Pontifícia Universidade Católica
do Rio de Janeiro

Alexandre Marangoni Costa

A Study on Neural Networks for 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
April 2019

Pontifícia Universidade Católica
do Rio de Janeiro

Alexandre Marangoni Costa

A Study on Neural Networks for 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. Thibaut Victor Gaston Vidal
Departamento de Informática – PUC-Rio

Prof. Bruno Feijó
Departamento de Informática – PUC-Rio

Rio de Janeiro, April 4^{th}, 2019

Alexandre Marangoni Costa

Bachelor's in Computer Engineer (2013) at the Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). In 2009, was scholarship holder of scientific initiation in mathematics of computer graphics. In 2010, worked in Laboratório de Inteligência Computacional Aplicada (ICA), which belongs to the Department of Electrical Engineering (DEE PUC-Rio). Joined the Master Program in Optimization and Automated Reasoning at the Department of Informatics (DI PUC-Rio), in 2017.

Bibliographic data

Costa, Alexandre Marangoni

A Study on Neural Networks for Poker Playing Agents / Alexandre Marangoni Costa; advisor: Marcus Vinicius Soledade Poggi de Aragão. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2019.

v., 59 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 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 real.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) Finance Code 001 and in part by Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ).

Abstract

Costa, Alexandre Marangoni; Poggi de Aragão, Marcus V.S. (Advisor). **A Study on Neural Networks for Poker Playing Agents**. Rio de Janeiro, 2019. 59p. 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.. **Um Estudo em Redes Neurais para Agentes Jogadores de Pôquer**. Rio de Janeiro, 2019. 59p. 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

ANN – Artificial Neural Networks
BN – Bayesian Networks
DL – Deep Learning
ML – Machine Learning
RL – Reinforcement Learning

"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 estimated to be 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:

Figure 1.1: Real world problems and poker aspects from [Billings1998]

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 follows, it wins. If it deviates, it loses. 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 the reinforcement learning, 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 discuss incremental learning using our methodology and framework, and show statistics of players from last and first phase to support our findings.

In summary, our contributions are:

- A framework and methodology to build Texas Hold'em agents and to control incremental learning of agents;
- A simulation of the game that can be used to generate input dataset in a controlled way;
- State representation and feature engineering 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**, **machine learning**, and the **related work**;

– Chapter 3 describes the research **methodology** and activities;

– 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**.

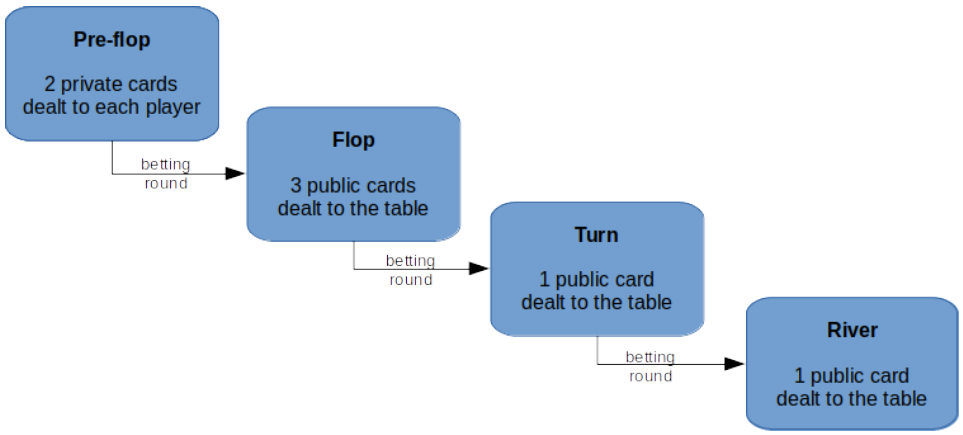
Next, 3 public cards are dealt to the table. This is called **flop**. The flop is public, and those 3 cards are shared among all players. A betting round happens again, but now the players bet who has the best combination of private hand and shared table, a total of 5 cards. Every bet goes to the table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

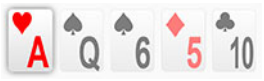
The bet can be a fold, a call of the previous bet or a raise. When a player folds, he is out of this game and loses the amount he has previously given to the table pot. Each player announces it's bet starting from the first player after the dealer until every player but one fold or every active player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

Figure 2.1: Sequence of rounds in Texas Hold'em Poker variant



2.1.1 Game Knowledge

The hands are listed below 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.

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 at 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 the dealer, small-blind, and big-blind, once every 5 games.

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**

2.1.2 Game Variants

Poker variants usually differ in orthogonal dimensions. The two most common are the 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 the 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 Machine Learning

Arthur Samuel coined the term "machine learning" in 1959, to describe "the fact that a computer can be programmed so that it will learn to play a better game of checkers than can be played by the person who wrote the program" [Samuel1959]. It was attributed to him the question "how can computers learn to solve problems without being explicitly programmed?".

Nowadays, machine learning is considered to be a subset of artificial intelligence techniques that build models from a set of data in order to make predictions and take decisions. Some ML applications worth mentioning are financial market analysis, general game playing, image recognition, medical diagnosis, natural language processing, online marketing, and recommendation systems [wikipedia/machinelearning].

Machine learning algorithms are often divided into supervised learning and unsupervised learning.

- **Supervised learning** algorithms build models from sample data that contains the desired output for each input and are able to generalize outputs to unseen examples. For example, in the game of poker, a possible input is the player hand and the bets, the output is the player gain or loss of chips. By building a supervised model out of a sufficiently large dataset, it's possible to predict the gain or loss of chips of an unseen scenario, with relative confidence.
- **Unsupervised learning** algorithms build models from inputs without the need of output data. Instead of focusing on the prediction, unsupervised models learn patterns of the data, like groups or clusters of data points. For example, in the game of poker, unsupervised models could be used to group similar scenarios in buckets, in order to reduce the search space and take a faster decision.

Another common subset of machine learning algorithms is the **reinforcement learning** paradigm. Those algorithms learn by taking decisions in dynamic environments, which result in positive or negative feedback received by the environment. By accumulating enough data about states, actions and feedbacks received, they optimize a policy in order to maximize the number of positive reinforcements. For example, in the game of poker, a reinforcement model could start by taking dummy decisions to states given and, by observing the chips gained or lost for each state-action tuple, it changes its strategy in order to receive more chips.

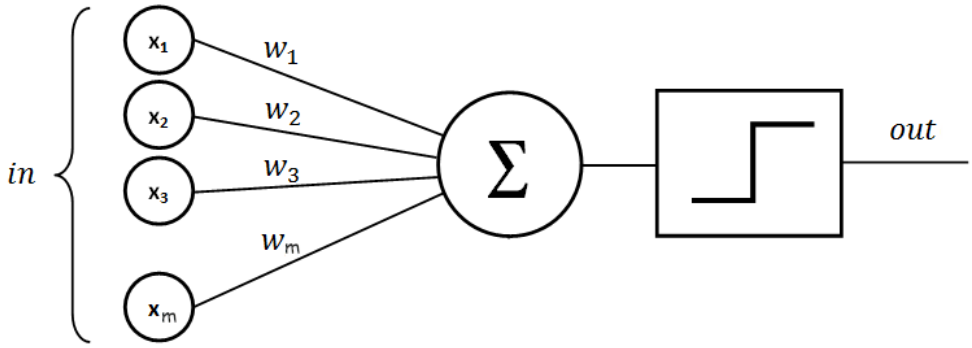
2.2.1 Neural Networks

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 from those biological systems and even more from human brains. Those differences make the study of Artificial Neural Networks incompatible with the study of neuroscience.

2.2.1.1 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.

Figure 2.2: The perceptron diagram



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

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

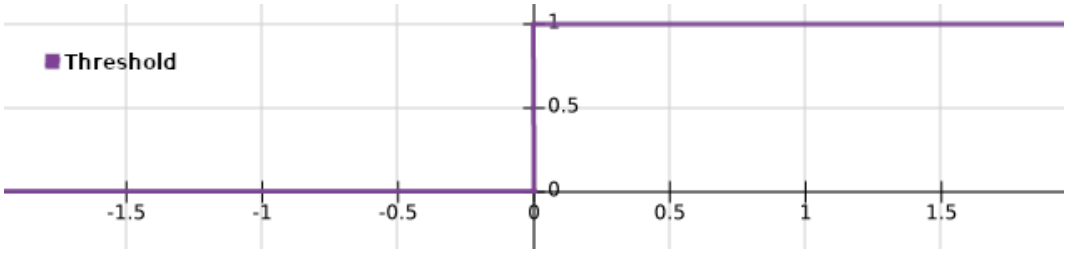
$\boldsymbol{w} \cdot \boldsymbol{x}$ is the dot product $\sum_{i=1}^{m} w_i x_i$,
w is a vector of weights
m is the number of inputs
b is the bias

This model was heavily criticized because it couldn't learn a simple XOR function, as it could 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.

As the name suggests, 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** function. This function is binary, it is equal to 1 (passes the signal) if the input signal is greater than 0, and it does not pass the signal otherwise, as in above perceptron description.

Figure 2.3: Threshold activation function

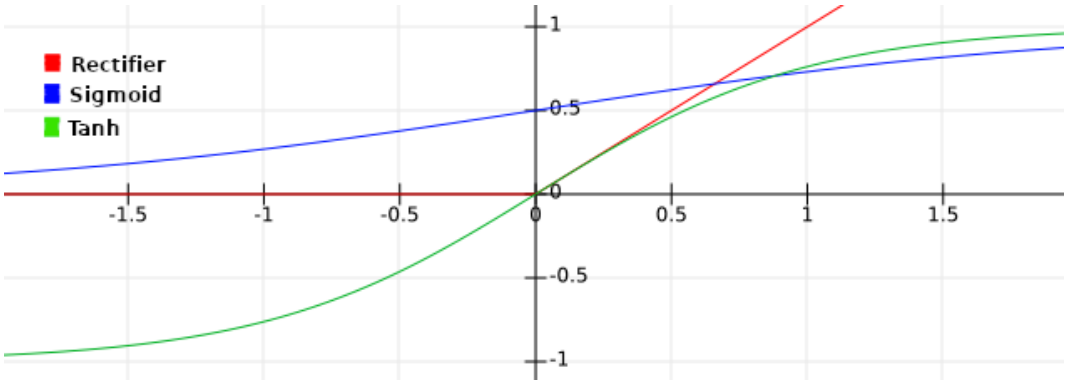


Nowadays, one of the of the most popular activation functions is the **rectifier** function $\phi(x) = max(0, x)$. The reputation is due to the work of Glorot, Bordes, and Bengio, who proved it allowed faster and better training of ANNs [Glorot2010].

Another common activation function is the **sigmoid**. Unlike the threshold function, the sigmoid function is smoother, and can be mathematically described as $\phi(x) = 1/(1 + e^{-x})$. This function is often used in the last layer of classification models to estimate the probability of the signal being 0 or 1 [Richard1991], and that's where the smoothness can be helpful instead of the binary threshold.

Finally, there is the **hyperbolic tangent** function, also referred as *tanh*. It is similar to the sigmoid function but goes from 1 to -1. The sigmoid function can "stuck" the training of ANNs, a problem called "vanishing gradients". When a strongly-negative input is provided, the sigmoid function outputs values near to zero instead of passing a negative signal [Glorot2010.2], which results in a very slow learning curve. Thus, the hyperbolic tangent function is usually preferable. Mathematically, it can be described as $\phi(x) = (1-e^{-2x})/(1+e^{-2x})$.

Figure 2.4: Rectifier, sigmoid and tanh activation functions



2.2.1.2 Backpropagation

Right now, one can understand that a multilayer perceptron simulates the behavior of any function, given the right weights. But how to arrive at the proper weights?

One way could be by brute-force: trying out an infinite number of weight combinations until the network outputs the right values of the function for any input. This is not feasible because it could take an eternity. If we have 25 neurons, and we try 1000 different weights for each neuron, there are a total of 10^{75} different combinations of weights for the network. As of today, the world's fastest supercomputer, Summit, can process 200 petaflops per second. Let's say hypothetically that it can test the result of a neural network in one floating operation, which in reality is a lot more. With this optimistic estimation, it would require $10^{75}/(200 \times 10^{15}) = 5 \times 10^{57}$ seconds to test all the weights possibilities. That's approximately 1.58×10^{50} years, longer then the time the universe has existed. And this is not even considering that the right weights might be different from the 1000 different weights tried out.

Backpropagation is an algorithm that relies on gradient descent to correct the weights. First, the cost function is chosen. The cost function describes how to calculate the error between the predicted output and the real output. A common cost function is the mean squared error: $Cost = 0.5 \times (y_{pred} - y_{real})^2$. The cost function shows how far the prediction is from the desired output. With this measure, the backpropagation algorithm uses gradient descent to calculate how much to change from the network parameters w in order to minimize the cost function, according to a learning rate α, $w = w - \alpha \nabla_w C$. When the error is acceptable, the backpropagation stops and the neural network is considered to be trained.

2.2.1.3 Data Mining and Preprocessing

When a data scientist receives third-party data, it usually comes with important information for the owners of the data, like database ids, meta-information that concerns the business problems, unnecessary timestamps, etc. The quality of a machine learning model depends heavily on the input data. When the input dataset is full of unnecessary, noise or wrong data, the models built from this data will reflect that and output unreliable information. An important part of building a consistent model comes before thinking about the architecture of the model and it's called data mining.

The goal of data mining is to extract crucial information from a database into a more comprehensible structure [Pang2018]. Thus, data mining concerns with all the techniques involved in the preparation of a dataset to be fed into a machine learning model. Some of those concerns are data access, data preprocessing, metrics extraction, data dimension, and data visualization. This work will focus on data preprocessing and metrics extraction.

Data preprocessing is referred to as the process of cleaning the data before feeding to a machine learning model. This usually involves:

- **Data filtering:** to remove unnecessary information, from which the output does not depend on;
- **Feature engineering:** to create new features that make models work better;
- **Handling of missing data:** to discard incomplete observations or transform missing variables using statistical values such as the mean, median or most frequent values;
- **Categorical data encoding:** to transform categorical data into new dimensions;
- **Feature scaling:** to standardize a variable of the data;

The first three are self-explanatory. Below there are some examples to understand encoding and scaling.

Given a database with people information, one possible variable is the person's genre. Some might be "male" and others "female". The string "male" and "female" have little numerical value. Data encoding is the process of transforming this information into numbers, for example, "male" to 0 and "female" to 1. If there are only two categories, this is sufficient to feed into a mathematical model.

A more complex problem arises when some of the observations contain the "other" string, referring to people that do not consider themselves either of those two genres. One might try to set "other" as 2, but this is a big mistake because now the data have scalar meaning: $2 > 1 > 0$. This confuses most of the known machine learning models, as they are essentially numerical. When there are only two categories, the numerical models work because they treat 1 as different from 0, not higher, thus enabling them to mathematically separate observations.

To solve this problem, a data scientist has to transform this information into new dimensions. He might substitute the genre dataset column into a "female" and a "male" column, and set 1 to men's "male" column, 0 to men's "female" column, 0 to women's "male" column, 1 to women's "female" column and finally 0 and 0 to "male" and "female" columns of the remaining people data. Notice that there is no need for a third column since the last step fully differentiates people who don't consider themselves either "male" or "female". This removes the scalar order from the categorical variable, hence contributing to the learning of a numerical model.

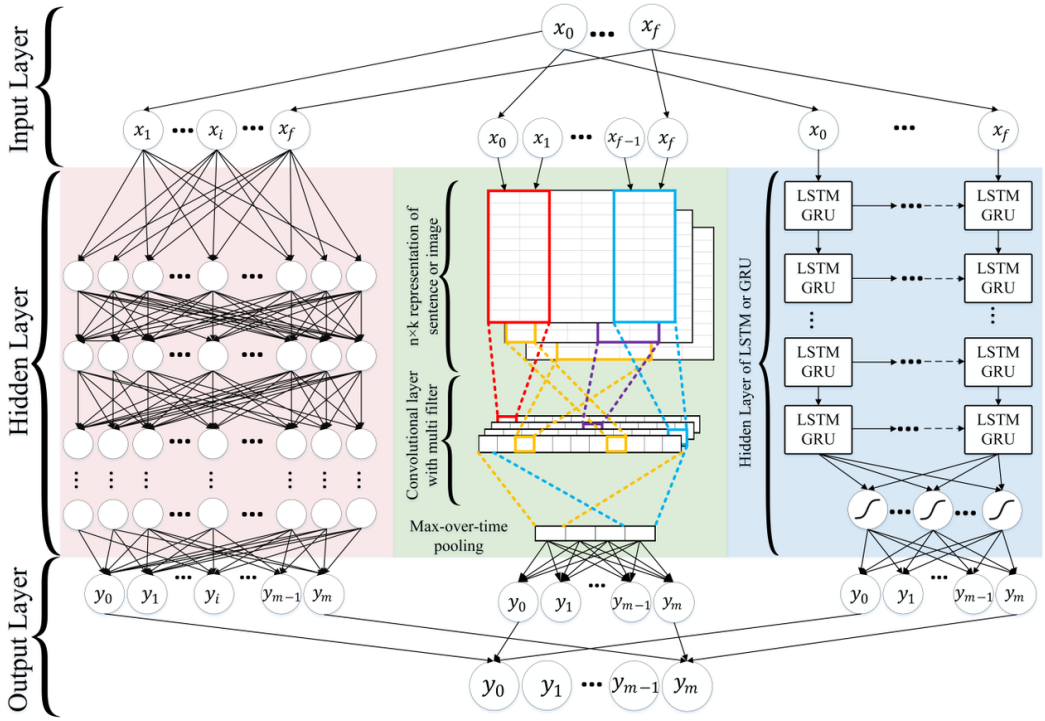
Feature scaling concerns about the mean and standard deviation of features. Depending on how an ML model works, a dataset feature with much higher value can dominate other features and disturb the model learning. For example, in neural networks (see figure 2.2), if the feature x_1 has observations from 0 to 1000 and the other features have a range from 0 to 1, the first feature will dominate the dot product $\sum_{i=1}^{m} w_i x_i$, as it is more sensible to a change in w_1 than in other weights.

2.2.1.4 Deep Learning

Nowadays, many Artificial Neural Networks are composed of multiple neurons, multiple layers of neurons, multiple types of layers such as convolutional or recurrent layers [Lavet2018.1], and can predict complex scenarios, for example, house pricing, insurance cost, weather forecasting, stock market prices, etc. Each neuron in a complex neural network architecture works in a similar way to a perceptron.

Deep Learning is a branch of machine learning responsible for studying complex architectures, with the purpose of learning patterns and making predictions out of data representations. The *deep* in the name refers to the depth and complexity of the architectures, as they can be very deep.

Figure 2.5: Diagram of a random multimodel deep learning network, from [wikimedia/deeplearning]



2.2.2 Reinforcement Learning

Reinforcement learning is rather known as a class of problems instead of a set of techniques. This class of problems has in common an agent interacting with a dynamic environment by observing the scenario, taking actions and receiving rewards (or punishments) [Kaelbling1996].

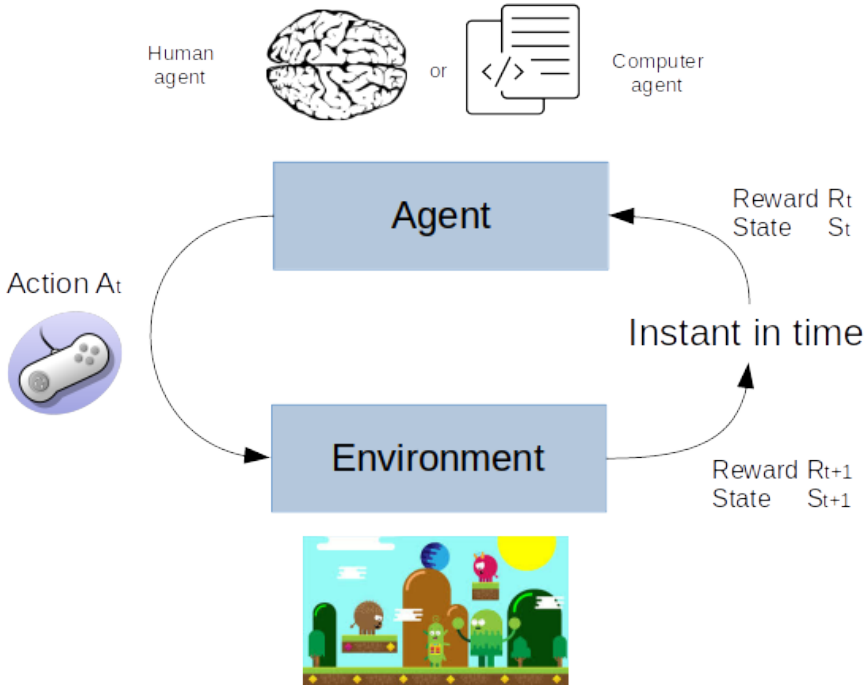
For example, in order to play a new video game, human players watch the screen, press a joystick button, receive/lose game points and watch the screen again, in a repetitive cycle. By trial-and-error, human players understand which actions led to more game points and which ones resulted in a loss. After some rounds of learning, they change their trial-and-error behavior, choosing more effective actions to win the game.

Reinforcement learning computer agents embrace this workflow. At each step, they observe a state, take an action, receive a positive/negative signal, and observe the new state from the external environment. Their goal is to optimize their future behavior to maximize the signals received.

This parallel between the computer and biological systems is not by chance. Since a long time, reinforcement learning plays a big hole in psychological studies about how neuro systems learn, through positive

and negative reinforcements controlled by the dopamine system in animal brains [Holroyd2002] [Dayan2002] [Reynolds2001] [Garcia1966].

Figure 2.6: Reinforcement learning diagram



Formally, basic RL is modeled as a Markov decision process:

- The set of possible states S
- The set of possible actions A
- The probability of going from a state s to s', by taking an action a: $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$
- The reward after transition from s to s' with action a: $R_a(s, s')$

At each step, an agent receives a state s_t, take an action a_t, receive a reward r_t and a new state s_{t+1}. And the process moves on to another step. The agent aims to receive as many rewards as possible.

The policy π is the probability of taking an action a given a state s: $\pi(a|s) = P(a_t = a|s_t = s)$. The goal of an agent is to find the best possible policy. To accomplish this, it calculates a value function for the policy given a state, $V_\pi(s)$. In other words, the value function is the expected return starting with a state s and following the policy π. Finally, it attempts to find a policy that maximizes the return by maintaining a set of estimates of expected returns for some policy.

In many cases, the reward remains zero until a step in the future. For example, a poker agent might receive its rewards only after the river

round of bets. Hence, $R_a(s, s')$ is associated with future discounted rewards $R = \sum_t^\infty \gamma^t r_t$, where γ is the discount $\gamma \in [0, 1]$. From this equation, the sum is more sensible to earlier rewards, since $\gamma^t > \gamma^{t+1}$

Usually, the agent is capable of observing the state completely, but that's not always the case. For example, in chess games, the whole state is observable, but in poker, an agent cannot see its opponents' cards. This is called a partially observable environment.

Reinforcement Learning differs from supervised learning because there is no prior need of a dataset of inputs and outputs and sub-optimal actions are not explicitly corrected. In RL, the focus is on performance and in the balance between exploration (obtaining information about the environment states, actions and rewards) and exploitation (to maximize the rewards given current knowledge). It differs from unsupervised learning in its goals. While unsupervised learning focuses on finding patterns and differences between data points, RL's goal is to find a good behavior.

Common RL implementations use dynamic programming to find the best policy. Deep Reinforcement Learning is the branch of RL that uses DL techniques (see 2.2.1) to find the best policy, thus maximizing the RL rewards [Lavet2018.2]. Pucker players are inspired in this concept, as it will be described in further sections.

2.3 Related Work

Most of the related work on computer poker relies on improving existing algorithms to win poker competitions against other computer agents or professional players. There is almost no work about the building blocks of a learning player, a methodology and a framework to build incrementally better players.

In University of Porto, Luís Teófilo, together with different researchers, discussed about the simulation and performance assessment of poker agents [Teofilo2013]. They also have created a framework to analyze logs of past human poker scenarios and measure progress of machine learning strategies [Teofilo2011]. Our work is different, it provides a whole pipeline for generating poker scenarios, with automatic agents instead of humans, and provides a framework and methodology to support incremental progress of automatic agents that learn from each other experiences and has no human input.

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. After that, University of Alberta, Carnegie Mellon and the 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 on 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, studies in the 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 hole in this field [Zikenvich2008], and Limit Texas Hold'em was weakly solved in 2015, by Cepheus, another AI developed at the Computer Poker Research Group at University of Alberta [Bowling2008]. Cepheus' along with other UoA's work also provide a profound discussion on how to assert the solving of extensive-form games such as poker. Those works are used to measure progress along their findings.

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 its strategy.

Libratus succeeded Claudico but was rewritten from scratch. It was built 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].

Finally, in 2017 the Computer Research Group (at the University of Alberta) 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.

Alberta's work differs from Carnegie Mellon's. Both had contributions from Oskari Tammelin for a faster and lighter Counterfactual Regret Minimization algorithm, but Alberta did a great job on optimizing the learning to run on modest machines, while Libratus and Claudico run on supercomputers. Our work run on a personal laptop, a quad-core with 2.50GHz and 16GB of RAM.

Regarding reinforcement learning, 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].

Our work is inspired by the reinforcement learning formulation, but adds domain specific logic to improve learning. Most notably, we add the notion of hand strength and hand potential to improve performance of the ML models, which were first proposed at UoA in their earlier works. Also inspired by UoA and DeepStack's work, we applied neural networks to predict our chances of winning, but focused on building a framework to support research in the still open field of multiplayer no-limit poker.

3
Methodology

The goal of this work is to propose a framework to build and test different neural networks that can play multiplayer no-limit Texas Hold'em poker against each other, learn from a supervised experience and maximize its rewards. It should be extensible to other machine learning models and preprocessing techniques.

The first step was to build the simulation of the game. We made several decisions to simplify this already complicated step, the remarkable ones are the abstraction of the pre-flop phase and the focus on a 5-player game. We took both decisions with a Software Engineering concern in mind, and they can be easily readdressed in future work to support the pre-flop round and different sizes of table.

Next, we built two trivial players as a foundation for more intelligent agents. The first one is an always-check player, called Player, which ignores the game state and place its bet as the minimum allowed bet, always. It has all the necessary methods to play the game and it is the super-class of all players. The second one, called DummyPlayer, plays randomly, choosing actions according to the generation of a random float between 0 and 1.

Those players could be used to build a dataset to feed ML models but, as discussed in section 3.1, data points built randomly or from rare situations produce less reliable ML models. To overcome this, we programmed three players using Bayesian network models. After some experiments, we chose two of them to build an initial dataset.

Afterward, we made the database component and connected it with the simulation of the game. The BN players played 40000 games, thus generating 566739 rows of state-action-reward information, from flop, turn and river poker rounds.

Then, we wrote four neural network architectures with different number of layers and nodes per layer, with the purpose of identifying the best architecture, and we created four NN-players out of those networks. Each NN-player uses three neural networks with the same architecture, one for each round of the game (flop, turn and river), hence they are fit with different data from different poker rounds, generating a total of 12 different neural networks.

Since the ML models were built in a different programming language than the simulation (more on 4.4), we developed an HTTP API to expose the model predictions. The ML players use this API to consult its ML models.

Finally, we added a statistic component to the simulation, to measure the players' performance and support the evaluation of this work.

With the system ready, we ran four phases of learning, and evaluated each player of each phase in four games of 3000 hands. The details and results are described in chapter 5. In each phase, the players learned with data generated from the previous phase. The first phase players learned from data generated from the bayesian network players.

The goal is to achieve incremental learning, thus the last phase players must win against first phase players. We also desire that machine learning players achieve a better performance than the simple bayesian network players. This demonstrates that the framework was successful in allowing the creation of increasingly good players, according to our initial goals.

3.1 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 reliable 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 an 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 an initial set of simple players that are put to play against each other. Rather than playing randomly, those players use Bayesian Networks [Heckerman1998] to analyze 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.

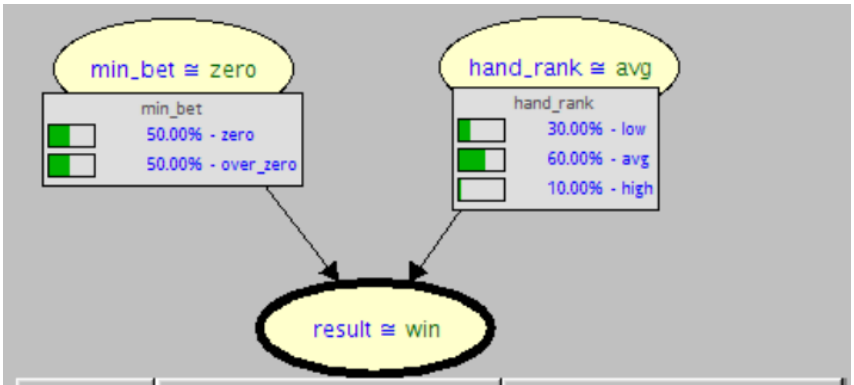
Bayesian networks are tree structures that represent conditional dependencies of variables in a directed acyclic graph and are capable of inferring from those variables [Neapolitan2003].

Nodes of this graph represent 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 an initial dataset. The first one is simpler, and relies only on the state minimum bet and hand rank. The second uses players' position, together with the 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 the minimum bet is zero.

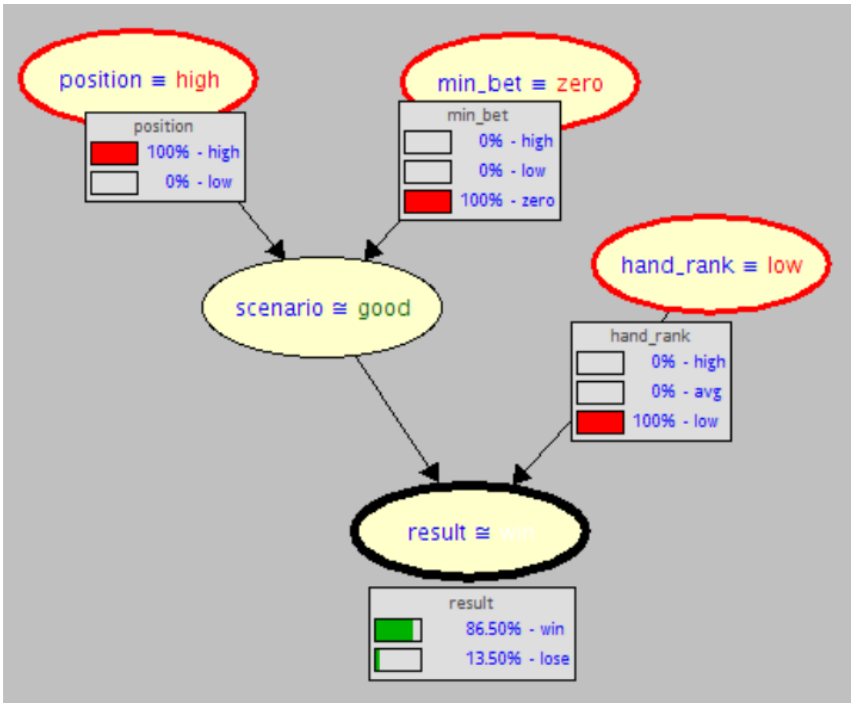
As an example, the bayesian player represented by figure 3.1 will fold with a bad hand when there is a minimum bet over zero. And it will re-raise when it has a great hand and the minimum bet is over zero.

Figure 3.1: Simple bayesian network with its probabilities



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 3.2: 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.

3.2 Application of Deep Reinforcement Learning

According to the reinforcement learning principles (more on 2.2.2), we used neural networks to predict the outcome of a poker scenario given an action. The NN-player chooses the action with the maximum expected return.

From this strategy, it's important to notice that when the model predicts:

- a win and the player wins, it's good because the player would commit to a winning scenario
- a loss and the player loses, it's good because the player would fold, thus he is less punished
- a win and the player loses, it's bad because the player would commit to a losing scenario

– a loss and the player wins, it's bad because the player would fold a winning scenario, thus denying a reward

If the model predicts a 200 chips win, and the player actually wins 150 or 300, it's less relevant. Same for a loss.

Although predicting the outcome is a regression problem instead of a classification, the above explanation led us to first evaluate the models in terms of the correct classification of the expected return. They were trained as a regression, but evaluated as a classification.

3.2.1 Accuracy, Precision and Recall

To evaluate classification models, we calculate metrics between the predicted and the real outputs.

Accuracy classification metric is the fraction of correct predictions over all the predictions. In a 5-people poker table, a regular player loses a lot more times than it wins. A model that predicts a loss for most of the cases will have high accuracy, hence the accuracy isn't important for this subject.

Precision is the fraction of correct predictions over all positive predictions. When a model classifies the scenario as a win but the player actually loses, he commits many chips to a losing pot, losing more money than he would have lost with a correct prediction. Hence, the precision is an important metric for this subject.

Recall is the fraction of correct predictions over all wins. When a model classifies the scenario as a losing game, but the player actually wins, he will fold a winning hand, thus he will receive a punishment from a situation he could have been rewarded.

F1 metric is a balance between precision and recall and it is also important to measure. It is equal to $2*(precision*recall)/(precision+recall)$.

3.2.2 Number of Layers and Neurons

We started the investigation of NN architectures with 4 layers of 100 neurons each. To simplify, we will call this architecture just 4×100.

Following the findings of previous work [Moravcik2017], we used rectifier activation functions in hidden layers, and the Adam optimization algorithm to correct the model weights. We used a linear activation function in the last layer because it is a regression problem, hence we did not restrict the prediction to the behavior of an activation function. By having rectifier activation functions

in the hidden layers, we already approached the non-linearity, which is intrinsic to the poker problem.

The Adam optimizer was responsible for the stocastic gradient descent of weights in backpropagation. At first, this choice was made by comparison with the performance of different optimizers, in terms of the classification described above. Then, it was confirmed by the same choice in [Moravcik2017]. Further, the Adam optimizer is computationally efficient, requires little memory and it is well suited for large and stochastic problems such as poker [Kingma2015].

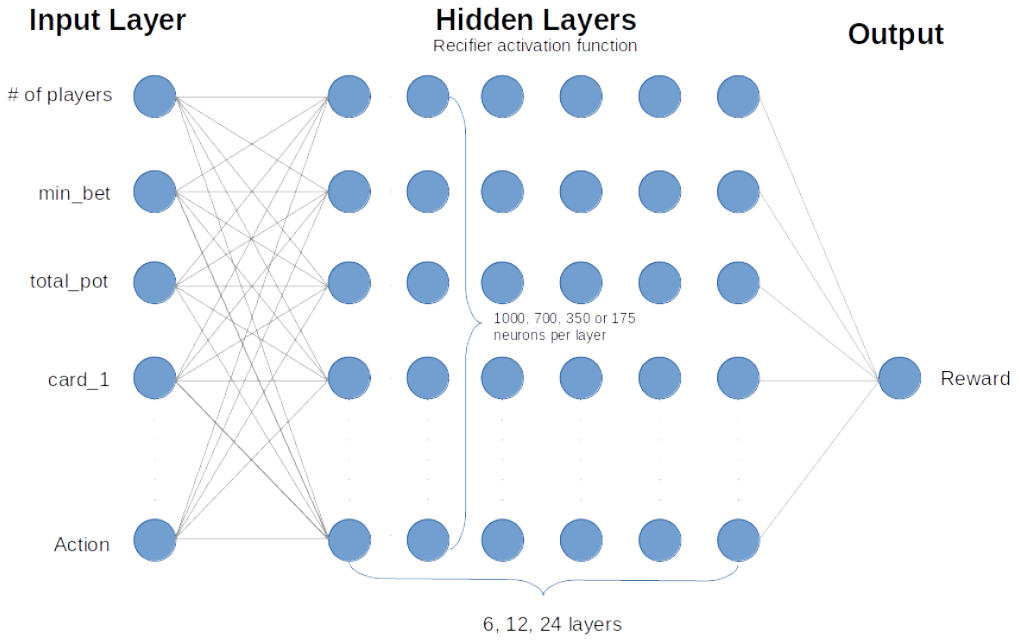
The 4×100 models had a fine accuracy (average of 0.7), but poor precision and recall(average of 0.5). In a 5-player poker table, usually a player loses more times than it wins. Hence, predicting a loss is easier than predicting a win, which explains the classification metrics and why this model was not reliable.

Then, we gradually increased the number of neurons and layers by experimentation and evaluation, until we reached the performance limit with a 6×1000 network. After that, increasing the number of neurons or layers didn't show any improvement in the classification metrics. Sometimes the metrics even decreased.

After fitting the 6×1000 network with the BN dataset described in 3.1, we investigated the neuron weights. We realized that an average of 300 neurons per layer were very close to 0, hence they had almost no influence in the network's output. This is an old technique called "pruning neural networks", which led us to remove redundant neurons to achieve faster training time [LeCun1990], without compromising the performance. Following that, we created a 6×700 network and evaluated the accuracy, precision, recall and loss on both networks, using separated train and test datasets, and we assured there was no loss in the quality of outputs.

To analyze the trade-offs between the number of layers and the number of neurons per layer, we kept the total of $6\times700=4200$ neurons, and created a 12×350 and a 24×175 neural network.

Figure 3.3: Neural network architectures implemented in Pucker framework



In summary, this methodology brought us to four deep architectures:

- **6 × 1000**: 6 layers of 1000 neurons each
- **6 × 700**: 6 layers of 700 neurons each
- **12 × 350**: 12 layers of 350 neurons each
- **24 × 175**: 24 layers of 175 neurons each

We kept the 6×1000 network to analyze if it has any gain in performance after multiple learning phases, to understand if more neurons demonstrates better performance after they are fit with more data.

We built the learning players by using those models to estimate the expected value of each action, given a poker scenario. Their metrics and performance in poker games will be further discussed in chapter 5: Evaluation.

3.2.3 Hand Rank, Strength and Potential

A significant gain in performance was achieved by the addition of three variables to the input dataset: hand rank, hand strength and hand potential. Without those variables the models have bad classification metrics and could not improve their performance over learning phases.

The hand rank denotes the ranking of a poker hand over all possible poker hands. It's used to verify which player have won, but it can also be used by players to evaluate their hands in order to take a better decision. Without

this information, the neural networks have to learn by themselves which hands are worth to be played and which ones put them in a bad scenario. By using a scalar to represent the ranking of a hand, the networks were able to predict the outcomes of their actions with much more confidence.

The hand strength denotes the probability that the current hand is the best hand in the table. It can be calculated by the enumeration of all possible other hands and a comparison between this other hand and the players' hand. By counting the number of better and worse hands, the player can estimate how good is his hand in comparison with his possible opponents' hands.

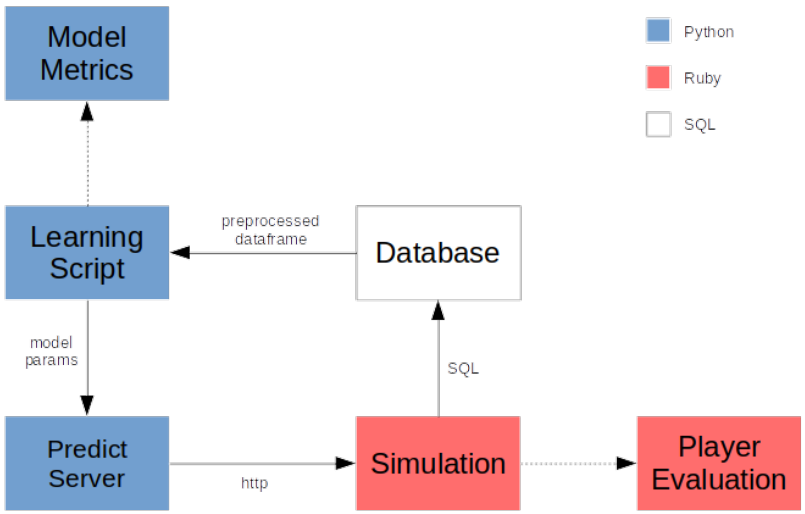
Hand strength is a measure of the present. But in turn and river, other cards are dealt to the table and can change the strength of the players' hands. Hand Potential denotes the probability that our hand will be improved over our opponents' hand, given a new card to the table. It can be calculated by the enumeration of all possible cards that can be dealt to the table, and counting how many of them improves our hand as opposed to our opponents' hand.

4
Framework Architecture

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

The simulation runs the game and inserts data about the states seen by a player, actions taken and his rewards (or punishments if negative) in the database. The states, actions and rewards are the learning variables. The learning script reads the database, fits the model, and stores the model parameters in the 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.

Figure 4.1: Pucker framework architecture diagram



4.1
Simulation

The Ruby programming language was chosen to write the simulation component. Ruby offers 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. 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 idiomatic property, 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 (see Listing 4.1) and a simple random player can be written in a few lines of ruby (see Listing 4.2).

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
```

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.

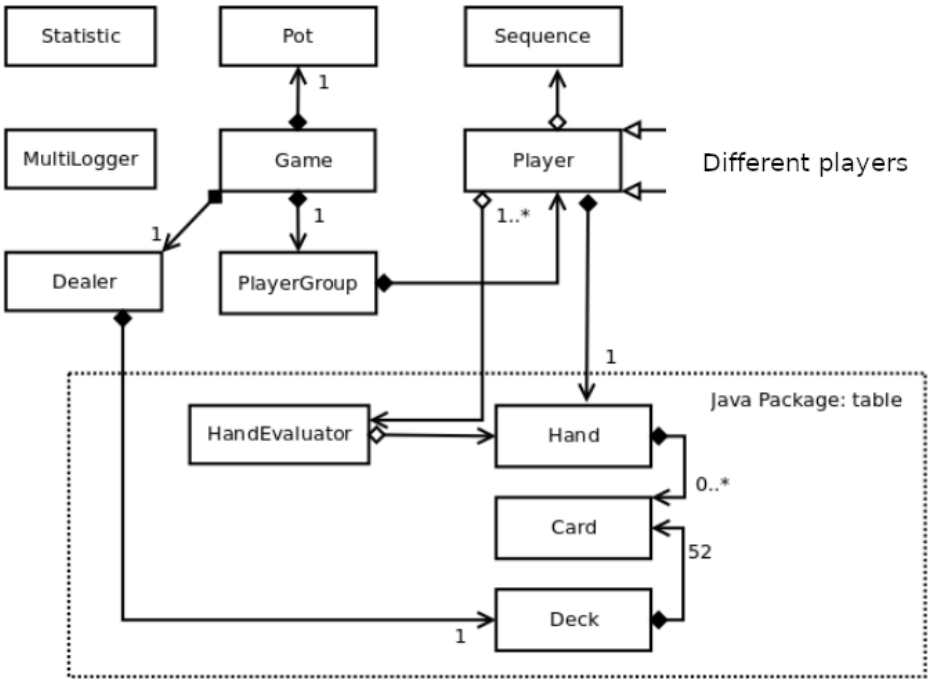
In the context of the Pucker framework, a game has a group of players, a dealer and a pot with the bets of the current game. The dealer deal cards. Players evaluate game state and choose an action based on their current state.

Listing 4.2: player.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
```

Figure 4.2: Pucker framework simulation diagram



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

In pre-flop, a player has 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 has abstracted the pre-flop phase: every player bets the same as the big-blind player.

Many simulators have been developed for poker research, the most no-

table one is the Poker Academy simulator. In comparisson with this simulator, our simulation runs two times slower. It simulates 100000 games in 26 minutes, while the Poker Academy simulator does in 13 minutes. This is due to the Hand Rank algorithm, which is slower in our simulator but can be easily exchanged in future work. Instead of using open-source solutions, we decided to build our own simulator to have more control over the code, possible bugs, and performance. For example, it's possible to run multiple simulations at the same time, one per CPU, with our simulation component. This reduced the simulation time four times, during the evaluation of this work.

4.2 Storage

In real poker, the state of the game have many 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.

To support machine learning, Pucker must store an abstraction of the state, the actions taken and rewards. Pucker is an extensible framework, it is easy to add or remove variables to the state abstraction in future work. 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

In poker, a player sees a game state, takes an action, and receives a reward (or punishment) at the end of the round. Pucker players remember the

states, actions, and rewards, and stores them in an 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 is written at once in a single insert query, after multiple games.

4.3 Learning

A machine learning prediction typically maps a set of variables to an output. In this work, the input will be the state and action, the output will be the reward seen at 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 maximizes its rewards. This is inspired in Reinforcement Learning but the policy optimization is done implicitly by the neural network.

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 for their choice in favor of other models, like Gradient Boosting Machines. Until today, popular Gradient Boosting Machine implementations cannot handle incremental learning [xgboost-github].

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, preprocesses, fits prediction models and stores their param-

eters in the disk. It uses [scikit-learn.org], [keras.io], [pandas.pydata.org], and [joblib.readthedocs.io] packages.

4.4 Prediction

The prediction component reads the stored model parameters and creates an instance of the model ready to predict unseen states and actions.

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-action query 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

According to the subsection 3.2.1, we will start the evaluation of the models by the precision, recall and F1 metrics of each model. To prevent overfitting, we will split the datasets in 80% for training and 20% for testing those metrics, and only when using them in game plays we will train the model with the complete database.

Then we will analyze the performance in action, by playing them against each other in tables of five players. We will do that in four rounds of 3000 games each and display a measure of rewards one by each player.

5.1
Metrics without feature engineering

The first Neural Networks developed had little information about the domain of poker. They were created to predict the outcome of actions given basic poker variables such as hand cards, table cards, opponents bets, pot size, number of players. They reached a precision lower than 0.6, and a very low recall as well, and were discarded right away. To overcome this, our first step was to separate the neural networks of flop, turn and river, the results are shown below.

Table 5.1: Classification metrics of Neural Networks without Feature Engineering

	flop			turn			river		
NN	prec.	recall	f1	prec.	recall	f1	prec.	recall	f1
6x1000	0.686	0.561	0.617	0.739	0.640	0.686	0.720	0.788	0.752
6x700	0.652	0.675	0.663	0.737	0.657	0.694	0.731	0.763	0.747
12x350	0.723	0.459	0.562	0.724	0.691	0.707	0.751	0.713	0.731
24x175	0.670	0.457	0.551	0.750	0.601	0.667	0.745	0.779	0.626

By the use of simple multilayer NNs, players were not able to action effectively, and the models presented poor metrics. Given this bad result, we added some domain specific knowledge to the models, as described in section 3.2.3. With hand strength and hand potential knowledge, the models performed much better, as you can see in next section.

To effective compare the metrics of this session with the next section, one must compare against phase 1, since we also used phase 1 database in this section.

5.2 Model metrics

Below are the metrics for each NN model in each learning phase. It's important to notice that phase 1 presents the overall best metrics among all phases. At first glance this seems a problem, but it can be explained by the fact that phase 1 dataset was built by BN players, which are more predictable than NN players.

Table 5.2: Classification metrics of Neural Network 6x1000

	flop			turn			river		
phase	prec.	recall	f1	prec.	recall	f1	prec.	recall	f1
1	0.682	0.661	0.671	0.750	0.720	0.735	0.770	0.772	0.771
2	0.735	0.353	0.478	0.730	0.624	0.673	0.760	0.704	0.731
3	0.700	0.558	0.621	0.729	0.653	0.689	0.787	0.805	0.796
4	0.695	0.624	0.658	0.747	0.721	0.734	0.758	0.750	0.754

Table 5.3: Classification metrics of Neural Network 6x700

	flop			turn			river		
phase	prec.	recall	f1	prec.	recall	f1	prec.	recall	f1
1	0.686	0.655	0.670	0.722	0.758	0.739	0.771	0.769	0.770
2	0.710	0.385	0.499	0.747	0.590	0.659	0.757	0.729	0.743
3	0.691	0.575	0.628	0.710	0.701	0.706	0.792	0.797	0.794
4	0.699	0.621	0.658	0.726	0.749	0.737	0.748	0.750	0.749

Table 5.4: Classification metrics of Neural Network 12x350

	flop			turn			river		
phase	prec.	recall	f1	prec.	recall	f1	prec.	recall	f1
1	0.654	0.696	0.674	0.721	0.749	0.735	0.757	0.779	0.768
2	0.752	0.326	0.455	0.727	0.624	0.672	0.789	0.646	0.711
3	0.700	0.546	0.613	0.713	0.695	0.704	0.782	0.813	0.797
4	0.725	0.533	0.614	0.741	0.719	0.730	0.751	0.761	0.756

Table 5.5: Classification metrics of Neural Network 24x175

	flop			turn			river		
phase	prec.	recall	f1	prec.	recall	f1	prec.	recall	f1
1	0.669	0.638	0.654	0.760	0.660	0.707	0.776	0.735	0.755
2	0.666	0.399	0.499	0.733	0.599	0.659	0.743	0.740	0.741
3	0.614	0.405	0.488	0.684	0.671	0.678	0.786	0.799	0.793
4	0.711	0.564	0.629	0.714	0.746	0.730	0.697	0.818	0.753

In flop and turn, there is a visible improvement of metrics from phases 2, 3 to 4, which shows incremental gain in performance between learning phases. In river, we cannot notice an improvement between learning phases. This is due to the models having more information about the table cards and the players bets, hence it needs less data to learn and additional learning phases didn't contribute to the overall performance.

In flop and turn, F1 metric increases over phases 2, 3 and 4, and the improvement is due to recall metric. According to section 3.2.1, the poor recall of early phases relies on the classification of winning scenarios as a loss. When a model classifies a winning situation as a loss, his player will fold a winning hand, preventing him from getting rewards.

Also according to section 3.2.1, precision goes down when a model predicts lost scenario as a win. It's important to notice that, in a five people table, most of the situations will be losing scenarios. Even when a model predicts a win, a player might deal with deception and luckiness, which are key factors in poker.

Another important observation is that turn and river present better metrics than flop. In those phases, there are less active players, since some players had already folded on flop. Also, the last phases are easier to predict since more actions have been taken by the opponents, and the observable state is composed of more table cards.

There is no significant difference in the metrics between the architectures, hence we will have to compare them in terms of game results.

5.3 Poker statistics

To evaluate the neural network players, we simulated 4 rounds of 3000 games each. Every player started with 2000 chips, and when they hit 0, they receive more 2000 chips, as a rebuy. The charts bellow account for the relative amount of chips they've have: the present amount minus the number of chips they've rebuyed.

5.3.1
Neural x Bayesian Networks

In the first evaluation, we simulated a table with the four different architectures and one bayesian agent, and measured how many chips they have accumulated.

Figure 5.1: Round 1 of the evaluation between neural and bayesian agents

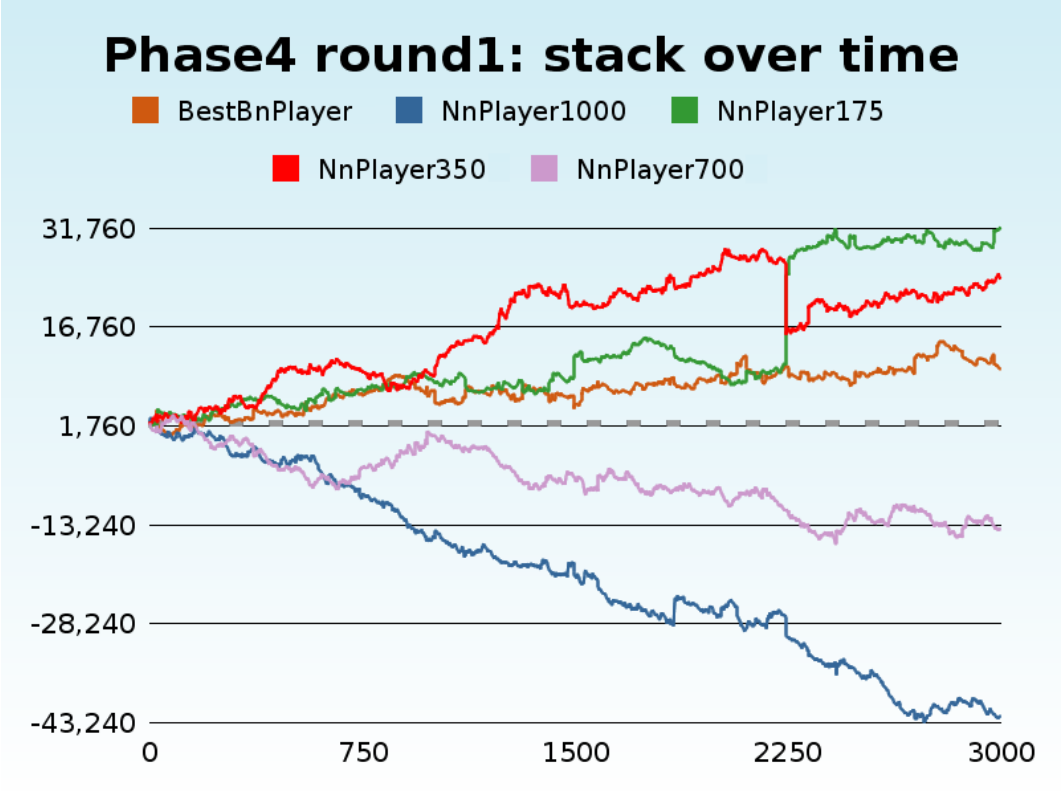


Figure 5.2: Round 2 of the evaluation between neural and bayesian agents

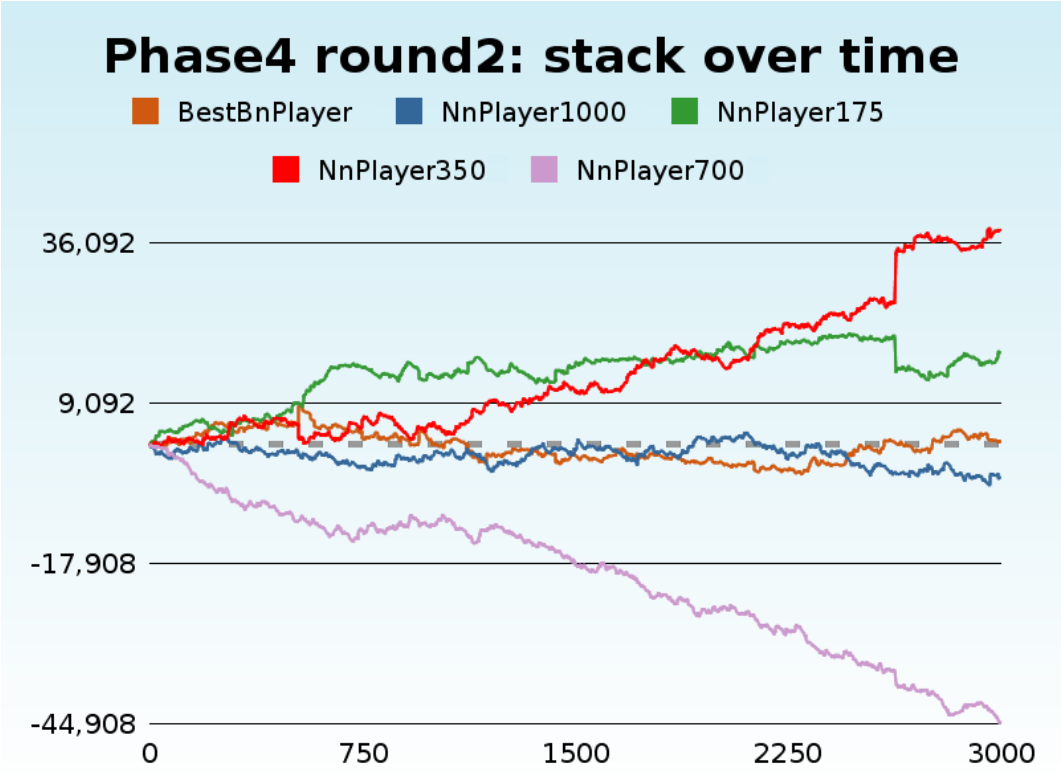


Figure 5.3: Round 3 of the evaluation between neural and bayesian agents

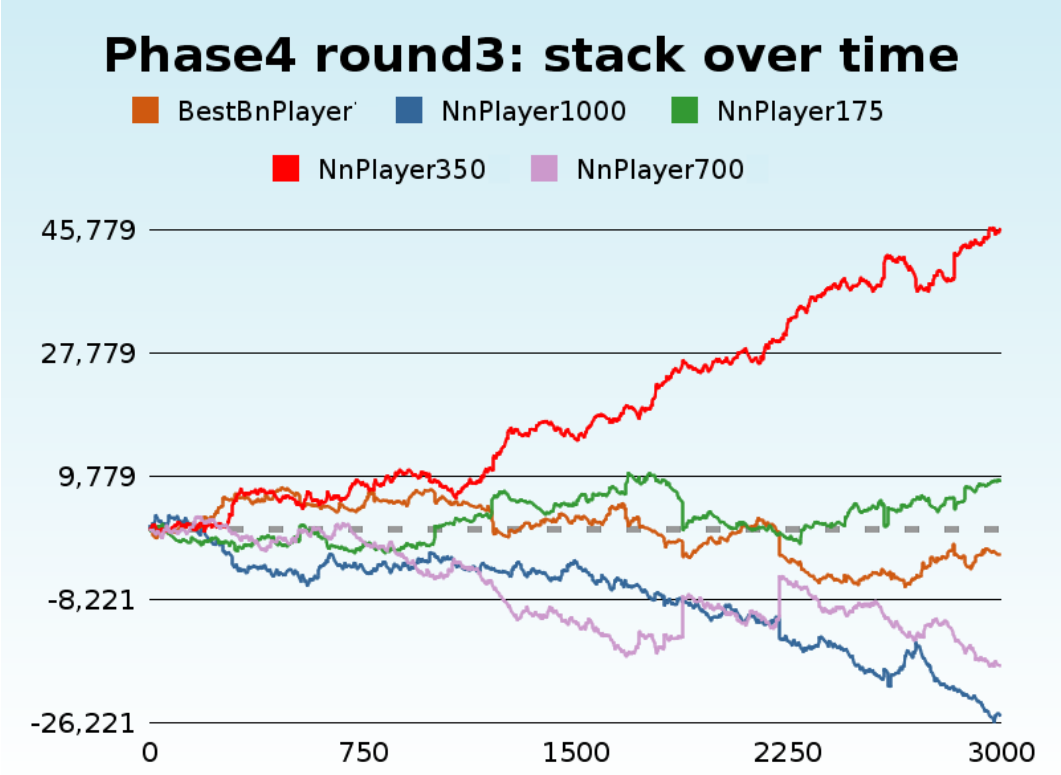
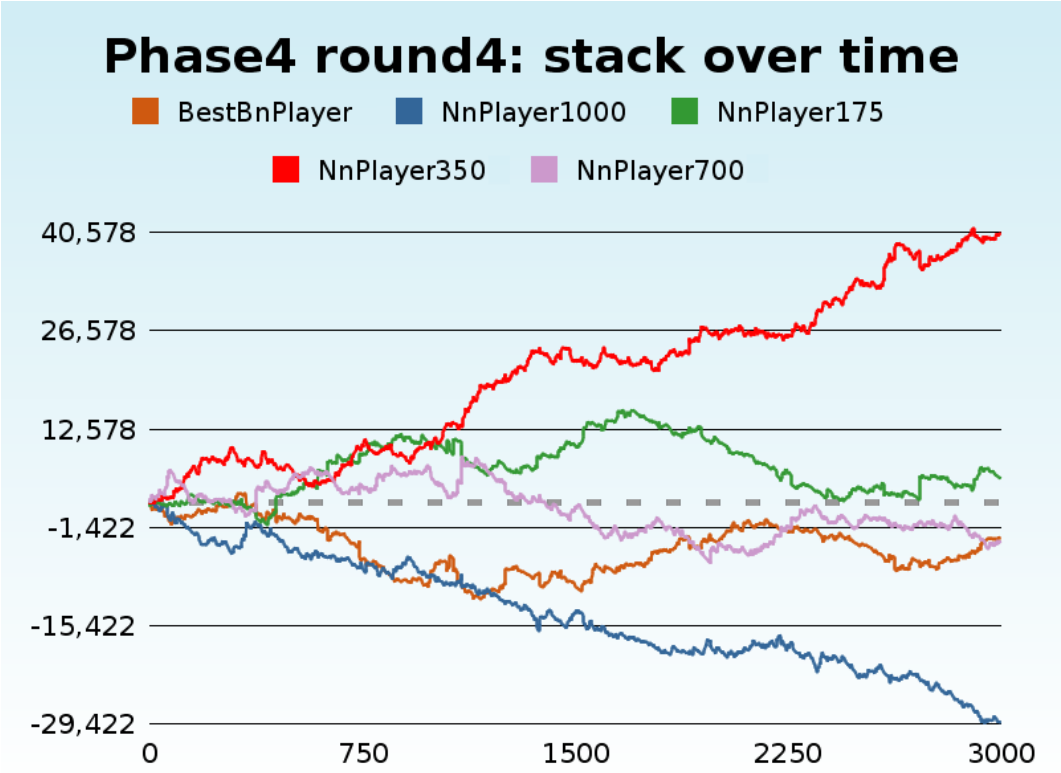


Figure 5.4: Round 4 of the evaluation between neural and bayesian agents



The player with the 12×350 architecture had the best performance, followed by the 24×175 agent, which suggest that wider architectures perform better than narrower ones with the same number of neurons. The narrower architectures performed worse than the bayesian player.

5.3.2 Incremental Learning Evaluation

Next, we evaluated the incremental learning. For this test, we created the same number of games but with two 12×350 agents in the fourth phase of learning, two of the same architecture in the second phase of learning and one dummy fifth player to serve as a benchmark for the performance of the players.

In the first two rounds, an always-call fifth player was used. The results show that phase 4 (P4) models performed better.

Figure 5.5: Round 1 of the evaluation between neural agents of different learning phases

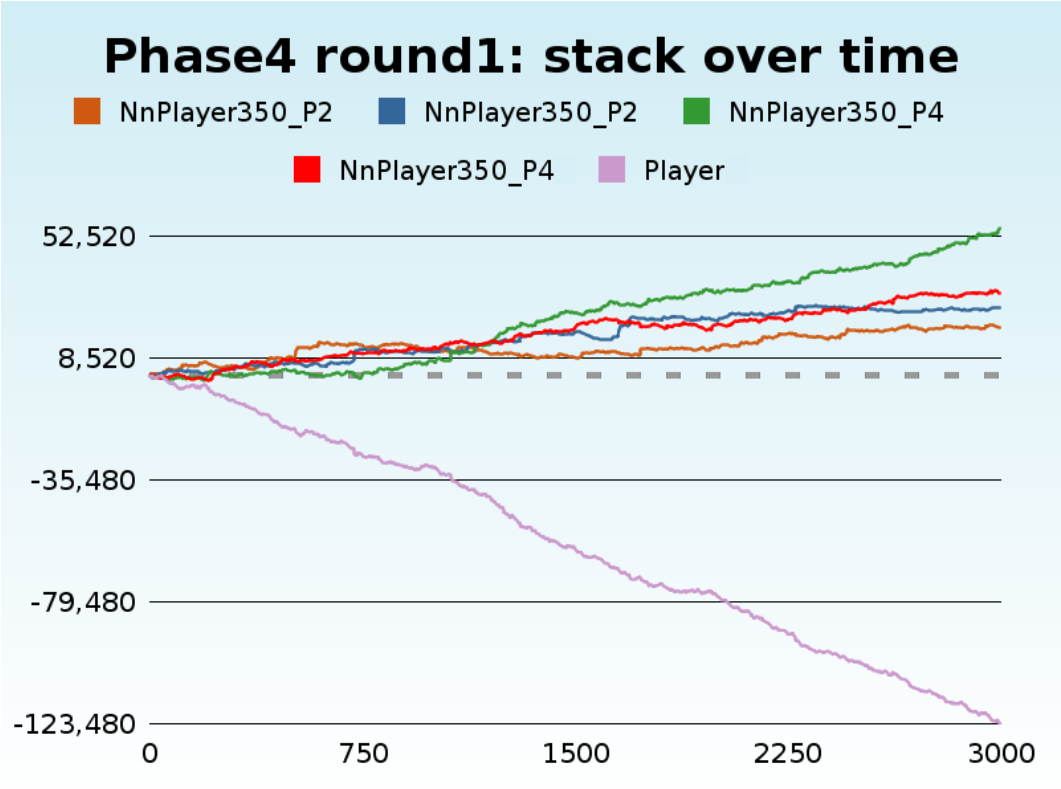
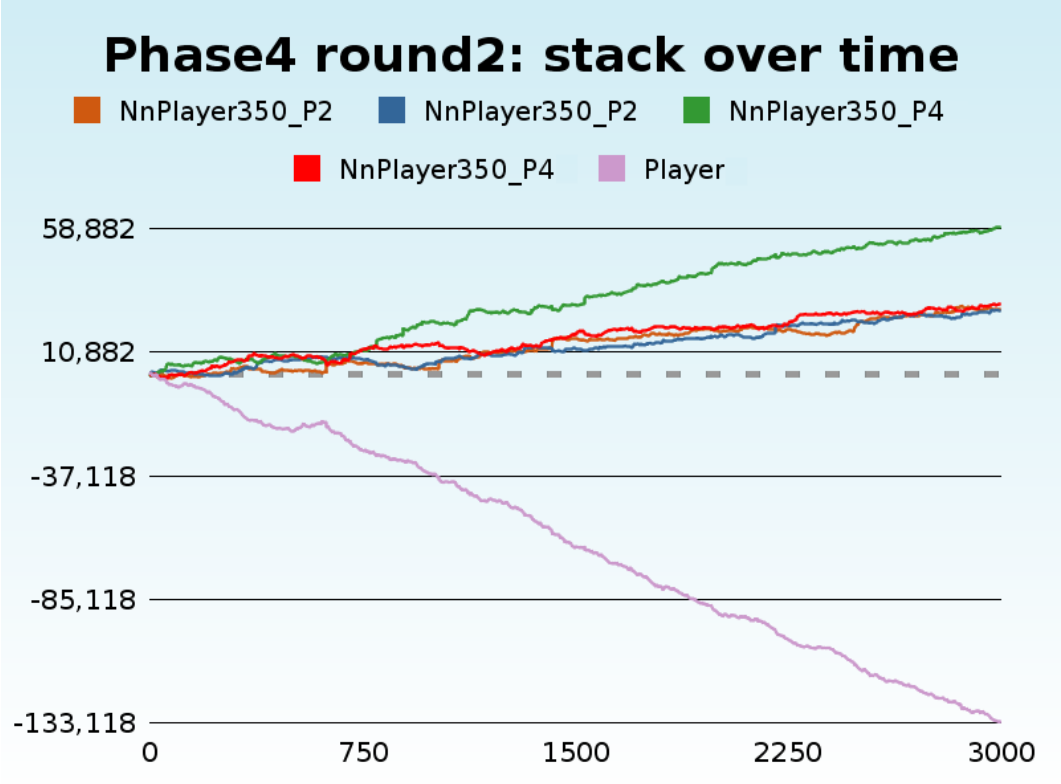


Figure 5.6: Round 2 of the evaluation between neural agents of different learning phases



In the last two rounds, a random fifth player was used. The results show that phase 4 (P4) models performed better again.

Figure 5.7: Round 3 of the evaluation between neural agents of different learning phases

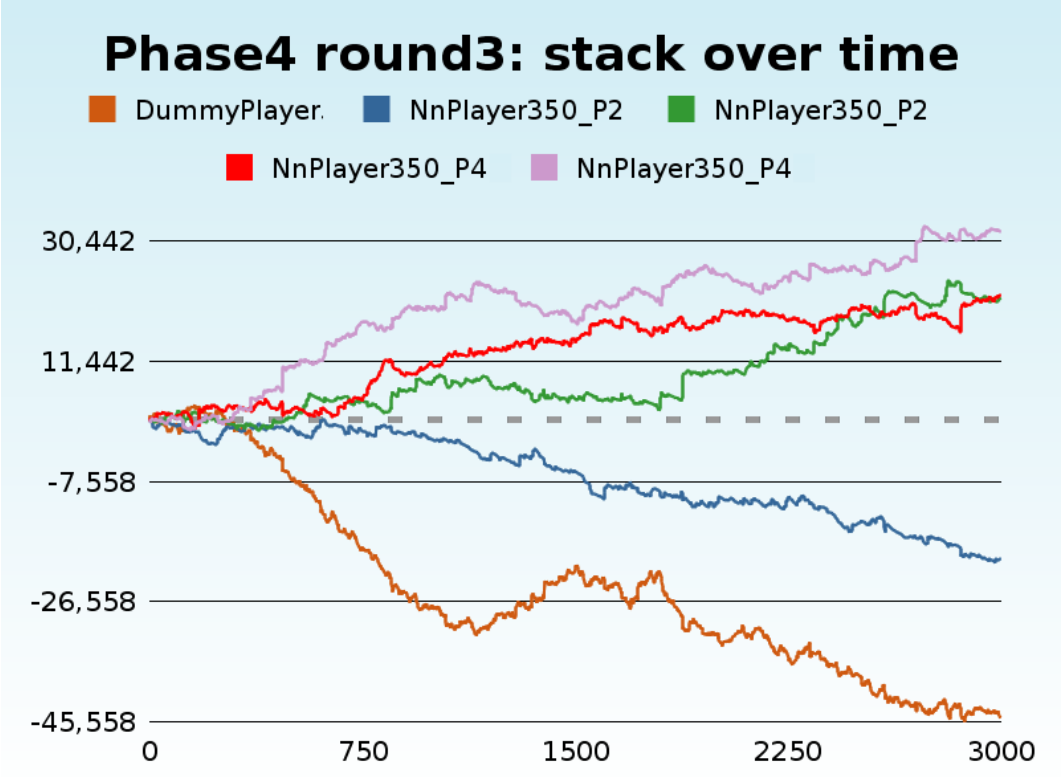
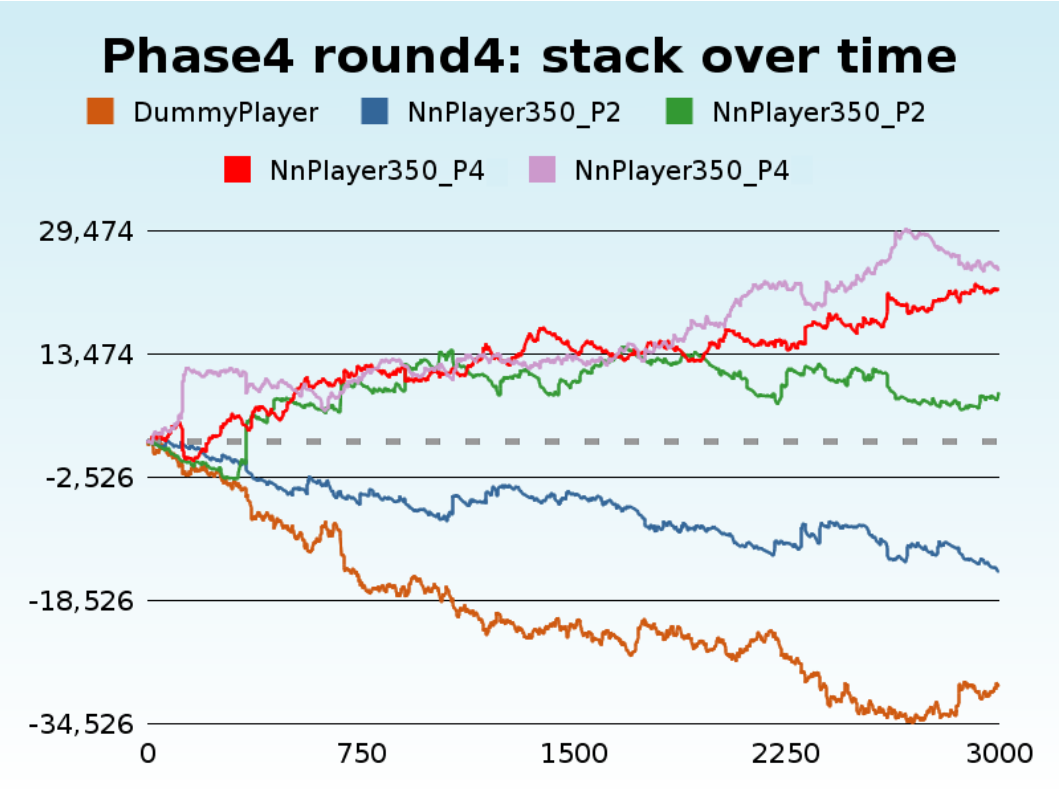


Figure 5.8: Round 4 of the evaluation between neural agents of different learning phases



This suggests P4 follows a better policy than P2, thus receiving more rewards. Both of them performed better than always-check and random players. Although this is an exciting result, it is more intended to evaluate the success of the framework development than the players. The framework allowed us to explore poker and many of its caveats, it empowered our research to get better at each step, and provided us a nice workflow over the learning phases.

The players present an increasing performance, but are worse than the best players developed at University of Alberta and Carnegie Mellon, since reaching this result was never the intention.

6
Conclusions

This chapter presents the concluding remarks about our work. Discuss our contributions in Section 6.1, and present directions for future works in Section 6.2.

In this dissertation, we presented theory and practice of deep learning and how to apply it to a complex field. We created a complete framework to support simulation, persistence, learning and prediction of multiplayer no-limit Texas Hold'em. We evaluated the methodology in practice, by creating several machine learning players and analyzing them in live-action.

Best response for multiplayer no-limit Texas Hold'em is still an open field. Academia can benefit from this work by understanding step-by-step how to approach computer poker, and by using the framework to create more robust poker agents and evaluate them against the neural network players developed in this research.

Although bluffing wasn't addressed in this research, the developed models showed that bluff is not a human behaviour, it is mathematically worth. The developed players bluffed in some different scenarios, more frequently when in higher position when no other players had placed a bet. This behaviour is also common in amateur poker tables.

We acknowledged that, without any domain specific knowledge, it is really hard to model a poker machine learning model. Besides researching and iterating through the different neural network architectures, what made most difference in our results were the separation of neural networks in flop, turn and river, and the development of domain specific features such as hand strength and potential.

It must be stated, however, that does not mean our approach is the best, specially regarding the performance of the created poker agents. The important goal was to support an improvement in the state-of-the-art by bringing all the aspects of computer poker in one work, from the simulation, to data generation, to deep learning and to reinforcement learning.

Another important goal was to incrementally build better players using the framework to support the research of better strategies, which was discussed in section 5.3.2.

6.1 Contributions

Our contributions can be summarized by:

- A review on how to use modern techniques to tackle a complex environment;
- A complete machine learning framework to simulate, generate data and evaluate computer poker;
- A set of deep reinforcement learning agents built using the techniques and the framework described;

The first contribution is theoretic and resides in the second and third chapter of this dissertation, where we describe the foundations of machine learning, deep learning and reinforcement learning, and how to use those techniques to build a poker playing agent. The second contribution is practical, and goes from the description of the system, in the fourth chapter, to the actual framework code [github/pucker]. The last contribution is both, theoretic and practical, and goes from section 3.2 to the deep learning code presented by the framework.

6.2 Future Work

To contribute to the state-of-the-art, a researcher should use the contents of this dissertation to improve the performance of the presented framework and agents.

One path is to automate the learning phases and the creation and evaluation of different architectures, inspired by evolutionary algorithms, in order to create neural networks from the evolution and progress of the learning phases.

Other way is to develop an opponent modeling component, to support the agents in building a probability distribution over an important part of the unobservable state: their opponents' hands.

Another interesting future work is to improve the reinforcement learning workflow, by an explicit programming of the policy optimization instead of relying on the neural network's learning.

Bibliography

[Billings1998] BILLINGS, D.; PAPP, D.; SCHAEFFER, 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] BILLINGS, D.; PAPP, D.; SCHAEER, J. ; SZAFRON, D.. **Opponent modeling in poker**. In AAAI National Conference, 1998.

[Bowling2008] BOWLING, M.; BURCH, N.; JOHANSON, M. ; 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] BROWN, N.; SANDHOLM, T.. **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] BROWN, N.; SANDHOLM, T.. **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] DAVIDSON, A.; BILLINGS, D.; SCHAEFFER, J. ; SZAFRON, D.. **Improved opponent modeling in poker**. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE (IC-AI'2000), LAS VEGAS, NV, 2000.

[Dayan2002] DAYAN, P.; BALLEINE, B. W.. **Reward, motivation, and reinforcement learning**. Neuron, 36(2):285 – 298, 2002.

[Garcia1966] GARCIA, J.; ERVIN, F. R. ; KOELLING, R. A.. **Learning with prolonged delay of reinforcement**. Psychonomic Science, 5(3):121–122, Mar 1966.

[Glorot2010] GLOROT, X.; BORDES, A. ; BENGIO, Y.. **Deep sparse rectifier neural networks**. In: Gordon, G.; Dunson, D. ; Dudík, M., editors, PROCEEDINGS OF THE FOURTEENTH INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND STATISTICS, volumen 15 de **Proceedings of Machine Learning Research**, p. 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[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.

[Holroyd2002] HOLROYD, C.; COLES, M.. **The neural basis of human error processing: Reinforcement learning, dopamine, and the error-related negativity**. Psychological Review, 109:679–709, 10 2002.

[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.

[Kaelbling1996] KAELBLING, L. P.; LITTMAN, M. L. ; MOORE, A. P.. **Reinforcement learning: A survey**. Journal of Artificial Intelligence Research, 4:237–285, 1996.

[Kingma2015] KINGMA, D. P.; BA, J.. **Adam: A method for stochastic optimization**. CoRR, abs/1412.6980, 2015.

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

[Lavet2018.1] FRANÇOIS-LAVET, V.; HENDERSON, P.; ISLAM, R.; BELLEMARE, M. G. ; PINEAU, J.. **An introduction to deep reinforcement learning**. CoRR, abs/1811.12560:229 – 230, 2018.

[Lavet2018.2] FRANÇOIS-LAVET, V.; HENDERSON, P.; ISLAM, R.; BELLEMARE, M. G. ; PINEAU, J.. **An introduction to deep reinforcement learning**. CoRR, abs/1811.12560, 2018.

[LeCun1990] LECUN, Y.; DENKER, J. S. ; SOLLA, S. A.. **Optimal brain damage**. In: Touretzky, D. S., editor, ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 2, p. 598–605. Morgan-Kaufmann, 1990.

[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] MORAVCÍK, M.; SCHMID, M.; BURCH, N.; LISÝ, V.; MORRILL, D.; BARD, N.; DAVIS, T.; WAUGH, K.; JOHANSON, M. ; BOWLING, M. H.. **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.

[Pang2018] TAN, P.-N.; STEINBACH, M.; KARPATNE, A. ; KUMAR, V.. **Introduction to Data Mining (2Nd Edition)**. Pearson, 2nd edition, 2018.

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

[Reynolds2001] REYNOLDS, J.; HYLAND, B. ; WICKENS, J.. **A cellular mechanism of reward-related learning**. Nature, 413:67–70, 10 2001.

[Richard1991] MICHAEL, D.; LIPPMANN, R.. **Neural network classifiers estimate bayesian a posteriori probabilities**. Neural Computation, 3:461–483, 1991.

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

[Rubin2009] RUBIN, J.; WATSON, I.. **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] RUMELHART, D. E.; HINTON, G. E. ; WILLIAMS, R. J.. **Learning internal representations by error propagation**. In:

PARALLEL DISTRIBUTED PROCESSING: EXPLORATIONS IN THE MICROSTRUCTURE OF COGNITION, VOL. 1, p. 318–362. MIT Press, Cambridge, MA, USA, 1986.

[Samuel1959] SAMUEL, A. L.. **Some studies in machine learning using the game of checkers**. IBM Journal of Research and Development, 3(3):210–229, July 1959.

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

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

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

[Teofilo2011] TEOFILO, L.; REIS, L.. **Holdemml: A framework to generate no limit hold'em poker agents from human player strategies**. p. 1–6, 01 2011.

[Teofilo2013] FILIPE TEÓFILO, L.; ROSSETTI, R.; REIS, L.; LOPES CARDOSO, H. ; ALVES NOGUEIRA, P.. **Simulation and performance assessment of poker agents**. volumen 7838, p. 69–84, 01 2013.

[Zikenvich2008] ZINKEVICH, M.; JOHANSON, M.; BOWLING, M. ; 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.

[github/pucker] COSTA, A. M.. **Pucker is a no limit texas hold'em emulator, together with a collection of different players, that can be used to play against each other.** `https://github.com/alexandremcosta/pucker`, 2019. [Online; accessed February-2019].

[joblib.readthedocs.io] **Joblib: running python functions as pipeline jobs**. `https://joblib.readthedocs.io/en/latest/`. Accessed: 2019-01-15.

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

[keras.io] **Keras: The python deep learning library**. https://keras.io/. Accessed: 2019-01-15.

[pandas.pydata.org] **Python data analysis library**. https://pandas.pydata.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.

[scikit-learn.org] **scikit-learn machine learning in python**. https://scikit-learn.org/stable/. 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.

[wikimedia/deeplearning] KOWSARI, K.. **Random multimodel deep learning (rdml) architecture for classification. rmdl includes 3 random models, one dnn classifier at left, one deep cnn classifier at middle, and one deep rnn classifier at right (each unit could be lstm or gru)**. https://commons.wikimedia.org/wiki/File:Random_Multimodel_Deep_Learning_(RMDL).png, 2018. File: Random Multimodel Deep Learning (RMDL).png; Online; accessed on February-2019.

[wikipedia/machinelearning] WIKIPEDIA. **Machine learning — Wikipedia, the free encyclopedia**. http://en.wikipedia.org/w/index.php?title=Machine%20learning&oldid=886780793, 2019. [Online; accessed February-2019].

[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.