Imperfect Information Games: Poker

Keith L. Downing

January 17, 2023
Classes of Games (from AI textbooks)

- **Perfect Info** - state of playing arena and other players holdings known at all times.
- **Imperfect Info** - some info about arena or players not available.
- **Deterministic** - outcome of any action is certain.
- **Stochastic** - some actions have probabilistic outcomes, e.g. dealt cards, rolled dice, etc.

<table>
<thead>
<tr>
<th>Deterministic</th>
<th>Stochastic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect Information</td>
<td>Imperfect Information</td>
</tr>
<tr>
<td>Stratego</td>
<td>Bridge</td>
</tr>
<tr>
<td>Scotland Yard</td>
<td>Risk</td>
</tr>
<tr>
<td>Battleship</td>
<td>Poker</td>
</tr>
<tr>
<td>Chess</td>
<td>Hearts</td>
</tr>
<tr>
<td>Checkers</td>
<td>Scrabble</td>
</tr>
<tr>
<td>Othello</td>
<td>Monopoly</td>
</tr>
<tr>
<td>Go</td>
<td>Backgammon</td>
</tr>
<tr>
<td>Ludo</td>
<td>2048</td>
</tr>
</tbody>
</table>
The terminology is slightly different.

**Imperfect Information Games**

- Games in which players make moves without knowing either a) the actions or holdings of other players, or b) the results of external processes (aka *nature* or *chance*).

- **Endogenous Uncertainty** - that related to the actions or holdings (e.g. hole cards, hidden pieces, etc.) of other players. This includes games where players make moves simultaneously, such as rock-scissors-paper and prisoner’s dilemma.

- **Exogenous Uncertainty** - that arising from external stochastic processes, such as the roll of dice, dealing of cards, etc.

- In these games, a player’s **beliefs** about the possible opponent’s states and actions, as well as possible outcomes of chance events, are crucial in determining behavior.

- Thus, a player’s strategy will normally a) account for probability distributions over endogenous states and exogenous events, and b) constitute a probability distribution over possible actions for that player.
**Game Theory: Strategies**

- **Pure** - For each game situation, a pure strategy has ONE particular action. Deterministic mapping from states to actions.
- **Mixed** - Probability distribution over complete, pure strategies.
- **Behavioral** - Probability distributions over actions at each game state.
Key Features of a Normal-Form Game

- A set of two or more players.
- A set of legal actions for each player.
- A set of payoffs: a mapping from each combination of player actions to rewards for each player.

<table>
<thead>
<tr>
<th>Player 1 Actions</th>
<th>Player 2 Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>(3,3)</td>
</tr>
<tr>
<td>Y</td>
<td>(4,0)</td>
</tr>
<tr>
<td>Z</td>
<td>(3,0)</td>
</tr>
</tbody>
</table>

- Cell entries / pairs = payoffs to players 1 and 2.

- **Best-Response (BR) action** - bold font - $BR_i(k) =$ highest-paying action choice for player $i$ when opponent chooses action $k$.
  - $BR_1(X) = Y; BR_1(Y) = Z; BR_1(Z) = Z$
  - $BR_2(X) = Y; BR_2(Y) = Z; BR_2(Z) = Z$

- **Nash Equilibrium** - blue font - a pair of actions, $k_1$ and $k_2$, where $BR_1(k_2) = k_1$ and $BR_2(k_1) = k_2$, i.e., $BR_1(BR_2(k_1)) = k_1$. Intuitively: a situation where neither player can choose a better action to counter that of its opponent.
Game Theory: Pareto Optima -vs- Nash Equilibria

- Define a **strategy profile** as a set of strategies \( (s) \), one for each player, e.g. \( s = (Y,X) \) in the game above.

- Let \( v_1(s) \) and \( v_2(s) \) be the payoffs for players 1 and 2 for profile \( s \).

- Profile \( s \) **pareto dominates** \( s' \) when \( \forall i : v_i(s) \geq v_i(s') \) and \( \exists k \ni v_k(s) > v_k(s') \). I.e., payoffs for each player are at least as good with \( s \) as \( s' \), and at least one player does better with \( s \) than with \( s' \).

- A **pareto optimal** profile is one that is not pareto dominated by any other profile. Note: This does not mean that it must pareto dominate all other profiles.

- In game above, there are 7 pareto optimal payoffs: \( 2 \times (0,4) \), \( 2 \times (4,0) \), \( 3 \times (3,3) \), but each only dominates one other payoff: \( (0,3) \) or \( (3,0) \).

- Deciding pareto optimality involves comparing a profile’s payoffs to those of every other profile, i.e., all cells in the table.

- Detecting Nash Equilibrium involves comparing all profile payoffs in one row and one column.

- Nash equilibria are **stable action profiles**, since if a) both players begin to use their strategy in the profile, and b) each believes that the other will not change strategies, then they too will not change (since they are currently playing the best response to the opponent).

- Pareto optimal solutions cannot guarantee that type of stability.
Prisoner’s Dilemma

<table>
<thead>
<tr>
<th>Player 1 Actions</th>
<th>Player 2 Actions</th>
<th>Cooperate</th>
<th>Defect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cooperate</td>
<td>(-1, -1)</td>
<td>(-4, 0)</td>
<td></td>
</tr>
<tr>
<td>Defect</td>
<td>(0, -4)</td>
<td>(-2, -2)</td>
<td></td>
</tr>
</tbody>
</table>

- **Nash Equilibrium (blue)** - Both defect. It’s the best-response strategy for each player: if P1 defects, P2’s best response is defection, and if P2 defects, P1’s best response is defection. It is **not** a pareto optimum, since it is pareto dominated by cooperate-cooperate.

- **Pareto Optimum (red)** - Both cooperate or exactly one defects - No strategy profile gives a better result for one player and a no-worse result for the other ⇔ none of them are pareto-dominated by any other profile. The single-defector profiles are a best response for the defector, but not for the cooperator.
Each information set is a context within which a player chooses actions.

Each player ($P_i$) has a collection of information sets: $h \in H_i$ with properties:

1. Each $h$ contains one or more game states.
2. If $h$ contains exactly one game state, then $P_i$ knows the exact game state when in $h$; otherwise it does not.
3. $P_i$ will choose the same action (or have the same probability distribution over actions) for every state in $h$.

- In perfect information games, all information sets are singletons (i.e. they house a single game state).
- Endogenous and exogenous uncertainty in imperfect information games create many non-singleton information sets.
Information Set = Context for Action Choice

Context

Future External Events
- Determinism, Predictability
- Uncertainty

State of Game Environment
- Observable Conditions
  - Uncertainty
  - Hidden parts of a game board

Opponent's Actions
- Observable Acts
  - Simultaneous moves
- Uncertainty

Opponent's Holdings
- Observable Holdings
  - Hole cards
  - Uncertainty

Dealt cards, rolled dice

Behavioral Strategy

More uncertainty =>
Larger information sets =>
Larger set of possible game states

Keith L. Downing
Imperfect Information Games: Poker
Information Sets in Auctions

- Open - Each player knows the other’s recent bid before making its own bid.
- Closed - Each player bids without knowing the bids of others. Hence, the information sets contain many possible states.
- Note: In this domain, a state ≈ a bid, i.e. the action.

Keith L. Downing | Imperfect Information Games: Poker
A Simple Closed Auction

**Rules**

- Each player places a particular value on the object to be auctioned off.
- Each player makes a closed bet of 5, 10 or 20 units.
- The highest bidder gets the object.
- In case of ties, a coin flip determines the winner.

Assume: Current object valued at 15 by P1, and 25 by P2.

Then, if P1 (P2) wins with a bid of B, the value of the auction itself is 15 - B (25 - B). E.g. If P2 wins with a bid of 5, she gains 25 - 5 = 20 units.

The payoff matrices, $M_1$ and $M_2$, for players 1 and 2 reflect these values (15 and 25).
Payoff Matrices (aka Utility Matrices)

Each cell entry = the value of the auction itself to player 1 (or 2) given a) the value placed on the object, and b) the pair of bids (actions).

\[
M_1 = \begin{array}{c|ccc}
\text{Acts (Bids)} & 5 & 10 & 20 \\
\hline
5 & 5 & 0 & 0 \\
10 & 5 & 2.5 & 0 \\
20 & -5 & -5 & -2.5 \\
\end{array}
\]

\[
M_2 = \begin{array}{c|ccc}
\text{Acts (Bids)} & 5 & 10 & 20 \\
\hline
5 & 10 & 15 & 5 \\
10 & 0 & 7.5 & 5 \\
20 & 0 & 0 & 2.5 \\
\end{array}
\]

- Player 2’s evaluations of the auction are higher, since P2 places a higher value (25) on the auctioned object than does P1 (15).
- Diagonal entries are \((0.5)(0) + (0.5)(\text{Value} - \text{Bid})\), due to 0.5 chance of winning the coin flip.
The closed auction creates non-singleton information sets.

So each player does not know the other player’s action when they choose their own action.

However, after observing the opponent in similar situations, each will have an idea of how they act.

That idea = belief = probability distribution over acts.

In this game, the distribution over one player’s actions = distribution over the states that the other player will experience.

Distributions over hidden states are called ranges.

\( R_1 \) = range of P1’s actions = range of hidden states that P2 will experience.

\( R_2 \) = range of P2’s actions = range of hidden states that P1 will experience.
Range x Payoffs = Values

Multiplying a payoff matrix by a range (assume it’s a row vector) produces a vector \( V_i \) of expected values for each action.

- \( V_2 = R_1 \cdot M_2 \) = Expected payoffs of each P2 action based on P1’s range.
- \( V_1 = M_1 \cdot R_2^T \) = Expected payoffs of each P1 action based on P2’s range.

By observing P1’s play, P2 estimates P1’s range as: \( R_1 = [0.2, 0.5, 0.3] \)

Then, \( V_2 = R_1 \cdot M_2 = [2.0, 7.5, 4.25] = [\text{expected payoff for P2 if P2 bids 5, expected payoff for P2 if P2 bids 10, expected payoff for P2 if P2 bids 20}] \)

This indicates to P2 that, given P1’s behavior, a bid of 10 by P2 is likely to yield the best payoff.

Thus, P2 will begin to bid 10 more often. P1 will notice this and adjust the estimate of P2’s range to: \( R_2 = [0.1, 0.7, 0.2] \)

Then, \( V_1 = M_1 \cdot R_2 = [0.5, 2.25, -4.5]^T = [\text{expected payoff for P1 if P1 bids 5, expected payoff for P1 if P1 bids 10, expected payoff for P1 if P1 bids 20}]^T \)
Adapting Behavioral Strategies

Ranges, values and behavioral strategies interact and evolve.

Generate Payoff Matrices, $M_1$ and $M_2$

- **P2** updates estimate of $R_1$ based on **P1**'s behavior
- **P2** changes own behavioral strategy based on $V_2$
- **P2** updates the values of its own actions ($V_2$), using $R_1$ and $M_2$
- **P1** updates estimate of $R_2$ based on **P2**'s behavior
- **P1** changes own behavioral strategy based on $V_1$
- **P1** updates the values of its own actions ($V_1$), using $R_2$ and $M_1$

Adaptive Auction Loop

This same generic loop applies to many games.
Four-Card Gambit

- The deck contains 4 cards: 2 Aces and 2 Kings.
- Players alternate being P1 and P2.
- Initially, 1 hole card is dealt to each player. Each looks at their hole card.
- P1 decides whether to fold (giving 2 points to P2) or continue play.
- If play continues, then the single public card is dealt and displayed.
- Player 2 then decides whether to fold or call.
- If P2 folds, P1 gets 2 points, regardless of hole cards.
- If P2 calls, a showdown determines the winner.
- Winning with 2 Kings gives 2 points; with 2 Aces, 4 points.
- Ties give 1 point to each player. The only possible tie is (A,K) for each player.
The two chance nodes represent the same information set for P2, since P2 does not know P1’s hole card. This is endogenous uncertainty. Conversely, exogenous uncertainty concerns the public cards to be dealt, i.e. information that will enter from outside.
Exogenous uncertainty still handled by chance nodes with multiple branches, but
Endogenous uncertainty incorporated in ranges, hence no branching based on
assumptions about hidden states.
Normal Form Representation for 4-Card Gambit

<table>
<thead>
<tr>
<th>Player 1</th>
<th>P(K)K</th>
<th>P(K)A</th>
<th>P(A)K</th>
<th>P(A)A</th>
<th>F(K)K</th>
<th>F(K)A</th>
<th>F(A)K</th>
<th>F(A)A</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK</td>
<td></td>
<td>(2,0)</td>
<td>(1,1)</td>
<td>(0,4)</td>
<td>(2,0)</td>
<td>(2,0)</td>
<td>(2,0)</td>
<td>(2,0)</td>
</tr>
<tr>
<td>PA</td>
<td>(0,2)</td>
<td>(1,1)</td>
<td>(4,0)</td>
<td></td>
<td>(2,0)</td>
<td>(2,0)</td>
<td>(2,0)</td>
<td></td>
</tr>
<tr>
<td>FK</td>
<td></td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td></td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>FA</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td></td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td></td>
</tr>
</tbody>
</table>

- P = play, F = fold, K = king, A = ace,
- Single card in parentheses = public card.
- Cell contents = (P1-payoff, P2-payoff)
- Each strategy choice = action + info available to player when choosing that action. E.g. PK for P1 = play (don’t fold) given that P1’s hole card is a King; also, F(A)K for P2 = fold given that the public card is Ace and P2’s hole card is King.
- Bold payoff value = **best-response (BR) action**. Given an opponent’s state and/or action, the BR is Player P’s action (among all possible actions) that yields the highest payoff for P.
- For P1 (P2), a BR cell will have the highest P1 (P2) payoff in that column (row).
- Blue pair = best responses for both players = **Nash Equilibrium**.
Public price: Player 1 must pay 6 to continue play, i.e. to buy the public card. The fee is returned to P1 if and only if P1 wins the hand, either by showdown or by P2 folding.

### Player 2

<table>
<thead>
<tr>
<th>Player 1</th>
<th>P(K)K</th>
<th>P(K)A</th>
<th>P(A)K</th>
<th>P(A)A</th>
<th>F(K)K</th>
<th>F(K)A</th>
<th>F(A)K</th>
<th>F(A)A</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK</td>
<td>–</td>
<td>(2,0)</td>
<td>(-5,1)</td>
<td>(-6, 4)</td>
<td>(2,0)</td>
<td>(2,0)</td>
<td>(2,0)</td>
<td>(2,0)</td>
</tr>
<tr>
<td>PA</td>
<td>(0,2)</td>
<td>(-5,1)</td>
<td>(4,0)</td>
<td>–</td>
<td>(2,0)</td>
<td>(2,0)</td>
<td>(2,0)</td>
<td>–</td>
</tr>
<tr>
<td>FK</td>
<td>–</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>–</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>FA</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>–</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>(0,2)</td>
<td>–</td>
</tr>
</tbody>
</table>

- Cell PK - P(A)A (orange) is no longer a Nash Equilibrium.
- P1 now finds that playing with a hole King gives a worse average payoff than simply folding.
- Hence, the PK-P(A)A combination occurs less often than in the original 4-card gambit version.
- The P1 and P2 strategies now yield the remaining 3 Nash equilibria (blue) more often.
A Nash Equilibrium is a situation where no player can benefit by changing their strategy, under the assumption that all opponent strategies remain static. Hence, any change by player P will be deleterious (for P) if all other players continue playing the same strategy.

In 4-card gambit, repeated play and rational behavior based on the payoffs achieved will lead players to choose strategies that, together, yield Nash equilibria.

E.g. When P1 has a hole-card King, it will find that playing generally leads to a better result than folding, so it will play more than fold (in that situation). Similarly, when P2 is in state (A)A, it will find that playing yields better payoffs than folding. Hence, when P1 has hole K and P2 has (A)A, they are both likely to play, and thus the payoff will be (0,4): one of the Nash equilibria.

A Nash Equilibrium Strategy is one in which the best responses in all contexts lead to Nash equilibrium payoffs.

More generally, **solving** poker entails finding a Nash Equilibrium Strategy for it.
DeepStack combines deep learning (DL) and self-play to achieve expert-level performance. The approach is very similar to deep reinforcement learning (DRL). It computes Nash-Equilibrium strategies for the states that arise in poker games via resolving. So although it does not solve Poker, it does solve for individual poker states by finding minimally-exploitable strategies.

It has a strong mathematical foundation. It was the first theoretically-sound heuristic approach to imperfect-information games. Below is a quick summary of that mathematics.

Nash Equilibria for Heads-Up (2-player) Poker

Let $\Sigma_1$, $\Sigma_2$ be the sets of possible strategies for P1 and P2, respectively.

A strategy profile ($\sigma$) is a combination of strategies for P1 and P2: $\sigma = [\sigma_1, \sigma_2]$, where $\sigma_i \in \Sigma_i$ for $i \in \{1, 2\}$.

Denote the expected payoff (aka utility) to player i under profile $\sigma$ as $u_i(\sigma)$. This utility is an average payoff of many rounds of poker.

A Nash Equilibrium is a strategy profile, $\sigma^* = [\sigma_1^*, \sigma_2^*]$ where:

\[ u_1(\sigma^*) \geq \max_{\sigma'_1 \in \Sigma_1} u_1([\sigma'_1, \sigma_2^*]) \quad \text{and} \quad u_2(\sigma^*) \geq \max_{\sigma'_2 \in \Sigma_2} u_2([\sigma_1^*, \sigma'_2]) \]

That is: $\sigma_1^*$ is the best response to $\sigma_2^*$, and vice versa.

An $\varepsilon$-Nash Equilibrium is a strategy profile, $\sigma^\diamond = [\sigma_1^\diamond, \sigma_2^\diamond]$ where ($\varepsilon \geq 0$):

\[ u_1(\sigma^\diamond) + \varepsilon \geq \max_{\sigma'_1 \in \Sigma_1} u_1([\sigma'_1, \sigma_2^\diamond]) \quad \text{and} \quad u_2(\sigma^\diamond) + \varepsilon \geq \max_{\sigma'_2 \in \Sigma_2} u_2([\sigma_1^\diamond, \sigma'_2]) \]

That is: $\sigma_1^\diamond$ is within $\varepsilon$ of the best response to $\sigma_2^\diamond$, and vice versa.
Regret and Counterfactual Utility

- Regret is the difference in payoff (utility) between using a given strategy and using the optimal strategy.
- Denote player i’s strategy at time t as $\sigma_i^t$ and the opponent’s strategy as $\sigma_{-i}^t$.
- Then the average overall regret of player i at time T is:

$$R_i^T = \frac{1}{T} \max_{\sigma_i' \in \Sigma_i} \sum_{t=1}^{T} u_i([\sigma_i', \sigma_{-i}^t]) - u_i([\sigma_i^t, \sigma_{-i}^t])$$

- Note that the $\sigma_i' \in \Sigma_i$ that maximizes the above summation is considered optimal, and the payoff for $\sigma_i^t$ is compared to that optimum at every timestep.

- Define counterfactual utility $u_i(\sigma, \Upsilon)$ as the average payoff that player i would receive if the game went through information set $\Upsilon$ and under strategy profile $\sigma$.

- The word counterfactual now creeps into the analysis since we are basing regret calculations on what if scenarios: what if we reached this information set under this strategy profile? What would the regret be then?

- $\sigma$ is actually a profile in which player i attempts to reach $\Upsilon$, while its opponent has no such bias. This is a technical detail that only adds unnecessary complexity to our current analysis, but it appears in the more formal (theorem - proof) discussions of Nash Equilibria in imperfect information games.
Immediate Counterfactual Regret

Let \( \sigma_{\gamma \rightarrow a} \) be a strategy profile identical to \( \sigma \) except that player i always plays action \( a \) \((a \in A(\gamma))\) in the context of information set \( \gamma \); \( A(\gamma) \) = the legal actions for information set \( \gamma \).

Immediate counterfactual regret, \( R_i^T(\gamma, a) \), is the regret associated with an information set and an action:

\[
R_i^T(\gamma, a) = \frac{1}{T} \sum_{t=1}^{T} \Pi_{\sigma_t}(\gamma)[u_i(\sigma_{\gamma \rightarrow a}^t, \gamma) - u_i(\sigma^t, \gamma)]
\]

where \( \Pi_{\sigma_t}(\gamma) \) = probability of reaching info set \( \gamma \) via the strategy used at time \( t \).

For minimizing regret, the only regret of interest is the positive, so define the positive immediate counterfactual regret as \( R_i^{T+}(\gamma, a) \):

\[
R_i^{T+}(\gamma, a) = \max(0, R_i^T(\gamma, a))
\]

Define the positive regret at a particular information set, \( R_i^{T+}(\gamma) \) as:

\[
R_i^{T+}(\gamma) = \max_{a' \in A(\gamma)} R_i^{T+}(\gamma, a')
\]
**The key result** (of several theorems and proofs):

\[ R_i^T \leq \sum_{\gamma \in \Gamma^*} R_{i^+}(\gamma) \]  

⇒ Overall regret is no larger than the sum of the positive immediate counterfactual regrets. So by reducing the latter, we reduce the former.

Define \( \bar{\sigma}^T \) as the average strategy profile over \( T \) timesteps. Then, another important game-theory theorem states:

*At time \( T \) in a zero-sum, two-player game, if each player \( i \) has \( R_i^T \leq \varepsilon \), then \( \bar{\sigma}^T \) is a \( 2\varepsilon \)-Nash equilibrium profile.*

→ By minimizing regret locally, we minimize overall regret, and that yields a Nash equilibrium profile, which is just the pair of average strategies for each player over the entire training period, i.e. the period during which games are played and the strategies are modified to reduce local regret.
So how do we minimize regret locally? It’s pretty easy!

Each player node (i.e. node where a player chooses an action) in the search tree represents an information set, $\Upsilon$.

And each such node houses a strategy, $\sigma$, which is a probability distribution over all legal actions.

To minimize regret at each node, simply modify the strategy to give higher probabilities to the actions with the highest positive regret:

$$\sigma_{i}^{T+1}(\Upsilon, a) = \frac{R_{i}^{T+1}(\Upsilon, a)}{\sum_{a' \in A(\Upsilon)} R_{i}^{T+1}(\Upsilon, a')}$$

Key Conclusion: By modifying local node strategies to reduce regret, an adaptive poker agent can attain a $2\varepsilon$-Nash equilibrium profile for vanishingly small $\varepsilon$, given a large enough $T$.

Now that we’ve covered the basics of game theory and the connections between information sets, Nash equilibria, and regret minimization, we can look at the details of poker players such as DeepStack and see how they can solve poker (at least in a theoretical sense), and, in fact, play very well.
Texas Holdem: The Pre-Flop Stage

Betting rules can vary for this stage, especially in 2-person ("Heads up") versus 3-or-more-player games.

Keith L. Downing
Imperfect Information Games: Poker
Texas Holdem: The Flop Stage

Flop

Turn

River

Player 1

Player 3

?

Public Cards

?

Player 2

Player 4

Dealer

Action Options:
Fold, Call (Check),
Raise, All-In

Check = A call when
no money has yet been
bet in the current stage.

Action Options:
Fold, Call (Check),
Raise, All-In

Action Options:
Fold, Call (Check),
Raise, All-In

Action Options:
Fold, Call (Check),
Raise, All-In

Action Options:
Fold, Call (Check),
Raise, All-In

Betting Rotation
(Clockwise)
Texas Holdem: The Turn Stage

Player 1

Player 3

Public Cards

Player 2

Dealer

Action Options:
Fold, Call (Check), Raise, All-In

Action Options:
Fold, Call (Check), Raise, All-In

Action Options:
Fold, Call (Check), Raise, All-In

Betting Rotation (Clockwise)

Keith L. Downing

Imperfect Information Games: Poker
Texas Hold'em: The River Stage

Flop
Turn
River

Player 1

Action Options:
Fold, Call (Check),
Raise, All-In

Player 2

Action Options:
Fold, Call (Check),
Raise, All-In

Player 3

Action Options:
Fold, Call (Check),
Raise, All-In

Player 4

Action Options:
Fold, Call (Check),
Raise, All-In

Public Cards

Flop

Turn

River

Betting Rotation (Clockwise)

Dealer

Keith L. Downing

Imperfect Information Games: Poker
To remain in a stage (and thus in the hand), a player needs to contribute the same amount to the pot as all the other players.

During any stage, if all players but one have folded, the non-folding player wins.

As long as two or more players are active (not folded), play progresses through all stages (pre-flop, flop, turn and river).

If two or more players remain active at the end of the river stage, a showdown occurs: all active players show their hole cards and make the best 5-card hand out of their 7 cards.

The highest ranking 5-card hands win.

The winner of the hand gets all the chips in the pot. Multiple winners divide the pot evenly.

Each player’s pile of chips is public: visible to all players.
Monte Carlo Methods for Poker

- A Monte Carlo method is one that uses repeated sampling of events to generate a probability distribution over a different (often higher level) set of events.

- In poker, one example is repeated random deals of cards to get the probabilities that particular card collections (e.g. a pair of 10’s) will form the best 5-card hand among a group of players.

- In chess, to assess the current board situation, perform many (somewhat) random simulations from that state to final states to get the probability of each player winning. AI Chess players have traditionally not done this, since they have good heuristics to evaluate board states. GO, however, had few good heuristics and thus invited sims earlier.

- These simulations are called rollouts, and they may be totally random or based on a stochastic policy / strategy.

- Basing rollouts on a purely deterministic policy is pointless, since each simulation would produce the same result (unless the simulated environment has non-determinism as well).

- In poker, many types of rollouts are possible. Some account for (stochastic) player actions (fold, call, raise, etc.), while others simply deal and re-deal cards, often thousands of times.
Rollout Hand Evaluation: Pre-Flop

Player 1

Private Cards

Player 2

Private Cards

Public Cards

Flop

Turn

River

Several thousand rollout deals

\[ \binom{50}{7} = 99,884,400 \]
Rollout Hand Evaluation: Flop

Player 2

Private Cards

Player 1

Public Cards

Flop

Turn

River

Several thousand rollout deals

\( \binom{47}{4} = 178,365 \)
Rollout Hand Evaluation: Turn

Player 2

Private Cards

Player 1

Public Cards

Flop

Turn

River

Several thousand rollout deals

$\binom{46}{3} = 15,180$
Rollout Hand Evaluation: River

Player 2

Private Cards

\[\begin{array}{c}
\text{A}\spadesuit \\
\text{Q}\heartsuit
\end{array}\]

Public Cards

\[\begin{array}{c}
\text{4}\spadesuit \\
\text{5}\heartsuit \\
\text{J}\spadesuit \\
\text{K}\spadesuit \\
\text{10}\spadesuit
\end{array}\]

Flop

Turn

River

Private Cards

\[\begin{array}{c}
? \\
? \\
\end{array}\]

Player 1

\[\binom{45}{2} = 990\]

Try all 990 remaining possible pairs
Generating A Poker Cheat Sheet

Use many pre-flop rollouts with varying number of opponents to get the winning probabilities of different hole pairs for different sized games. There are 169 different types of pairs for a normal 52-card deck.

<table>
<thead>
<tr>
<th>Hole Cards</th>
<th>2 players</th>
<th>3 players</th>
<th>4 players</th>
<th>5 players</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ace, Ace</td>
<td>0.86</td>
<td>0.74</td>
<td>0.64</td>
<td>0.54</td>
</tr>
<tr>
<td>Ace, King Unsuited</td>
<td>0.65</td>
<td>0.48</td>
<td>0.38</td>
<td>0.33</td>
</tr>
<tr>
<td>7, 8 Suited</td>
<td>0.47</td>
<td>0.33</td>
<td>0.27</td>
<td>0.23</td>
</tr>
<tr>
<td>2, 7 Unsuited</td>
<td>0.35</td>
<td>0.21</td>
<td>0.14</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Rollout Process (one of several approaches)

1. Choose any instance of an equivalence class. E.g. for class (Q, 4, unsuited) choose (Q ♦, 4♠) as hole cards.
2. Do thousands of rollouts to the remaining 5 (flop, turn, river) public cards and K-1 opponent hole pairs (for a K-person game).
3. Count the fraction of rollouts for which (Q ♦, 4♠) produces the highest-ranked 5-card hand.
4. This gives the winning probability for (Q, 4 unsuited) in a K-person game.
5. Repeat for all 169 equivalence classes.
Strategies and Rollouts

- A strategy is a mapping from a game state to a probability distribution over possible actions.
- Rollouts are important tools for building strategies, particularly for stochastic domains.
- In poker, the simple repeated-dealing rollouts provide basic probabilities that a player’s hand will win in a showdown, and those probabilities can strongly affect the probability distribution over a player’s actions (fold, call, raise, etc.).
- However, the repeated-dealing rollouts do not take players’ conditional actions into consideration.
- But to give more accurate probabilities of success, they should.
- Note that a player’s conditional actions are their strategy.
- So to build a good strategy, we need rollouts based on a strategy.

=> We need to bootstrap the strategies: iteratively refine strategies based on previous strategies.
Each rollout may try SOME (or ALL) of the possible player actions.

Each rollout may generate SOME (or ALL) of the possible chance events.

Each rollout should only end at nodes that can provide a quantitative evaluation of the root player’s winning probability.

Nodes where rollouts can end include:
1) actual game-ending states (showdowns and all-but-one-folds),
2) nodes containing some sort of function approximator, such as a neural network critic, that maps states to evaluations.
In some games, like poker, the complete set of legal actions at any tree node is **not** dependent upon the **hidden** information. **Bad cards do not prevent betting.**

The action set only depends upon the public information: public cards, pot size, each player’s chips, etc. **A player’s low chips can constrain their betting.**

Thus, the entire search tree can be generated **without** considering the endogenous uncertainty: private / hidden information such as hole cards in poker.

Limit tree size by limiting action set and number of raises per stage.

Each rollout can then involve random choices (aka **samples**) of, or ranges (i.e. probability distributions) over, the private information, which are passed through the same search tree, but returning different results to the root on each rollout.
The strategy can be updated for every state represented by a node in the search tree.
The Context for Each Node’s Strategy

The public context combines with each hole pair to yield the complete context for each probability distribution. Each node in tree has a different public context, so each needs a different strategy.

Keith L. Downing
Imperfect Information Games: Poker
Randomly rollout some private and/or public card information.

Use the evolving strategy to determine the probabilities of taking each action at each player-choice node in the search tree.

If using a prob distribution over private info (instead of a sampled value), then modify the distribution based on actions taken at each node (and Bayes’ Law, as detailed later).

Compute evaluations of each player’s private hand (or prob distribution over private hands) at all leaf nodes of the tree.

Pass these evaluations back to root; modify on the way up.

Use the action probabilities of the current strategy (at each node) as basis for these modifications.

Then use the evaluations recorded at each node to compute the node’s **counterfactual regret** (explained earlier, and later).

At each player node, use counterfactual regret to update the node’s strategy.
Main Sampling Schemes

- Comprehensive (Complete) Sampling
- Self-Public Sampling
- Opponent-Public Sampling
- Public (only) Sampling

* All schemes sample the public information, but they vary as to whether they a) sample particular private info (i.e. hole cards) or b) work with a probability distribution over all possible private info states.

* Player-action sampling (fold, call, raise, etc.) can also be combined with any of these schemes.
Everything is sampled: all private (hole) cards and all public cards.

P2 wins a showdown with a straight flush

P1 wins a showdown with 3 queens

Keith L. Downing
Imperfect Information Games: Poker
Sample hole pairs for focal player (P1) and all public cards.
Opponent-Public Sampling

Sample public cards and opponent’s (P2) hole cards

The expected winnings for Player 2 based on P1’s Range (modified on the way down to the leaf), along with the sample private and public info.

Keith L. Downing

Imperfect Information Games: Poker
Two vectors of expected winnings (aka value vectors) for P1 and P2 based on the ranges of P1 and P2 (modified on the way down to the leaf), along with the sampled public info.

Only sample the public cards. DeepStack does this.
Action Sampling

- Each rollout may involve a complete traversal of each branch of the tree.
- Alternatively, it may involve just one (or a few) choices of children from each parent.
- For a parent chance node, that means just dealing out some of the possible cards.
- For a parent player-action node, choose children based on the aggregate action probabilities, derived from the matrix product of range and strategy.

\[
P1\text{'s Strategy at Node N} = P1\text{'s Range} \times \text{Strategy} = 1 \times H \times A = 1 \times A
\]

\[
P1\text{'s Aggregate Action Probabilities at Node N} = 1 \times A
\]

\[
H = \# \text{Hole Pairs} \quad A = \# \text{Actions}
\]
Modern AI poker players (e.g. DeepStack) find Game Theory Optimal (GTO) strategies.

Even though the AI pokerbot knows its own cards, it calculates with a range, not a single pair.

This reflects knowledge that the opponent could have about AI’s hole cards, based on public cards and the pokerbots recent actions.

So the AI’s strategies are based on what it thinks the opponent knows about the AI’s cards. Beliefs are key!
Computing Expected Winnings from Ranges

Expected Winnings = Evaluation = Value (of the hole pair)
A Player’s Evaluations

Expected winnings when holding each pair, given:
- the current public cards, and
- the RANGE of the opposing player

* Values are typically the size of the final pot scaled by a standard amount, such as the average pot size.

Keith L. Downing
Imperfect Information Games: Poker
*All values (1,-1,0) are from the perspective of player 1.*
Evaluations at an End State

Values in both evaluation vectors are from the perspective of player 1.

Multiply by -1 to change the perspective to player 2.

Normally, we want each evaluation vector to represent the perspective of its player.
* When using a solver, this strategy is generated from scratch for each action taken in the real game: the AI resolves at each decision point. Each player node has a strategy matrix.
* Each player’s total chips are part of the state and can restrict legal actions. E.g. You can’t raise if you only have enough chips to call.
States in the Search Tree

Each state from which a player will act should contain:

- Public cards
- Pot - number of chips in the pot
- Chips - number of chips that each player possesses
- History - of folds, calls, raises and all-in in this hand (or at least this stage)
- Stage - current stage of the hand (optional)
- Depth - depth within the current stage (optional)

ALL of the information in a state is freely available to both players.
Resolving: A Bootstrapping Process

- Start at Root
- Traverse tree many times
- Recurse through entire search tree
- Save root node's strategy after each traversal
- Basis for whole search tree

Strategy

Ranges

Values

Regrets

Strategy Series

Average Strategy

Current Game State: S

Ranges for S

Next Game State: S(a*)

Next Action (a*)

Ranges for S(a*)

Inputs

Outputs

Current Game State: S

Can return as inputs during future calls to re-solve during the same hand of poker

Keith L. Downing

Imperfect Information Games: Poker
Each call to the resolver includes the current state (S) of an actual game (i.e. an episode) and ranges (R1,R2) for each player.

Assume the root player is P1.

The resolver begins by generating a (typically large) subtree rooted at S.

R1 and R2 are the ranges for the root of the subtree.

In traversing the tree, individual nodes will have their own modified versions of R1 and R2, which influence calculations of value vectors, which affect regrets, which affect strategies.

After many subtree traversals, many strategies result.

The average strategy $\Phi$ is then used to choose P1’s **actual action** (a*) for state S.

Bayesian Updating (see below) uses $\Phi$, P1 and a* to modify R1 to R1*. R2 is not changed.

The resolver returns a* along with the range pair: (R1*,R2).

A future call to the resolver in the same episode (i.e., poker *hand*) will use a new state (S’) and (R1*,R2).

This next call uses S’ to generate a subtree, with the root’s ranges = (R1*,R2).
Resolving: A Node’s Perspective

Downward Pass

Ranges
P1
P2

P1 range modified based on strategy, branch action, and Bayes rule

Act-0
Act-1
Act-k

Strategy

For P1’s values, the weights are based on the strategy: the prob of choosing the branch action

Updated Values based on weighted averages of incoming value vectors.

Upward Pass

Values

Strategy modified based on regrets

Regrets

Updated Values

P1
P2

Values

P1
P2

P1
P2

Values

P1
P2

Keith L. Downing

Imperfect Information Games: Poker
Traversing the Search Tree (Downward)

Ranges

Evaluations

(R1, R2)

act(i)

(R1*, R2)

Update R1 (Range player 1) using the Strategy, Bayes Rule and fact that player 1 chose the ith action

(R1*, R2*)

act(k)

(R1**, R2@)

Update R2 (Range player 2) using the Strategy, Bayes Rule and fact that player 2 chose the kth action

(R1**, R2@)

act(j)

Use Utility Matrix and both ranges to compute both evaluations: V1 and V2.
Bayesian Update (of Acting Player’s Range)

- **0.45**
- **0.05**
- **1326 possible pairs**
- **0.03**
- **0.47**
- **0.099**
- **0.001**

<table>
<thead>
<tr>
<th>Actions</th>
<th>Fold</th>
<th>Call</th>
<th>Raise</th>
<th>All In</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.05</td>
<td>0.45</td>
<td>0.47</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>0.33</td>
<td>0.57</td>
<td>0.099</td>
<td>0.001</td>
</tr>
</tbody>
</table>

- **0.57**
- **0.001**
- **0.099**
- **0.33**

**The Strategy Array**

- **prob(raise) = sum of this column / sum of entire array**

- **prob(and | pair) * prob(pair)**

**Bayesian Update**

\[
\text{prob(pair | act)} = \frac{\text{prob(act | pair) * prob(pair)}}{\text{prob(act)}}
\]

- **Updated Range**

**Assume:**

\[
\text{prob(raise)} = 0.15
\]

**Updated Range**

- **0.003**
- **0.011**
- **0.0067**
- **0.002**

**Keith L. Downing | Imperfect Information Games: Poker**
At any fold state, let $v_f = (\text{pot size}) / (\text{average pot size})$. Then fold vectors are: $V_{\text{loser}} = [-v_f, -v_f, 0, \ldots, 0, -v_f]$ and $V_{\text{winner}} = [+v_f, +v_f, 0, \ldots, 0, +v_f]$. In both vectors, the only 0's are for the impossible hole pairs (as determined by the public cards). Note: fold vectors are value vectors.

At any end state that is also a final (but non-folding) state, apply the utility matrix to both ranges to produce evaluations, which are sent back up the tree.

At any stage-ending state that is not a final state (of the hand), either a) apply an NN, b) do multiple chance acts and continue, or c) do multiple chance acts and then apply an NN once for each new chance card at start of next stage.
Use both ranges and either fold vectors or utility matrix as basis for evaluation vectors: \( V_1 \) and \( V_2 \).

To compute evaluations, add the evaluations sent up from the child nodes, but weighted by the probabilities (from the strategy) of doing the action that leads to each child.
Averages weighted by the current strategy (\(\sigma\))

\[\forall p \in [1, 2], \forall k \in [1, 1326] : V^p_k(S^*) = \sum_{i=0}^{3} \sigma[k, a_i] \times V^p_k(S_i)\]

where:

- \(k\) = index in the value (V) and range (R) vectors = index of a hole pair.
- \(\sigma[k, a_i]\) = probability (given by current strategy) of choosing action \(a_i\) when holding the kth hole pair.
- \(V^p_k(S_i)\) = value (for player p) of the kth pair at child node \(S_i\) (the child reached via action \(a_i\))
Chance nodes perform chance events, typically many. In Texas Hold’em, a chance node for starting the turn stage might branch $K$ ways to deal $K$ possible cards, where $K \leq 49$.

The two ranges sent down to the chance node are then passed to each child node, but they must be filtered to account for the turn card on that edge. E.g. Any range pairs involving $Q\heartsuit$ will have probabilities set to 0 for the child state created by the $Q\heartsuit$ turn card.

On the upward pass, a chance node simply averages the evaluations received from each of its children.
Regret = Degree to which a particular action \((a_i)\) from state \(S\) is better (or worse) than the average result over all possible actions taken from \(S\).

\[
\forall k \in [1, 1326], \forall i \in [0, 3] : r^p_k(S, a_i) = V^p_k(S(a_i)) - V^p_k(S)
\]

- \(p\) = player making a move at state \(S\).
- \(S(a_i)\) = state attained when doing action \(a_i\) in state \(S\).
- \(V^p_k(S(a_i))\) and \(V^p_k(S)\) - kth item in the value vector for player \(p\) at states \(S(a_i)\) and \(S\), respectively.
- \(k\) = index of a hole pair in the value vector (V).
- \(r^p_k(S, a_i)\) = regret for player \(p\), if holding pair \(k\) and doing action \(a_i\) in state \(S\).
- These are called counterfactuals, since they ask (for each \(k\)): What if the player had hole pair \(k\)?
At each node of the search tree, store one regret matrix (R), for the player who moves from that node; rows = hole pairs; columns = actions.

After each full traversal of the search tree (down and up), calculate a new regret value \( (r_{k,i}) \) for hole-pair \( k \) and action \( i \); and update \( R \):

\[
R_{k,i} = R_{k,i} + r_{k,i}
\]

In addition, create the positive regret matrix \( (R^+) \) at each node via:

\[
\forall k, i : R^+_{k,i} = \max(0, R_{k,i})
\]

A node’s \( R^+ \) is then used to update the node’s strategy (\( \sigma \)), with one \( \sigma \) entry for each \( R^+ \) entry.

Normalized positive regret = an action probability.
The strategy entry corresponding to each $R^+$ item is:

$$\sigma_{k,i} = \frac{R_{k,i}^+}{\sum_m R_{k,m}^+}$$

which expresses the relative advantage of choosing action $a_i$ when in the state represented by this node and holding hole pair $k$, compared to choosing the other actions; $\sum_m$ is over all actions.

The full strategy consists of the strategies in all nodes, since each node represents a different game context. Within each node, the context is further refined by all of the possible hole pairs.

So the complete tree contains action probability distributions for a wide range of public-information contexts, each of which is further elaborated with all possible hole pairs.

This is still not a complete poker strategy, since the tree includes only a small subset of all public poker situations.

On each iteration, a new strategy is created for each node. The complete series of strategies for the root node is typically saved, with the final root strategy being the average over that root strategy series.
Neural Networks in Resolver Search

P1 Range  Public Cards  Pot  P2 Range

P1 Values  P2 Values

Hidden Layers

Dot Layer  Dot Layer

EP1  P1 Target Values

Addition Layer

EP2  P2 Target Values

E  P1

E  P2

+  +

-  -

0
Neural Network Details

The NN’s purpose: map situations (ranges and public info) to evaluations (value vectors) for each player.

**Layers**
- Inputs: P1, P2 ranges + Encodings of public cards + (Relative) pot size
- Hidden Layers - several (3-5) standard dense layers.
- Value Layers - Derived vectors of values for both players.
- Dot Layers - Compute the dot product of each player’s range and value vectors.
- Addition Layer - This sums the outputs of the two dot products. Since poker is a zero-sum game, the output of this layer should be zero.

The output layers are P1 Values, P2 Values and the Addition Layer. The first two are compared to the target value vectors to yield error terms $E_{P1}$ and $E_{P2}$, while the Addition Layer is implicitly compared to zero to yield the zero-sum error, $E_0$. All three errors contribute to the total loss term used by backpropagation.

Building these networks is easy. Generating enough training data is HARD.
The payoff matrix \( M \):

\[
M = \begin{pmatrix}
0 & 2 & -3 \\
-2 & 0 & 1 \\
3 & -1 & 0
\end{pmatrix}
\]

\( M_{i,j} \) = payoff to P1 when it has hidden state \( i \), while P2 has state \( j \).

\( M \) is assumed to be zero-sum, so the payoffs for P2 are the negative of those for P1 in each cell: \(-M_{i,j}\)

\( M \) is symmetric: \( M_{i,j} = -M_{j,i} \)

- \( r_1 = [0.2, 0.5, 0.3] \) = Range for P1
- \( r_2 = [0.6, 0.35, 0.05] \) = Range for P2
- \( v_1 = M \cdot r_2^T = [0.55, -1.15, 1.45] \) = Evaluations for P1 and P2 (below)
- \( v_2 = -1 \times r_1 \cdot M = [0.1, -0.1, 0.1] \) (The -1 enforces zero-sum conditions)
- \( r_1 \cdot v_1 + r_2 \cdot v_2 = 0 \) (This is the key consequence of zero-sum conditions)
Symmetric Payoffs not Needed

This relationship holds even when the payoff matrix is not symmetric.

\[ M = \begin{pmatrix} 1 & -1 & 2 \\ 2 & 3 & -3 \\ -1 & 2 & -4 \end{pmatrix} \]

- \( r_1 = [0.2, 0.5, 0.3] \)
- \( r_2 = [0.6, 0.35, 0.05] \)
- \( v_1 = M \cdot r_2^T = [0.35, 2.1, -0.1] \)
- \( v_2 = -1 \times r_1 \cdot M = [-0.9, -1.9, 2.3] \)
- \( r_1 \cdot v_1 + r_2 \cdot v_2 = 0 \) (The key zero-sum consequence still holds.)
Generating Training Data - Cheap Method

1. Randomly generate (3, 4 or 5) public cards and a *relative* pot size.
2. Generate random ranges for P1 and P2, filtered with the public cards (i.e. probabilities for all pairs containing a public card are zero)
3. Generate a utility matrix, M, based on the public cards.
4. Multiply ranges by M to produce 2 value vectors.
5. Case: Features = (2 range vectors, public cards and relative pot), Targets = (2 value vectors)

- This approach can be used for the flop, turn or river stage, since each stage has enough public cards to create a 5-card hand (when combined with two hole cards), thus giving a reasonable evaluation of each hole pair.
- However, these evaluations are weak, based only on cards, not player actions.
Generating Training Data - Expensive Method

1. Randomly generate (3, 4 or 5) public cards and an **absolute** pot size.
2. Generate random ranges for P1 and P2, filtered with the public cards (i.e. probabilities for all pairs containing a public card are zero)
3. Run the Resolver using public cards and absolute pot size as basis for the root state. **Costly!!**
4. For each traversal of the tree during resolving, cache the value vectors for the root node. Average these vectors for both players to produce $\tilde{V}_1$ and $\tilde{V}_2$.
5. Case: Features = (2 range vectors, public cards and **relative** pot), Targets = ($\tilde{V}_1$, $\tilde{V}_2$)

- In theory, this can be used for any stage: pre-flop, flop, turn or river. However, since resolution only produces values at game-ending states (using utility matrices), some branches of the tree will extend all the way to the river stage, and the resolving search space becomes huge.
- **Solution:** Bootstrapped neural network training.
- This greatly reduces depth of the resolving trees, and it integrates player actions into evaluations, but it is **still expensive.**
1. Generate random states for start of river stage.
2. Resolve from river start to end-game states.
3. Use returned value vectors as targets.
4. Use these cases to train the river-stage neural network: $NN_{river}$. 
Generate random states for start of **turn** stage.

2. Resolve from turn-stage start to either a) end-game states in turn stage, or b) start states of river phase, where $\text{NN}_{r\text{}iver}$ produces value vectors.

3. Use returned value vectors as targets.

4. Use these cases to train the turn-stage neural network: $\text{NN}_{\text{turn}}$. 
1. Generate random states for start of **flop** stage.
2. Resolve from flop-stage start to either a) end-game states in flop stage, or b) start states of turn phase, where $NN_{\text{turn}}$ produces value vectors.
3. Use returned value vectors as targets.
4. Use these cases to train the flop-stage neural network: $NN_{\text{flop}}$. 
Overview: Training and using Neural Networks

Training NNs

1. Use random river states and resolving to train the river-stage network: $NN_{river}$
2. Use random turn states and resolving (using $NN_{river}$) to train the turn-stage network: $NN_{turn}$
3. Use random flop states and resolving (using $NN_{turn}$) to train the flop-stage network: $NN_{flop}$

Deploying NNs in the Resolver

1. Each call to the Resolver involves a particular state and stage of the game. For each stage, use a different NN.
2. For the pre-flop stage, the hole-card cheat sheet is probably a good enough basis for choosing actions, but you can also resolve to the start of the flop stage and then apply $NN_{flop}$ to produce value vectors.
3. For the flop stage, resolve to the start of the turn phase, then use $NN_{turn}$ to produce value vectors.
4. For the turn stage, resolve to the start of the river phase, then use $NN_{river}$ to produce value vectors.
5. For the river stage, resolve to end-game states.
Coarse System Overview

Poker Oracle
- Classify Poker Hands
- Gen Poker Cheat Sheet
- Gen Utility Matrix
- Evaluate Showdowns
- Basic Rollout Hole-Pair Evaluation

Solver
- Gen Subtrees
- Update Ranges (Bayesian)
- Update Regrets
- Update Strategies
- Gen Neural Networks (NNs)
- Train NNs
- Resolve via Rollouts

Create Poker Game Manager
- Gen Poker Players
- Run Poker Games

Poker State Manager
- Gen Poker (Root and Child) Search States

Deck of Cards

Evaluations

Ranges

Real Actions

Hypothetical Actions

Game Situations

Poker Player
Hand Classifier (not described above) - Given a set of 5, 6 or 7 cards, determine the best 5-card hand, for example *full house*, *aces and 8’s*, or *pair of queens, with jack, 10 and 4*. The extra cards, e.g. jack 10 and 4, are needed as tie-breakers in case an opponent has a similar hand. You can find these by searching for ”poker-hand ratings” online. The classifications produced must include the 10 primary hand types and enable a comparison of any two card sets to determine winner or tie.

Rollout Hole-Pair Evaluator - Given a hole pair (h), 0-5 public cards, K opponents, and rollout count N, compute the winning probability of h based on N rollout deals of hole cards to K opponents.

Poker Cheat-Sheet Generator - Using the hole-pair evaluator, produce a table $T_{j,k}$ where entry $T_{j,k}$ = the probability of winning with the jth hole-pair type in a game with k opponents. For a standard card deck, there are 1326 possible hole pairs but only 169 hole-pair types, such as *(pair of 7’s)*, *(queen, 10 same suit)*, *(queen, 10 unsuited)*, etc.

Utility Matrix Generator - Given 3, 4 or 5 public cards, produce an $H \times H$ matrix (where $H =$ number of possible hole pairs) where entry $H_{j,k}$ is either 1, -1 or 0 indicating that hole-pair j wins, loses or ties hole-pair k, respectively. Two hole pairs that are not possible (either because they overlap one another or the public cards) also have entries of 0. Multiplying entries by -1 to convert perspective from P1’s to P2’s.
Poker Game Manager: Main Methods

- **Generate Poker Agents** - An agent may use pure rollouts as the basis of action choice, or a resolver, or a combination of both. Resolving is only relevant for a 2-agent game. This can also generate a human-player object to serve as an interface between a person and the game manager.

- **Manage Game** - Keep track of a card deck, public cards, the pot, etc. for each game.

- **Monitor Players** - Run all player actions through the manager (optional).

- **Texas Hold’em Simulator** - A high-level module that accepts K poker agents and simulates G games.
Poker State Manager: Main Methods

- **Generate Root States** - Given important information about the current game, from the Poker Game Manager, this produces a state object to be used in the root node of a resolver subtree.

- **Generate Child State** - Given an action and a state from the node of a resolver subtree, this produces the child state.

- **Generate Child States** - Given a state from the node of a resolver subtree, this produces a list of child states, based on the set of actions that can legally be applied to the parent state.

- The State Manager needs to handle **player** states and **chance** states differently.
- **Bayesian Range Updater** - Given a range (prob. dist over all hole pairs), a node’s strategy, and the action (A) taken from that node, produce an updated range (to be passed to the child node associated with action A).

- **Regret Updater** - Each search-tree node (for an acting player, not for a chance event) should house a regret matrix (R), where $R_{j,k}$ = the cumulative regret (across all rollouts) associated with hole-pair j and action k. This module uses the evaluation vectors returned from each child node to update R. The positive regret matrix ($R^+$) is easily computed from R (as described earlier).

- **Strategy Updater** - Given $R^+$ for a search-tree node, compute a new strategy for that node (as described earlier).
Subtree Generator - Given the current (public) state of a two-player Texas Hold’em Game (pot size, public cards (PC), chips for each player), generate a complete search subtree.

The subtree should contain all reachable states using all legal actions but constrained by any reasonable assumptions, such as fixed-size bets, maximum number of raises (e.g. 4 per stage), number of stages to span (e.g. start of flop stage to start of turn stage).

Based on the original public cards plus any new ones added by chance nodes, this module should also be able to generate utility matrices (by calling the Poker Oracle) for all end-state nodes (i.e. those where a showdown is needed to determine a winner).

Generate Neural Networks for different game stages, and incorporating the zero-sum condition.

Train Neural Networks using datasets attained by either the cheap or expensive (bootstrapping) method described above.
Action choice depends critically upon the evaluation (i.e., win probability) of a hole pair at each stage of the game (pre-flop, flop, turn, and river). However, this does not mean that a rollout player cannot bluff!
A Hybrid Agent could use the resolver with prob $p$, and rollouts with $1-p$.

Resolving that crosses a stage border, e.g. from Flop to $NN_{\text{turn}}$, must account for the new public card(s) (e.g. the turn card) by including chance nodes in the resolver subtree and producing many children of each chance node, with each corresponding to a different public card.