Heriot-Watt University

Research Report

Opponent exploitation with neural network in imperfect game

Author:
Noé Rivals

Supervisor:
Dr. Michael Lones

A thesis submitted in fulfilment of the requirements for the degree of MSc.

in the

School of Mathematical and Computer Sciences

August 2021

Declaration of Authorship

I, Noé Rivals, declare that this thesis titled, 'Opponent exploitation with neural network in imperfect game' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

Date: 06/04/2021

“The first step is to establish that something is possible; then probability will occur.”

Elon Musk

Abstract

Imperfect game solving is an important field of research in AI. Research has been done on finding or approximating the equilibrium of a game. Once an equilibrium is found, the algorithm does not lose in the long term in a zero-sum game. Opponent exploitation is another approach to game solving, it tries to model the opponent's behavior and exploit it resulting in a higher reward. We will present a way of exploiting the opponent using neural network. We will evaluate our algorithm on the Kuhn poker variant. We will also explore its scalability for large information games with limit texas holdem poker.

Acknowledgements

I would like to thank Dr. Michael Lones for supervising my project, giving me precious advice, and guiding me.

Contents

List of Figures

List of Tables

Abbreviations

RNN	**R**ecurrent **N**eural **N**etwork
LSTM	**L**ong **S**hort **M**emory
EA	**E**volutionary **A**lgorithms
NN	**N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork

Chapter 1

Introduction

Imperfect game solving is the term used to solve a game where some information is unknown such as cards in poker. Imperfect game solving is an interesting field because it can be transposed easily to real-world problems. For example, in military, sometimes the opponent's resources are unknown or with a vague estimate, imperfect game solving algorithms are designed to handle state with unknown information.

Research in this field has been mostly focused on finding the equilibrium of a game. An equilibrium strategy played on the long term can not lose. It is the probabilistic solution to the game. On larger game state equilibrium can not be computed, algorithms are so trying to approximate it and get closer to it.

However, an equilibrium strategy does not guarantee the higher reward against an opponent. It just guarantees that on the long term, it will not lose. To improve the reward against an opponent, we need to exploit it. This is done by modeling the opponent's behavior and exploiting the mistakes made by him, this is called a sub-optimal opponent.

The problem is that whenever we try to exploit an opponent we are exposed to exploitability, this is referred to as the get-taught-and-exploited problem [Sandholm, 2007]. On another side, some research has proven that in some imperfect games, safe exploitation is possible introducing the notion of gift [Ganzfried and Sandholm, 2015]. Safe exploitation ensures that the opponent can be exploited without being exposed to exploitability. In this research we will not focused on having a safe exploitation, this is a choice made to fully explore the capacity of neural network and not limit it by the safe exploitation constraints.

Existing research has been done on opponent modeling in imperfect games. As said previously one demonstrates that safe opponent exploitation exists in some imperfect games such as poker [Ganzfried and Sandholm, 2015], however, the method used in this paper do not scale for large game state or complex games. One research has been done for modeling large game state (poker limit texas hold'em) [Ganzfried and Sandholm, 2011]. Both methods use a similar method. It uses a decision tree with a pre-computed equilibrium approximation, it plays a certain number of games against an opponent and updates the statistical difference between the equilibrium and the opponent choices. This difference defines the opponent model. This algorithm have multiple limitations. First, it requires playing a relatively high number of times against an opponent to start having a good result. Secondly in a large game tree, only the first branches will have an accurate opponent modeling, further we go in the tree less the algorithm has data. This is a major problem because it only models the opponent early game. Third, this method does not encode complex behavior, it is limited to the statistical mistake of the opponent. Fourth, it does not have a spatial representation, it means that the first or last game played by the opponent has the same weight in the opponent modeling. For example, it will not be able to model anger, opponent betting higher than usual after losing.

The goal of this project is to demonstrate that opponent modeling is possible with neural network and to assess the performances. We are aiming to have an algorithm that is not game-dependent and which is working with information that is available or computable for every imperfect game. This information is for example equilibrium and the game result. This will allow this algorithm to be re-used for other use cases.

We will at first works on a small game state to makes our experimentation. It would allow us to iterate faster and have rapid feedback on the architecture performance. We will use Kuhn poker which is an extremely simple version of poker [Kuhn, 1950]. Then we will try to scale up the algorithm to a larger game state to assess the scalability. We will use limit texas holdem poker which has an info set size of 10^{14}.

We will use recurrent neural network (RNN) because it will be able to use its output as input and so update the opponent model given the previous model. Recurrent neural network has proven to be able to learn complex problems. We are going to use another neural network to take actions within the game given the opponent model.

We will evaluate our model playing Kuhn poker. It will play against simple opponents that are easily exploitable and more advanced opponent that has already been exploited with other technique to have a comparison point. To evaluate the exploitability of our algorithm we will make it play against an equilibrium opponent. In addition, we will use the true best response algorithm that is known for exploitability testing [Johanson et al., 2011].

The source code for this project is available at https://github.com/noeRls/opponent-exploitation

This report is structured as follows:

- **Chapter 2, Literature Review**: This chapter covers the literature regarding machine learning models that had been considered for this project. It also explains the research made in imperfect game solving.
- **Chapter 3, Methodology**: This chapter explains the architecture of the two neural networks, the methods that will be used to train them, and the evaluation process.
- **Chapter 4, Kuhn poker experiment**: This chapter details how the algorithm performed playing Kuhn poker and assess the potential of NN to model opponent.
- **Chapter 5, Scale-up, limit texas holdem poker experiment**: This chapter explores the scalability of the algorithm by making it play limit texas holdem poker.

Chapter 2

Literature Review

In this section, we will present multiple topics that have been studied to make the implementation choices.

In the first section we will present machine learning models that will be used in this project, Decision Tree and RNN. We will also cover the training methods that has been considered, which are reinforcement learning and evolutionary algorithm.

The first sections will be dedicated to a general presentation of some machine learning model that has been considered for this project. In addition we will present In addition we will present some data models that are used in games solving. Those ones will be necessary to fully understand the work done on this project.

Then we will discuss the current state of the research about imperfect game solving. Presenting the different approaches to this problem.

2.1 Machine learning models

2.1.1 Decision Tree

Decision trees are a way of representing multiple choices and their outcome. As suggested by its name it uses a tree. The root node represents the initial state. Each link represents a decision, and the linked node is the outcome of this decision, usually represented by a new state.

This data structure is particularly useful to represent games. For example, a chess game can be represented by a decision tree where links are moved played and nodes the resulting game state.

However, one major drawback of this data structure is that it can quickly become complex if there is an important depth (consecutive decision). For instance, for chess, it is impossible to represent a full decision tree because it contains too many different states. A common technique to overcome this limitation is to set a fixed depth and update the decision tree when it is traversed.

Metadata is often added to the link and the nodes to be processed easier by algorithms.

When decision trees are applied to games, an important notion is the Nash equilibrium [Nash et al., 1950]. It represents the optimum decision to take for every state, acknowledging that the opponent is playing perfectly. When a Nash Equilibrium is found for game, it is said that the game is solved. Given the actual computer limitation, such equilibrium can not be computed (or even stored) for complex games. In this case, algorithms try to approximate it, we will discuss it in more detail in section 2.2.2.

2.1.2 Recurrent Neural Network

Recurrent Neural Network (RNN) is a type of neural network that uses a part of its output for a part or all its input.

It is particularly interesting for his capacity to have an internal memory and remember the past input. A simple RNN is said to have a short memory and has been used for use-cases involving sequential data.

2.1.2.1 LSTM

Long Short-Term Memory (LSTM) is a type of RNN that has increased the popularity of RNN by providing a long memory layout to the existing RNN. It has a specific design that allows having a way better space representation of the input and has proven to works well in many domains. In addition, it is also worth mentioning that it is by design opportune for creativity because it uses its output as input. For example, one project has been made to generate Game Of Throne book [Thoutt, 2017]. Or also generating music [Kotecha and Young, 2018].

Given its design, RNN can process and train on a large input given that it is a suitable input for RNN. Where other neural network architectures would not be able to works with it efficiently.

2.1.3 Reinforcement Learning

2.1.3.1 Overview

Reinforcement Learning is a method used to train an algorithm based on experiences. An agent evolves in an environment, makes action, and learns from his mistake.

The basic idea is to have a reward given for an action or a set of actions. While learning the agent will try different actions and the reward will be used to update the algorithm. The update rules are specific to the algorithm used. It is common to use an extra formula to take into consideration the future expected reward.

While learning when the agent needs to take action, there is a set of methods that are common to all algorithms. It either **exploit**, take the best action (not necessarily optimal yet), or **explore**, pick a random action. A well-known method is epsilon greedy, it chooses to exploit or explore given predefined probability.

2.1.3.2 Q-learning

Q-learning [Watkins and Dayan, 1992] is a popular reinforcement learning method. It learns the Q-values, which are the expected reward of the actions given a state. The formula of a Q-value is: $Q^*(s,a) = \max_\pi \sum \big[r_t + yr_{t+1} + y^2 r_{t+2} + ... | s_t = s, a_t = a, \pi\big]$. Where a the the action, s is the state, π the policy and y the discount which represent how much the future reward should be take in consideration.

This formula has been limited to small state space due to its learning complexity. Recently research has been done to learn these Q-values with a large state space using neural network, it is Deep Q Network (DQN) [Mnih et al., 2015]. This method uses another formula to compute the Q-values, making neural networks way more efficient to train. This method has proven to be successful, it has been used to makes an agent learn how to play 2600 Atari games using only the screen pixels for input [Bellemare et al., 2013].

DQN are only using the current state to take action, so it performs poorly when it requires the agent to remember the previous states. Research has been done to overcome this limitation combining DQN with Recurrent Neural Network (DRQN [Sorokin et al., 2015] Chen et al. [2016]. Both of these methods use an attention system. [Chen et al., 2016] use an attention layer that takes all the previous state of the RNN and selects which information to take (figure 2.1). On the other hand [Sorokin et al., 2015] use a part of the output of their RNN to define the next region of attention (figure 2.2). On the figures you would notice that it is built with a CNN before, it is because these NN were designed to learn how to play Atari games from the screen pixels.

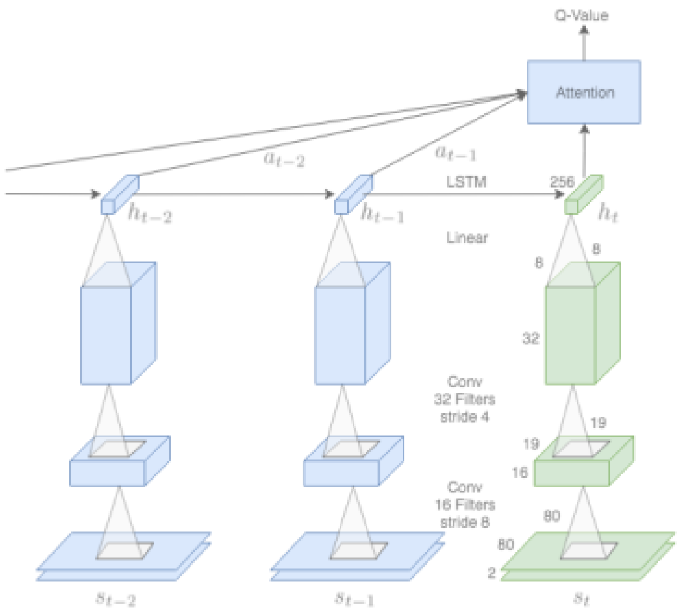

FIGURE 2.1: DRQN with attention layer [Sorokin et al., 2015]

2.1.4 Evolutionary Algorithms

2.1.4.1 Overview

Evolutionary algorithms are a wide category of AI. The idea is inspired by evolutionary biology and natural selection.

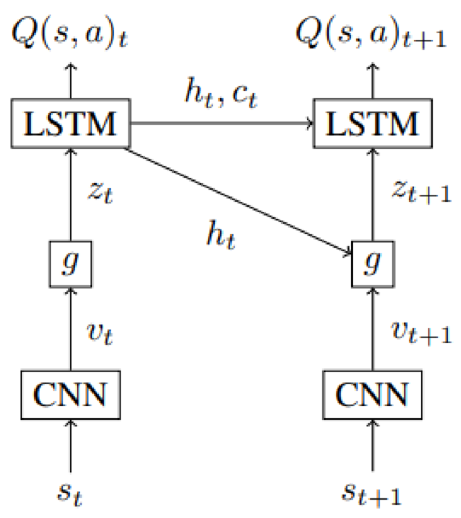

FIGURE 2.2: DRQN with attention [Sorokin et al., 2015]

The main idea is to have multiple agents (population) evolving together. Each agent represents a potential solution. Agents are evaluated thanks to a fitness function, it defines how well an agent performed. They are evaluated, ranked, a new population is created then a selection is applied to the old generation. This process repeats until a solution is found.

There are multiple methods to perform the selection once the fitness has been calculated. The most commons are:

- **Roulette Wheel Selection** (also called Fitness Proportionate Selection): the probability of selecting an individual is $\frac{f_i}{\sum f_k}$. It can be represented as a wheel being spin to select the individual. [Golberg, 1989]
- **Tournament Selection**: randomly take individuals from the population, take the best one of them [Goldberg and Deb, 1991]
- **Ranked based Selection**: The fitness of the individuals is modified to their rank. For example, the worst individual gets a fitness of one, and the best individual a fitness equal to the population size. Then a roulette wheel selection is done with this new fitness. [Goldberg and Deb, 1991]

In addition, all these methods have parameters that allow them to either apply more selective pressure or encourage diversity.

To create a new population, multiple methods can be used some may be specific to an EA category. First, an agent will inherit from one individual, all the data from an agent will be copied. From there you can apply mutation, randomly modifying a part of the data. The new agent can also inherit from multiple agents. In this case, there is a crossover/recombination, this mechanic is specific to the implementation of the EA. For example, a simple idea to inherit from two agents would be to take half of the data from one and half of the data from the other one. In addition, crossovers create a lot of diversity. However, if crossovers are made without much consideration it can lead to poor results, this is why there are often rules to guide the crossover.

Co-evolution is a notion we will rely on for this project. Co-evolution learning defines a way of calculating the fitness. It calculates the fitness based on the interaction between the individuals. So, the fitness is subjective and depends on the population. It is particularly useful the cases where a single fixed fitness function is difficult to establish. This differs from traditional EA.

2.1.4.2 Collaborative co-evolution

Collaborative co-evolution consists of evolving multiple agents working together to find a solution, each agent is a part of the solution.

Collaborative co-evolution has been studied for evolving neural network. One research worked on symbiotic adaptive neuro-evolution (SANE) system [Moriarty and Miikkulainen, 1997]. Instead of having each agent representing a neural network, neurons evolve together by being mixed up at every iteration (figure: 2.3). In addition, some neurons form symbiotic relationship, it means that they will likely be matched together in the same layout for the next generation.

This co-evolution method helps to keep a high diversity and claim to have better performance and adaptation than standar elite/tournament method.

I think that such an algorithm is hard to setup, it requires a fine-tuning of the relations between neurons and how you create neural network with them. In addition, in nowadays development environment, such a method can lack performances compared to more supported algorithms.

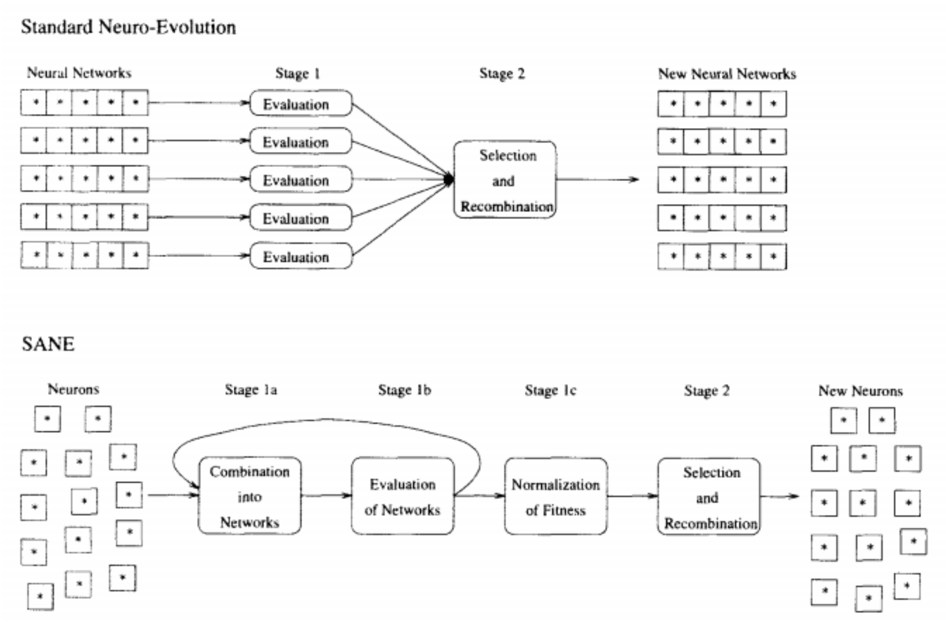

FIGURE 2.3: SANE method compared to Standard Neuro-Evolution [Moriarty and Miikkulainen, 1997]

2.1.4.3 Competitive co-evolution

Competitive co-evolution consists of multiple agents fighting together to have a higher reward. The fitness is directly based on the competition among the individuals.

One of the major problems of this technique is that it can quickly converge or and get stuck on a sub-optimal solution. In fact, if an agent happens to be stronger than the others then it will ultimately converge to it, even if it is a weak solution [Rosin and Belew, 1997]. To reduce this effect, methods should be used to ensure diversity within the population and speed up the convergence to a perfect solution.

Teaching tests is a method that consists of injecting agents that have a fixed strategy. Those would help distinguish an optimal agent from a sub-optimal agent dominating the population [Rosin and Belew, 1997].

It is important to keep track of the previous agents, it would ensure that the algorithm is not making cycles and that future agents defeat the previous generation [Rosin and Belew, 1997]. This can be done by injecting agents into the teaching set while training.

Fitness sharing is a method to promote diversity within the population. It is defined to take into account similarity between individuals and lower the fitness given the number of similarities. An idea is to consider each agent as a source of reward and that the reward needs to be shared among all individuals that defeat it. The fitness assigned to an agent is defined by $\sum_{j \epsilon X} \frac{1}{N_j}$ where N_j is the number of agents defeating agent j and X is the population set [Rosin and Belew, 1997]. Given this method, if a promising agent emerges and defeats an agent that only a few ones can defeat, it will likely be kept in the future generation. In addition, it reduces the convergence to a sub-optimal solution that exploits only a part of the population because the fitness will be divided.

2.2 Imperfect game solving

2.2.1 Overview

Imperfect game solving is an active field of research, this is a challenging topic for AI. It requires to deal with a high number of possibilities, this encourages to push the scalability limits of some AI models. In addition, it has to deal with unknown information. Furthermore, the evaluation process is tricky, there is no universal way to judge if one action is good or not. Usually, the evaluation is done by playing against other agents/individuals.

Imperfect game solving knowledge is not only about games. It is a way to use AI to works with unknown information. Real-world use cases often deal with unknown information, for example, finance, negotiation, strategic pricing, etc.

One of the most studied game in imperfect game solving is poker. This game has multiple advantages:

- It has multiple variants from simple ones such as Kuhn poker to complex ones such as texas hold'em no limits.
- It has a wide community with a lot of professional players. This is an important point because the evaluation process may involve playing against professionals players with money, which makes the game results non-biased.
- Strategy matters, player choices have an important impact. A good agent playing poker against a weaker opponent will have a high chance to win.
- It is a complex game. Bluffing is an important part of the game and is a complex notion.

In the following section, we will discuss two main models used in imperfect game solving, Decision Tree and Neural Network. Then we will introduce the notion of exploitability.

2.2.2 Decision Tree

In imperfect game solving decision trees are widely use and most of the existing approaches use it.

Decision trees are tricky to build for imperfect games because to compute the outcome of a decision it requires to take into account all the possible hidden states.

Ultimately the Nash Equilibrium strategy does not care about hidden state because there is no information about it. An optimum action is given a known state. However, during the equilibrium solving or estimation, the hidden state must be used to estimate/calculate the reward.

For a medium/large game state, abstractions are made to limit the number of representations. For example, in poker, similar combinations of cards are grouped [Johanson et al., 2013].

Successfully poker bot such as Libratus [Brown and Sandholm, 2018] or Baby Tartanian [Brown and Sandholm, 2016] estimates equilibrium on a large game state using MCCFR (Monte Carlo Counterfactual Regret Minimization). It is a reinforcement learning method consisting of playing against itself and updating the strategy based on how much the AI regrets not have selected an action in the past. This process is pre-computed for the first stages, then due to the complexity, it needs to be run in real-time.

Decision tree with equilibrium finding is nowadays the way that has proven to works the best for imperfect game solving.

2.2.3 Neural Network

Neural network has been used for imperfect game solving. We will present two use cases, one using supervised learning and the other one reinforcement learning.

One of the most successful uses of Neural Network in imperfect game solving is DeepStack [Moravčík et al., 2017], a poker bot. It uses neural network to approximate the opponent's counterfactual values and so decide which action to take (figure 2.4). One neural network is trained for each stage of the game, each one uses the output of the previous neural network as input. They trained these neural networks using supervised learning from randomly generated poker situations.

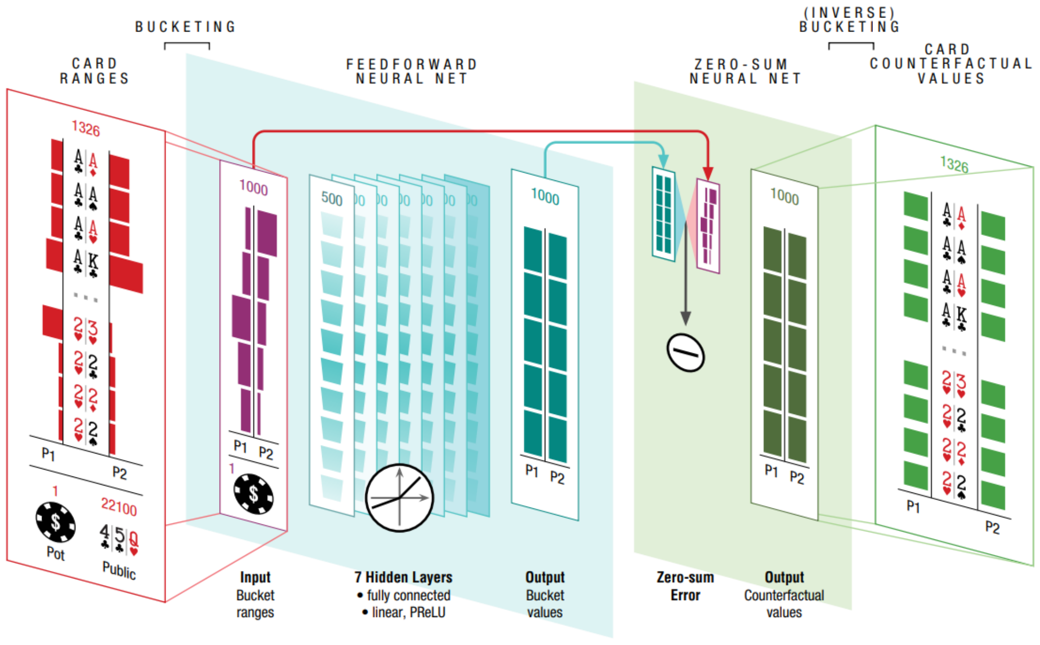

FIGURE 2.4: DeepStack counterfactual value network [Moravčík et al., 2017]

Rlcard [Zha et al., 2019] is an implementation of Deep-Q learning [Mnih et al., 2013] for imperfect game solving. It also uses NSFP (Neural Fictitious Self-Play) to manage the learning environment and DeepCFR to train the agent. The neural network is always fed with the same "info set" which is an absolute representation of the game. The idea is to bind an action to an info set thanks to neural network. It has proved that neural network and reinforcement learning can perform well in this area. In addition, this implementation is not specific to a game and can be applied to any imperfect game. On the other hand, due to this not specific implementation, it can not have abstraction. So, games with a large game state such as poker texas hold'em are not expected to have good performance.

2.2.4 Exploitation in imperfect game solving

Exploitation aims to identify and exploit the weaknesses of an opponent. Opponent modelling is the process of identifying the way the opponent is playing. If an opponent model is accurate exploiting it is straightforward.

Exploitation is a tricky process because it comes with exploitability. While deviating from the optimum to maximize the reward based on the previous action of the opponent there is a chance to be exploited. This is known as the "get taught and exploited problem" [Sandholm, 2007].

It is called safe opponent exploitation when exploitation is done without possibility of being exploited and unsafe opponent exploitation when the possibility of being exploited is assumed.

2.2.4.1 Unsafe opponent exploitation

One research has been done on modelling the opponent in large information games [Ganzfried and Sandholm, 2011]. They decided to assume that by exploiting an opponent they were going to be exposed to exploitation. They used an algorithm called Deviation-Based Best Response (DBBR), which statistically learns the deviation made by an opponent from an approximated equilibrium. In this way, it can exploit the statistical mistake made by an opponent.

One interesting thing about this research is that it relies on concepts available in every game, which is Nash-equilibrium and best response. This means that their method can be generalized to other imperfect games.

On the other hand, this algorithm has major issues, it requires to play a high number of games against an opponent to have a relatively good model. Even with that, the model will only be accurate in the early stages because it will not have enough sample of the opponent strategy in the middle/late. Indeed, decision trees are immense for imperfect games and it is unrealistic to have a usable model opponent for all the stages of the game with this algorithm.

The exploitation is also purely statistical, it can not model complex behavior. It also treats all the games with the same weight, it does not take an account the temporally of each game. For example, it would be impossible to model anger, the fact that if an opponent loses the previous game it will play aggressively the next one.

2.2.4.2 Safe opponent exploitation

Safe opponent exploitation is the idea of exploiting an opponent without the capacity of being exploited. It was stated that it was not possible [Ganzfried and Sandholm, 2011].

However, a recent research on safe opponent exploitation has proven that it was possible on some game [Ganzfried and Sandholm, 2015]. They introduce the notion of gift, profitable strategies that can be used by identifying sub-optimal opponent behavior. These gifts are defined formally for each game and can not be generalized.

Safe opponent exploitation relies heavily on equilibrium and plays the equilibrium strategy most of the time. During the evaluation process of this study, they played against dump agents, that for example always fold. It is hard to estimate how important does safe exploitation would make a difference with a more subtle sub-optimal agent.

The main issue with safe exploitation is that the extra reward from exploiting the opponent is limited by an important set of rules. In addition, the safe exploitation is game-specific and is impossible to generalize, it needs to be re-proven for each use case.

Chapter 3

Methodology

In this section, we will cover the implementation details of this project.

First, we will discuss the different neural network architectures then we will discuss the training of these neural networks and finally how we are going to evaluate them.

We are assuming to know the equilibrium. It is a shortcut we are making because a lot of research had been done on equilibrium finding (section: 2.2) and if we assume to know an equilibrium or a partial equilibrium this would help to reduce the input of the neural networks and improve the learning time.

We are aiming to have an algorithm that is universal to every imperfect game. To do so we are relying on notions that are common to all imperfect games, such as equilibrium and the reward of a game.

3.1 Neural Network Architecture

There will be two neural networks, one that builds the opponent model and one that decides which actions should be taken during a game. I have decided to use two NN because I believe that the opponent modeling NN requires to have the knowledge of a full game. In addition that will allow to have a specific set of information for the input of these NN and so improve the learning speed.

For optimization reasons, we do not want that these neural networks have to learn how the game works. Instead, we will try to process the input feed to these neural networks with the game knowledge. For example, giving the equilibrium of an action instead of giving the info set (the info set is the representation of the public and private information). It is a challenging constraint but we believe that it will enable this NN to scale way better.

3.1.1 Opponent modelling NN

For the opponent modeling neural network, it needs to learn how the opponent play over time. To do that it needs to have multiple knowledge.

3.1.1.1 Knowledge of previous games

It needs to know previous games. It is done by having memory. Either an internal memory such as LSTM (section: 2.1.2.1) or by having a manual memory which is using information from the output of the neural network as its input. These kinds of neural networks are RNN (section: 2.1.2).

3.1.1.2 Knowledge about game history

It needs to know the game history. The game history consists of the multiple actions that had been done during a game. With a game history, you're supposed to be able to know every step of the game. A simple idea would be to give the different info set and the different actions taken by each player. However as said previously we do not want to feed the info set to the neural network.

Instead our idea is to represent the game history with a succession of:

- agent action
- opponent action
- equilibrium for the opponent action

Like this, the game knowledge is encapsulated in the *equilibrium for the opponent action.*

The equilibrium for an opponent's action is not always known. Indeed, to know it we need to have access to the private information of the opponent. In some cases, this information will not be known and we can not ignore this case.

One approach to this problem is to treat the two cases separately [Ganzfried and Sandholm, 2015]. However, because we are using a neural network the size and meaning of the input are rigid. That would require having two different neural networks. For the sake of simplicity and optimization reasons, we decide to not treat this.

One thing worth mentioning is that even if we do not know the private information of the opponent we can still use the equilibrium to give game knowledge. Indeed, instead of having a specific equilibrium for the information set, you can have a global equilibrium. For example in poker, if you're the first player you should bet 50% of the time (the % is just here, for example, it depends on the poker variant). So, if you notice that the opponent bet only 10% of the time you can exploit it. It means that this global equilibrium is valuable information.

We decided to represent both of these equilibrium knowledge for the opponent's action. The global equilibrium and the specific equilibrium. If the private information of the opponent is unknown we can not have a specific equilibrium and so will just set the specific equilibrium values to -1 to indicate that no values are available. Figure 3.1

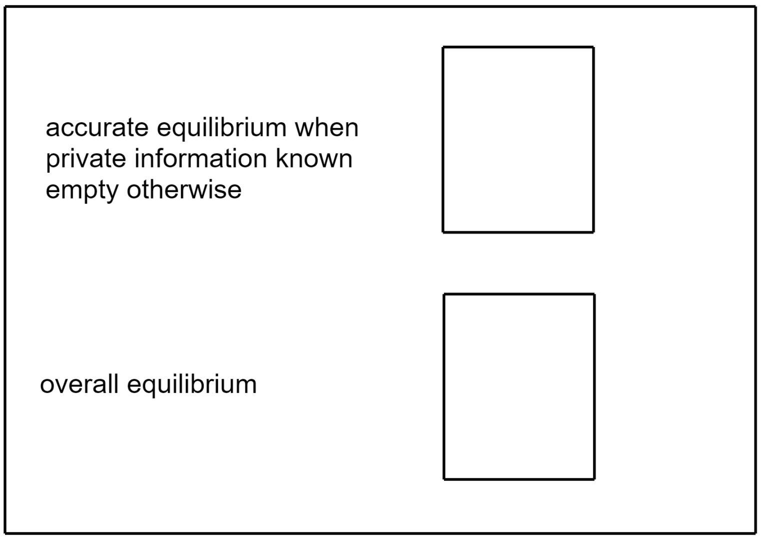

Figure 3.1: Equilibrium for the opponent action

Also, another representation that we are thinking of is representing either the specific or the global equilibrium depending if we know the private information of the opponent.

And to indicate the NN that it's the specific equilibrium we will have a flag (a binary value) set to 1 if it is the specific equilibrium.

Also to not mess up with the order of the input, if the opponent start we leave empty the first agent action.

Finally, because the input size of NN is constant we can not change the size of the input depending on the number of actions done in the game. To overcome this issue we will left empty a part of the input if the game stopped before.

3.1.1.3 Knowledge about game result

In the effort of feeding up the NN with game knowledge, we believe that the game result is valuable information that will greatly improve the performance of the NN.

The game result is as its name suggests the outcome of the game. For example in poker who won and how much money the player won.

This is the only part of the NN that is game-specific because each game can have valuable outcome information. However, we believe that it is not a problem for the generalization of this algorithm to other imperfect games. Indeed the outcome information is easy to retrieve and in most of the games, it only represents a small set of information.

3.1.1.4 Conclusion

In figure 3.2 you can find a sum-up of this information with a RNN architecture where the opponent model also represents the NN memory. In figure 3.3 it is the same information however the memory is managed internally as an LSTM would do.

3.1.2 Decision making NN

The decision-making NN is the one that will use the opponent model to decide which action to take within a game.

It needs to have information about the game history, we use the same representation as for the opponent modeling (section 3.1.1.2) however we do not add information about the

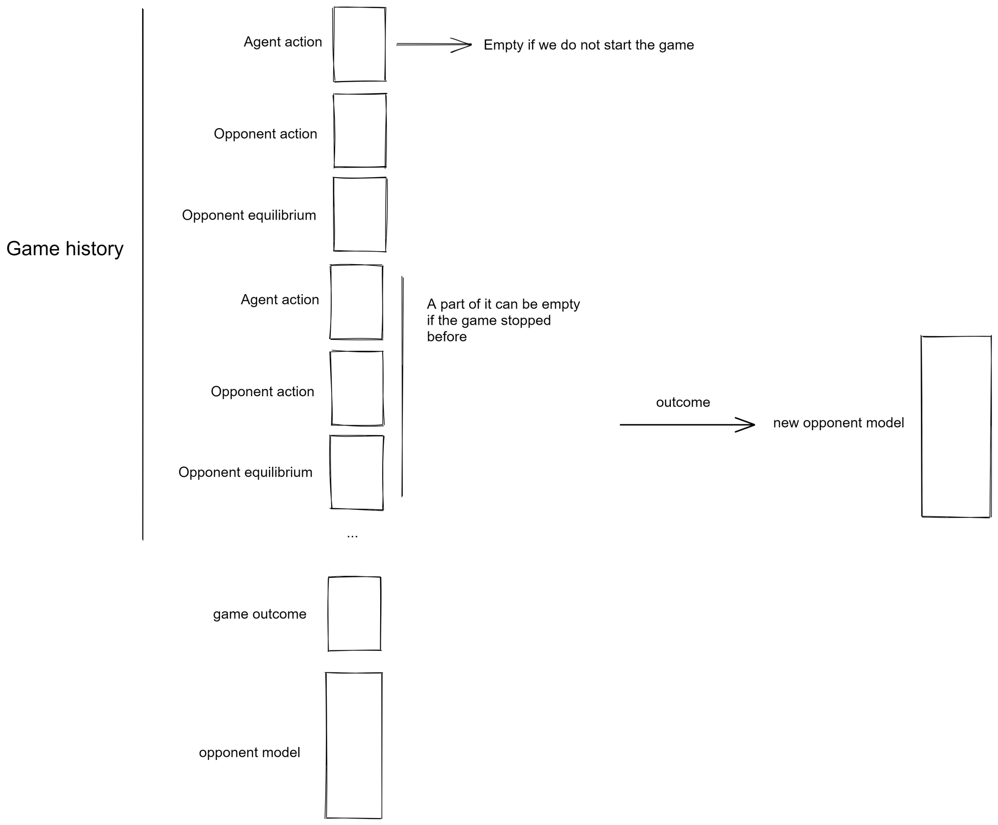

FIGURE 3.2: Opponent modelling neural network, simple RNN

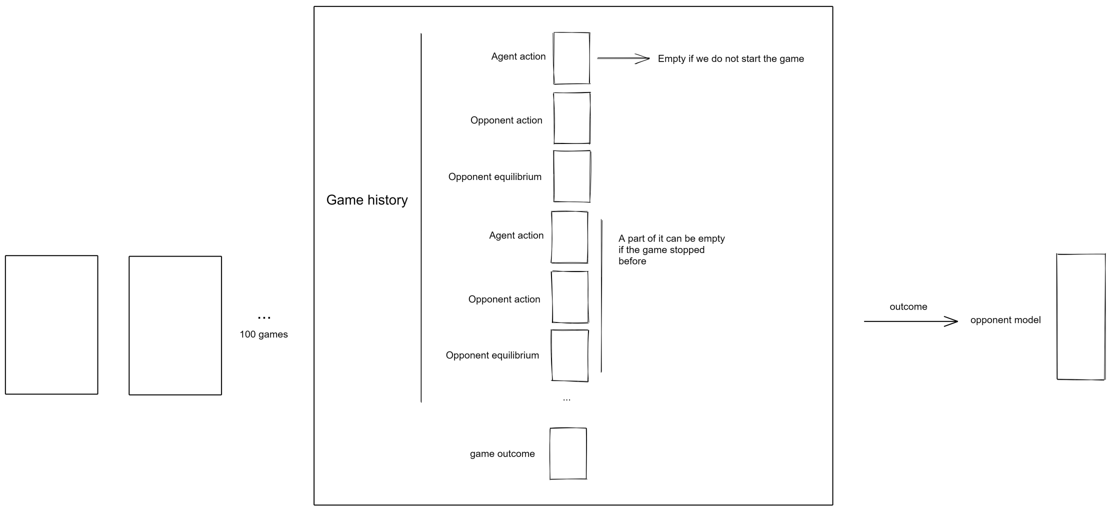

FIGURE 3.3: Opponent modelling neural network, LSTM kind

opponent equilibrium. This might change if we see that including the global equilibrium for each agent action improve the performance.

In addition, we are giving the equilibrium of the current action for the agent. It continues the effort of feeding knowledge of the game to the NN.

You can find a schema of this neural network in figure 3.4

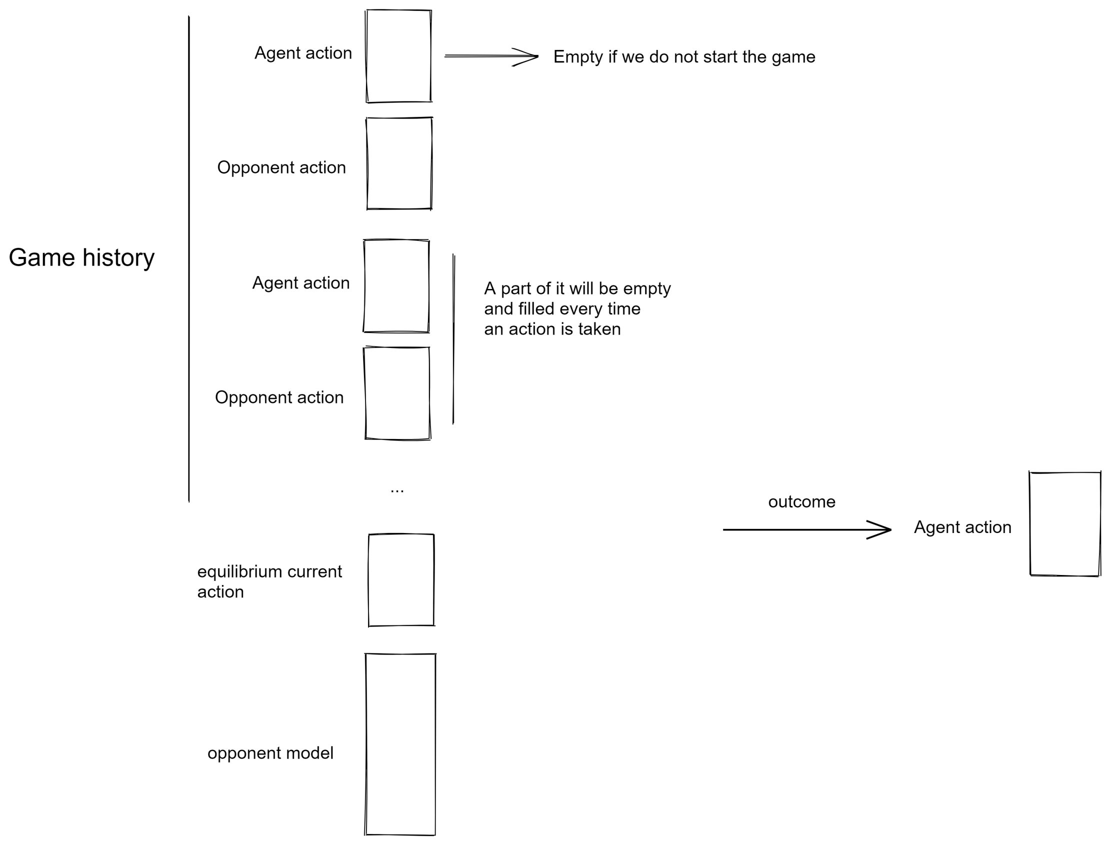
Game history
Agent action
Empty if we do not start the game
Opponent action
Agent action
A part of it will be empty and filled every time an action is taken
Opponent action
...
outcome
Agent action
equilibrium current action
opponent model

FIGURE 3.4: Decision making NN

3.2 Training Neural Networks

The training of these neural networks is challenging and is the core difficulty of this project. Indeed we do not just expect our algorithm to be the best on the long term. We also expect it to be able to detect and exploit opponents.

3.2.1 Attribution of reward

We can not attribute or pre-compute the expected result given an input. Indeed the opponent modeling and decision making are too complex to calculate this. Instead, we attribute a reward given the actions taken by the agent.

The attribution of a reward is subtle and should be done carefully. We can not have a reward for every action the agent is making, it is not an instant reward.

For example, if the agent bet with bad cards maybe it is his strategy to exploit the opponent who is afraid when the agent bet. Given so we can not assign a negative reward to this action because the action takes part in a more advanced strategy. The reward needs to be given when the agent loses or wins something. In our case, if the agent ends up winning the game it will have a positive reward and a negative reward if it ends up losing the game.

It is a general idea of how attributing the reward, this can be tune given the learning algorithm used.

3.2.2 Learning environment

The learning environment needs to make sure that the agent can learn how to exploit complex behavior. This is a difficult constraint because it means that the agent must play against multiple sub-optimal opponents.

This requires to have generated opponents created by the learning environment or/and have opponents with a fixed strategy defined before the learning starts. Having only opponents with a fixed strategy will cover only a few behaviors. Also, it makes complicated for this algorithm to be used in other imperfect game because it will require to find/create other opponents. So having only fixed strategies while learning should be avoided.

3.2.3 Training algorithms

We consider two kinds of algorithms that fit the training constraints. Competitive co-evolution and reinforcement learning.

3.2.3.1 Competitive co-evolution

Competitive co-evolution (section: 2.1.4.3) is the most promising kind of algorithm in its capacity of generating complex opponents and attributing a dynamic reward.

Competitive co-evolution makes multiples agents compete against each other. So it generates a set of sub-optimal opponents by design and makes it evolve into more complex agents over the training. Also, these agents start with simple behavior and the more the training continues more complex these agents became. It allows having incremental learning from simple to complex behavior.

Competitive co-evolution allows by design to have shared fitness. As said in section 2.1.4.3 it allows to promote diversity and encourage the development of promising behavior.

For evolving a NN with this algorithm, every weight needs to be encoded in a solution. Usually, EA has trouble scaling when the search space is too large. Neural networks tend to have a lot of weight, this can lead to performance issues if we are using EA to evolve it.

3.2.3.2 Reinforcement Learning

As said in section 2.1.3, reinforcement learning is made to work with rewards and have really good performance for evolving NN.

The main issue with this method is that it uses backpropagation to evolve the NN and it is hard to implement if multiple NN are involved. In our case, I think that backpropagation can be done even if we are using two NN. The reason is that a part of the decision-making input is the output of the opponent modeling NN, so we can theatrically propagate the error from one NN to the other. However, this would require extra research and work to implement it.

The second issue is that by design reinforcement learning does not work with multiple agents, in fact, it only evolves one NN. To meet our learning environment constraint (section: 3.2.2) we would need to manually find/create opponents with a fixed strategy. This is something we should avoid.

3.2.4 Conclusion

To conclude Competitive co-evolution perfectly fit the learning environment but may suffer from performance issue if the size of the NN is too important. On the other hand, reinforcement learning has really good performance for evolving NN but the solution is likely to be incomplete and suffer from a lack of diversity in the learning environment.

3.3 Evaluation

The evaluation is a crucial process for this research. It is the only way to demonstrate the capacity of neural network to model opponent behavior. Better the testing will be, the better these results will be generalized, without the need to run more experiments.

We will use poker to run our experiments. It is the most used game in imperfect game solving research as it includes complex behavior and strategy. The poker variant used will depend on the size of the game state we want to test.

Performances will be measured in milli-big blinds per hand (mbb/hand), the average number of big blinds win per 1000 hands. This is a common unit in poker research. Following this standard will allow these results to be comparable with other research papers.

3.3.1 Exploitation testing

The exploitation testing will ensure that the algorithm successfully extracts simple opponent behavior. Even "stupid" opponents can be interesting to study because they can give metrics on how fast the algorithm exploits them.

Here is the list of the different algorithms:

- Always fold
- Always rise
- Random: plays random moves

3.3.2 Exploitability testing

Exploitability testing will provide information on how much the algorithm is exploitable.

It will play against an opponent following the equilibrium strategy. Given that, we will be able to see how reasonable our algorithm is and how well it managed to not fall into trying to exploit the equilibrium.

It will also be evaluated using the true best response with access to the private game state. In other words, at each step, the true best response will be computed with access to the card of every player. This is a non-realistic testing because it is cheating but it provides an interesting insight into how much the algorithm is exploitable.

We will also evaluate how far the algorithm is flexible and able to adapt if the opponent drastically switch his strategy. To do so we will have an opponent playing random for 100 games then playing the true best response for the remaining of the day.

3.3.3 Advanced opponent testing

In this testing category, we are trying to reproduce real-world poker examples. We will challenge our algorithm with different sub-optimal opponents.

We will use made up sub-optimal strategy by playing each action with a probability chosen uniformly randomly within 0.2 of the equilibrium probability. This type of sub-optimal opponent has been used in previous research in opponent modelling [Ganzfried and Sandholm, 2015]. It will be easier to compare the results.

We discard the usage of human testing for this research because it will require financial resources to have non-bias and conclusive results.

Chapter 4

Kuhn poker experiment

In this chapter, we will present and explain the experiment on a simplified version of poker, the Kuhn poker. First, we will describe how we setup the neural network and how we trained it. Then we will present the result and the different metrics on how well the neural network performed. Finally, we will analyze the result and try to generalize them to any imperfect game.

4.1 Setup

4.1.1 environment

We created a Kuhn poker environment that gives us all the necessary data to correctly feed the neural network. We have defined 3 actions possible: check, bet and fold. We have decided to regroup raise and bet in the same action ”bet” to reduce the complexity. If at any time the decision-making NN predicts an incorrect action, the default action ”bet” or ”check” would be done.

4.1.2 NN

We decided to first use a simple RNN for the opponent modeling, you can find more information about the specific input of this NN in section 3.1.1, see figure 3.2. We used the decision making NN describe in section 3.1.2, figure 3.4.

Different layers size has been tested, from 10 to 20 for the hidden layer size of the descision making NN and from 5 to 20 for the opponent modelling NN. We also tried to add an other hidden layer for both NN. activation function, and opponent model size has been tested. We did not notice any improvement while increasing the layer size above 10 for both NN.

Different activation function has been tried, linear, sigmoid and relu. The result between the sigmoid and the linear function are pretty similar. However the relu had bad performance compared to the linear.

Different opponent model size has been tested, from 5 to 20. While analyzing the neural networks we have seen that the decision making NN uses at most 5 input, the others were not impacting the decision. However it is worth to notice that due to the way our training is done, it makes sense to use a larger opponent model size because we can not except all the opponent model output to be meaningful.

Only the most successful architecture will be presented, their results will be detailed in the section 4.2. The descriptions of their NN layers are detailed in table 4.1. The size of the input layer is defined by the environment and the opponent model size. It has been detailed in the section 3.1.

NN name	opponent-modelling NN	descision-making NN
input layer	x	x
hidden layer activation	10 linear	10 linear
output layer activation	opponent model size sigmoid	3 (check, bet, fold) softmax

Table 4.1: NN layer (kuhn poker)

4.1.3 Training

To train the NN we have used competitive co-evolution, general information about it can be found in section 3.2.3.1.

We have used these strategy in the teaching set:

- **Equilibrium**: playing the equilibrium strategy
- **Random**: playing a random action in the set of the available ones

- **Always bet**
- **Always fold**
- **Close Equilibrium**: playing a strategy close to the equilibrium. It select each action with a probability chosen uniformly randomly within 0.2 of the equilibrium probability. It aims to produce realistic sub-optimal opponent.

We have by default added 3 different *Close Equilibrium* strategies to the teaching set. We have not used a system to dynamically add training agents to the teaching set. We are making each agent playing against each other and using a shared fitness to promote diversity (ref: 2.1.4.3). To make the result of the games not based on cards luck, we are first making the two agents play a set of rounds. Then, we erase their memory and make them play with the opposite cards. A temporary reward is attributed, the agent winning 1, the one losing -1, then the final reward is attributed after the shared fitness calculation.

The EA has been used with these parameters. These parameters have been decided after multiple iterations, those seem to have good training performance in this context. For reference, we have put in italic the range of the values that we looked at.

- population size: 20 ; *10 to 25*
- individual kept each iteration: 5 ; *2 to 5*
- mutation rate: 0.1 ; *0.1 to 0.2*
- iteration: 100 ; *10 to 100*
- number of rounds (games): 25 ; *5 to 25*

You can find pseudo-code for the EA in the appendices A.1.

4.2 Result

To evaluate the NN we apply the evaluation method detailed in the section 3.3.

In this section, results of multiple NN models will be presented, the NN and EA parameters are by default the one described in the previous section 4.1.

- **Classic**: No parameters changes.
- **More Games**: Playing 50 games per iteration (instead of 25). Only 10 individuals and 3 are kept at each iteration.
- **Bigger Model Size**: The opponent model size is 10 (instead of 5)
- **Larger Teaching Set**: Having 6 *Close Equilibrium* agents in the teaching set

Due to the limited resources, the models are from a single run of the EA. For the record, one run takes approximately 2h, performances are discussed in section 5.

4.2.1 Win rate

In this section we will present the win rate of the different tested models against multiple strategies, the choices of these strategies are detailed in the section 3.3. We made the agents played together 500 games, erased their memory then played another 500 games with the opposite cards. You can find the results in the table 4.2.

It is worth mentioning that the *Close Equilibrium* strategy during the evaluation is not one of the training set.

Opponent Model	E	Close E	Random	Always Bet	Always Fold	TBR	Random then TBR
Classic	0.001	0.053	0.202	0.293	0	-0.182	-0.139
More Games	0.006	0.001	0.312	0.200	0.646	-0.175	-0.116
Bigger Model Size	-0.02	0.027	0.309	0.180	0.106	-0.087	-0.065
Larger Teaching Set	0.001	0.054	0.196	0.350	0	-0.172	-0.120
Equilibrium	0	0.007	0.150	0.104	0.238	-0.125	-0.118

TABLE 4.2: Win rate in mbb/hand of the opponent model against evaluation opponent for the kuhn poker (Equilibrium has been shortened to E and true best response to TBR)

4.2.1.1 Sub-optimal opponent exploitation

We can see that the model *Classic* have really good performance against *Close Equilibrium* (0.054) compared to the equilibrium (0.007). It demonstrates the capacity of the NN model to exploit sub-optimal opponent.

In the study [Ganzfried and Sandholm, 2015], they have an agent playing the best response according to the opponent model that they use for reference to demonstrate how unsafe it is. This algorithm have similar result as **Classic** regarding the *Close Equilibrium* (called sophisticated static in this study). However, the main difference is that the model *Classic* is less exploitable. Indeed comparing the difference between equilibrium and the algorithm for the random then TBR (called dynamic in the other study). It gives us 0.021 for *Classic* and 0.085 for the best response in the study. To sum up, using NN we are capable of achieving as good results as algorithms fully committed to opponent exploitation while being less exploitable.

4.2.1.2 Impact of training

We can notice that model *Classic* and *Larger Teaching Set* perform poorly against always fold, an opponent that is supposed to be easily exploitable. We can assume it is linked to the way NN are trained. In competitive EA with shared fitness, it is likely that a strategy exploiting a stronger opponent (always bet) will have a better fitness compared to exploiting a simple opponent (always fold). Polishing the EA training would likely improve and stabilize the results over multiple runs.

In addition EA are known to be stochastic, meaning that the results are hard to be predictable. That is because of the random elements such as the initialisation, selection and mutation. To have a good and predictable result, you are likely going to need to run multiple times the process and select the best of them.

4.2.1.3 Exploitability

The true best response help understands how much an algorithm is exploitable, it has access to every player's cards and plays the best response every time. It is not an absolute metric but highlight if an algorithm is likely to be exploitable.

An interesting thing about the *Bigger Model Size* model result is that it performs better than the equilibrium against the *TBR* and **Random then TBR**. Both of these strategies are "cheating" by having access to the opponent private information to produce the true best response. We can assume that the NN is noticing that the opponent is particularly good and decide to play super safe minimizing the loss. It demonstrates that using NN to model the opponent is not easily exploitable. Most important, it demonstrates that it can reduce the exploitability if the opponent is aware of a part of your private information.

In real-life use cases, it is frequent that some of your private information is known by the opponent. For example, if you are negotiating a deal, the person may know the budget of your company while you think it is private information.

4.2.2 Reward over time

In this section, we will present the interesting plots of the reward over time for different models.

The different plots showcase in this section are based on 1000 played games. It is worth to note that the EA training has been run having only 25 games played between each agent. Using a different scale (from 25 to 1000) demonstrate how the NN model handle keeping an opponent model through more game than usual.

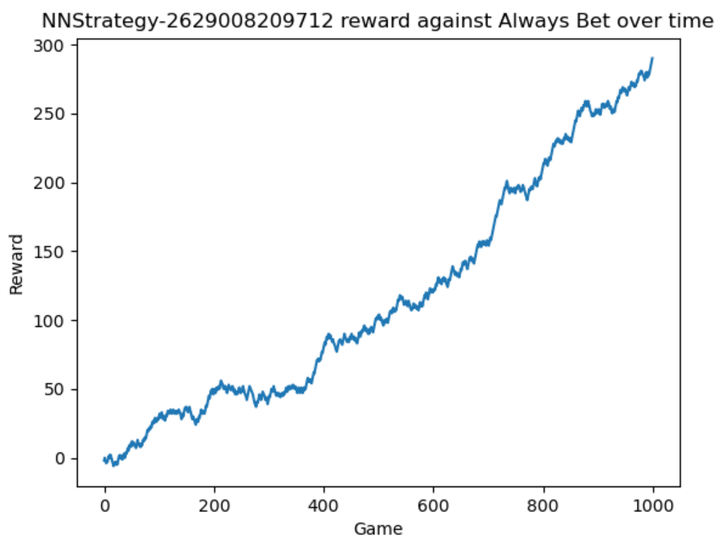

FIGURE 4.1: Model Larger Teaching Set, reward against Always Bet over time

In figure 4.1, you can see that at the very beginning, the reward is neither increasing nor decreasing. Then after some games, we can assume that the NN draws a model of the opponent, and start to exploit it by having a relatively stable increase of the reward.

In figure 4.2, you can see that there is not a straight tendency of increasing or decreasing. It's what you expect from a game against two opponents without net exploitation. It

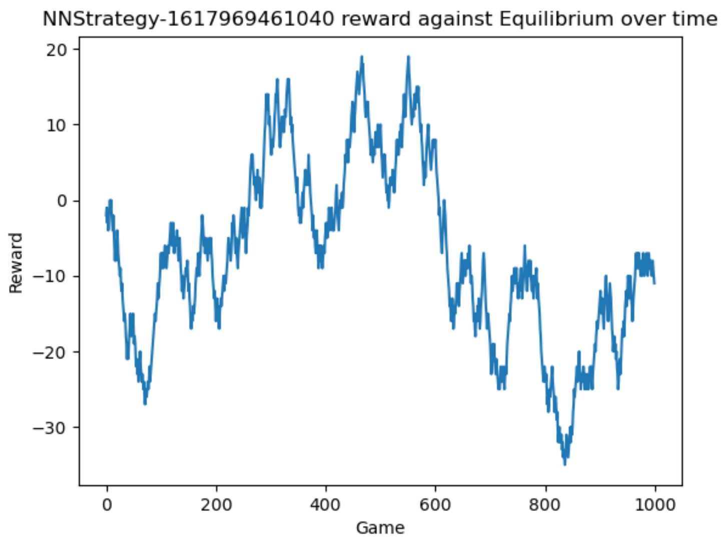

FIGURE 4.2: Model More Games, reward against Equilibrium over time

demonstrates the ability of the model to not fall into a wrong interpretation or at least recover/correct the opponent model.

4.2.3 Input impact on the NN

We have measured the impact of each input to every output of the neural network to determine how much the input is taken into consideration in the decision. To do that we used SHAP [Lundberg and Lee, 2017], a game-theoretic approach to explain the output of any machine learning model. On the plots, inputs are ordered by impact. Each blue dot represents the impact (positive or negative) of one input for one activation.

For the model *Bigger Model Size*, we can see that on the opponent-model-0 input is part of the most impacting input for the bet and check action, figure 4.3 and 4.4. It demonstrates that the NN relies on the opponent model to take its decision.

For the model *Bigger Model Size* we can also analyze how the ”opponent-model-0” is made by analyzing the opponent model NN, figure 4.5. We can see that the most used inputs are the game outcome, the first opponent action, and the first agent action. Interpreting the purpose of this opponent model output is hazardous because NN are complex. For example, a possible analysis is that this part of the opponent model aims to define if the opponent is often winning when he is betting.

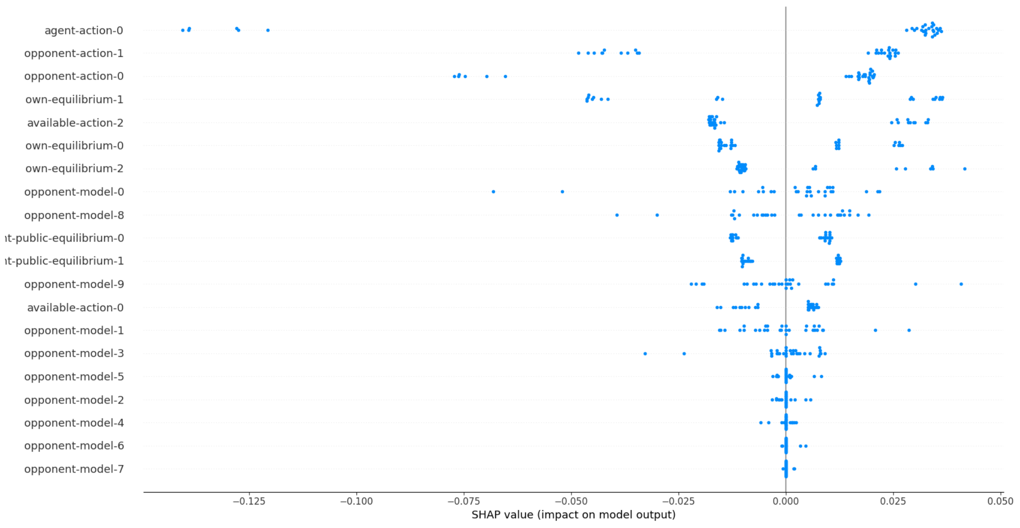

FIGURE 4.3: Bigger Model Size, input impact for the check action

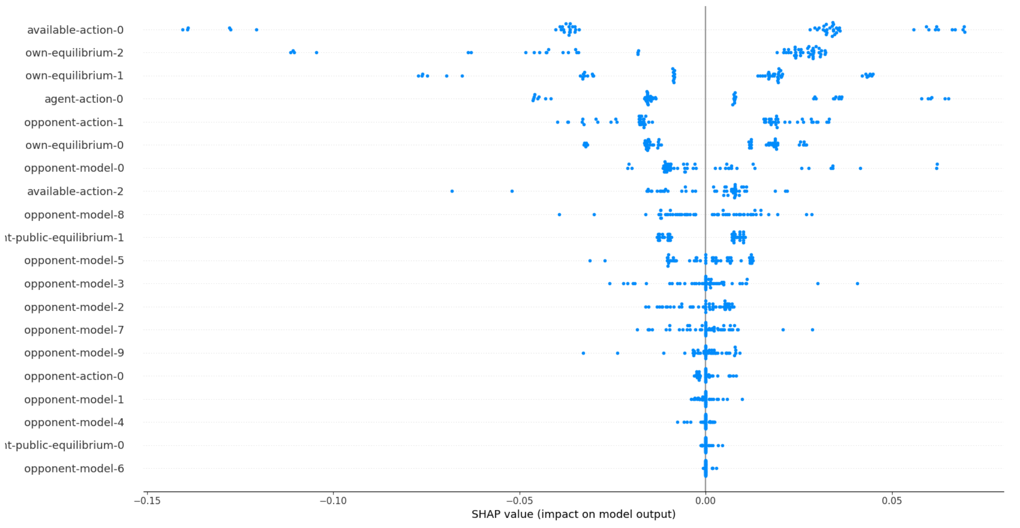

FIGURE 4.4: Bigger Model Size, input impact for the bet action

4.3 Analysis

4.3.1 Approximation or incorrect Equilibrium

During the development, I had a bug that allowed cards to be draw multiple times. It was possible to have the player one king and the second with another king (which is not possible in the Kuhn poker). So it makes the equilibrium incorrect. Even if

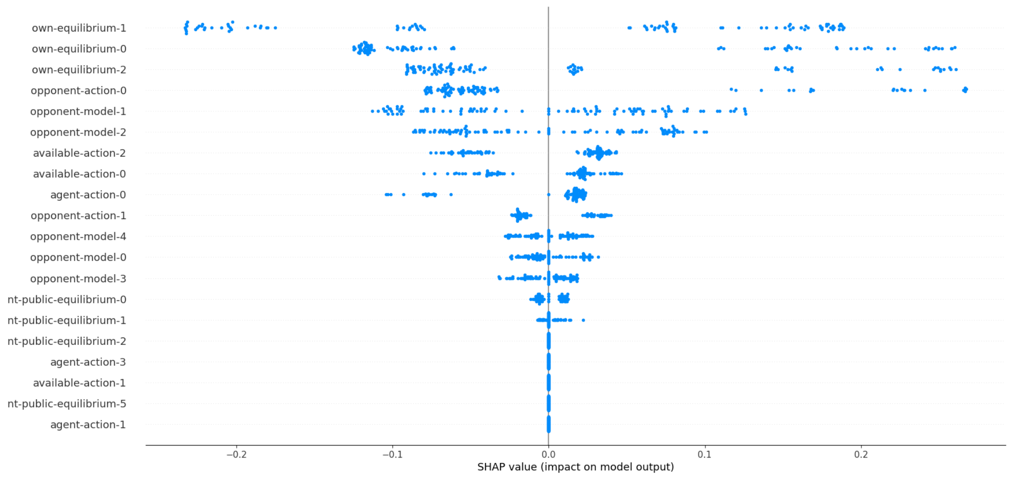

FIGURE 4.5: Bigger Model Size, input impact for opponent-model-0

the equilibrium was incorrect it managed to overcome this and exploit the different strategies. This means that using NN to exploit opponents in imperfect games using this NN architecture can manage and overcome incorrect or approximated equilibrium.

It is worth mentioning that the performances are likely to be degraded because the NN will need to learn the game. Indeed it will likely rely on the game history such as the previous action instead of relying on the equilibrium.

Obviously, this bug has been fixed and the presented result in the previous section are with the correct version.

4.4 Conclusion

With this Kuhn poker experiment, we can draw some general conclusions about opponent exploitation with neural networks in imperfect games.

We have demonstrated the capacity of a NN to model and exploit an opponent.

We have seen that it is way less exploitable than other existing algorithms used to model the opponent. Some models were having really good exploitability results, even better than equilibrium. Demonstrating that they were able to reduce their exploitability if the opponent knew some private information.

We have also highlighted the importance of the training phase and its impact on the quality of the model.

To conclude using NN to model opponents achieves as good results as existing algorithms while being way less exploitable than them.

Chapter 5

Scale up, limit texas holdem poker experiment

5.1 Introduction

In this chapter, we will study how well our algorithm is scaling up on more complex games. To do this we are going to adapt our algorithm to make it works with the limit texas holdem poker variant. This game has an info set size of 10^{14} and 5 available actions.

5.2 Equilibrium approximation

One of the most complex tasks is to approximate the equilibrium. As discussed previously, a lot of algorithm aims to find a good approximation of the equilibrium and it is an active field of research (ref: 2.2). There is a lot of complex algorithm with specific optimization. We decided to implement an algorithm called CFR (Counterfactual Regret Minimization) [Neller and Lanctot, 2013].

Approximating the equilibrium takes a relatively high amount of time, to increase the performances we implemented it in C++ and bound it to our python program.

5.2.1 CFR

CFR algorithm is a machine learning decision tree algorithm. For each iteration, it explores the whole tree with a given starting state. Once it reaches the terminated nodes it can assign a regret to it given the result of the game. This regret is then back-propagated taking into account the probability to reach this node. The probability to reach this node is calculated by creating strategies for each state transition. The goal of these strategies is to find the optimal probability for the player playing. These strategies are updated at every iteration given the regret of the leading nodes. Because this algorithm is applied in a two-player game, it requires paying extra attention to inverse the regret at every step so that each player will try to minimize their regret. You can find the pseudo-code of this CFR in B.1.

5.3 Experiment

5.3.1 Setup

We used the same setup as for the kuhn poker experiment (ref: 4.1). We tried to run it with the following hyper-parameters. They were choose to balance the time a game takes by reducing the overall number of games in one iteration.

- population size 10
- individual kept each iteration 3
- mutation rate 0.2
- number of games 10
- iteration 200

Running one iteration takes 2h30. Due to a lack of resources, I was not able to properly run all the iterations. However, we can still think about how realistic it is to run this algorithm.

5.3.2 Number of trainable parameters

The number of trainable parameters is simply the number of weights and bias in the neural network. Using the same NN layers like the one for the kuhn experiment 4.1, we were able to extract these numbers, you can find it in table 5.1.

Game	Number of trainable parameters
Kuhn poker	754
Limit texas holdem poker	6000

Table 5.1: Number of trainable parameters

We can notice that there are only 8 times more trainable parameters ($6000/754 \approx 8$). We can assume that the training will need 8 times more iterations, so we can approximate the training time: $2h30 * 8 * 100 \approx 83days$. However there is a huge margin to reduce this training time, it will be discussed in the next section 5.4.

5.4 Optimisation

The $83days$ of training were calculated given the specs of my computer (i7-9750H CPU @ 2.60GHz, rtx2060 GPU and 16GB of RAM).

5.4.1 Multi-threading

The first way to drastically improve the performance would be by multi-threading all the games. Indeed almost all the training time is spent making agents playing against each other. Given this, the training time could theoretically be lowered to at least less than 30s per iteration in an optimum setup where each game has a dedicated thread. Even in a non-optimum setup, each thread reduces the training time.

5.4.2 Equilibrium

Another way to improve the algorithm performance on this poker version would be to optimize the time to compute the equilibrium approximation. Indeed, in this version, at least 80% of the game time is spent computing the equilibrium. There is a famous algorithm called MCCFR (Monte Carlo Counterfactual Regret Minimization), studies have shown that its performance is at least 10 times better than the classic CFR [Lanctot et al., 2009], [Lanctot et al., 2009]. Just with this update, this could theoretically bring the training time to $\approx 17days$ taking my computer for reference.

5.5 Conclusion

Scaling up the algorithm is doable, the number of trainable parameters seems to reasonably increase allowing complex games to be trainable. The scalability of the algorithm heavily relies on how fast it is possible to approximate the equilibrium. The more difficult the game will be, the more time it would require to compute the equilibrium and so slow down the training. By multi-threading the algorithm, with enough hardware resources, it is possible to train this algorithm on complex imperfect games in a reasonable amount of time.

Appendix A

EA training algorithm

```
% data structure
Struct Individual {
    won_against = []
    lost_against = []
    fitness = 0
}

% The TeachingIndividual is only here for indication,
% you can have a different set of metadata
Struct TeachingIndividual extends Individual {
    strategy_name: string
}

Struct LearningIndividual extends Individual {
    values: array representing the training parameters
}

% hyperparameters
iterations = 100
mutation_rate = 0.1
trainable_params = 10
nb_population = 10
new_member_per_population = 7
individual_kept_nb = nb_population - new_member_per_population

% initialization
training_individuals = []
FOR _ in nb_population:
    training_individuals.append(LearningIndividual(
        values: random number array from -1 to 1 of size trainable_params,
    ))
teaching_individuals = [
    TeachingIndividual(strategy_name: "equilibrium")
    TeachingIndividual(strategy_name: "random")
    TeachingIndividual(strategy_name: "bet")
    TeachingIndividual(strategy_name: "fold")
    TeachingIndividual(strategy_name: "close_equilibrium")
]

% run
FOR _ in iterations:
    population = training_individuals + teaching_individuals

    % make them play against eachother
    FOR i = 0; i to population.size():
        FOR j = i + 1; j to population.size():
            a, b = population[i], population[j]
            result = evaluate(a, b)
            IF result == WIN:
                a.won_against.append(b)
                b.lost_against.append(a)
            ELSE IF result == LOST:
                b.won_against.append(a)
                a.lost_against.append(b)

    % assign fitness
    FOR individual in population.size():
        individual.fitness = 0
        FOR looser_individual in individual.lost_against:
            individual.fitness += 1 / looser_individual.lost_against.size()

    % generate new population
    selected_individuals = randomly select new_member_per_population
                             of training_individuals proportionaly to their fitness
    FOR new_individual in selected_individuals:
        new_individual.values = randomly change the values from -1 to 1
                                     with a probability of mutation_rate

    % process selection and assign a new population
    individual_kept = select individual_kept_nb best training_individuals
                        based on their fitness
    training_individuals = selected_individuals + individual_kept
```

Listing A.1: EA training algorithm

Appendix B

CFR pseudo-code

```
Class PokerEnv {...}
% A class representing the poker env, it's implementation
% will not be shown in this pseudo-code

Struct StategyInfo {
    float value;
    string action;
}

Struct NodeInfo {
    % you should store an hash key of this env intead of
    % the full PokerEnv, we are doing this for readability
    PokerEnv env
    float regret
    Map<PokerEnv, strategys> strategys
}

nodes = {}

void runOnce(env, proba):
    currentNode = nodes[env]

    if (env.isOver()):
        currentNode.regret = env.reward * proba
        return

    actions = env.getAvailableActions()
    leadingNodes = {}
    total = 0

    % explore and compute values of the leading nodes
    FOR action in actions:
        copy = env.copy()
        copy.play(action)
        if (!currentNode.strategys[copy]):
            currentNode.strategys[copy] = {
                value: 1 / actions.size
                action: action
            }
        runOnce(copy, currentNode.strategys[copy].value * proba)
        leadingNodes.append(nodes[copy])

    sumNegativeRegret = 0
    sumPositiveRegret = 0
    for FOR node in leadingNodes:
        regret = -node.regret
        if (regret > 0):
            sumPositiveRegret += regret;
        else:
            sumNegativeRegret += regret;

    % update strategies
    FOR node IN leadingNodes:
        regret = -node.regret
        if (sumPositiveRegret > 0):
            currentNode.strategys[node.env].value = regret > 0 ? regret / sumPositiveRegret : 0;
        else if (sumNegativeRegret != 0):
            currNode.strategys[node.env].value = regret == 0 ? 1.0f : 1 - abs(regret / sumNegati
        else:
            currNode.strategys[node.env].value = 1;

    % weight the stratgies so it sum up to 1
    float totalStrategy = 0
    FOR node in leadingNodes:
        totalStrategy += currentNode.strategys[node.env].value
    FOR node in leadingNodes:
        currNode.strategys[node.env].value /= totalStrategy;

    % compute the regret of this node
    FOR node in leadingNodes:
        currentNode.regret += proba * -node.regret * currentNode.strategies[node.env].value

env = PokerEnv(public_cards,private_cards,...)
% Initial environemnt of which you want the CFR Value
% It takes all the necessary information to create a moment in the game.
for _ in NB_ITERATION:
    runOnce(env, 1)

% CFR values are in nodes[env].strategies
FOR strategy in nodes[env].strategies:
    env.generateRandomCardsFor2ndPlayer()
    print("Action {strategy.action} -> strategy.value")
```

Listing B.1: CFR pseudo-code

Bibliography

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Brown, N. and Sandholm, T. (2016). Baby tartanian8: Winning agent from the 2016 annual computer poker competition. In *IJCAI*, pages 4238–4239.

Brown, N. and Sandholm, T. (2018). Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424.

Chen, C., Ying, V., and Laird, D. (2016). Deep q-learning with recurrent neural networks. *Stanford Cs229 Course Report*, 4:3.

Ganzfried, S. and Sandholm, T. (2011). Game theory-based opponent modeling in large imperfect-information games. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 533–540. Citeseer.

Ganzfried, S. and Sandholm, T. (2015). Safe opponent exploitation. *ACM Transactions on Economics and Computation (TEAC)*, 3(2):1–28.

Golberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989(102):36.

Goldberg, D. E. and Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms*, volume 1, pages 69–93. Elsevier.

Johanson, M., Burch, N., Valenzano, R., and Bowling, M. (2013). Evaluating state-space abstractions in extensive-form games. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 271–278.

Johanson, M., Waugh, K., Bowling, M., and Zinkevich, M. (2011). Accelerating best response calculation in large extensive games. In *IJCAI*, volume 11, pages 258–265.

Kotecha, N. and Young, P. (2018). Generating music using an lstm network. *arXiv preprint arXiv:1804.07300*.

Kuhn, H. W. (1950). A simplified two-person poker. *Contributions to the Theory of Games*, 1:97–103.

Lanctot, M., Waugh, K., Zinkevich, M., and Bowling, M. (2009). Monte carlo sampling for regret minimization in extensive games. *Advances in neural information processing systems*, 22:1078–1086.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

Moravčík, M., Schmid, M., Burch, N., Lisỳ, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M., and Bowling, M. (2017). Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513.

Moriarty, D. E. and Miikkulainen, R. (1997). Forming neural networks through efficient and adaptive coevolution. *Evolutionary computation*, 5(4):373–399.

Nash, J. F. et al. (1950). Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49.

Neller, T. W. and Lanctot, M. (2013). An introduction to counterfactual regret minimization. In *Proceedings of Model AI Assignments, The Fourth Symposium on Educational Advances in Artificial Intelligence (EAAI-2013)*, volume 11.

Rosin, C. D. and Belew, R. K. (1997). New methods for competitive coevolution. *Evolutionary computation*, 5(1):1–29.

Sandholm, T. (2007). Perspectives on multiagent learning. *Artificial Intelligence*, 171(7):382–391. Foundations of Multi-Agent Learning.

Sorokin, I., Seleznev, A., Pavlov, M., Fedorov, A., and Ignateva, A. (2015). Deep attention recurrent q-network. *arXiv preprint arXiv:1512.01693*.

Thoutt, Z. (2017). Game of throne book 6 generator. https://github.com/zackthoutt/got-book-6.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

Zha, D., Lai, K.-H., Cao, Y., Huang, S., Wei, R., Guo, J., and Hu, X. (2019). Rlcard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376*.