Pontifícia Universidade Católica
do Rio de Janeiro

Alexandre Marangoni Costa

Deep learning overview and application to build poker playing agents

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Marcus Vinicius Soledade Poggi de Aragão

Rio de Janeiro
February 2019

Pontifícia Universidade Católica
do Rio de Janeiro

Alexandre Marangoni Costa

Deep learning overview and application to build poker playing agents

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the undersigned Examination Committee.

Prof. Marcus Vinicius Soledade Poggi de Aragão
Advisor
Departamento de Informática – PUC-Rio

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática – PUC-Rio

Prof. Thibaut Vidal
Departamento de Informática – PUC-Rio

Prof. Bruno Feijó
Departamento de Informática – PUC-Rio

Prof. Márcio da Silveira Carvalho
Vice Dean of Graduate Studies
Centro Técnico Científico – PUC-Rio

Rio de Janeiro, February 26^{th}, 2019

Alexandre Marangoni Costa

Bachelor's in Computer Engineer (2013) at the Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Joined the Master Program in Informatics at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in 2017.

Bibliographic data

Costa, Alexandre Marangoni

Deep learning overview and application to build poker playing agents / Alexandre Marangoni Costa; advisor: Marcus Vinicius Soledade Poggi de Aragão. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2019.

v., 37 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Deep Learning. 2. Redes Neurais. 3. Aprendizado de Máquina. 4. Pôquer. 5. Simulação Multiagente. I. Poggi de Aragão, Marcus V.S.. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

To FAPERJ and CAPES, for financing this research.
To Marcus Poggi, who is an advisor but also a close friend.
To my family, for unconditional love.
To my friends, who turn life into a beautiful journey.
To Alan Turing, for making computation a reality.
To God, for making everything a reality.

Abstract

Costa, Alexandre Marangoni; Poggi de Aragão, Marcus V.S. (Advisor). **Deep learning overview and application to build poker playing agents**. Rio de Janeiro, 2019. 37p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Data science research needs real examples to test and improve solutions. Games are widely used to mimic those real-world examples. Poker rounds are a good example of imperfect information state with competing agents dealing with probabilistic knowledge, risk assessment, and possible deception, unlike chess, checkers and perfect information brute-force search style of games. By using poker as a test-bed we can analyze different approaches used in real-world examples, in a more controlled environment, which should give great insights on how to tackle those real-world scenarios. We propose a framework to build and test different neural networks that can play against each other, learn from a supervised experience and maximize its rewards.

Keywords

Deep Learning; Neural Network; Machine Learning; Poker; Multi-Agent Simulation;

Resumo

Costa, Alexandre Marangoni; Poggi de Aragão, Marcus V.S.. **Visão teórica e prática sobre a aplicação de Deep Learning na construção de agentes de pôquer**. Rio de Janeiro, 2019. 37p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A ciência de dados precisa de uma grande quantidade de dados para testar e melhorar soluções. Jogos são largamente usados para abstrair situações da vida real. Rodadas de pôquer são um bom exemplo pois, por não saber as cartas dos oponentes, o jogador analisa um cenário de informação incompleta numa competição de agentes que envolve conhecimento probabilístico, análise de risco e brefe. Isso o diferencia de xadrez, damas e jogos de conhecimento perfeito e algoritmos de busca em forca bruta sobre o espaço de soluções. Usar o pôquer como um caso de teste possibilita a análise de diferentes abordagens usadas na vida real, porém num cenário mais controlado. Esta dissertação propõe um arcabouço de funcionalidades para criar e testar diferentes algorítimos de Deep Learning, que podem jogar pôquer entre sí, aprender com o histórico e maximizar suas recompensas.

Palavras-chave

Deep Learning; Redes Neurais; Aprendizado de Máquina; Pôquer; Simulação Multiagente;

Table of contents

List of figures

List of tables

List of abreviations

ANP – Agência Nacional do Petróleo
CENPES – Centro de Pesquisas Leopoldo Américo Miguez de Mello
LAMP – Local Affine Multidimensional Projection
MDS – Multidimensional Scaling
MP – Multidimensional Projections
N_p – Cumulative oil production
P_{10} – Percentile 10
P_{50} – Percentile 50
P_{90} – Percentile 90
PETROBRAS – Petróleo Brasileiro S.A.
Q_o – Oil flow
Q_w – Water flow
W_p – Cumulative water production

"Hard" is everything you still don't know

Hamazaki, Geiza

1
Introduction

1.1
Motivations

Different games were historically used to benchmark and explore Artificial Intelligence algorithms. In the last twenty years, several games like Chess, Checkers, Go, Backgammon and even Atari games were solved [Kocsis2006], with algorithms reaching strategies that could win from professional players. To solve these style of games, a player has to take a decision given a perfect information scenario. In other words, all the information concerning the game state is available to the player: board configuration in case of board games, screen state in case of Atari games. Since there is no hidden information about the game, a brute-force search can retrieve the best action [Billings1998] with confidence.

But real life is not like that. According to von Neumann, founder of modern game theory, "real life consists of bluffing, of little tactics of deception, of asking yourself what is the other man going to think I mean to do. And that is what games are about in my theory." [Bronowski1973]

In poker, a player's private cards give asymmetric information about the state of the game. Each player sees a different state of the game, and none of them sees the complete state. This is why poker is called an Imperfect Information game and why it's hard to model and solve. Even if a better strategy is played, it can lose from a worse strategy because it has better cards, or it has bluffed or even changed strategy. To give a perspective, an example of an incomplete information board game is Battleship, while Chess is a perfect information game.

Along with that, the number of possible states in Heads-Up No-Limit Texas Hold'em Poker is approximately 10^{160} . Heads-Up means there are only two players in the table, so the number of states in a multiplayer table is even bigger. In comparison, chess and backgammon have 10^{47} and 10^{20} game states respectively [Johanson2013]. The universe has approximately 10^{80} atoms.

Limit Texas Hold'em Poker, a game variation in which players can only raise bets to a fixed amount, have around 10^{13} decision points in a heads-up

game, and for that, it is a lot easier to solve a Limit poker variant. In fact, it was solved in 2015 by the Cepheus algorithm, developed by the Computer Poker Research Group at University of Alberta, and a joint effort with Oskari Tammelin, a Finnish software developer [Tammelin2015].

Furthermore, card games have an element of chance when the deck is shuffled, before dealing the cards. Also, in Texas Hold'em early rounds (named pre-flop, flop, and turn) players have to act before seeing all dealt cards. A player's chance might change when a public card is revealed. This is why poker is classified as a stochastic game, in other words, not deterministic. An example of a stochastic board game is Backgammon, while Chess and Checkers are examples of deterministic games.

Real-world problems, like network and airport security, financial and energy trading, traffic control, routing, business negotiations and forecasting (weather, politics, etc) involve decision making with imperfect information and high-dimensional information states with a huge number of decision points [Silver2016], like poker. Those problems have some characteristics that require intelligent behavior [Billings1998]. We can map many of those characteristics to poker aspects:

Real world	Poker
Imperfect knowledge	Opponents' hands are hidden
Multiple competing agents	Many competing players
Risk management	Betting strategies consequences
Agent modeling	Identify patterns in opponent's play
Deception	Bluffing and varying style of play

An optimal solution to these problems would be a Nash equilibrium, a strategy that if an agent deviates, it loses. If it follows, it wins. If an opponent also follows, both tie. Simple machine learning methods achieve near-optimal solutions to perfect information games but fail to converge in imperfect information games. Using a controlled environment such as poker, allows one to measure progress in a domain where simple machine learning does not converge to near-optimal solutions.

1.2 Goals

The imperfect information property, the stochastic property, along with the size of the game makes solving of multiplayer No-Limit Texas Hold'em Poker an interesting milestone for Computer Science, not reached until now. This work proposes Pucker, a framework to help scientists reach this goal, providing a consistent simulation of the game, storage of past scenarios in an efficient way, a learning and prediction strategy for machine learning algorithms, and some examples of algorithms to help the development of better strategies.

The best Pucker agent is inspired by reinforcement learning, similar to David Silver's work [Silver2016], but adds domain knowledge and state abstraction to learn multiplayer No Limit Texas Hold'em. In the end, we show how to measure progress and display a consistent evaluation of the agent's incremental learning.

Unfortunately, there are several caveats in learning from a generated dataset. A learning model requires a fine representation of poker state, with information such as private and public cards, previous opponents' actions, and any known public information the enables a gradient descent model to approach incremental learning. Also, the training data obtained from simulation of deterministic weak players are not sophisticated enough to generate a model competitive against serious players.

To deal with these challenges, we propose a high dimensional representation composed with most of the information used by professional players, along with statistical features built by simulation of future rounds, that improved early stage poker actions such as flop actions. Also, we improved the dataset by running multiple phases of learning, whereas the first phase relies on data generated by deterministic players, and further phases learn from data generated from deep learning sophisticated players. Finally, we propose a method inspired by genetic algorithms, to prevent reaching a local best response by generating populations of algorithms from the most winners. In the end, we acknowledge that the last descendants of this population are capable of better actions.

In summary, our contributions are:

- A framework to build Texas Hold'em agents;
- A simulation of the game that can be used to generate input dataset in an efficient way;

- Several deterministic players used to generate initial data and measure learning progress;
- A state representation that players can use to learn best actions;
- A learning and prediction neural network that leverages reinforcement learning to learn a multi-step game such as poker;

1.3 Dissertation Structure

The remainder of this thesis is organized as follows:

- Chapter 2 we describe **how to play poker** and the **related work**;
- Chapter 3 shows how **neural networks** work to gradually improve its predictions and how we used reinforcement learning to adapt the model to multi-step games;
- Chapter 4 describes the project **architecture**, simulation, data storage, learn and predict modules;
- Chapter 5 presents the experiments and the associated **results**;
- Chapter 6 presents some **concluding remarks** and directions for the future work;

2
Background

2.1
Poker Foundations

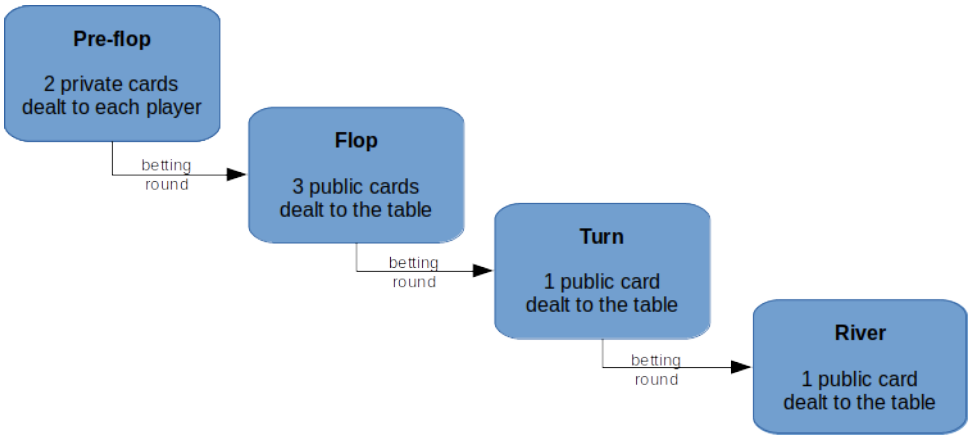
Texas Hold'em poker variant starts with every player receiving 2 private cards. Then, there is a betting round: every player bets some amount that they have the best hand. This first round is called **pre-flop**.

Next, 3 public cards are dealt to the table. This is called **flop**. The flop is public, and those 3 cards are shared among all players. A betting round happens again, but now the players bet who has the best combination of private hand and shared table, a total of 5 cards. Every bet goes to the table pot.

Afterward, there is the **turn** round. A card is dealt to the table, followed by a betting round. Finally, another card is dealt to the table - the **river** round - and the last betting round finishes the game.

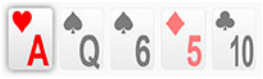
A player with the best combination of private and public cards wins the pot. Also, a player wins if all other players have given up. In case of tie, winners share the table pot.

The bet can be a fold, a call of the previous bet or a raise. When a player folds, he is out of this game and loses the amount he has previously given to the table pot. Each player announces it's bet starting from the first player after the dealer until every player but one fold or every active player bet the same amount (call).



2.1.1 Hand Ranking

The hands are listed bellow in ranking order:
High card: all five cards of different rank and a variety of suits;



One pair: two cards of the same rank and three different cards;

Two pairs: two pairs and one different card;

Three of a kind: three cards of the same rank and two different cards;

Straight: five cards in sequence of rank with different suits;

Flush: five cards of the same suit;

Full House: three cards of the same rank and two cards of the same rank;

Four of a kind: four cards of the same rank;

Straight flush: a straight of the same rank;

If two players have the same type of hand, the hand with higher card wins. A player can use a combination of its private cards and the public cards in the table. In the end, a player has 7 cards available but can combine only 5 in a poker hand.

2.1.2 Positions

It is trivial to understand that player's position matters. The last player to bet have more information about other players' bets and can take a better decision in order to maximize its rewards.

In each round, each player is the **dealer** of the deck, in sequence. The player next to the dealer is the **small-blind** and the next player is the **big-blind**. In pre-flop, the small-blind have to place a minimum bet and the big-blind has to place twice the small-blind. To participate, every player has to bet as least the big-blind amount. In the next round, the flop, bets start from zero.

In this work, we will consider a game of 5 players, therefore, there will be 5 positions, and each player will be dealer, small-blind, and big-blind, once every 5 games.

2.1.3 Actions

Given a state of the game, a player can

- **Bet**: to place an amount to the pot when no one has done it before
- **Raise**: to raise a previous bet
- **Call**: to bet the same amount of a previous bet
- **Fold**: to give up, and lose the money placed in the pot

– **Check**: when no bet has been made, a player can pass (it's the same as betting an amount of 0)

Also, in no-limit poker a player can bet all of its money. This is called **all-in**

2.1.4 Variants

Poker variants usually differ in orthogonal dimensions. The two most common are number of players and betting structure.

A **heads-up** variant includes two players. The **ring** variant includes more than two players and we will refer to it as multiplayer variant. This work concerns about multiplayer poker but can be extended to the heads-up variant.

In the **limit** variant, also called *fixed limit*, the players can only raise to a fixed amount, usually the big-blind amount or twice the big-blind. In **no-limit** poker, a raise can be anything from the last bet to the players total stack.

2.2 Related Work

The first computer program to play poker was called Orac, and was created by Mike Caro to compete in World Series of Poker in 1984.

Afterward, University of Alberta, Carnegie Mellon and University of Auckland led the development of poker bots.

In 1998, the Computer Poker Research Group at University of Alberta, led by Michael Bowling, released Loki, an artificial intelligence capable of playing Limit Heads-Up Texas Hold'em [Billings2016]. Next, they improved their work and created Poki, in 2000 [Davidson2000]. Both of them focused in modeling the opponent's strategy, but still relied in "search" by simulation to find the best decision.

In 2003, scientists began to shift from the chess methodology model, and in 2008 a poker bot developed in University of Alberta, called Polaris, played 6 heads up No Limit Hold'em matches against humans, with 3 wins, 2 losses and 1 tie.

Next, in 2009, University of Auckland introduced Sartres, its first poker AI. It was designed to play specifically heads up Limit Texas Hold'em, and was the first one to use a case-based reasoning methodology [Rubin2009].

Counterfactual Regret Minimization started to play a big whole in this field [Zikenvich2008], and Limit Texas Hold'em was essentially solved in 2015,

by Cepheus, another AI developed at the Computer Poker Research Group at University of Alberta [Bowling2008].

Also in 2015, a professor at Carnegie Mellon University developed Claudico [Brown2015], a No Limit Texas Hold'em bot, but it lost heads up matches against pro players. It required a Pittsburgh supercomputer with 16 terabytes of RAM to learn the its strategy.

Libratus succeeded Claudico, but was rewritten from scratch. It was build with more than 15 million core hours of computation, compared to 3 million performed for Claudico. It applied a new variant of Counterfactual Regret Minimization, namely CFR+, developed by Oskari Tammelin, a scientist involved in the Cepheus project [Brown2017].

In 2016 David Silver introduced Neural Fictitious Self Play, a deep reinforcement learning method composed of two neural networks that learn approximate Nash equilibria of imperfect-information games. Silver's goal was to not rely on engineering abstractions or any domain knowledge, and still be capable of learning in a complex environment. It was applied to Limit Texas Hold'em and approached the performance of state-of-the-art [Silver2016].

Finally, in 2017 the Computer Research Group (UoA) together with several scientists from the Czech Republic, introduced DeepStack, an algorithm for imperfect information scenarios. This work combined many different deep learning strategies, like recursive reasoning, to handle information asymmetry, decomposition to focus computation on the relevant decision, and a form of intuition algorithm to automatically learn from self-play [Moravcik2017]. For the first time, a computer program defeated with statistical significance professional poker players in heads-up no-limit Texas Hold'em.

3
Learning

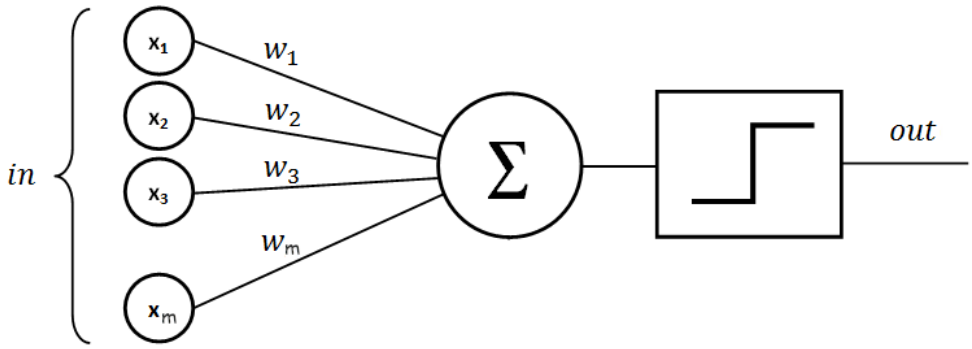
3.1
Deep Learning

Deep Learning is a branch of machine learning responsible for studying complex architectures, with the pourpose of learning patterns in data representations. The *deep* in the name refers to the depth of layers in the architectures, as it can be very deep.

Most of the models in Deep Learning are composed of Artificial Neural Networks. Those neural networks were first proposed by McCulloch and Pitts [McCulloch1943], in 1943, and were inspired by the processing and communication patterns in biological nervous systems, although there are many differences and even more from human brains. Those differences make the study of Artificial Neural Networks incompatible with the study of neuroscience.

3.1.1
The perceptron

The first ANN model was implemented by Frank Rosenblatt [Rosenblatt1958], and was called *the perceptron*. The perceptron was composed of multiple inputs, an output, a collection of weights and an activation function. It was capable of learning binary classification by supervised learning.



Mathmatically, the perceptron is a function that maps the input $\boldsymbol{x}$ to a binary output $\boldsymbol{f(x)}$.

$$f(x) = \begin{cases} 1 & \text{if } \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

$\boldsymbol{w} \cdot \boldsymbol{x}$ is the dot product $\sum_{i=1}^{m} w_i x_i$,
w is a vector of weights
m is the number of inputs
b is the bias

This model was heavily criticized because it couldn't learn a simple XOR function, as it can only simulate linearly separable functions. To overcome this, Rumelhart, Hinton and Willians [Rumelhart1986] proposed the Backpropagation algorithm, which implemented another layer and finally was capable of simulating non-linearity.

Nowadays, Artificial Neural Networks are composed of multiple neurons, and multiple layers of neurons. Each neuron works similar to a perceptron.

3.1.2 Activation Functions

As the name sugests, the activation functions are responsible for setting the activation of a neuron, in other words, it sets how the input signal is passed to the output.

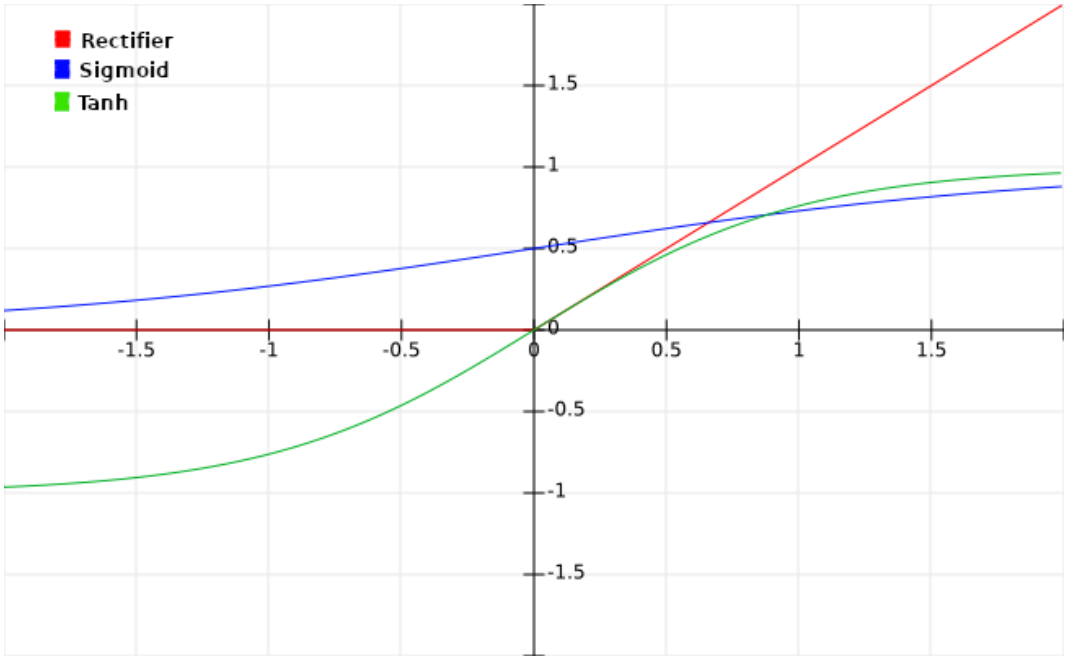
The first activation function, proposed by McCulloch and Pitts, was the threshold fuction. This function is binary, it is equals to 1 (passes the signal) if the input signal is greater than 0, and it does not passes the signal otherwise, as in above perceptron description.

Another common activation function is the sigmoid. Unlike the threshold function, the sigmoid function is smoother, and can be mathmatically described as $\phi(x) = 1/(1 + e^{-x})$. This function is usually used in the last layer of classification models to describe the probability of the signal being 0 or 1, and that's where the smoothness is helpfull instead of the binary threshold.

Then, there is the rectifier function $\phi(x) = max(0, x)$. Nowadays, it is one of the most popular functions for Artificial Neural Networks, thanks to the work of Glorot, Bordes and Bengio, who proved it allowed faster and better training of ANNs [Glorot2010].

Finally, there is the hyperbolic tangent function, which is similar to the sigmoid function, but it goes from 1 to -1. The sigmoid function can "stuck" the training of ANNs, because when a strongly-negative input is provided, the sigmoid function outputs values very near to zero, inactivating the neuron instead of passing a negative signal [Glorot2010.2]. Thus, the hyperbolic

tangent function is usually preferable. Mathmatically, it can be described as $\phi(x) = (1 - e^{-2x})/(1 + e^{-2x})$.

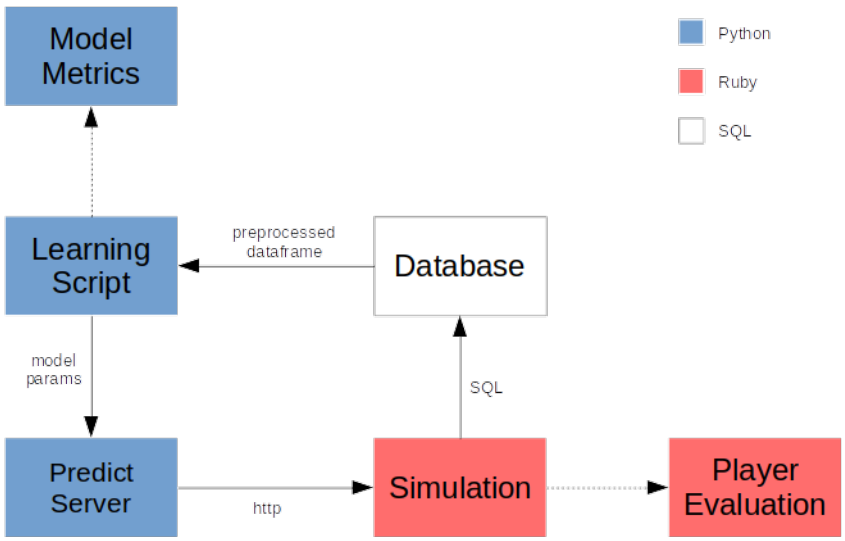


3.2 Reinforcement Learning

4
Architecture

The Pucker framework has 4 components: a no limit Texas Hold'em simulation written in JRuby [jruby.org], a SQLite storage [sqlite.org], a learning and a prediction script written in Python programming language [python.org].

The simulation runs the game, inserting data about the states seen by a player and his rewards (or punishments if negative) in the database. The states and rewards are the learning variables. The learning script reads the database, fits the model, and stores the model parameters in disk. The prediction script loads the model and exposes predictions through an HTTP API. The simulation queries the prediction API when a machine learning player needs to take a decision.



4.1
Simulation

The Ruby programming language was chosen to write the simulation component. Ruby offers a great syntax to write game simulations because it is idiomatic, succinct, and object-oriented.

The game of poker, as many other games, is composed of independent reusable components that answer to messages, such as player, dealer and a deck

of cards. According to Ampatzoglou and Chatzigeorgiou [Ampatzoglou2007], games demand great flexibility, code reusability and low maintenance costs. Consequently, the application of design patterns in them can be beneficial. Ruby is heavily object-oriented and allowed fast development of the simulation.

Due to its idiomacy, the main method of the simulation, the *play* method on the *Game* class, can be understood by anyone. It seems like an english description of the poker game:

Listing 4.1: game.rb

```
def play
  setup_game
  collect_blinds

  # FLOP
  3.times { deal_table_card }
  bets = collect_bets
  main_pot.merge!(bets)

  # TURN
  deal_table_card
  bets = collect_bets
  main_pot.merge!(bets)

  # RIVER
  deal_table_card
  bets = collect_bets
  main_pot.merge!(bets)

  winners = eligible_players_by_rank
  reward winners

  rotate_and_reset_states
end
```

A simple random player can be written in few lines of ruby:

Listing 4.2: game.rb

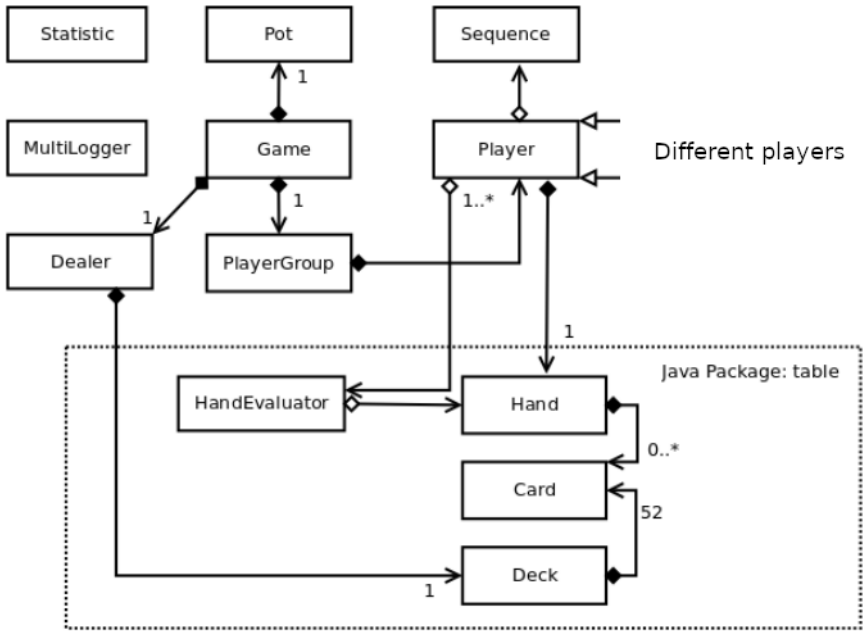
```
class DummyPlayer < Player
  def bet(state)
    min_bet = state.min_bet

    choice = rand

    if min_bet > 0 && choice < 1/3.0 # FOLD
      fold
    elsif choice < 2/3.0 # CHECK
      get_from_stack(min_bet)
    else # RAISE
      raise_from(min_bet)
    end
  end
end
```

Through the past years, many reusable poker components were written by the Computer Poker Research Group, at the University of Alberta [spaz.ca/poker/doc]. Those components were written in the Java programming language. To reuse those components, we chose the JRuby platform to run the poker simulation. This platform runs the Ruby syntax on the Java Virtual Machine [jruby.org], and simplifies the calling of Java poker components from Ruby simulation code.

In the context of the Pucker framework, a game has a group of players, a dealer and a pot with the bets of the current game. The dealer deal cards. Players evaluate game state and choose an action based on their current state.



As previously said (see 2.1), Texas Hold'em poker variant has 4 phases: pre-flop, flop, turn and river.

In pre-flop, a player have to take an action with little information about its opponents, since few bets have been committed to the pot. Given that, in pre-flop a player must deal with more imperfect information than in further rounds of the game. Due to its complexity, this research have abstracted the pre-flop phase: every player bets the same as the big-blind player (see 2.1.2).

4.2 Storage

In real poker, there are many game variables, such as how much time a player delayed to take a decision, history of opponent's decisions, cards on the table, cards in player's private hand, how strong is the combination of table and hand cards, player position, and many others.

In Pucker, we consider a state as composed by:

- Count of players
- Count of active players in this round (players who have not folded)
- Player's position
- Amount in the pot
- Amount each player has bet

- Number of raises per player
- Self amount committed to the pot
- Self number of raises
- Hand cards
- Table cards
- Combination of Hand and Table ranking
- Combination of Hand and Table strength
- Combination of Hand and Table potential
- Action taken

The reward (or punishment if negative) a player has received in the end of a round is also part of the state. This is the dependent variable our models are trying to predict. Pucker is an extensible framework, it is easy to add or remove variables to the state in future work.

In poker, a player sees a game state, takes an action, and receives a reward (or punishment) in the end of the round. Pucker players remember the states, actions and rewards, and stores them in a SQLite database, one row per action taken.

This is a complex game. To learn a fine strategy, a machine learning player must be fitted from a very large dataset. To accomplish that, the simulation needs to be fast and can't be stopped every round by a slow operation such as database inserts. To overcome this problem, a batch of states are written at once in a single insert query.

4.3 Learning

A machine learning prediction typically maps a set of variables to an output. In this work, the input will be the poker state and the output will be the reward seen in the end of this state. In short, the machine learning algorithm will learn to predict the reward, given a state and an action.

To choose the right action, a machine learning player will predict the reward of different actions (fold, call, raise), and choose an action that maximises its rewards. This is inspired in Reinforcement Learning, but without policy optimization [Silver2016].

Since the number of states in no limit Texas Hold'em is 10^{160} [Johanson2013], it is impossible to store every state. It is even hard to store the number of states to take a good decision. To overcome this, we

will store only the model parameters and erase the database of states after the learning process.

It is crucial for the model to be extensible: it may correct its parameters according to new data, it will not be able to access the full history. Neural Networks are naturally capable of incremental learning, and this is a big reason of their choice in favor of other models, like Gradient Boosting Machines. According to [xgboost-github], until today popular Gradient Boosting Machine implementations cannot handle inclemental learning.

Instead of considering the game round (pre-flop, flop, turn or river) a variable of the state, past work [Sirin2008] and [Moravcik2017] has seen better results by creating a separate model for different rounds, and we will also consider that. The main reason is that different rounds of poker are played in very different ways because they have a different state. Additionally, the number of possible states in poker is huge [Johanson2013], by using different models in different rounds we are reducing the number of possible states a model has to learn from.

The learning component is a Python script that reads states from a database, preprocess, fits a prediction model and stores its parameters in disk.

4.3.1 Neural Networks Architectures

4.3.2 Bayesian Networks and Initial Dataset

In order to train a supervised model, it is needed a dataset. The performance of a machine learning model relies heavily on the quality of this dataset [Polikar2001]. A realiable dataset must represent reality with confidence, thus it should be generated from real situations.

In Pucker, after the models are trained, new datasets can be generated from subsequential games played by those models. But still remains the problem of how to generate a initial dataset with good quality.

Untrained Neural Networks could be put to play. Since their parameters are not fitted yet, they will play randomly according to the random generation of its initial parameters. The quality of this dataset is worse than a dataset generated from good players, because it represents random situations, hardly seen in professional games.

Pucker Framework introduces a initial set of simple players that are put to play against each other. Rather than playing randomly, those players use Bayesian Networks [Heckerman1998] to analyse state and choose better poker

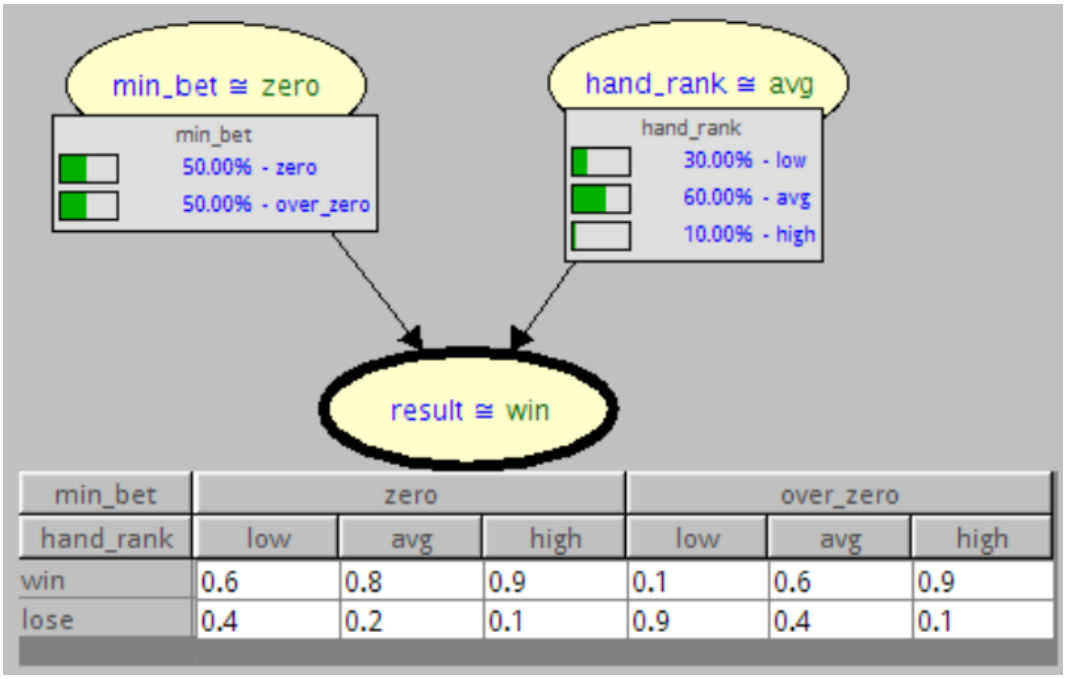
actions. To certify that Bayesian players perform better, they were put to play against random players and won by a large margin in the long run. They were also tested against "always check" players, and won by a large margin.

Bayesian networks are tree structures that represent conditional dependencies of variables in a directed acyclic graph, and are capable of infere from those variables [Neapolitan2003].

Nodes of this graph represents Bayesian variables. In poker, those nodes can represent the poker state: hand ranking, number of players, position, etc. Unconnected nodes represent variables that are conditionally independent of each other, like a player hand and position. Nodes are associated with a probability function that represents the chance of possible states of that node. Edges represent conditional dependencies, for example, a good poker hand depends on the private player hand and the public table cards.

After some experiments, two different architectures were chosen to compose poker players in order to generate a initial dataset. The first one in simpler, and relies only on the state minimum bet and hand rank. The second uses players' position, together with minimum bet, to compose a scenario conditional variable. When the Bayesian Network detects a high winning condition, bayesian players raise. When it detects a low winning condition, bayesian players check. If it's a losing condition, it folds, or checks if minimum bet is zero.

As an example, the bayesian player represented by figure 4.1 will fold with a bad hand when there is a minimum bet over zero. And it will re-raise when it has a great hand and the minimum bet is over zero.



min_bet	zero			over_zero		
hand_rank	low	avg	high	low	avg	high
win	0.6	0.8	0.9	0.1	0.6	0.9
lose	0.4	0.2	0.1	0.9	0.4	0.1

Figure 4.1: Simple bayesian network with its probabilities

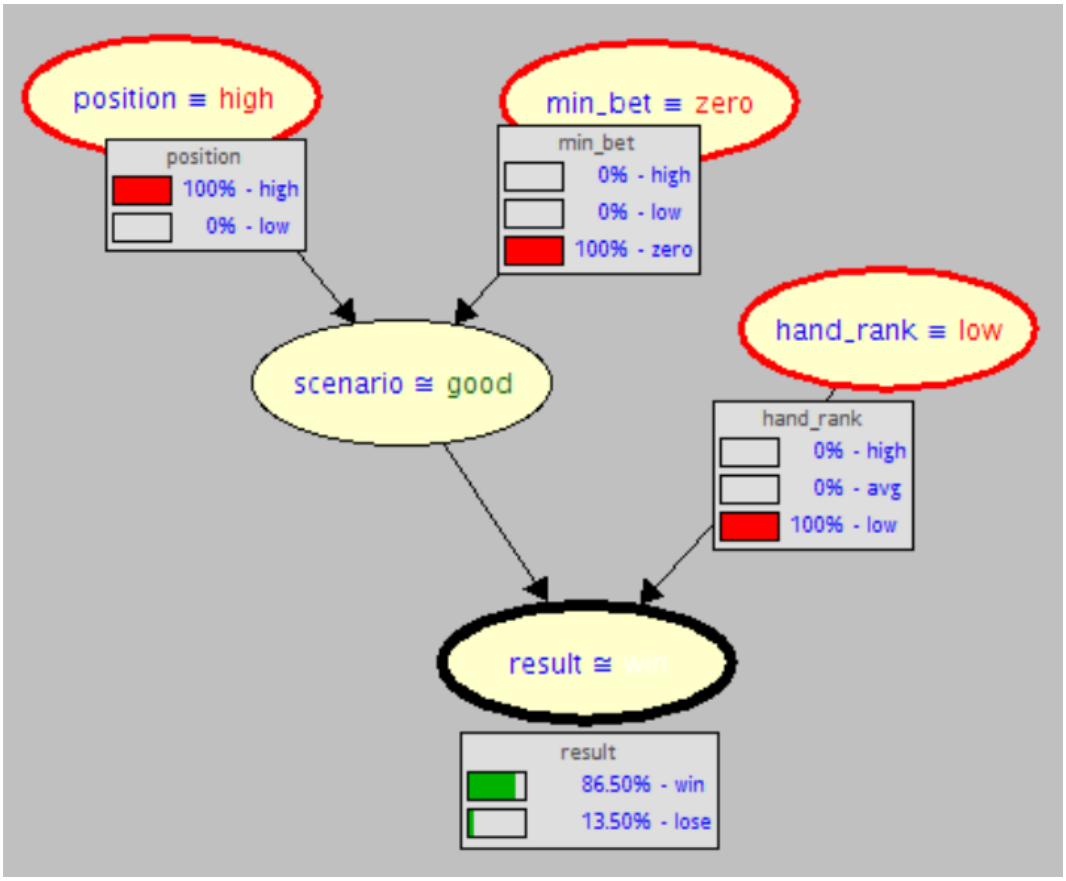


Figure 4.2: Better bayesian network. It is represented a situation of high position, zero bets and low hand rank. This player might bluff in this situation.

Those two bayesian networks played 40000 games in a five players Texas Hold'em table. This table also had a random player to generate diverse situations in order to generalize the initial dataset. The result is a 566739 row database that was used to train the initial neural networks. After that, the neural networks played against each other, generated even better quality datasets, and learned incrementally from their self-plays.

4.4 Prediction

The prediction component is trivial. It reads the stored model parameters and creates an instance of the model ready to predict from new states.

There are two problems: this is a slow process, and the Ruby simulation cannot run Python code seamlessly, as they are different languages running in different virtual machines. To overcome those problems, the prediction component keeps an instance of the model in memory and exposes an HTTP API that receives a state and returns a prediction.

This component is capable of exposing many different models, one per HTTP endpoint. This way, Pucker supports the simulation of many different machine learning algorithms playing at the same time.

To build the HTTP API, Pucker uses the Flask library [flask.pocoo.org].

5
Evaluation

5.1
Model metrics

5.2
Poker statistics

6
Conclusions

This chapter presents the concluding remarks about our work. Discuss our contributions in Section 6.2, and present directions for future works in Section 6.3.

In this thesis, we presented a novel graphical approach to scenario reduction on time series ensembles. We evaluated the feasibility of our proposal by performing an empirical study with a series of potential users, both experienced in the area and not. By observing their interactions and interviewing them afterwards, we obtained valuable insights on the usefulness of our proposal. Following from the results of our previous publication we have expanded our work in two aspects: (i) use glyph sizes to represent the time in the Time-lapsed LAMP chart; (ii) in the same chart, encode the uncertainty inherent to the data. The first expansion was done in order to fix an issue related to the abstraction created by employing multidimensional-projections using time-varying data. The second expansion aims to help users to quickly identify time ranges with high variance in the data.

Besides the user study, we also compared our results with other approaches in the literature and industry. proposes to select a set of representative scenarios under different well configurations. Their approach handles several simulation parameters and responses simultaneously, thus, selecting representative scenarios based on multiple criteria. The Industry approach selects scenarios with cumulative production closest to the target references at the end of the simulation. Both approaches are well suited depending on the post-processing tasks and the data itself. In our tests, our approach selected scenarios with consistently smaller error when compared to both the Industry and Clustering approaches. It must be stated, however, that this does not mean that our approach is better, only that it yields good scenarios when the objective function is the proximity to a reference in a context where the scenarios' evolution must be taken into consideration.

6.1
Publications

6.2 Contributions

6.3 Future Work

The remainder of this thesis is organized as follows: Chapter 2.2 presents the state-of-the-art in Artificial Intelligent poker agents. Next, in Chapter 2 we describe poker foundations and explain computer poker features that can help convergence of neural networks in early poker stages. Chapter 3 shows how neural networks work to gradually improve its predictions and how we used reinforcement learning to adapt the model to multi-step games. Chapter 4 describes the project architecture, simulation, data storage, learn and predict modules. Chapter 5 presents details about the experiments and the associated results. Chapter 6 presents some concluding remarks and directions for future work.

Bibliography

[Ampatzoglou2007] AMPATZOGLOU, A.; CHATZIGEORGIOU, A.. **Evaluation of object-oriented design patterns in game development**. Information and Software Technology, 49:445–454, 05 2007.

[Billings1998] BILLINGS, D., P. D. S. J.; SZAFRON, D.. **Poker as a Testbed for Machine Intelligence Research**. In: PROCEEDINGS OF THE 12TH BIENNIAL CONFERENCE OF THE CANADIAN SOCIETY FOR COMPUTATIONAL STUDIES OF INTELLIGENCE ON ADVANCES IN ARTIFICIAL INTELLIGENCE, p. 228–238, june 1998.

[Billings2016] D. BILLINGS, D. PAPP, J. S.; SZAFRON, D.. **Opponent modeling in poker**. In AAAI National Conference, 1998.

[Bowling2008] M. BOWLING, N. BURCH, M. J.; TAMMELIN, O.. **Heads-up limit hold'em poker is solved**. In: SCIENCE, 347 (6218), 2015.

[Bronowski1973] BRONOWSKI, J.. **The ascent of man, Documentary Episode 13.**, 1973.

[Brown2015] N. BROWN, T. S.. **Claudico: The World's Strongest No-Limit Texas Hold'em Poker AI**. In: IN PROCEEDINGS OF THE NEURAL INFORMATION PROCESSING SYSTEMS: NATURAL AND SYNTHETIC (NIPS) CONFERENCE, DEMONSTRATION TRACK, 2015.

[Brown2017] N. BROWN, T. S.. **Libratus:The superhuman ai for no-limit poker**. In: IN PROCEEDINGS OF THE TWENTY-SIXTH INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, IJCAI-17, 5226–5228, 2017.

[Davidson2000] A. DAVIDSON, D. BILLINGS, J. S. D. S.. **Improved opponent modeling in poker**. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE (IC-AI'2000), LAS VEGAS, NV, 2000.

[Glorot2010] X. GLOROT, A. B.; BENGIO, Y.. **Deep sparse rectifier neural networks**. volumen 15, 01 2010.

[Glorot2010.2] GLOROT, X.; BENGIO, Y.. **Understanding the difficulty of training deep feedforward neural networks**. In: PROCEEDINGS OF THE THIRTEENTH INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND STATISTICS, volumen 9 de **Proceedings of Machine Learning Research**, p. 249–256, May 2010.

[Heckerman1998] HECKERMAN, D.. **A Tutorial on Learning with Bayesian Networks**, p. 301–354. Springer Netherlands, Dordrecht, 1998.

[Johanson2013] JOHANSON, M.. **Measuring the size of large no-limit poker games**. Technical Report TR13-01, Department of Computing Science, University of Alberta, march 2013.

[Kocsis2006] L. KOCSIS, C. S.. **Bandit Based Monte-Carlo Planning**. In: PROCEEDINGS OF THE SEVENTEENTH EUROPEAN CONFERENCE ON MACHINE LEARNING, 4212, p. 282–293. Springer-Verlag Berlin Heidelberg, 2006.

[McCulloch1943] MCCULLOCH, W. S.; PITTS, W.. **A logical calculus of the ideas immanent in nervous activity**. The bulletin of mathematical biophysics, 5(4):115–133, Dec 1943.

[Moravcik2017] M. MORAVCIK, M. SCHMID., B. N. L. V. M. D. B. N. D. T. W. K. J.-H. M.; BOWLING, M.. **Deepstack: Expert-level artificial intelligence in heads-up no-limit poker**. In: SCIENCE 356, p. 508–513, 2017.

[Neapolitan2003] NEAPOLITAN, R. E.. **Learning Bayesian Networks**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[Polikar2001] R. POLIKAR, L. UPDA, S. U.; HONAVAR, V.. **Learn++: An incremental learning algorithm for supervised neural networks**. Trans. Sys. Man Cyber Part C, 31(4):497–508, Nov. 2001.

[Rosenblatt1958] ROSENBLATT, F.. **The perceptron: A probabilistic model for information storage and organization in the brain**. Psychological Review, p. 65–386, 1958.

[Rubin2009] J. RUBIN, I. W.. **SARTRE: System Overview, A Case-Based Agent for Two-Player Texas Hold'em**. In: WORKSHOP ON CASE-BASED REASONING FOR COMPUTER GAMES, EIGHTH INTERNATIONAL CONFERENCE ON CASE-BASED REASONING (ICCBR 2009). Springer, Heidelberg, 2009.

[Rumelhart1986] D. E. RUMELHART, G. E. H.; WILLIAMS, R. J.. **Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1**. chapter Learning Internal Representations by Error Propagation, p. 318–362. MIT Press, Cambridge, MA, USA, 1986.

[Silver2016] J. HEINRICH, D. S.. **Deep reinforcement learning from self-play in imperfect-information games**. NIPS Deep Reinforcement Learning Workshop, 2016.

[Sirin2008] V. SIRIN, A. P.. **A machine learning approach to the poker playing problem**. 2008.

[Tammelin2015] O. TAMMELIN, N. BURCH, M. J.; BOWLING, M.. **Heads-up limit hold'em poker is solved**. In: SCIENCE VOLUME 347, p. 145–149, january 2015.

[Zikenvich2008] M. ZINKEVICH, M. JOHANSON, M. B.; PICCIONE, C.. **Regret minimization in games with incomplete information**. In: NEURAL INFORMATION PROCESSING SYSTEMS, VOLUME 21, 2008.

[flask.pocoo.org] **Flask is a microframework for python based on werkzeug, jinja 2 and good intentions**. `http://flask.pocoo.org/`. Accessed: 2019-01-15.

[jruby.org] **The ruby programming language on the jvm**. `https://www.jruby.org`. Accessed: 2019-01-15.

[python.org] **Python is a programming language that lets you work quickly and integrate systems more effectively.** `https://www.python.org/`. Accessed: 2019-01-15.

[spaz.ca/poker/doc] **Package ca.ualberta.cs.poker**. `http://spaz.ca/poker/doc`. Accessed: 2019-01-15.

[sqlite.org] **What is sqlite?** `https://www.sqlite.org/index.html`. Accessed: 2019-01-15.

[xgboost-github] **Is it possible to update a model with new data without retraining the model from scratch?** `https://github.com/dmlc/xgboost/issues/3055`. Accessed: 2019-01-15.