Combining Methods in AI for Imperfect Information Games: An Autonomous Agent for Six-Player Poker

Submitted by Spencer Eick to the faculty of University of London, Goldsmiths in partial fulfilment of the requirements for the degree of BSc Computer Science.

Abstract

For the past generation, poker has been a popular testbed for AI research involving imperfect information. While significant progress has been made in recent years, the amount of computation time required for state-of-the-art methods makes them challenging to implement in a competitive online real-money environment where an agent can arguably be best evaluated. Here a simpler time-efficient alternative approach for six-player no-limit holdem poker is presented, borrowing from both newer and older methods. Also discussed is the topic of how the game state may be read from external software such as a commercial poker client. An evaluation of the application's critical components and overall performance is then presented.

1 Introduction

Many real-world problems resemble games of imperfect information. Consider, for example, a negotiation involving multiple parties where each party is attempting to exert leverage and persuade the other parties to compromise. The objective of any game, negotiation included, is to behave optimally with the goal of maximising one's utility. However, factors such as what the other parties know or do not know, as well as personality traits which may or may not be known, are often of great importance when determining an optimal strategy for negotiation. A game such as this can be very complicated and difficult to model indeed. That is why many researchers throughout the years have focused their attention on the development of autonomous agents that can play games such as chess, and more recently video games and poker, as a means of benchmarking the success of artificial intelligence and gaining new insight into how to extend the techniques to real-world applications including elections, security interactions, and financial markets.

Poker itself has been compared to "policy decisions in commercial enterprises and in political campaigns" [1]. Aside from imperfect information, there are several aspects to the game that make it an interesting topic of research, such as opponent modelling and management of risk. There is also a significant deceptive element to the game in the form of wagering with a weak hand (i.e. *bluffing*) or passing one's turn (i.e. *checking*) with a strong hand. Early approaches focused on modelling opponents along with computing best-response actions through formulae determined by those with expert domain knowledge of the game [3, 4]. In the past decade however there has been a divergence from exploitation-based opponent modelling to game theoretic approaches focused on minimising the exploitability of the agent itself rather than attempting to exploit opponents. This work uses both types of approaches depending on the situation.

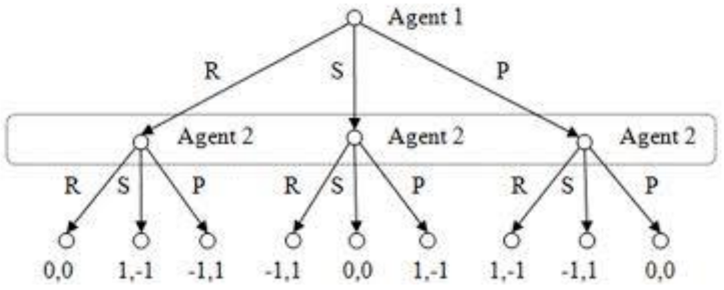
Whilst recent state-of-the-art techniques have shown remarkable success, they have not strictly done so under the conditions imposed by a typical online real-money setting with short time limits for decisions and varying chip stack sizes. Moreover, the evaluation approach of paying a group of top professional human players to incentivise them to perform their best is unsuitable to research projects with limited resources. This work therefore aims to develop an agent capable of competing in a typical online micro-stakes real-money game where it can be evaluated against a large player pool of human opponents.

2 Overview of Concepts and Related Work

"The analysis of a more realistic poker game than our very simple model should be quite an interesting affair." -John Forbes Nash, 1951

2.1 The Challenge of Large Imperfect Information Games

Unlike perfect information games such as chess and go, the states in an imperfect information game are not well-defined. A state-of-the-art approach to solving chess would involve real-time depth-limited solving of a sub-section of the game-tree, using values at leaf nodes that are estimates of the expected value of being in that state of the game tree assuming that both players will play optimally going forward [2]. As per Noam Brown et al in their 2018 paper [5], this does not work for imperfect information games. To understand why, we need only consider the simple game of rock-paper-scissors (RPS), for which the extensive form can be seen below [6]:



The rectangle surrounding the states wherein Agent 2 acts indicates that Agent 2 does not know which state it is in. At the leaf nodes we have the rewards for Agent 1 and Agent 2 respectively. The optimal strategies for both players or *Nash equilibrium* for the game of RPS is for both players to choose any action $a \in \{R, P, S\}$ with a probability of one-third. However, the assumption that Agent 1 plays this optimal strategy is by itself insufficient for determining an optimal strategy for Agent 2, since the value of any action performed by Agent 2 would be zero in that case. We might therefore surmise that Agent 2 could take any action with any probability, such as playing R 100% of the time. This would indeed achieve the same reward if Agent 1 always plays the optimal strategy. The problem is that Agent 1 could then adjust to Agent 2's strategy by always playing P.

When we furthermore consider the enormity of possible states in the no-limit variant of poker (there are an estimated 10^{75} states [7] with two players alone), we can understand the profound challenge that it has posed as well as develop an appreciation for the success of recent state-of-the-art milestone research.

2.2 Early Approaches to Poker AI

In 1998, Darse Billings et al presented a foundational early poker agent dubbed *Loki* that used opponent modelling, formula-based strategies, along with expert-defined rule-based logic [3, 15]. An *opponent model* in poker can define a probability distribution over an opponent's likely hole cards [15], a probability distribution over a set of actions available to an opponent [16], or both. Opponent modelling has usually been achieved via a neural network using features of the game state. There is moreover a distinction between *generic opponent modelling* that models an average opponent and *group-specific opponent modelling* whereby certain classes of opponents – obtained through clustering techniques – each have their own model [17].

The researchers introduced the notions of *hand strength, hand potential,* and *effective hand strength* [15]. Hand strength measures the strength of a hand versus an opponent's distribution of hole cards. Hand potential is a measure of how likely the hand is to improve in strength. They also defined an alternative metric to hand potential representing the likelihood of the hand strength decreasing. Combining all these, a formula for two types of effective hand strength were presented, representing the value of betting and the value of calling given future board cards to come. These were used as a metric for determining actions following rule-based logic.

Billings would later present *Poki*, an improved version of Loki that notably performed nested subgame simulations using Monte Carlo tree search [4]. A few years afterwards, the trend in poker-related research began to shift toward methods very firmly based in the realm of game theory.

2.3 Extensive-Form Imperfect Information Games

Poker can also be modelled as a game tree of sequential multi-agent interactions in the same manner that is depicted above for RPS. A formal description of an extensive-form imperfect information game has the following components [8, 9]:

- The finite set $\mathcal{P}$ of players. A special player $c \notin \mathcal{P}$ also exists to represent the probability that some action is taken (c may be thought of as nature or chance).
- The finite set $\mathcal{H}$ of all game tree nodes. Each node $h \in \mathcal{H}$ represents a *history* of a possible sequence of actions, as well as any information private to one player. The leaf nodes $\mathcal{Z} \subseteq \mathcal{H}$ are terminal histories at which rewards are realised. A sequence of actions leading from h to h' may be denoted as $h \sqsubset h'$, and the node following h after an action a is chosen is written as $h \cdot a$.
- The function A, where $A(h)$ maps a node or history to a set of actions available at that history.
- The player function P, where $P(h)$ denotes the player who acts at a particular node.
- The function f_c available at all nodes where $P(h) = c$, whereby $f_c(a|h)$ is the probability $a \in A(h)$ occurs given h.
- The *utility* or reward function $u_i : \mathcal{Z} \to \mathcal{R}$ for each player $i \in \mathcal{P}$, the range of which is denoted by L.
- An *information partition* $\mathfrak{T}_i$ of $\{h \in \mathcal{H} : P(h) = i\}$ for each player $i \in \mathcal{P}$. Each member of the partition $I_i \in \mathfrak{T}_i$ or *information set* (infoset) represents the set of histories that are indistinguishable to player i. In other words, for all $h, h' \in I_i$, the player i is unable to know whether she is in h or h'. Furthermore, since $A(h) = A(h')$ for all $h, h' \in I_i$, we can instead write $A(I_i)$.

Apart from information partitions and infosets, the definition is the same as it is for extensive-form perfect information games. In the RPS diagram of the section 2.1, the encircled nodes constitute an infoset for Agent 2. Furthermore, if $|\mathcal{P}| = 2 \ \wedge \ (z) + u_2(z) = 0$ then we say that the game is two-player zero-sum [9]. Note that the provided formal definition allows for players to forget previously known information [8]. In poker we assume that all players have *perfect recall* and remember all their actions.

2.4 Additional Game Theoretic Definitions

In addition to the main components of an extensive-form imperfect information game, let us take note of the following definitions [9]:

A *strategy* or *policy* $\sigma(I_i)$ is a discrete probability distribution (probability vector) over $A(I_i)$. The probability of a given action a occurring at $h \in I_i$ can be written as either $\sigma(h, a)$ or $\sigma(I_i, a)$. A strategy for a particular player is denoted as σ_i. A *strategy profile* σ is a tuple of strategies for every $i \in \mathcal{P}$. We denote the strategy profile for all players other than i as σ_{-i}.

If, in response to σ_{-i}, player i plays a strategy BR that maximises her utility (expected value), this is known as a *best response* strategy. More formally, $BR(\sigma_{-i}) = argmax_{\sigma_i'} u_i(\sigma_i', \sigma_{-i})$ where σ_i' denotes some alternative strategy for player i.

We define a *public state* K as a set of histories in which $h \in K \ \wedge \ h, h' \in I_i \ \to \ h' \in K$. In a public state, all players are aware that the true history or state of the game is some $h \in K$, but due to hidden information, they may know which history it is. An *imperfect information subgame* S, often simply called a *subgame*, represents a union of public states such that the following holds true: $h \in S \ \wedge \ h'' \in S \ \to \ h \sqsubset h' \sqsubset h''$.

2.5 Nash Equilibria

Suppose that in the game of RPS the strategy of Agent 1 is to play rock 40% of the time and paper and scissors both 30% of the time. The best response of Agent 2 would then be to play paper 100% of the time. However, Agent 1 could then adjust to Agent 2's new strategy by always playing scissors. At some point, they would both realise that to prevent the other player from hard countering their strategy, they should never take any action with a relatively high probability. In fact, as previously mentioned, the *Nash equilibrium* of RPS for which neither player is incentivised to deviate from their strategy is to take each action with an equal probability of one-third.

A Nash equilibrium σ^* is a strategy profile where $\forall\, i, u_i(\sigma_i', \sigma_{-i}^*) = max_{\sigma_i'} u_i(\sigma_i', \sigma_{-i}^*)$ [10]. More simply, $\forall\, i, \sigma_i = BR(\sigma_{-i})$, or in other words, every player's strategy σ_i in σ^* is a best response to σ_{-i}.

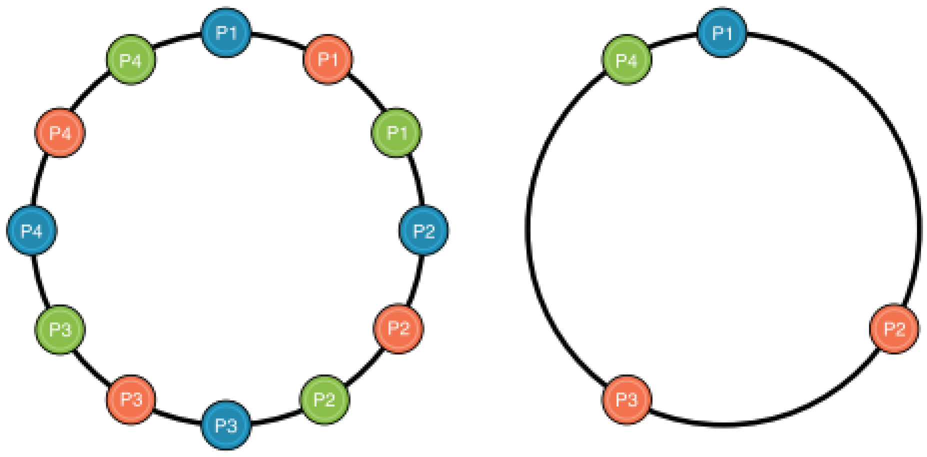
For large games, we typically wish to compute an approximation of a Nash equilibrium or ϵ-Nash equilibrium defined by some sufficiently small constant ϵ where $\forall\, i,\ u_i(\sigma_i', \sigma_{-i}^*) + \epsilon \geq max_{\sigma_i'} u_i(\sigma_i', \sigma_{-i}^*)$ [13].

In two-player zero-sum games specifically, there is exactly one Nash equilibrium [11], and provided that the game rules do not bias a certain player, each strategy σ_i^* is essentially unbeatable, and σ^* is a solution to the game itself [9]. The AI systems that have bested top human players in games like chess, Go, and two-player no-limit poker have thus far all done so by approximating σ^* rather than attempting to exploit perceived weaknesses in the human opponent's strategy [9].

2.6 Limitations of Nash Equilibria in Multi-Player Games

For multi-player games which we define as games where $|\mathcal{P}| > 2$, there may be multiple or possibly an infinite number of Nash equilibria, none of which are even guaranteed to be a favourable strategy [9, 11]. Unlike two-player zero-sum games where an agent i may independently compute a Nash equilibrium without the cooperation of $\mathcal{P}_{-i}$, doing so when $|\mathcal{P}| > 2$ may not in fact result in a strategy profile that is a Nash equilibrium [9, 11].

An illustrative example of this is the Lemonade Stand Game [12]. Each player simultaneously sets up a lemonade stand along a circle as far away from the other players as possible to avoid business competition. In a Nash equilibrium for the game, the players are all evenly spaced apart from one another. Since there are an infinite number of ways this can happen, the Nash equilibria for the game are infinite. Notice furthermore in the diagram below [12] that if each player independently chooses a position corresponding to a particular Nash equilibrium, then the resulting strategy profile may not be a Nash equilibrium.



Despite this uncertainty of their reliability, in multi-player poker at least, the approximation of Nash equilibria has empirically shown strong performance. In 2019, researchers developed the Pluribus agent that defeated a group of top professionals in six-player poker for the first time [11]. The main caveat to its success was that the agent and its human opponents played every hand using the same chip stack size of 100 big blinds. In reality, stack sizes in poker become imbalanced as chips are gained and lost, resulting in varying optimal strategies that may indeed be quite different. Nevertheless, Pluribus was a milestone achievement that demonstrated that Nash equilibria can yet be very effective in practice for multi-player games despite the theoretical ambiguity.

2.7 Regret Minimization

The term *regret*, in plain terms, is a measure of how much we regret playing a particular strategy compared to playing some alternative strategy. There are different notions of regret depending on the problem domain [9]. The one used by state-of-the-art poker AI systems considers an agent i playing a sequence of strategies from a set of possible strategies $\mathcal{S}_i$ over T iterations. An outer loop iterates over $\mathcal{S}_i$ as an inner loop iterates over T. For each alternative strategy $\sigma_i' \in \mathcal{S}_i$, we measure the difference in utility between σ_i' and σ_i^t. The *average overall regret* of agent i after T iterations is then given by the following formula [9, 13]:

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

For two-player zero-sum games, it has been proven that the minimisation of regret for each player i such that $R_i^T \leq \epsilon$ results in an ϵ-Nash equilibrium [9, 13]. There have been a variety of state-of-the-art algorithms for doing this, most of them based on the *counterfactual regret minimization* (CFR) algorithm first introduced by Zinkevich et al [13] in 2007.

2.8 Abstraction

As mentioned in section 2.1, there are an estimated 10^{75} states [7] in two-player no-limit poker alone – an intractably large size. We wish to simplify the game to an *abstraction* of a manageable size, with the hope that the utility of each player playing an ϵ-Nash equilibrium in the simplified game does not differ drastically when the same strategy profile is applied to the unabstracted game. Formally, an abstraction for player i has the following components [14]:

- A partition $\alpha_i^{\mathfrak{I}}$ of $\{h \in \mathcal{H} : P(h) = i\}$ for which each set in the partition is a subset of the corresponding set in $\mathfrak{I}_i$. The elements of $\alpha_i^{\mathfrak{I}}$ are called *abstract information sets*, and $\alpha_i^{\mathfrak{I}}$ itself is an *information abstraction*.
- A function α_i^A on histories such that for each abstract information set $I_i' \in \alpha_i^{\mathfrak{I}}$, the following holds true: $\forall\, h, h' \in I_i',\ \alpha_i^A(h) \subseteq A(h) \land \alpha_i^A(h) = \alpha_i^A(h')$. A set of actions $\alpha_i^A(h)$ is then referred to as an *action abstraction*.

In plain terms, an action abstraction limits the number of actions a player may take at each history. For example, in poker we may choose to restrict each player to choosing between n number of bet-sizing options. Since the size of the game increases exponentially with the cardinality of the set of actions $A(h)$ [14], this alone can have a significant impact.

Information abstraction involves limiting the size of the infosets. In the game of poker, this is done via a technique called *bucketing* [4] whereby we merge hands of equal or similar strategic value. As an example, imagine that the board cards are the following: **Qd Tc 2c** (Queen of diamonds, Ten of clubs, 2 of clubs). It makes no difference to us whether we hold either **9h 8h** or **9s 8s** in this situation since the suits of our hole cards do not match that of any card on the board, and we would choose to play the same exact strategy in both cases. We may therefore bucket together the histories on this type of board where we were dealt either holding and treat them as one individual history.

3 System Design and Analysis

The approach taken with the AI system presented here is to have the agent play as closely as possible to an ϵ-Nash equilibrium (NE) solution, whenever possible. However, computing an ϵ-Nash equilibrium is PPAD-complete in most cases [18], which, on the first and second betting rounds, is very challenging if not infeasible to do in real-time within the 14-second timeframe permitted by the third-party external poker client software. The agent instead accesses two separate databases of pre-computed NE solutions compiled by third parties [19, 20] depending on the public state K of the game. When K is such that no solution exists in either database, the agent then employs a separate decision-making system that uses an opponent model combined with formula-based strategies and rules, which is based on the early Billings et al [15] system. The entire AI system could be summarised into the following six main components:

- A **screen reader** that uses computer vision techniques to read relevant information within the game window.
- A **state manager** responsible for recording all actions and relevant information for each history.
- An **opponent model** that predicts a probability vector over an abstracted set of actions $\alpha_i^A(h)$.
- An **NE-based decision maker** that issues HTTP requests to retrieve a solution from one of the two third-party databases and chooses the most appropriate action $a \in \alpha_i^A(h)$.
- A **model-based decision maker** that uses the opponent model along with formula-based strategies and rules.
- An **actor** that performs actions by sending keyboard input to the game window.

A diagram depicting the interactions between these components is presented in the appendix (1).

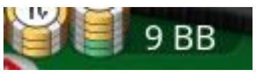
As a research project, the system has not been designed with any target user in mind and it uses a command line interface. The following sections detail the inner workings for each of these components.

3.1 Reading the Screen

The agent cannot directly access the information pertaining to the state of the game in the third-party poker software. One option might be to locate exactly where the software is storing the information in memory, and then read from those memory addresses. This however would most likely require much investigation and would seemingly be less scalable for use with other external software. The approach taken here is to instead process sub-sections of screen images containing the relevant information which are then forwarded to a predictive *optical character recognition* (OCR) model. The screen reader uses the popular open-source OCR engine *Tesseract* [21], the newest version of which is based on LSTM neural networks.

Upon running the application, the screen reader loads a JSON file into memory. This file contains the bounding box coordinates for every area of the screen that should be captured. Since it would be tedious to create this JSON file manually, a separate module was written to record the bounding boxes based on mouse clicks, which draws rectangles on the screen to aid with accuracy.

Before feeding the images to the OCR model, they must first be processed to produce accurate results. Tesseract's results are only reliable when feeding it images of black text on a white background. In addition, it often produces inaccurate results when numbers are combined with letters, such as in the below image:



In the poker client, the sizes of real-money chip stacks and wagers are represented in big blinds, hence the text **BB** following the number. The goal then is to somehow produce an image of a black **9** on a white background, with nothing else. Note that the area around the number cannot simply be captured because the number is not always in the same exact position, and it may have multiple digits. Observing the image closely however, we can notice that the background surrounding the text is a darker shade of green. The screen reader exploits this dark-green background using techniques such as *thresholding* and *contour detection*, which will be further detailed in section 4.1.

To detect where the button is on the screen, the reader uses *template matching*, whereby it searches for a sub-image within the captured area of the screen. This technique is also used to detect images such as the back of cards, indicating the presence of a player still involved in the hand, or the timebank of a certain player, indicating that it is their turn to act. In other cases, such as detecting the card suits, a simpler approach is used: since each suit has a unique colour, the reader simply averages the pixels of the card and checks if it falls within a certain range.

3.2 Managing the Game State

To make decisions, the agent must know what has happened. The state manager runs at one-second intervals, recording new information pertaining to the public game state such as the following: the sequence of actions that have transpired, the total investments of each player for each round of betting, the pot and stack sizes, and the wager amounts. Each player starting from the last active player to the currently active player is sequentially examined. If they are no longer in the hand, it means that they folded. Otherwise, we look at the money in front of them. If the money is equal to (or less than in the case of a player with little money) the size of the last wager, this indicates a call – and otherwise, a raise.

One tricky aspect of this occurs on transitions from one betting round to another. In these cases, the last player to act becomes the first player to act on the next round. All the wagers disappear and are added to the pot. The speed with which human opponents act also cannot be controlled. For example, it could be possible that after the agent makes a bet, two opponents very quickly call and one of them immediately bets on the following round, all before the system has a chance to update. The state manager can infer what actions took place using an algorithm detailed in section 4.2.

The state manager is moreover responsible for making calls to the appropriate decision-making sub-system depending on both the public state K and what we will refer to as the *assumptive state* K'. We can think of K' as the result of applying a function to all $h \in K$ such that every action within $h' \in K'$ is a member of an action abstraction set $\alpha_i^A(h)$. This is useful because the NE solutions that the agent uses were computed with highly abstracted (simplified) actions, and each bet or raise amount must therefore be rounded to the nearest amount in the abstraction set. At some point, if the true actions taken in K differ substantially from their corresponding actions in K', then K' becomes unreliable for determining a strategy via the NE-based decision maker (in which case the model-based decision maker is used). The reliability of K' is measured by computing a mean-squared error across all corresponding histories in K and K' wherein the action taken at each history is a bet or raise. To formalise this, let s and s' be corresponding equal-length sequences of bet or raise amounts for every history in K and K' at which a bet or raise occurred. If $\frac{1}{|s|}\sum_{i=1}^{|s|}(s_i - s_i')^2 > c$ where c is some threshold constant, then the assumptive state K' is considered to be unreliable and is disregarded from that point onward in the hand (the model-based decision maker, which is then used, has no notion of assumptive state). This reliability is checked after every update of the assumptive state by the NE-based decision maker.

3.3 Predicting the Opponents' Actions

A supervised learning model was created using hand history data obtained from the internet. The appendix (2) contains one example hand from the dataset. A parser was written to extract a feature vector at each *post-flop* history of the public state. *Pre-flop* and *post-flop* are poker-specific terms that respectively refer to the first betting round (with no board cards) and any following betting round. The reason why pre-flop hands were not considered when developing the model is two-fold. First, the style of play that occurs pre-flop is quite different; each player only has two cards, and there are no board cards. Billings et al reported higher accuracy for their model when disregarding pre-flop hands [16]. Secondly, being the first betting round, there are much fewer states to account for compared to subsequent rounds. As a result, the NE-based decision maker is relatively much more robust at resolving pre-flop strategies, seeing as it is much more likely for a solution to exist in the pre-flop database compared to the post-flop database.

For each feature vector, there is one of three labels: the current player folded, the current player acted passively (checked or called), the current player acted aggressively (bet or raised). The features used are provided in the appendix (3). These features were fed through a deep neural network to produce a model that can predict a probability vector over the three generic types of actions used as the labels. The AI system uses a module dedicated to accessing this model and returning predictions. For example, it may use this to predict a probability vector over the generic actions of the next player to act in response to a specific hypothetical action taken by the agent. In other cases, we may wish to predict the probability that all remaining opponents fold to an aggressive action taken by the agent, which is useful when deciding whether to bluff. This can be computed by predicting the fold frequency of each opponent and multiplying them all together. Alternatively, we may wish to predict the complement of the probability of all remaining opponents checking following a check from the agent, which is the probability that at least one player makes a wager. This is useful when deciding how often to check a hand that may be worth betting. The model module copies relevant info from the public state and modifies it, each time calling the prediction method for the next opponent.

3.4 The NE-Based Decision Maker

This component uses modules that make HTTP requests to third-party servers for NE strategies. Due to abstraction, the real wager amounts must be modified to request a valid game tree node. The servers return a JSON response containing the strategy for the specific node requested along with its abstracted action set. There are two third-party data providers that are used: one for pre-flop and one for post-flop. Notably, for post-flop situations, the NE-based decision maker is only used when not more than two players progressed past the first betting round.

One issue is that the pre-flop raise actions assumed by the post-flop solutions only include one sizing. The approach taken is to strike a balance between choosing an appropriate pre-flop raise amount and choosing amounts that are close to the ones assumed by the post-flop solutions. The way this is handled is by using the average raise amount over the entire probability distribution of hole cards. Another way to think about this is than an extra layer of abstraction is added whereby all raise actions are merged into one (all-in raise actions are the only exception to this and are not merged).

Before sending a request to either server, a node path must be constructed using valid actions at each node. The sequence of real actions that have occurred in K are iterated and the nearest wager amounts are used when constructing both the node path and the assumptive state. A cache is maintained that maps node paths to corresponding action sets, which is how it can know what actions are available at each intermediary node in K'. In the case that the node path does not exist in the cache, that intermediary node – and all intermediary nodes that follow – must be requested before requesting the actual current node of K'.

Additionally, because these external web-based systems are primarily intended for browser use, an authenticated status must be maintained using an HTTP session along with JSON web tokens, as well as periodic requests to refresh the access token.

3.5 The Model-Based Decision Maker

This component uses the neural network model along with betting and calling formulas and rules to determine a best course of action. The first thing it does is to calculate several metrics to be used in each of the formulae. Hand strength (HS), hand potential, and effective hand strength (EHS) are all calculated using the algorithms documented by Billings in his PhD thesis [4], reproduced here in the appendix (4). Note that HS is afterward raised to the power of $|\mathcal{P}_{-i}|$, the number of remaining opponents, to produce HS_n. There are furthermore two types of hand potential: the probability of our hand improving in strength ($PPOT$), and the probability of the hand's strength decreasing ($NPOT$). There are moreover two types of effective hand strength: one used for betting or raising (EHS_v) *for value* (i.e. with at least a relatively strong hand) and one used for calling (EHS_c), the formulae for which are as follows:

$$EHS_v = HS_n + (1 - HS_n) * PPOT$$
$$EHS_c = EHS_v - HS_n * NPOT$$

Whilst it is used by the model-based decision maker to compute EHS values, the notion of hand potential by itself is perhaps not the most useful metric. In some cases, the potential of our hand improving may be quite low; however, when it does improve, it may improve to a level of hand strength that is extremely high. An example of this is a flush or straight draw. In fact, it may often be better to call with these types of draws with the chance of winning additional bets on later rounds than to call with marginal hand with higher hand potential. The approach taken is therefore to use a metric that will be referred to as *nutted hand potential* (NHP), the chance of improving to a hand for which $HS_n > k$ where k is some constant near 1. The algorithm to compute NHP is presented in section 4.4.

There are believed to be other factors that should affect EHS besides the number of remaining opponents. These factors are *pot odds*, *post-flop aggression*, and *call count*. Pot odds (PO) is the amount in the pot divided by the amount to call. In the case that no one has bet, the pot odds can be represented as an infinite number. Post-flop aggression (PA) will be defined as the number of bets or raises that occurred on any round after the first. The call count (CC) is the number of players who called the last bet or raise. Using these metrics, the new formulae for EHS values used by the agent are as follows:

$$EHS_v' = (EHS_v)^{1+PA+CC*PO^{-1}}$$
$$EHS_c' = (EHS_c)^{PA+CC*PO^{-1}}$$

A new metric EHS_s is also used to represent the *showdown value* of a hand, which is a measure of how much the agent would like to show its hand down without bluffing or betting for value:

$$EHS_s = (EHS_v)^{1+PA}$$

EHS_v' is compared to a constant k_v to determine whether the hand can be bet for value. If it can, then the hand's *worth* is determined by linearly interpolating EHS_v' from the interval $[k_v, 1]$ to the interval $[w_0, w_n]$ where w_0 and w_n are the smallest and largest wager amounts in the action set. The *worth* represents how much the agent is willing to bet and is rounded to w_v, the nearest amount in w. Wager amounts in w are then iteratively tested up to w_v, computing an expected value (EV) for each of them based on how frequently the model predicts the opponent(s) will fold. The highest EV wager amount is then selected.

In the case of calling, the k value, k_c, is no longer constant; it is determined by fitting PO to a non-linear function. The function $f(x) = (x - 10)^{-2}$ is used, which is based on an upper bound of 20 set for PO and was found through experimentation. Additionally, the formula $EV_c = RP * NHP - CA(1 - NHP)$ is used to estimate the expected value of calling with nutted hand potential. If either $EHS_c' > k_c$ or $EV_c > 0$, then the agent considers its hand to be a profitable call.

When considering whether to bluff, the agent considers the predicted success rate SR of bluffing, which is the probability that all remaining opponents fold. Let RP be the size of the raked pot (the slightly reduced pot size that accounts for *rake*, which is the percentage taken by the poker room as a game host fee), and let PIP be the additional amount that we put into the pot as a result of betting or raising to some percentage of the pot w_b. We can estimate the EV of bluffing on the flop or turn (the second or third betting rounds) as follows:

$$EV_b = RP * SR + (1 - SR)(RP * NHP - PIP(1 - NHP))$$

Note that NHP is undefined on the river (the final betting round), but in that case, EV_b can be calculated more simply as $RP * SR - PIP(1 - SR)$.

3.6 Performing Actions

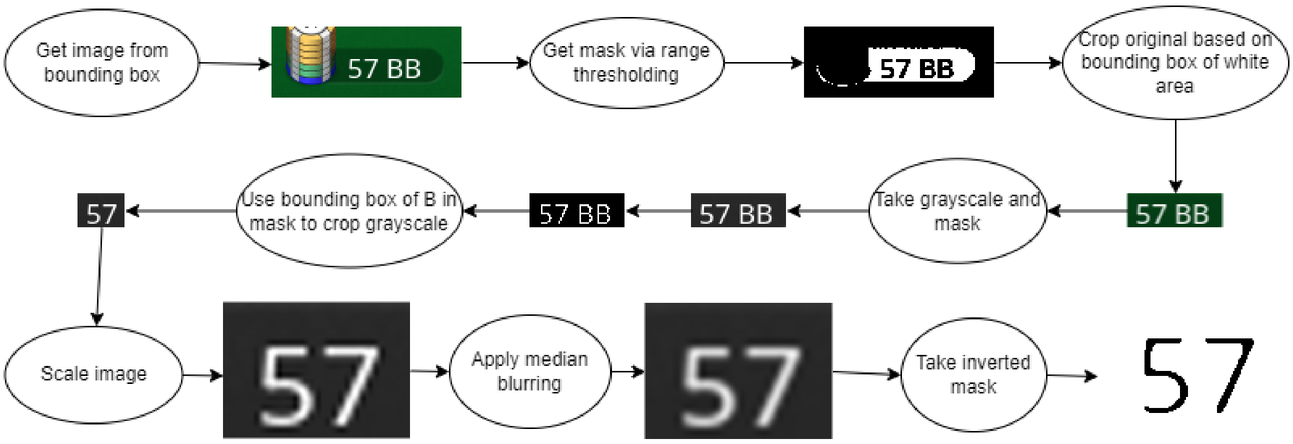
The actor component randomly samples an action from the strategy returned by either of the two decision makers. The action is then performed within the poker client using keyboard input. Wager amounts in increments of 5% of the pot are mapped to hotkeys that were set up within the poker client, and the hotkey for the nearest increment is used. The actor may wait a random amount of time before performing an action. Note that the keypress duration for a human is typically between 50 and 300 milliseconds [19]. After initiating a keypress, the actor attempts to imitate this human behaviour by waiting some random number of milliseconds within that range before releasing the key.

4 Implementation

The poker AI system has been implemented using the Python programming language, along with open-source libraries such as Tesseract and OpenCV for the screen reader, and TensorFlow for the opponent model. Following sub-sections contain some noteworthy implementation details and algorithms.

4.1 Reading Wagers: Image Processing Pipeline

By *thresholding* an image using a range of pixel values, a binary mask of the image is produced. From that binary pixel array, using the OpenCV library [22], the bounding boxes for the contours of the area of the dark-green region as well as that of the first **B** in a wager image can be detected.

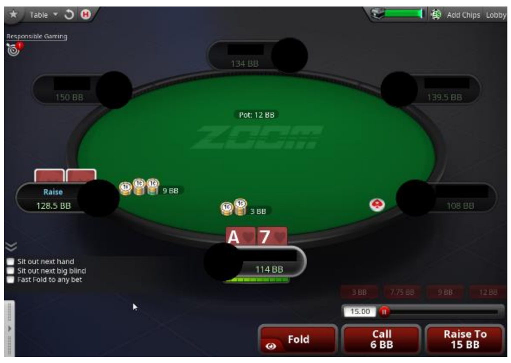


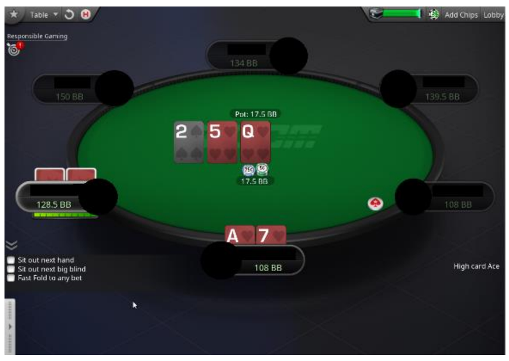
Get image from bounding box
57 BB
Get mask via range thresholding
57 BB
Crop original based on bounding box of white area
57 BB
Take grayscale and mask
57 BB
57 BB
Use bounding box of B in mask to crop grayscale
57
Scale image
57
Apply median blurring
57
Take inverted mask
57

The resulting image is then passed to Tesseract's OCR model. This process differs depending on the type of element on the screen to be read, but reading the wager is the most involved.

4.2 Inferring Past Actions

Supposing that the state manager updates on the following two consecutive frames, it can infer the correct sequence of actions that has occurred so far in the hand:





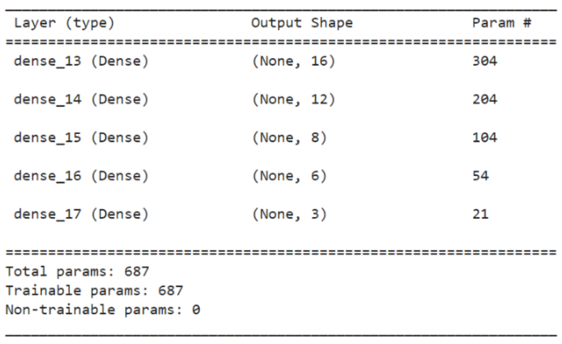
In other words, with only two frames, it can know that all players before the small blind player folded, the small blind player raised, the big blind player re-raised, the small blind player called and then checked on the flop (the next round). To infer actions that occurred between betting rounds, it uses the following algorithm:

```
function UpdateLastRoundActions:
    # Handle cases where the remaining players checked or folded
    if last unraked pot recorded >= current raked pot:
        for each player from the last active to the currently active:
            if the player is in the hand:
                if there is no recorded action for the player on the last round:
                    UpdateState(player, "check")
            else:
                UpdateState(player, "fold")
        # Nothing further to consider, so just return
        return

    remaining = list()  # This will store any remaining players to account for
    actions = dict()  # A map of players to actions
    for each player from the last active to the currently active:
        if the player is in the hand:
            remaining.append(player)
        else:
            actions[player] := (player, "fold")

    # Handle cases where any remaining players called
    total := last recorded unraked pot
    if there is any recorded wager:
        for player in remaining:
            total += player's recorded investment on the last round

    # If, by adding to the prior-round investments of the remaining players to match the prior-round last wager, we can equal or
    # exceed the raked pot, then we can assume that any remaining players whose prior-round investment was lower than the last
    # wager had called
    if total >= raked pot:
        for player in remaining:
            if player's prior-round investment = last wager:
                break
            actions[player] = (player, "call", last wager)
    else:
        # The agent is assumed to have bet or raised
        unrakedPot := min(rakedPot plus rake cap, rakedPot / (1 - rake))
        inv := total investments of all remaining players for the last round
        wager := (unrakedPot - last unraked pot + inv) / length(remaining)
        actionString := "bet" if no last wager else "raise"
        actions[agent] = (agent, actionString, wager)
        remaining.pop(0)  # Remove the agent
        for player in remaining:
            actions[player] = (player, "call", wager)

    # ensure that actions are added in the proper order
    for each player from the last active to the currently active:
        if player in actions:
            UpdateState(action[player])
```

4.3 Training the Opponent Model

The parser processed roughly five million poker hands in text format like the example in the appendix (2). Features of the post-flop game state for every non-terminal node were appended to a CSV file which was later converted to a Numpy feature matrix. All columns of the matrix were normalised to the interval [0, 1] using one of two techniques. Min-max normalisation was used for columns whose values fall within a certain range. That is to say, the function $f(x) = \frac{x - x_0}{x_n - x_0}$ was applied to each element of the column, with x_0 and x_n being the min and max of the range of possible values of the feature. For a few of the features whose values could theoretically be unbounded (e.g. the stack to pot ratio) the function $f(x) = \frac{x}{1+x}$ was used.

The feature vectors in the matrix totalled roughly 26.2 million, and the data was shuffled and split using 80% for training, 10% for validation, and 10% for testing. The size of the input layer – or the number of features – was 18, and four densely connected hidden layers were used, as can be seen in the model architecture below:



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_13 (Dense)            (None, 16)                304

 dense_14 (Dense)            (None, 12)                204

 dense_15 (Dense)            (None, 8)                 104

 dense_16 (Dense)            (None, 6)                 54

 dense_17 (Dense)            (None, 3)                 21

=================================================================
Total params: 687
Trainable params: 687
Non-trainable params: 0
_________________________________________________________________
```

The length of the output layer, which is 3, corresponds to the number of labels. This is because the output is a probability vector over these labels. The loss function used was *categorical cross-entropy*, which measures the distance between probability distributions. Other hyperparameters used are presented in the table below:

Hidden layer activation function	ReLU
Last layer activation function	Softmax
Optimizer	RMSProp
Learning rate	0.005
Batch size	2^{14}

Hyperparameter tuning largely involved experimenting with the learning rate and the batch size. Regularization was briefly toyed with, but it did not appear to be useful, perhaps due to the very large size of the dataset.

4.4 Model-Based Decision-Making Algorithms

The algorithm to compute nutted hand potential is presented below:

```
function NuttedHandPotential(CARDS holeCards, CARDS boardCards, INT numOpponents, FLOAT k) -> NULL or FLOAT:
        if length(boardCards) = 5:
                return NULL

        # remove known cards from the deck
        SET deck := copy(_DECK_)
        for card in holeCards + boardCards:
                deck.remove(card)

        INT nutted := 0
        INT total := 0

        # consider all possible combinations of board runouts
        for CARDS combo in Combinations(deck, 5 - length(boardCards)):
                for card in combo:
                        deck.remove(card)
                CARDS board := boardCards + runout
                FLOAT handStrength := HandStrength(holeCards, board, numOpponents, deck)
                for card in combo:
                        deck.add(card)
                total += 1
                if handStrength > k:
                        nutted += 1

        return nutted / total
```

Note that the algorithm is computation intensive on the flop. It makes calls to the implementation of Billings et al's *HandStrength* algorithm, which compares the agent's hole cards to every possible pair of cards that the opponent may hold. *HandStrength*, in turn, makes calls to a *Rank* function, which, given a list of cards, finds the best five-card hand and outputs an integer such that stronger hands receive a higher value and hands of equal strength receive the same value. For the *Rank* function, the open source eval7 poker library is used. *Rank* must be called $\binom{47}{2}\binom{45}{2} + \binom{47}{2} = 1{,}071{,}271$ times on the flop, which takes roughly 6.5 seconds with Python. The algorithm can however be easily modified to consider only the very next card to come, resulting in only $47\binom{46}{2} + 47 = 48{,}692$ calls to *Rank* on the flop. A default value for k of 0.955 is used.

Below is the general algorithm that the model-based decision maker uses to determine a strategy:

```
function Make(state) -> LIST<TUPLE>:
        if the round is pre-flop:
                return MakePre(state)

        metrics := GetMetrics(state)
        unopened := True if no one bet yet else False

        # Check to pre-flop aggressor on flop when acting before
        if unopened AND Agent acts before last aggressor AND round = flop:
                return [("Check", 1)]  # Check with 100% probability

        valuePct := GetValuePct(state)  # The percentage of pot to be bet or raised
        if valuePct > 0:
                if unopened:
                        if Agent is not last to act:
                                checkGetsAction := 1 - Model.predictAllCheck(state)
                                return [("Check", checkGetsAction), ("Bet", 1 - checkGetsAction, valuePct)]
                        else:
                                return [("Bet", 1, valuePct)]  # Bet with 100% probability
                else:
                        return [("Raise", 1, valuePct)]
```

```
if not unopened and CanCall(state):
          return [("Call", 1)]

bluffPct := GetBluffPct(state)  # The percentage of pot to be bluffed
if bluffPct > 0:
          if unopened:
                    return [("Bet", 1, bluffPct)]
          else:
                    return [("Raise", 1, bluffPct)]

if unopened:
          return [("Check", 1)]

return [("Fold", 1)]
```

The function *MakePre* is infrequently used, because the NE-based decision maker is usually quite competent with resolving a pre-flop strategy. If *MakePre* does need to be used, the strategy is quite a simple one that is based on how many raises occurred and how strong the hole cards are. The functions *GetValuePct* and *GetBluffPct* use the ideas and formulae discussed in section 3.5 to determine what percentage of the pot should be wagered, whereas *CanCall* uses them to return a Boolean.

5 Evaluation

The opponent model was evaluated separately from the AI system. The metrics of categorical cross-entropy, precision, and recall were used for model evaluation. To evaluate the AI system as a whole, the bot was tested at the lowest stakes available within the poker client.

5.1 Evaluating the Opponent Model

As mentioned previously, the three labels used by the classifier involve folding, acting passively (checking or calling), and acting aggressively (betting or raising), which will be referred to as F, C, and R respectively. The distribution of actions in the dataset was as follows:

F	16.8%
C	57.6%
R	25.7%

Due to the imbalanced nature of the dataset, accuracy could not be relied on as an evaluation metric. Instead, the metrics of precision and recall were both used in addition to categorical cross-entropy (the loss).

It was reasoned that a classifier predicting each label with a probability corresponding to its frequency in the dataset would serve as a useful baseline for comparison. The table below shows the results of the baseline classifier on the test set. Note that the loss is inversely proportional to the model's performance, with a higher value indicating worse performance.

Baseline Loss	**96.7%**
Baseline Precision	**42.5%**
Baseline Recall	**42.5%**

The trained model was unable to be overfit. One reason that this may be is that there was already a high degree of linear separability in the data. In fact, even when using zero hidden layers, the validation loss on the best epoch was not much worse than it was with the architecture of four densely connected layers that was later adopted – suggesting this to be the case. When using zero hidden layers, it achieved loss, precision and recall scores of 68.8%, 66.0% and 61.7% respectively. Another reason may be due to the nature of the classification problem, which is also a reason why precision and recall themselves are not fully representative of the model's performance.

5.1.1 Limitations of Precision and Recall

For a *multi-class* supervised learning task, what are we trying to predict? We are trying to predict a single solitary label: for example, whether an image is of a lion, a tiger, or a bear, given that only one animal is present in each image. In a *multi-label* supervised learning task, there may be multiple animals present in each image, and so we wish to predict one or more labels for each image. The classification task being dealt with in this report, however, does not fit neatly into either of these two categories. What the opponent model actually predicts is not a subset of labels, but rather a probability distribution over all the labels: i.e., how often is the average opponent expected to perform each type of action? Moreover, it is unclear what a target probability distribution should be. It is only known that the human player performed a certain action; the player's strategy itself is unknown.

To make this limitation clearer, imagine that the average player's true strategy at a certain history is to fold 40% of the time, check or call 55%, and bet or raise 5%. The probability vector for this distribution is $[0.4, 0.55, 0.05]$. Suppose that the classifier predicts a probability vector of $[0.05, 0.55, 0.4]$. This would indeed correctly predict the highest probability label of C, but it may not be very useful. The knowledge that the average player folds with 40% frequency in this situation could be quite beneficial, for example, if the pot odds are such that 40% is a high number.

5.1.2 Final Model Results

Despite the limitations discussed above, the multi-class precision and recall scores, when compared to the baseline results, does at least give some sense of the model's performance, and were thus included as evaluation metrics. The results of the final trained model on the test set can be seen below:

Model Loss	65.7%
Model Precision	68.7%
Model Recall	64.5%

The best metric for evaluation is ultimately the categorical cross-entropy loss function itself, which measures the error of the probability distribution returned by the last-layer Softmax activation function. The final model showed a reduction of 32.1% in loss from that of the baseline classifier.

5.2 Evaluating the Agent's Performance

The agent was evaluated by having it play the lowest real-money stakes offered within the poker client, i.e. 2-cent big blind games (USD currency). The results, which were obtained by parsing the hand history text files generated by the poker client, are presented below:

Hands played	847
Net won (BB)	32
Rake Paid (BB)	43
Net won (USD)	0.64
Rake paid (USD)	0.86
Raked win-rate (BB per 100 hands)	0.08
Unraked win-rate (BB per 100 hands)	0.18

The agent made a very small profit of 0.64 USD, or 32 big blinds. This is a statistically meaningless result over only 847 hands. If we assume that the agent's true win-rate is 0.08 BB/100, and assuming that the standard deviation of a six-player no-limit poker win-rate is 90 BB/100 as reported here, then, using a confidence level of 95%, the margin of error for net winnings after only 847 hands is ± 524 BB. The agent would need to play very many more hands to make any kind of conclusion about its overall performance. Unfortunately, time did not permit that to happen.

6 Conclusions and Remarks

Although no conclusion can be made regarding the agent’s overall performance, it could be argued that the creation of a poker AI system that can make decisions which at least seem reasonable is not an insignificant task, particularly when its decisions must essentially be based on images generated by external software. To that end, this project combined knowledge from a few different areas such as machine learning, computer vision, game theory, and web-based systems. It was moreover observed that supervised learning can be a useful method of predicting probability distributions over a set of labels.

6.1 Further Work

Through the combined use of information abstraction and supervised learning, the agent could likely be improved with an additional model that predicts a probability distribution over an opponent’s likely hole cards [4]. The categorisation of opponents based on observed actions should also be considered, whereby each class of opponent could have its own model [17]. Playing out a subgame through Monte Carlo simulation may also enable more accurate estimations for the expected value of actions [17]. Moreover, when facing a bet or raise, the NE-based decision maker could be improved by requesting the nodes for the nearest two wager amounts in the abstracted action set and interpolating them based on the true wager amount.

6.2 Code Availability

All source code for this project is available in the following Github repository:

https://github.com/eicksl/cm3070

To set up the agent, first decompress the README.zip file and then follow the instructions therein.

6.3 Statement on Ethics and Impact

In the realm of financial markets, automation through the use of trading algorithms is often encouraged. In the poker-playing community however, it is not well-embraced, most likely due to the zero-sum nature of the game. Additionally, whilst there are no legal issues with running a poker bot, the use of automated decision-making software may be in violation of the poker room’s terms of service, depending on the company. With these issues in mind, the evaluation approach taken here should be reconsidered. To prevent potential abuse, the source code repository will be made private after a final mark has been released. Whilst the code does remain public, the readme file has been compressed to limit search engines from leading unwanted traffic to the repository.

References

[1] Findler et al. Studies on decision making using the game of poker. *Proceedings of IFIP Congress 1971*, pages 1448-1459 (1972).

[2] Murray Campbell et al. Deep Blue. *Artificial Intelligence*, 134(1-2):57-83 (2002).

[3] Denis Papp. Dealing with imperfect information in poker. Master’s thesis, University of Alberta (1998).

[4] Darse Billings. Algorithms and assessment in computer poker. PhD thesis, University of Alberta (2006).

[5] Brown, N., Sandholm, T., and Amos, B. Depth-limited solving for imperfect-information games. *Advances in Neural Information Processing Systems* (2018).

[6] Korukhova, Y. and Kuryshev, S. Training agents with neural networks in systems of imperfect information. *ICAART (1)* (2017).

[7] Johanson, M. Measuring the size of large no-limit poker games. Technical Report TR13-01, Department of Computing Science, University of Alberta (2013).

[8] M. Osborne and A. Rubenstein. *A Course in Game Theory.* The MIT Press, Cambridge, Massachusetts (1994).

[9] Noam Brown. Equilibrium finding for large adversarial imperfect-information games. PhD thesis, University of Michigan (2020).

[10] John Nash. Non-cooperative games. *Annals of Mathematics*, 54:289-295 (1951).

[11] Noam Brown et al. Superhuman AI for multiplayer poker. *Science* volume 356, issue 6456 (2019). DOI: 10.1126/science.aay2400.

[12] M. ZInkevich, M. Bowling, and M. Wunder. The lemonade stand game competition: solving unsolvable games. *ACM SIGecom Exchanges*, 10(1):35-38 (2011).

[13] M. Zinkevich et al. Regret minimization in games with incomplete information. *Neural Information Processing Systems (NeurIPS)*, pages 1729-1736 (2007).

[14] K. Waugh et al. Abstraction pathologies in extensive games. *AAMAS (2) 2009*, 781-8 (2009).

[15] Billings et al. Opponent modeling in poker. *Aaai/iaai*, 493(499):105 (1998).

[16] Billings et al. Improved opponent modeling in poker. *International Conference on Artificial Intelligence*, ICAI’00 (2000).

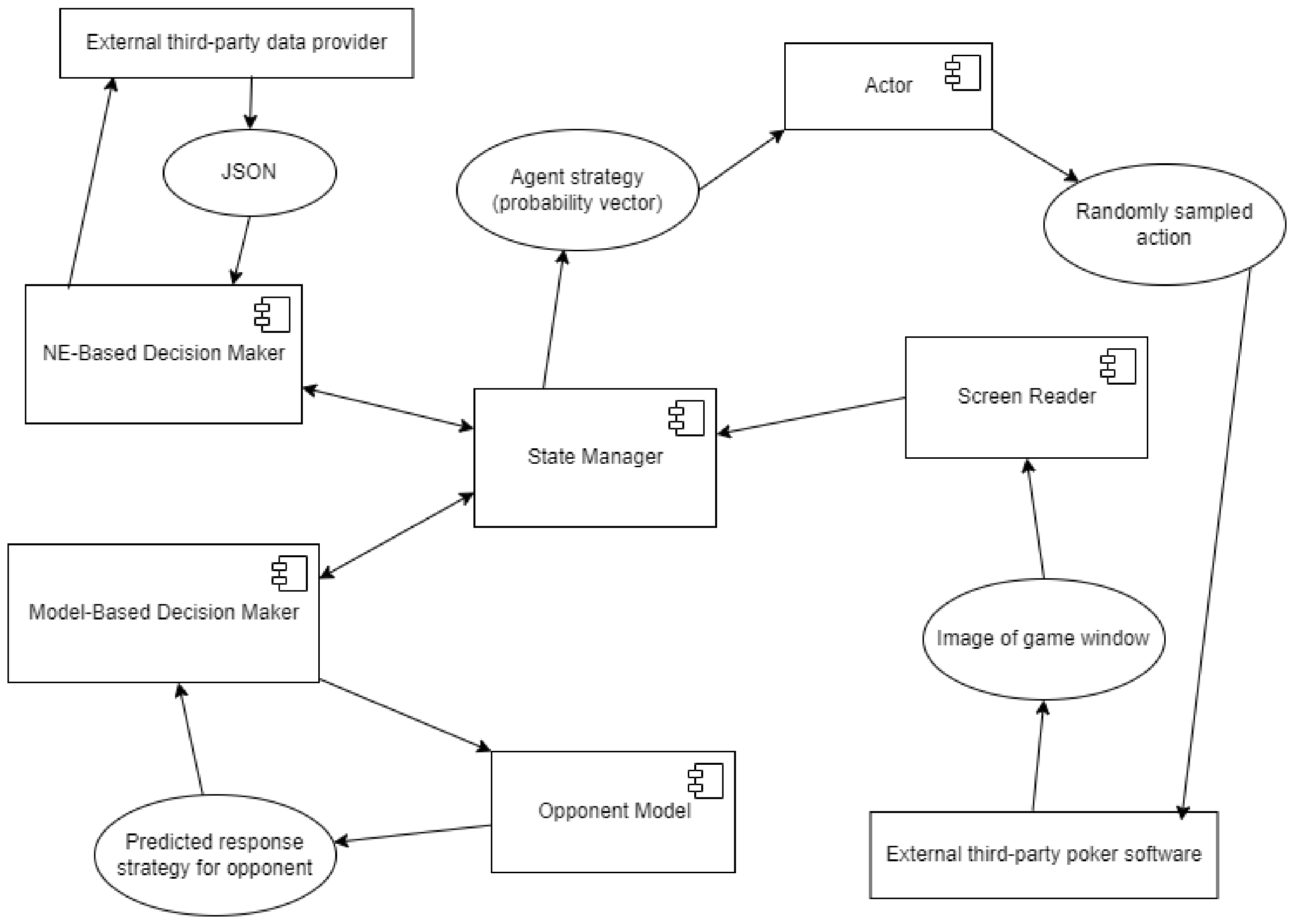
[17] A.A.J. van der Kleij. Monte Carlo Tree Search and Opponent Modeling through Player Clustering in no-limit Texas Hold’em Poker. Master’s thesis, University of Groningen (2010).

[18] Aviad Rubinstein. Inapproximability of Nash equilibrium. *SIAM Journal on Computing,* 47(3):917-959 (2018).

[19] Kevin Killourhy and Roy Maxion. Comparing anomaly-detection algorithms for keystroke dynamics. *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pp. 125-134 (2009).

Appendix

1 System Architecture Diagram


External third-party data provider
JSON
NE-Based Decision Maker
Agent strategy (probability vector)
Actor
Randomly sampled action
Screen Reader
State Manager
Model-Based Decision Maker
Image of game window
Opponent Model
Predicted response strategy for opponent
External third-party poker software

2 Example Hand from Dataset

```
PokerStars Hand #233578654693:  Hold'em No Limit ($0.01/$0.02 USD) - 2022/01/29 23:39:42 ET
Table 'Aaltje III' 6-max Seat #6 is the button
Seat 1: Martin-NH14 ($3.14 in chips)
Seat 2: jsvh55 ($2.17 in chips)
Seat 3: galamtor ($1.37 in chips)
Seat 4: TracyTheNuts ($2.39 in chips)
Seat 5: mangalisa ($2.23 in chips)
Seat 6: Rez9I ($2.15 in chips)
Martin-NH14: posts small blind $0.01
jsvh55: posts big blind $0.02
*** HOLE CARDS ***
galamtor: calls $0.02
TracyTheNuts: raises $0.04 to $0.06
mangalisa: calls $0.06
Rez9I: calls $0.06
Martin-NH14: folds
jsvh55: calls $0.04
galamtor: calls $0.04
*** FLOP *** [Th 5s Jc]
jsvh55: checks
galamtor: checks
TracyTheNuts: checks
mangalisa: checks
Rez9I: checks
*** TURN *** [Th 5s Jc] [Qh]
jsvh55: checks
galamtor: checks
TracyTheNuts: bets $0.21
mangalisa: folds
Rez9I: calls $0.21
jsvh55: folds
```

```
galamtor: calls $0.21
*** RIVER *** [Th 5s Jc Qh] [3c]
galamtor: checks
TracyTheNuts: checks
Rez9I: checks
*** SHOW DOWN ***
galamtor: shows [8d Jd] (a pair of Jacks)
TracyTheNuts: shows [Qs Ah] (a pair of Queens)
Rez9I: mucks hand
TracyTheNuts collected $0.91 from pot
*** SUMMARY ***
Total pot $0.94 | Rake $0.03
Board [Th 5s Jc Qh 3c]
Seat 1: Martin-NH14 (small blind) folded before Flop
Seat 2: jsvh55 (big blind) folded on the Turn
Seat 3: galamtor showed [8d Jd] and lost with a pair of Jacks
Seat 4: TracyTheNuts showed [Qs Ah] and won ($0.91) with a pair of Queens
Seat 5: mangalisa folded on the Turn
Seat 6: Rez9I (button) mucked
```

3 Features

Feature	**Datatype**	**Description**
Round	Integer	Betting round
HighCard	Integer	Highest card on the board
AverageRank	Float	Rank of the average card on the board
FlushPossible	Boolean	Flush is possible
BoardPaired	Boolean	Board is paired
HasFour	Boolean	Board has four to a straight or four to a flush
LC3Flush	Boolean	The last board card brought three to a flush
LC4Flush	Boolean	The last board card brought four to a flush
LC4Straight	Boolean	The last board card brought four to a straight
LCOvercard	Boolean	The last board card is of the highest rank
PlayerAggression	Float	$(1 + R)/(1 + C)$ where R is the total bets or raises of player to act and C is the total checks or calls
OpponentAggression	Float	$(1 + R)/(1 + C)$ where R is the total bets or raises of the last aggressor and C is the total checks or calls
NumAggCurrentRound	Integer	Number of total bets or raises on the current round
IsLastAggressor	Boolean	Player to act is the last aggressor
SPR	Float	Ratio of player’s stack to the pot size
AmountToCall	Float	Amount to call (zero if no one bet)
NumPlayers	Integer	Number of players in the hand
RelativePosition	Float	A measure of a player’s position relative to the button, such that 0 is assigned to the first player to act and 1 is assigned to the player nearest to the button

4 Hand Strength and Potential: Algorithms of Billings et al

```
HandStrength(ourcards,boardcards)
{
   ahead = tied = behind = 0
   ourrank = Rank(ourcards,boardcards)
   /* Consider all two-card combinations
      of the remaining cards.  */
   for each case(oppcards)
   {
      opprank = Rank(oppcards,boardcards)
      if(ourrank>opprank)          ahead += 1
      else if(ourrank==opprank)     tied += 1
      else /* < */                behind += 1
   }
   handstrength = (ahead+tied/2) / (ahead+tied+behind)
   return(handstrength)
}
```

```
HandPotential(ourcards,boardcards)
{
   /* Hand Potential array, each index represents
      ahead, tied, and behind.  */
   integer array HP[3][3]   /* initialize to 0 */
   integer array HPTotal[3] /* initialize to 0 */

   ourrank = Rank(ourcards,boardcards)
   /* Consider all two-card combinations of the
      remaining cards for the opponent.  */
   for each case(oppcards)
   {
      opprank = Rank(oppcards,boardcards)
      if(ourrank>opprank)       index = ahead
      else if(ourrank=opprank)  index = tied
      else /* < */              index = behind
      HPTotal[index] += 1

      /* All possible board cards to come.  */
      for each case(turn)
      {
         for each case(river)
         {  /* Final 5-card board */
            board = [boardcards,turn,river]
            ourbest = Rank(ourcards,board)
            oppbest = Rank(oppcards,board)
            if(ourbest>oppbest)        HP[index][ahead] += 1
            else if(ourbest==oppbest)   HP[index][tied] += 1
            else /* < */              HP[index][behind] += 1
         }
      }
   }

   /* PPot:  were behind but moved ahead.  */
   PPot = (HP[behind][ahead] + HP[behind][tied]/2
        + HP[tied][ahead]/2) / (HPTotal[behind]+HPTotal[tied]/2)
   /* NPot:  were ahead but fell behind.  */
   NPot = (HP[ahead][behind] + HP[tied][behind]/2
        + HP[ahead][tied]/2) / (HPTotal[ahead]+HPTotal[tied]/2)
   return(PPot,NPot)
}
```