MIT ES.S20 Lecture 11: Poker and Gambling

Lecture Outline

1. Rules of Poker
2. High-Level Model of a Simple Poker Agent
3. Human Algorithms
4. Linear Programming
5. Monte-Carlo Tree Search
6. Sampling and Regret Minimization
7. Opponent Modeling
8. Variance Propagation

Rules of Poker

- The Main Problem: Texas Hold'em

Each player is dealt two cards and the game consists of four betting rounds (called *preflop*, *flop*, *turn*, and *river*), during which players have the opportunity to bet counterclockwise from the dealer. Any player who does not match the highest bet must fold the round.

Each player is initially dealt two secret cards, which can be used in conjunction with public cards (the *board*) to form a hand. After the flop, three cards are placed on the board, then one after the turn and river. Hands are ranked as such:

High Card < One Pair < Two Pairs < Three of a Kind < Straight < Flush < Full House < Four of a Kind

At the end of the final betting round, the highest-raked hand of all remaining players will receive all betted money.

- Simplified Model: Kuhn Poker

For the abstract analyses, I will use *Kuhn Poker*, a vastly simplified two-player game in which there are only three cards (J, Q, K), two actions (bet 1 or pass), and the highest card wins. This

- Other simplified models

Other possible models include one-card poker (Kuhn with a larger deck); Goofspiel (bidding on point cards using hand cards); and Liar's Dice (bluff on dice in play).

High-Level Model of a Simple Poker Agent

- Raw probability of winning

We're given our two cards, and a board of 0-5 cards shared by the players. We get to choose the best hand we can make out of them

```
bestHand :: (SecretCards, BoardCards) -> Hand
```

so that Hands can be compared using the usual operators (<, ==, etc.). We can define the *spectrum* of a set of board cards to be a distribution over the best hands that could be made from it, assuming that the rest of the 5 cards on the board and the cards in an opponent's hand are dealt in a uniformly random way.

```
spectrum :: BoardCards -> (Hand -> Probability)
```

Our probability of winning with our current cards against one opponent can then be quantified

```
probWin secret board =
    sum (filter (\x -> bestHand secret >= x)
            spectrum board)
```

And the probability of winning against n opponents is multiplicative.

```
probWinAll nOpponents secret board =
    1.0 - (probWin secret board) ** nOpponents
```

Clearly in poker, these computations are complicated, so our first task is to be able to estimate probWin. In poker bots, it is usually interpolated over tables.

- Revising Spectra

We also need to react to our opponents' bets. If we assume some basic opponent rationality, we can say that our opponent will bet in a certain way if s/he has certain types of cards. So, there should be some subset of hands in the spectrum that are inconsistent with our opponent's betting history.

Therefore, we can add a weight adjustment based on priors of what our opponent would bet given certain types of hands into the spectrum multiplicatively:

```
spectrum :: BoardCards -> Prior -> (Hand -> Probability)
```

The prior is also a function of the pot, player behavior, etc.

- Decision-Making

Our decision model will be such that on our probWinAll assessment of the situation, we will define thresholds F and B such that we just choose

```
    pWin < F  => Fold
F < pWin < B  => Call/Check
    pWin > B  => Bet/Raise
```

Clearly, the threshold B should be some function of the amount of the bet, which turn it is, and how much money we have. Practically, the folding threshold is determined by the odds implied by how much each player has contributed to the pot:

```
impliedOdds = remainingBets / (finalPot - futureBets)
foldThresh  = impliedOdds / (1.0 + impliedOdds)
```

So, it's basically a ratio measuring what percentage you stand to lose by folding.

- Opponent Modeling

One of the key steps in the process above is inferring a prior on our opponents' hand from the betting information, given the state of the game. The traditional approaches look very ad-hoc to me, so I would recommend an approach based on Bayesian Networks (basically, learn a prior on the hand evaluation as before):

```
Opponent Capital -> Opponent Bets <- Opponent Hand
```

We can learn weights on the edges.

Human Strategies

- How to Play

As in many other games we've seen, humans generally engage in the relatively un-mathematical practice of using *features* and *rules of thumb* to characterize Poker. These collections of wisdom are collected into various *systems*, of which *The Theory of Poker* is the only one I've found useful. In addition to the obvious "when to raise", some of the features people care about are:

```
Looseness/Tightness       (players' bias toward certain behavior)
Signaling           (getting information from bets, sometimes intentionally)
"Free" Cards              (checking to get more information)
Bluffs/Slowplaying        (playing unlike your hand)
Semi-Bluffs               (when you think you'll get a good hand)
Implied/Effective Odds    (as above)
```

Half the battle is fighting against psychology. A useful technique is the Fundamental Theorem of Poker (which is actually just a mental technique):

> A mistake is when you play differently than the way you would have if you could see your opponents' cards.

There are two goals: accurately reading your opponents' hands and getting your opponents to read your hand inaccurately.

- When to Play

Deciding when to play a hand, however, is a bit a matter of your probability of winning. The common wisdom is to use a big table of pre-computed near-optimal decisions, or use a simple system like SAGE:

```
Ratio       = small stack / big blind
Power Index = Card value (+2 if suited) (+22 if pair)
```

Then the PI threshold for playing is:

```
          ratio:       1   2   3   4   5   6   7
pushing from SB:   T = 17  21  22  23  24  25  26
calling from BB:   T = *   17  24  26  28  29  30
```

Linear Programming

The classic way to compute Nash Equilibria in Extensive-Form Games is by converting the problem to a linear program: we will discretize the space of all combinations of hand strengths into buckets, probabilities to each possible event, and discretize the strength of hands (along with the transitions between the strengths of hands). This will yield a computable (but coarse) linear program which results in an efficient mixed strategy.

The problem, of course, is the complexity of solving the linear programs; but we should also be concerned with whether or not choosing a predictable strategy makes us *exploitable*.

Monte-Carlo Tree Search

Let's construct a tree representation of a poker game out of two types of nodes: *action nodes*, in which we make a decision; and *chance nodes*, in which a random event happens. We can use this method to serialize random events such as adding cards to the board. We will want to maintain a *value* of each action node, the average reward of all simulated games after this action; and we will maintain a *visit count*, the number of simulations run after a given action.

The MCTS process, just like in the case of Go, proceeds in four steps, with the added complexity that we have to maintain *imperfect information* and *random events*:

1. *Selection*: select the next node to expand. In this sampling approach, we will choose the *action* nodes leading to the highest expected value (in the paper, `argmax_a (v_a + c * sqrt(ln (n_p / n_a)))` for some tunable constant c, with p being the parent).
2. *Expansion*: we expand the selected node into a number of possible actions.
3. *Simulation*: we simulate the rest of the round randomly (*sample* in the space of possible hidden states), with our opponents making choices according to a probability distribution determined from the current estimated state.
4. *Backpropagation*: we propagate the results of the simulation up to all of the ancestors of the simulated game.

MCTS is inferior in this application: it will oscillate around a NE, and in the case of poker, we can do better. It will actually beat pre-computed strategies, though: MCTS will learn what the strategy is and will adjust accordingly. We need an algorithms that will change with our opponents' behavior.

Sampling and Regret Minimization

based on "Computing Approximate Nash Equilibria and Robust Best-Responses Using Sampling" by Ponsen et al.

- Regret Minimization

Define the *Regret Value* as the amount of payoff we miss out on by using some strategy s at time I. Let Z(s) be the set of histories of the game such that I is the first event and we use s:

```
v_I(s) = sum_{z in Z(s)} Pr(other players' decisions at I)
                         * Pr(the rest of history z)
                         * Payoff(z)
```

Then, we say that our *counterfactual regret* for not choosing strategy a is:

```
r_I(a) = v_I(a) - v_I(s)
```

Minimizing this regret at each time step will give a Nash Equilibrium. Note, however, that this will be very complicated for long games.

- Monte-Carlo Counterfactual Regret Minimization

Let's say that we only care about approaching a Nash Equilibrium *in expectation*. Then, we can *sample* some subset of the space of histories Q(s) in Z(s). The equation will be:

```
v_I(s | Q) = sum_{z in Q(s)} Pr(other players' decisions at I)
                             * Pr(the rest of history z)
                             * Payoff(z)
                             / Pr(history z was sampled)
```

We can use this directly as the regret calculation. Lancot et al. chose to sample regions based on outcome, but I think a uniform strategy is easier to justify.

Adopt a randomized strategy: weight all choices by their proportion of regret at that time and choose randomly.

This works well, but it converges more slowly than MCTS. Note that most of the computation (especially with many players) will come from our computing our opponents' bluff strategies.

Opponent Modeling

- Restricted Strategies

What if we could use the same sampling method to compute strategies offline? Let's combine MCCRM with a Restricted Nash Response: we say that with some probability p, our opponent is restricted to a fixed strategy, and otherwise plays with a CRM strategy. The choice of p is a tunable parameter: if 0, then it solves the game analytically; and if 1, it exploits the opponent's strategy. In this way, p looks like our *confidence* in our model.

Given a p, we can make an extra step in our sampling algorithm which chooses the fixed or optimal strategy. This means that this fits directly into MCCRM, and represents a speedup when we don't have to compute our opponents' regret.

The key insight is that we can *learn the opponent's strategy* (and theoretically, p). We've seen that technique before… this is a very powerful way to learn to exploit opponents' weaknesses in behavior.

Note that this can also be used in the simulation step of MCTS, but the models would have to be much simpler.

- Model Distributions

In this approach, we should note that there is no mathematical restriction on how, exactly, we restrict our opponent's strategy – we can, for example, have an arbitrary number of restricted strategies. This is very much in-keeping with the idea of

Point being that against any restricted strategy, we can *precompute best responses* offline, storing all of them in a distribution over models. Then, we can learn on-line which of these work best.