed in [2], is to link the Q-table, and thus the action and reward information, with the E-step of the classic EM algorithm. Specifically, the vanilla posteriors are used to generate a probability tuple into the state space. The Q-table then uses that tuple to select an action, and collect an appropriate reward, as generated by the game simulation. The updated Q-table is then used to compute a new, weighted posterior, and it is this modified value that is used by the M-step of the EM algorithm. This process continues for each new datapoint received, with the agent, at each point, refining its model parameters Θ to reflect the knowledge it has incorporated.

A pseudocode implementation would be:

- Initialize model parameters, Θ, and Q-table
- The E-Step:
  1. Compute posteriors, $p(s|x^n)$, using Bayes rule
  2. Convert Q-table to $p(a|s)$

3. Compute $p(a|x^n) = \Sigma_s p(a|s)p(s|x^n)$
4. Select an action by sampling $p(a|x^n)$
5. Collect reward
6. Update Q-table $\hat{Q}(s,a) \Leftarrow \gamma\hat{Q}(s,a) + \delta(a,a^n)p(s|x^n)r$
7. Convert Q-table to $p(a|s)$
8. Compute improved posteriors: $p(s|x^n,a^n) = \frac{p(a^n|s)p(s|x^n)}{p(a^n|x^n)}$

- The M-Step
  1. Update Θ to maximize log likelihood, $p(x^n,s)$, with respect to improved posteriors $p(s|x^n,a^n)$

4 Experiment

Our experiment was to train various agents against randomly generated computer opponents and evaluate their performance based on their winnings.

4.1 Agents

The agents were generated using various combinations of our parameters. One parameter is the number of states. These states are represented by the number of Gaussians in the mixture model part of our algorithm and by the rows in the Q-table. The features that we train on, described in §2.2, were varied to determine the optimal mix and number of features. The constant γ, the discount on the previous Q-table value in eqn. (1), was also varied.

4.2 Opponents

Our agents were trained using games of nine opponents. Each opponent was assigned a probability triple, $\{P(fold), P(call), P(raise)\}$ where $P(fold) + P(call) + P(raise) = 1$. These random opponents would choose their action based on this probability triple. The randomness of these opponents should force our agents not to adapt to their specific opponents but just to learn sound poker strategy. Once these basic strategies are learned, the agent can be used against real people and adapt to their unique playing style.

5 Results

After testing many agents, we found that if we used too many states and input features, that $p(s|x)$ would become too small to be represented with a Java double variable. These numbers could not be trivially zero, so we were limited in the number of states and features that were used. There were also problems when the agents were trained for many iterations, again $p(s|x)$ would become too small. There were many unsuccessful agents and few that were successful.

Figure 2a shows three of our unsuccessful agents. The first was trained using two features, hand strength and hand potential. It used five hidden states. The second agent also used the same features but used seven hidden states. The third agent used four features, hand strength, hand potential, the current round, and total amount that our agent has wagered. This agent uses seven states.

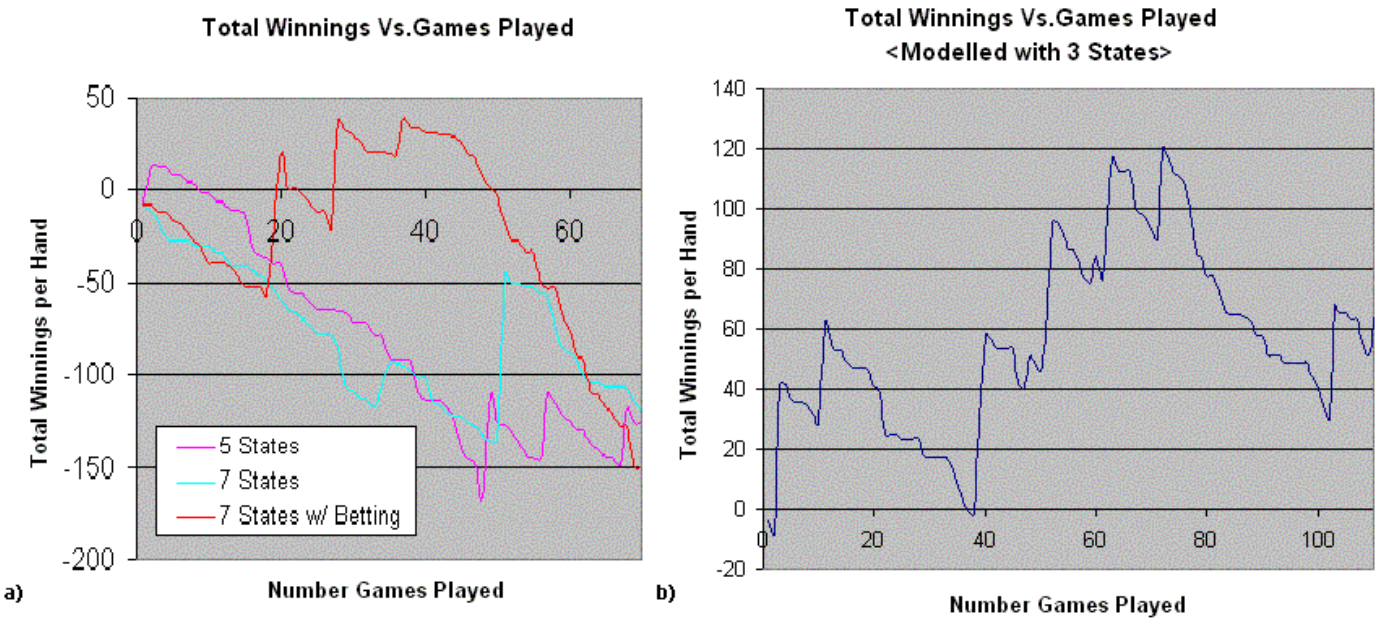

Figure 2: a) A comparison of the performance of the agent during training using a varying number of states and feature representations; b) one well performing agent.

Figure 2b is an example of one of our successful agents. This agent used only hand strength and hand potential for its features and had three hidden states.

Figure 3 shows the probability triples $P(fold), P(call), P(raise)$ of an agent trained with four features hand strength, hand potential, round, and agent's amount wagered. It used seven states. The graph shows for each decision that must be made what the probability of fold, call, or raise is. This can clearly be seen changing based on the learning and what the state is at a given time. A high probability of fold would correspond to a state with poor relative cards. A high probability of raise should correspond to a state with good relative cards. A high probability of call would correspond to middle of the road cards. A mixture of probabilities would correspond to a hand somewhere on a state boundary.

6 Conclusion

One of the most important aspects to finding a successful agent was trying many different agents. We must deal with the randomness of initialization, the randomness of game play, and local maxima that occur in the Q-table and the EM. A vast number of parameters must be tried out to find a good mix. All of this creates the need for many trials. Our intuition was that more states and more features would create a more successful agent. However, after looking at the results, our best agent was the simplest one. It used only three states and two features. We were having tractability issues with higher dimension input spaces, and higher dimension state spaces. There were also tractability issues with the number of iterations to train an agent. If these were solved we might find successful agents more in line with our intuition.

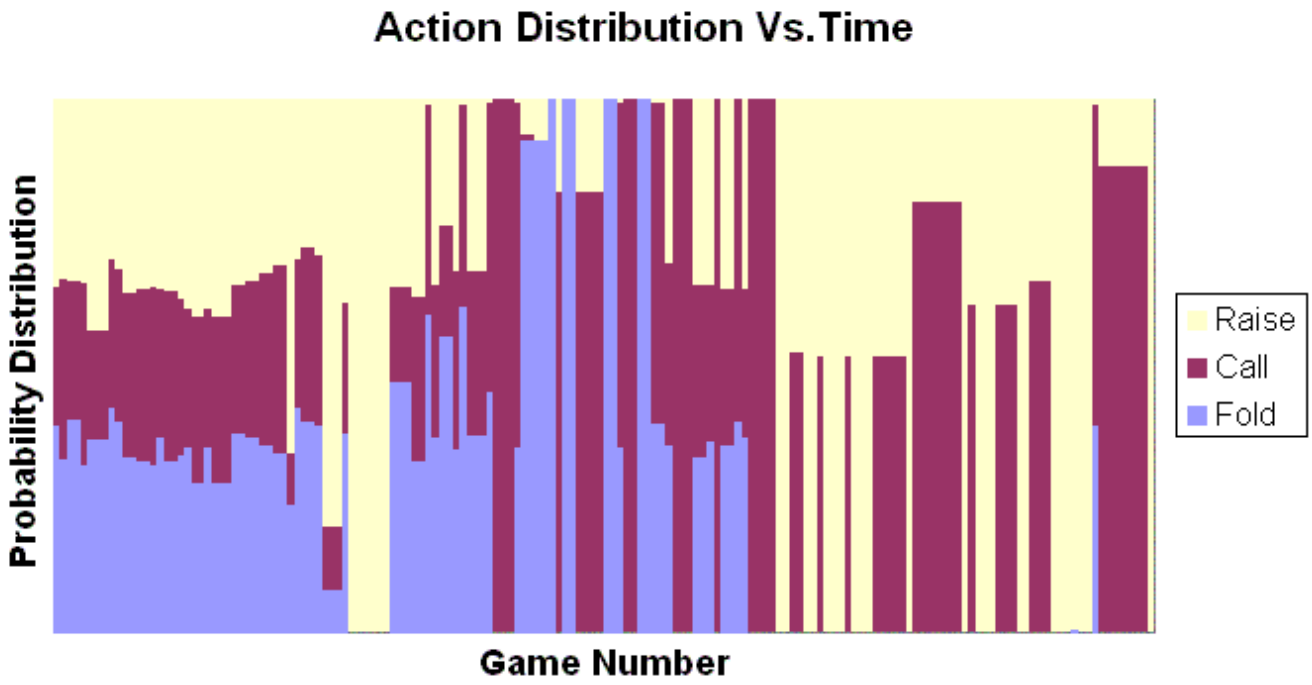

Figure 3: Samplings of the action probability distributions, $p(a|x)$, at various stages of the agent's training.

7 Future Work

There is much future work that can be done within our learning framework. The most important issue that needs to be addressed is creating a method to allow for more states and features. This could possibly be accomplished by scaling certain probabilities that are causing errors. Another possible improvement would be using an on-line variant of the EM algorithm. This would reduce computation time during training and speed up decision processes. Feature selection can also be improved by incorporating more *a priori* information based on poker strategy. Finally, we would like to train and test against real players by implementing a standard socket based protocol for playing on IRC[1].

References

[1] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence Journal*, 2001.

[2] Y. Ivanov, B. Blumberg, and A. Pentland. Em for perceptual coding and reinforcement learning tasks. In *8th International Symposium on Intelligent Robotic Systems*, 2000.

[3] M. Jordan and C. Bishop. *Introduction to Graphical Models*. Forthcoming.

[4] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.