Monte Carlo Neural Fictitious Self-Play: Achieve Approximate Nash equilibrium of Imperfect-Information Games

Li Zhang
Department of Computer Science
Zhejiang University
Hangzhou, China
zhangli85@zju.edu.cn

Wei Wang
Department of Computer Science
Zhejiang University
Hangzhou, China
21621272@net.zju.edu.cn

Shijian Li[*]
Department of Computer Science
Zhejiang University
Hangzhou, China
shijianli@zju.edu.cn

Gang Pan
Department of Computer Science
Zhejiang University
Hangzhou, China
gpan@zju.edu.cn

March 25, 2019

Abstract

Researchers on artificial intelligence have achieved human-level intelligence in large-scale perfect-information games, but it is still a challenge to achieve (nearly) optimal results (in other words, an approximate Nash Equilibrium) in large-scale imperfect-information games (i.e. war games, football coach or business strategies). Neural Fictitious Self Play (NFSP) is an effective algorithm for learning approximate Nash equilibrium of imperfect-information games from self-play without prior domain knowledge. However, it relies on Deep Q-Network, which is off-line and is hard to converge in online games with changing opponent strategy, so it can't approach approximate Nash equilibrium in games with large search scale and deep search depth. In this paper, we propose Monte Carlo Neural Fictitious Self Play (MC-NFSP), an algorithm combines Monte Carlo tree search with NFSP, which greatly improves the performance on large-scale zero-sum imperfect-information games. Experimentally, we demonstrate that the proposed Monte Carlo Neural Fictitious Self Play can converge to approximate Nash equilibrium in games with large-scale search depth while the Neural Fictitious Self Play can't. Furthermore, we develop Asynchronous Neural Fictitious Self Play (ANFSP). It use asynchronous and parallel architecture to collect game experience. In experiments, we show that parallel actor-learners have a further accelerated and stabilizing effect on training.

1 Introduction

With rapid develop of deep reinforcement learning, AI already beats human expert in perfect-information games like Go. However, researchers haven't make same progress in imperfect games like starcraft or xxx. One of the main problem in such researches is that they always did't consider to evaluate their training and results in a theorical and quantitive way, so the effects are difficult to guarantee.

Game theory[1] is the cornerstone of human behavior patterns in real world competitions. It studies how agents can maximize their own interests through competition and cooperation, and can measure the quality the decisions in game. It has become an attractive research task in computer science, the intersection research topic called "algorithmic gaemt theory" has established, and gets more and ore interact with the development of artificial intelligence [2, 3]. One main

[*] Corresponding Author

motivation is as computational dimensions get exponentially bigger for realworld complext problems, like transaction and traffic control, it's necessary to leverage ideas from algorithms and artificial intelligence to make them work in practice.

In Game theory, Nash Equilibrium[1] would be an optimal solution in games, i.e. no one can gain extra profit by alleviating their policy. Fictitious play[4] is a traditional algorithm for finding Nash Equilibrium in normal-form games. Fictitious players repeatedly choose best response to the opponent's average strategy. The average strategy of players would converge to Nash Equilibrium. Heinrich et al.[5] proposed Extensive Fictitious Play, extending the idea of fictitious play to extensive-form games. However, the states is represented in the form of look-up table in each tree node, so that the generalization training (of similar states) would be unpractical; And the update of average policy needs the traverse of the whole game tree which results in dimension disaster for large games. Fictitious Self-Play(FSP)[6] addresses these problems by introducing sample–based machine learning approach. The approximation of best response is learned by reinforcement learning and the update of average strategy is processed by sample-based supervised learning. But to make sampling efficient, the interaction between agents are coordinated by a meta controller and is asynchronous with learning.

Heinrich and Silver [6] introduced Neural Fictitious Self-Play(NFSP), which combines FSP with neural network function approximation. A player is consisted of Q-learning network and supervised learning network. The algorithm calculates a "best response",by ϵ–greedy deep Q-learning, as well as an average strategy by supervised learning of agents' history behaviors. It solves the coordinated problem by introducing anticipatory dynamics — players behaves according to a mixture of their average policy and best response. It's the first end-to-end reinforcement learning method which learns approximate Nash Equilibrium in imperfect games without any prior knowledge.

However, NFSP has bad performance in games with large-scale search space and search depth, because of the nature that opponents' strategy is complex and DQN learns in an offline mode. In this paper, we propose Monte Carlo Neural Fictitious Self Play(MC-NFSP). Our algorithm combines NFSP with Monte Carlo Tree Searches[7]. We evaluate our method in two-player zero-sum board game. Experimentally we show that MC-NFSP would converge to approximate Nash Equilibrium in Othello while NFSP can't.

Another drawback is in NFSP the calculation of best response relies on Deep Q-learning, which takes a long time to run until convergence. In this paper, we propose Asynchronous Neural Fictitious Self-Play(ANFSP), which uses parallel actor learners to stabilize and speed up training. Multiple players choose actions in parallel, on multiple copies of the environment. Players share Q-learning network and supervised learning network, accumulate gradients over multiple steps in Q-learning and calculate gradients of mini-batch in supervised learning. This reduces the data storage memory needed compared to NFSP. We evaluate our method in two-player zero-sum poker games. We show that the ANFSP can approach approximate Nash Equilibrium more stable and quickly compared to NFSP.

In order to show the effect of the advantage of the techniques of MC-NFSP and ANFSP in more complex game, we also evaluated the effectiveness in a FPS team combat game, in which an AI agent team fights with a human team, and our system provided good tactic strategies and control policies to our agent team, and help it to beat humans.

2 Background

In this section we provide a description of related game theory concepts, current AI systems to solve perfect and imperfect games, relationship between reinforcement learning and Nash Equilibrium, and the Neural Fictitious Self Play (NFSP) techniques. For a better brief introduction we refer the reader to[8, 9, 6]

2.1 Related Game Theory Concepts

Game in Study. In this paper, we mainly research on two-player imperfect-information zero-sum game. A zero-sum game is a game in which the sum of each player's payoff is zero, and an imperfect-information game is a game in which each player only observes partial game state. For example, Texas Hold'em, real-time strategy games and FPS games are imperfect-inforamtion zero-sum games. Such game is often represented in "Normal form". Normal form is a game representation schema, which lists payoffs that players get as a function of their actions by way of a matrix. In our studied games, players take action simultaneously. The goal of each player is to maximize their own payoff in the game. Assume $\pi^i(a|U^i)$ is the action distribution of player i given the information set U^i he observes, $\pi = (\pi^1, ..., \pi^n)$ refers to the strategy set of all players, Σ^i is the behavior set of player i, π^{-i} is the strategy set in π except π^i, $R^i(\pi)$ is the expected payoff the player i gained following strategy π in game. The ϵ-best responses of player i to opponent's strategy π^{-i},

$$BR^i_\varepsilon\left(\pi^{-i}\right) = \left\{\pi^i \in \Sigma^i : R^i\left(\pi^i, \pi^{-i}\right) \geq max_{\pi' \in \Sigma^i} R^i\left(\pi^i, \pi^{-i}\right) - \epsilon\right\}$$

contains all strategies whose payoff against π^{-i} that is suboptimal by no more than ϵ.

Nash equilibrium. Nash equilibrium refers to the strategy that satisfies any player in the game can't obtain higher profit by changing his own strategy when the others don't change their strategy. Nash proved that if we allow mixed strategies, then every game with a finite number of rational players in which each player can choose from finitely many pure strategies has at least one Nash equilibrium. In fact, Nash equilibrium is the only strategy that rational players can expect to converge in self-play.

Theorm 1. ***(Minimax(Neumann, 1928))*** *For two-player zero-sum game,there exists value v that*

$$max_{\pi^1\in\sigma_1} min_{\pi^2\in\sigma2} R^1(\pi^1,\pi^2) = min_{\pi^2\in\sigma^2} max_{\pi^1\in\sigma^1} R^1(\pi^1,\pi^2) = v$$

v is the game value, which means player1(player2) gains at least $v(-v)$ in Nash Equilibrium.

Exploitability. The distance between a strategy and Nash Equilibrium can be measured by the distance from game value. The difference is called the exploitability of policy. For zero-sum game with two player, policy π is exploitable by ϵ if and only if

$$\varepsilon = \frac{R^1\left(\mathrm{BR}^1\left(\pi^2\right), \pi^2\right) + R^2\left(\pi^1, \mathrm{BR}^2\left(\pi^1\right)\right)}{2}$$

In the equation above, $R^1\left(\mathrm{BR}^1\left(\pi^2\right)\right)$ means the reword of $player1$ by making best response to his opponent. It is obvious that an exploitability of ε yields at least an ε-approximate Nash Equilibrium (distance to Nash Equilibrium no larger than ε). Exploitability measures the extent to which the opponent benefits from a player's failure to adopt a Nash Equilibrium strategy,i.e. the player does not adopt a Nash Equilibrium and the extent to which the opponent punishes him with the best response. Nash Equilibrium is unexploitable, that is, 0 utilization.

2.2 Reinforcement learning and Nash Equilibrium

Reinforcement learning agents learn how to maximize their expected payoff during the interaction with the environment. The interaction can be modelled as Markov Decision Process(MDP). At time step t, agent observes current environment state S_t and selects an action a_t according to policy π , where π is a mapping from state to action. In return, agent receives reward r_t and next environment state S_{t+1} from environment. The process ends until the agent reaches the terminal state. Agent learns from transition tuples (S_t, a_t, r_t, S_{t+1}).The goal of agent is maximizing the accumulated return $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ for each state S_t with discount factor $\gamma \in (0, 1]$. The action-value function $Q^\pi(s, a) = E\left[R_t | s_t = s, a\right]$defines the expected gain of taking action a in state s. It's the common objective function for most reinforcement learning.

An agent is learning on-policy if the policy it learns is what it currently follows, otherwise it's learning off-policy. Deep Q-learning[10] is an off-policy method which aims to update the action-value function $Q(s, a|\theta_i)$ toward the one step return $r + \gamma max_{a'} Q\left(s', a'; \theta\right)$. The policy for choosing next action and updating Q are not same. At each time step, agent takes ϵ-greedy policy, which selects a random action with probability ϵ, otherwise selects an action with highest estimated Q value. The loss is defined as

$$L(\theta) = E\left(r + \gamma max_{a'} Q\left(s', a'; \theta_{i-1}\right) - Q\left(s, a; \theta_i\right)\right)^2$$

Monte Carlo Tree Search algorithm is an on-policy method which aims to update the action-value function $Q(s_t, a_t)$ towards the accumulated return $G_t = \sum_{i=t}^{T} r_{i+1}$. In an episode agent chooses action according to policy π and updates the action-value function till the episode ends. Asynchronous Deep Q-learning[11] is a multi-threaded asynchronous variant of Deep Q-learning. It uses multiple actor-learners running in parallel on multiple copies of the environment. Agents share Q-learning network and apply gradient updates asynchronously.

The relationship between MDP and game theory or the relationship between reinforcement learning (RL) solution and Nash Equilibrium is stated like: 1) MDP/RL adopts differential learning mechanism, which in theory finally achieves Bellman optimality (or Markov perfect equilibrium, a refinement of the concept of Nash equilibrium), so it can learn the subgame optimization substructure including Nash Equilibrium; 2) However, in practice it's very difficult to measure how near a trained strategy to a Nash Equilibrium (or exploitability) in large scale games, because either training and evaluation cost much labor and time, so the reinforcement learning researchers simply use convergence to control termination of their training.

2.3 Modern RL systems for games

In these years, reinforcement learning has great breakthrough in more complex games. The most significant is the DeepMind AlphaGo and AlphaZero which beated world champions in the game Go. AlphaGo is initialized with human anotated training datas, after achieved certain levels, it improves itself by RL and self-play. AlphaZero can teach itself the wining strategy by playing the game purely with itself using a Monte Carlo search tree. DeepMind has shown the effectiveness of Monte Carlo techniques in games, but the game Go is a sequential perfect-information game.

Recently, RL has more researches on imperfect-information games. StarCraft is a hot point of research, many researches like CommNet and BicNet focus on small map combat, and DeepMind's AlphaStar can play a whold game, and has defeated top human players. The AlphaStar used the supervised training with human data at first, then use a group of agents (league) play with each other in RL to improve to superhuman level, but its current performance is still not stable enough. So it is valuable to think about whether we can get some tools in game theory to measure and control quality of trained strategy, e.g. Nash Equilibrium.

In the study of AI for Poker games, researches often consider Nash Equilibrium. In 2014, Heinrich and Silver [2] of University College London proposed a SmoothUCT algorithm that combines the Monte Carlo search [3], converges to approximate Nash equilibrium, and wins three silver medals in the annual Computer Poker Contest (ACPC). In 2015, the University of Alberta in Canada developed an online Counterfactual regret minimization algorithm similar to reinforcement learning [4], which can be used to solve Nash Equilibrium in the upper limit betting Texas Hold'em. The artificial intelligence based on this algorithm Cepheus is a near perfect player, human In the long run, the result can only be a tie, or the computer wins. In the same year, in the first no-limit Texas Hold'em game between computers and humans, although poker experts won the artificial intelligence "Clauk-do[5]" developed by Carnegie Mellon University with a slight advantage, the academic community believes that Clau- Do has achieved great success. Clau-do's strategy is based on the approximate equilibrium of the poker game, and its results are attributed to algorithmic game theory. In 2016, Johannes Heinrich and David Silver [6] proposed the Neural Fictitious Self-Play algorithm, which combines supervised learning and reinforcement learning to approximate the Nash equilibrium of imperfect information games without any prior knowledge. 2017 Carnegie Mellon's artificial intelligence "Libratus[7]" defeated top Texas Hold'em players in one-on-one No-Limit Hold'em, and Libratus developed a balanced game to bring the strategy to Nash equilibrium.

2.4 Neural Fictitious Self Play

Fictitious Play (FP) [4] is a classical game theory model of learning Nash Equilibrium from self-play. At each iteration players determine best response to others' average strategy and update their average strategy. In some specific scenes, like zero-sum game[12], the average strategy of players in Fictitious Play can reach Nash Equilibrium. Because FP is mainly for normal form of game, Heinrish, etc. has extended FP to Fictitious Self-Play (FSP), which works to traverse game tree of extensive form of games, and is possible to find Equilibrium in larger games. However FSP methods need player and opponent to follow an order of action, which makes it not suitable in imperfect-information games.

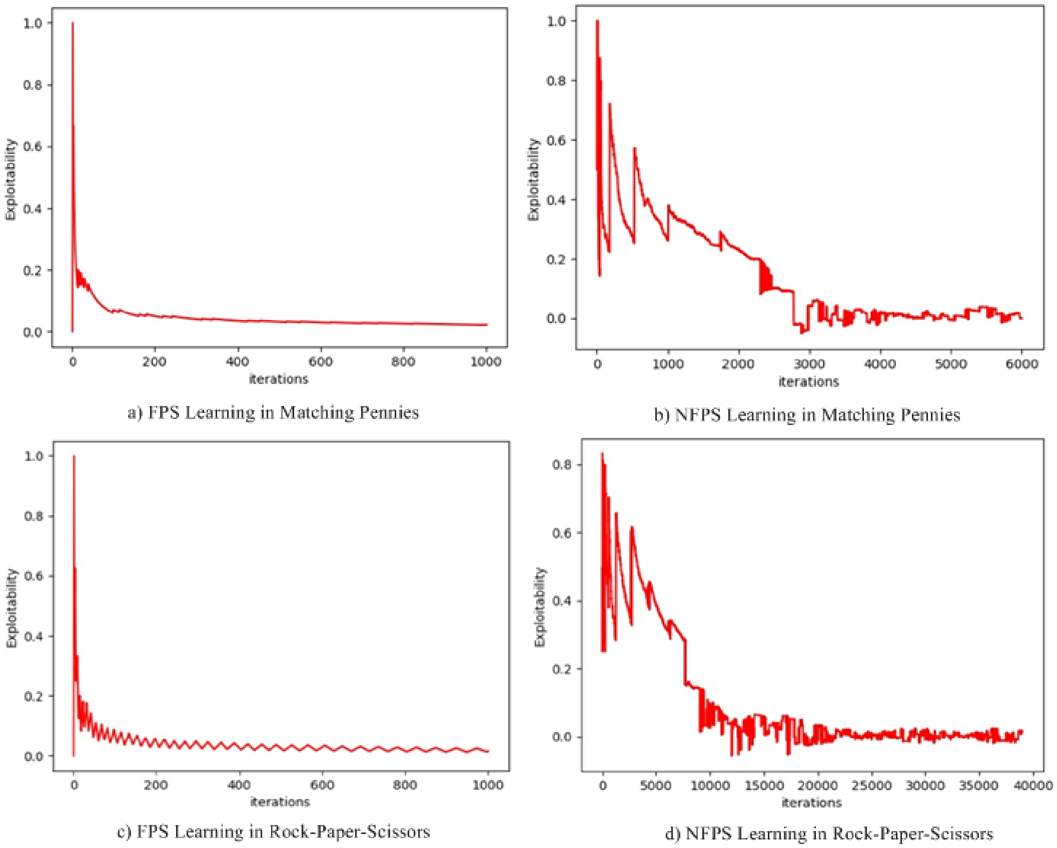
Neural Fictitious Self-Play[6] is a model of learning approximate Nash Equilibrium in imperfect-information games. The model combines fictitious play with deep learning. At each step, players choose a mixture of best response and average strategy. Players approximate best response by deep Q learning and update their average strategy by supervised learning. The state-action pair (S_t, a_t) is stored in supervised learning memory only when player determines action from best response. The transition tuple (S_t, a_t, S_{t+1}, r_t) is stored in reinforcement learning memory whichever the policy player follows when taking action. NFSP updates the average strategy network with cross entropy loss L_1,

$$L_1\left(\theta^{\Pi}\right) = \mathbb{E}_{(s,a)\sim\mathcal{M}_{SL}}[-\log \Pi(s, a|\theta^{\Pi})]$$

and updates the best response network with mean squared loss L_2. Both uses Stochastic Gradient Descent (SGD).

$$L_2\left(\theta^{Q}\right) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{M}_{RL}}\left[\left(r + \max_{a'} Q\left(s', a'|\theta^{Q'}\right) - Q(s, a|\theta^{Q})\right)^2\right].$$

Neural Fictitious Self-Play uses anticipatory dynamics[] method, i.e. player selects best response to opponent's short-term strategy prediction $\Pi_t^{-i} + \eta \frac{d}{dt} \prod_t^{-i}$, with η as antipatory parameter. Given B_{k+1}^i the best response of player i at iteration $k+1$, we have $B_{k+1}^i - \Pi_k^i \approx \frac{d}{dk}\Pi_k^i$. NFPS use an off-policy methods DQN in it, so it has problems in on-policy games like RTS where we need to sample opponents' changing strategy while we play, and enumerating opponents' strategy is too costly. As shown in Figure 1, we compared the training efficiency of FSP and NFSP, subfigures a) and b) show in the game "Matching Pennies", FSP converges in 200 iterations, but NFSP in more than 3,000. And in the game "Rock-Paper-Scissors", NFSP converges in more than 10,000 iterations as subfigures c) and d) show.



a) FPS Learning in Matching Pennies

b) NFPS Learning in Matching Pennies

c) FPS Learning in Rock-Paper-Scissors

d) NFPS Learning in Rock-Paper-Scissors

Figure 1: Training Efficiency of FSP and NFSP

3 Monte Carlo Neural Fictitious Self Play

3.1 Network Overview

Considering that Monte Carlo Tree Search (MCTS) algorithm consists of a policy network that generates an action probability, and a value network that evaluates the value of a state, we can see it doesn't suffer the complexity in DQN to score each action under each state. We believe MCTS can be used in high-dimensional and continuous problems. Moreover, MCTS directly uses reward that the player gets after each game to train its networks, it can avoid the inaccurate problem of DQN to evaluate Q-value ($Q(s_{t+1}, a|\theta')$) in the early stages. So we combines MCTS and NFSP to propose a new algorithm more suitable to larger imperfect games.

Monte Carlo Neural Fictitious Self Play(MC-NFSP) combines Monte Carlo Tree Search with Neural Fictitious Self-Play. MC-NFSP learns best response to opponents' average strategy by Monte Carlo Tree Search and updates average strategy by supervised learning with collected best response history. The dataset is generated from self-play in MC-NFSP. Agent plays a mixture of best response and average strategy as NFSP. Most of time they play an average strategy to the policy p' , but with some probability($\eta = 0.1$) they play a best response to MCTS.

The algorithm makes use of two neural networks: a policy-value network for Monte Carlo Tree Search (i.e. best-response network), a policy network for supervised learning (i.e. average-policy network). The best-response network is shown in Figure. 2. The input of neural networks is board state. The policy-value network has two outputs: a policy p, which is a mapping from current state to action probability, and a value v , which is the predicted value of the given state. The value is in $[0, 1]$, where losing game corresponds to 0 and winning game corresponds to 1. In our network, relu activation is used in convolution layers; dropout used in fully connection layers to reduce overfitting; and for policy probability, softmax is used. The policy network is almost same with best-response ntework except that it only outputs a policy p' (no value output), which is the average policy of player.

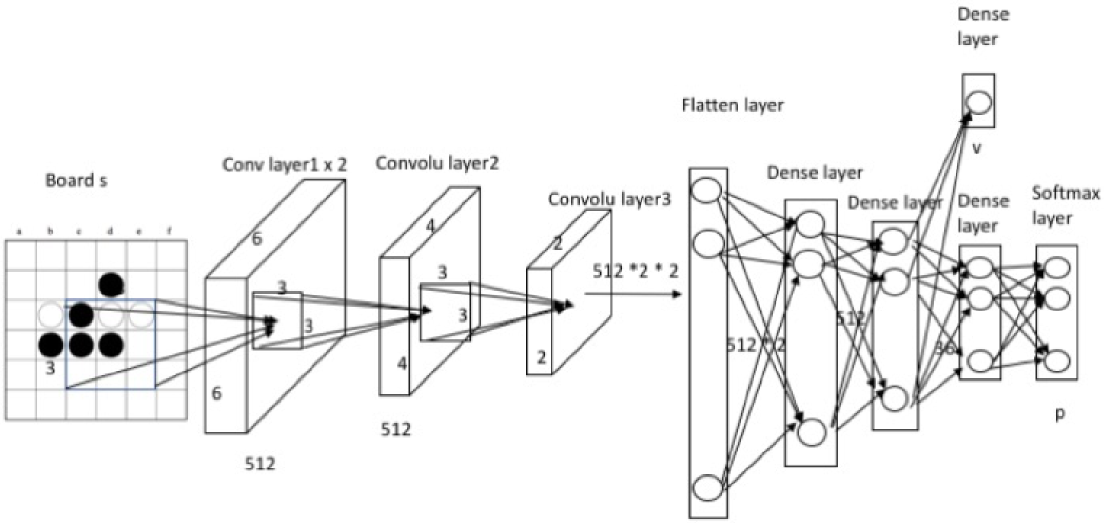


Figure 2: Best Response Network for MCTS

3.2 Algorithm Training

The networks are trained along with the game self-play. The self-player adopts a mixed policy: $\sigma = (1 - \eta)\Pi + \eta \mathrm{B}$. In each game, the player chooses to use the result from best-response network based MCTS, or from average-policy network, with a probalilities $\eta, 1 - \eta$.

MC-NFSP modifies Monte Carlo Tree Search(MCTS) algorithm for calculating best response to opponent's average strategy. Within the MCTS, agent simulates the play following the game tree. At each time, agent chooses action a maximizing $U(s, a)$.

$$U(s,a) = Q(s,a) + c_{put} \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)}$$

$Q(s, a)$ is the expected payoff taking action a in state s, $P(s, a)$ is probability of taking action a from state s according to the policy. $N(s)$ is the number of visits to state s across simulation. $N(s, a)$ counts the number of times action a been chosen at state s across simulation, c_{put} is a hyperparameter that controls the degree of exploration. Agent takes action a and reaches next state s'. If s' is the terminal state, then the player's final score is used as the node score. If s' is not the terminal state, then opponent takes action in s' according to average strategy and reaches next state s''. If s'' is the terminal state, then player's final score is used as the node score. If s'' is not the terminal state and s'' has been visited, then player chooses action a maximizing $U(s'', a)$ and simulates the play following the game tree. If s'' has not been visited before, then s'' is added in the game tree, $\mathrm{P}\left(s'', \cdot\right) = \vec{p_\theta}\left(s''\right)$ is calculated using neural network as initial node value, $Q\left(s'', a\right)$ is initialized as 0 , $N\left(s'', a\right)$ is initialized as 0.

Node value V is propagated along the path visited in current simulation and used to update corresponding $Q(s, a)$ value.

$$Q(s,a) = (N(s,a) * Q(s,a) + V)/(N(s,a) + 1)$$

After multiple simulations, the $N(s, a)$ values are a better approximation for the policy. $\frac{N(s_\mathrm{b})}{\sum_b (N(s,b)}$ is normalized as the improved policy $\vec{\pi}(s)$. Agent picks an action by sampling from the $\vec{\pi}(s)$. After an episode, tuples $(s, \vec{\pi}(s), v)$ are stored in reinforcement learning memory to train best response network. Pairs $(s, \vec{\pi}(s))$ are stored in supervised learning memory to train average strategy network. After a certain number of episodes, the best response network is trained with loss l_1 (in our loss functions, s_t is current game states, p_t is the output of the average network, z_t is the result of the game, value is 1 or -1),

$$l_1 = -\sum_t \left(\pi_t \log p_t - \left(\mathrm{v}\left(s_t\right) - z_t\right)^2\right)$$

the average strategy network is trained with loss l_2.

$$l_2 = -\sum_t \vec{\pi}_t \log \vec{p_t}$$

Algorithm 1 MC-NFSP algorithm

```
Initialize$\Gamma$,execute function $InitGame()$, $RunAgent(\Pi, B)$;
function InitGame()
    Initialize policy-value network $B(s|\theta^B)$ randomly
    Initialize policy network $\Pi(s|\Theta^\Pi)$ randomly
    Initialize experience replay $M_{RL}$ and $M_{SL}$
    (players share networks $B$ and $\Pi$)
end function
function RunAgent()
    for each iteration do
        its := its + 1
        policy $\sigma \leftarrow \begin{cases} B, & with\, probability\, \eta \\ \pi, & with\, probability\, 1-\eta \end{cases}$
        observe initial state $s$ and reward $r$
        while not terminal do:
            If policy comes from $\pi$,choose action $a$ in state $s_t$ according to $\pi$
            If policy comes from $B$, choose action according to adapted MCTS
            Execute action $a$, observe next state $s_{t+1}$
            if $terminal$ then:
                store $(s_t, \vec{\pi_t}, Z_t)$ in $M_{RL}$, store $(s_t, \vec{\pi_t})$ in $M_{SL}$ if policy comes from $B$
            end if
        end while
        if $its\%update == 0$ then:
            update best response network with $l = -\sum_t \left(\vec{\pi_t} \log p_t - (\mathrm{v}(s_t) - z_t)^2\right)$
            update average network with $l = -\sum_t \vec{\pi_t} \log \vec{p_t}$
        end if
    end for
end function
```

3.3 Experiment

We compare MC-NFSP with NFSP in Othello. Our experiment investigate the convergence of MC-NFSP to Nash equilibrium in Othello and measure the exploitability of learned strategy as comparative standard. To reduce the calculation time of exploitability, we choose 4×4 Othello board.

3.3.1 Othello

The neural network in MC-NFSP takes the 4×4 board position s_t as input and passes it through two convolutional layers and a flatten layer. Then, the resultant 120 dimensional vector is passed through many fully connected layers, and output both a vector p, representing a probability distribution over moves, and a scalar value v, representing the probability of the current player winning in position s_t ,for best response network. The architect of average strategy's neural network is same as best response's neural network except the output. It only outputs a vector p representing probability distribution over moves for average strategy network. We set the sizes of memory to 400w and 40w for M_{RL} and M_{SL} respectively. M_{RL} was updated with a circular buffer containing recent training experiences, M_{SL} was updated with reservoir sampling [13]to ensure an even distribution of training experiences. The reinforcement learning and supervised learning rate were set to 0.01 and 0.005, and both used Adam optimizer. Players perform gradient update every 100 episodes of play. MC-NFSP's anticipatory parameter was set to $\eta = 0.1$.

Figure 3 shows MC-NFSP approaching Nash equilibrium in Othello. The exploitability achieved 0 after 25000 episodes of play.

3.3.2 Exploitability in MC-NFSP

In our experiments, we also use exploitability to evaluate the distance between policy and Nash Equilibrium. Exploitability is calculated by making the best-response network to play with the average-policy network. Then the total reward of best-response network can be accumulated to the root of the game tree. And this value is used as exploitability.

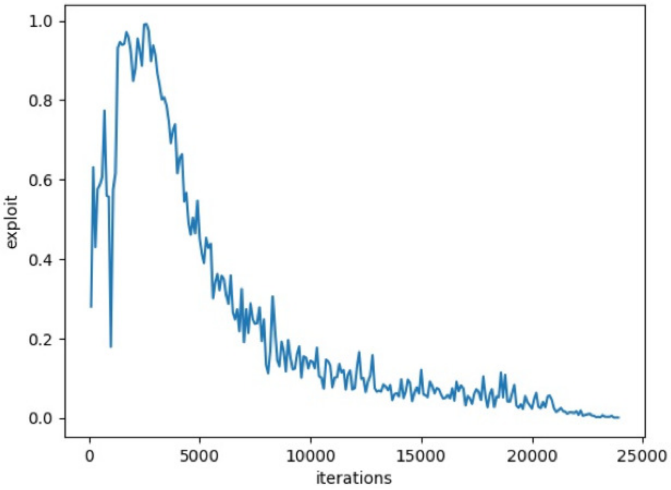


Figure 3: Exploitability of MC-NFSP in Othello

3.3.3 Comparison MC-NFSP with NFSP

The neural network in NFSP is the same with MC-NFSP' except the output. The output of best response network in NFSP is a vector Q, representing the value of each action in state s_t. The output of average strategy network is a vector p, representing the probability distribution over moves for average strategy network. We set the size of memory to 400w and 40w for M_{RL} and M_{SL} respectively. The reinforcement learning and supervised learning rate were set 0.01 and 0.005. Each player performed gradient updates of mini-batch size 256 per network for every 256 moves. The target network of best response network was refitted every 300 training. NFSP's anticipatory parameter was set to $\eta = 0.1$. The ϵ-greedy policy's exploration rate started at 0.6 and decayed to 0, proportionally to the inverse square root of the number of iterations.

Figure 4 shows NFSP can't approach Nash equilibrium in Othello. The exploitability of strategy oscillates during the training time. The reason about NFSP doesn't converge to approximate Nash equilibrium in Othello is NFSP players rely on Deep Q-learning approximating best response, which don't have good performance in scenes with large-scale search space and depth.

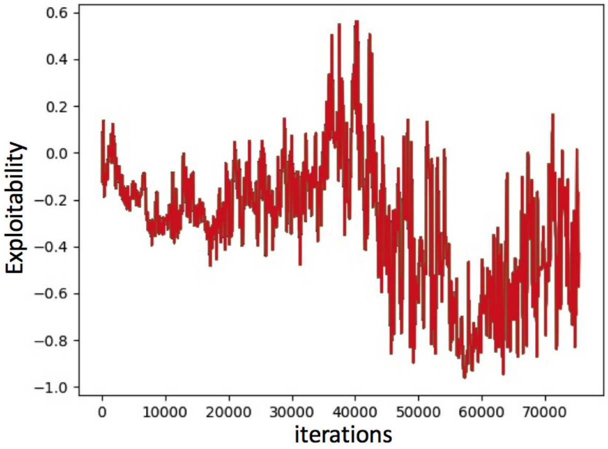


Figure 4: Exploitability of NFSP in Othello

4 Asynchronous Neural Fictitious Self Play

4.1 Algorithm Overview

Based on MC-NFSP, we further improve the time-efficiency by proposing a multi-thread learnig mechanism called Asynchronous Neural Fictitious Self-Play(ANFSP), which asynchronously execute multiple players in parallel, on multiple instances of the game environment. Players run different exploration policies in different threads, and share the nerual network and performs gradient update asynchronously.

Inspired with A3C algorithm, our algorithm also starts multi-threads of plays as Algorithm 2 shows. In each thread, similar as MC-NFSP, the players choose a mixture of average-policy and best-response networks ($\Pi and \mathrm{B}$) with a probability η, $\sigma = (1-\eta)\Pi + \eta\mathrm{B}$. Most of time they play an average strategy, but with some probability($\eta = 0.1$) they play a best response. The state-action pair (S_t, a_t) is stored in supervised learning memory only when the player determine action from best response. Each thread computes gradient of best response network using transition tuple (S_t, a_t, S_{t+1}, r_t) each step and accumulate gradients over multiple timesteps to certain number before they are applied, which is similar to mini-batches. And we compute gradient of average strategy network using mini-batch of supervised learning memory after multiple timesteps after accumulated to certain number. After a global counter achieves certain count, the networks are updated. The loss of best response network is defined as l_1, and the loss of average strategy network is defined as l_2.

$$l_1 = -\left(r + \gamma max_{a'} Q\left(s', a'; \theta^{Q^-}\right) - Q\left(s, a; \theta^Q\right)\right)^2$$

$$l_2 = -\sum_i a_i \log\left(\Pi\left(p_i | s\right)\right)$$

Algorithm 2 Asynchronous-Neural-Fictitious-Self-Play

```
InitGame(), Init game $\Gamma$, execute multiple thread $RunAgent()$
function InitGame()
    Init average strategy network $\Pi(s, a|\theta^{\Pi})$
    Init Q-value network $Q(s, a|\theta^Q)$
    Init target network $\theta^{Q'} \leftarrow \theta^Q$
    Init global anticipatory parameter $\eta$
    Init global count T = 0
    Init global iteration count iterations = 0
    return
end function
function RunAgent()
    Init thread count $t \leftarrow 0$
    repeatFor each iteration
        policy$\sigma \leftarrow \begin{cases} \epsilon - greedy(Q), & with\ probability\ \eta \\ \Pi, & with\ probability\ 1-\eta \end{cases}$
        observe state $s$ and reward $r$
        determine action $a$, observe reward $r_{t+1}$, next state $s_{t+1}$
        $y = \begin{cases} r \\ r + \gamma max_{a'} Q\left(s', a'; \theta^-\right), & if\ s_{t+1}\ is\ not\ terminal \end{cases}$
        accumulate gradient $d\theta^Q \leftarrow d\theta^Q + \frac{\partial\left(y - Q\left(s,a;\theta^Q\right)\right)^2}{d\theta^Q}$
        If policy $\sigma$ comes from $\epsilon - greedy(Q)$, store pair $(s_t, a_t)$ in
        $s_t \leftarrow s_{t+1}$
        $T \leftarrow T + 1$
        $t \leftarrow t + 1$
        if $T\ mod\ I_{target} == 0$ then
            update target network $\theta^{Q'} \leftarrow \theta^Q$
        end if
        if s is terminal then
            iterations += 1
            if iterations $mod\ I_{\text{Asyncupdate}} == 0$ then
                update $\theta^Q$ with $d\theta^Q$ asynchronously
                update $\theta\Pi$ with $L\left(\theta^{\Pi}\right) = \mathrm{E}_{(s,a)\sim M_{SL}}[-\log \Pi(s, a|\theta^{\Pi})]$
                $d\theta^{Q} \leftarrow 0, d\theta^{\Pi} \leftarrow 0$
            end if
        end if
    until $T > T_{\max}$
    return
end function
```

4.2 Experiment

We compare ANFSP with NFSP in modified Leduc Hold'em. For calculation simplification, we limit the maximum bet size each round in Leduc Hold'em is 2. Our experiment investigates the convergence of ANFSP to Nash equilibrium in modified Leduc Hold'em and measures the exploitability of learned strategy as comparative standard.

4.2.1 Leduc Hold'em

The experiment is on Leduc Hold'em with two player. In the game, the bet history is represented as a tensor with 4 dimensions, namely players, round, bet, action taken. Leduc Hold'em contains 2 rounds. Players usually have three actions to choose from, namely fold, call, raise. As the game ends once a player gives up, then the betting history is represented as a $2 \times 2 \times 2 \times 2$ tensor. We flatten the 4-dimensional tensor to a vector of length 16. Leduc Hold'em has a card deck of 6 cards. We represent each rounds' cards by a k-of-n encoding. E.g. LHE has a card vector of 6 cards and we set public cards to 1, the rest to 0. Concatenating with the cards input, we encode the information state of LHE as a vector of length 22.

We started 4 threads with exploration rate randomly chosen from $[0.4, 0.6, 0.5, 0.7]$. The exploration rate decayed to 0, proportionally to the inverse square root of the number of iterations. We set the sizes of M_{SL} to 200w. We train network every 32 iterations of play. We update Deep Q-learning network with accumulated gradients, and update average strategy network with mini-batch size of 128. The reinforcement learning and supervised learning rate were set 0.01 and 0.005, and both used SGD. The target network of Deep Q-learning was updated every 50000 actions. ANFSP's anticipatory parameter was set to $\eta = 0.1$.

Figure 5 shows ANFSP approaching Nash equilibrium in modified Leduc Hold'em. The exploitability declined continually and appeared to stabilize at around 0.64 after 140w episodes of play. The training costed about 2 hours.

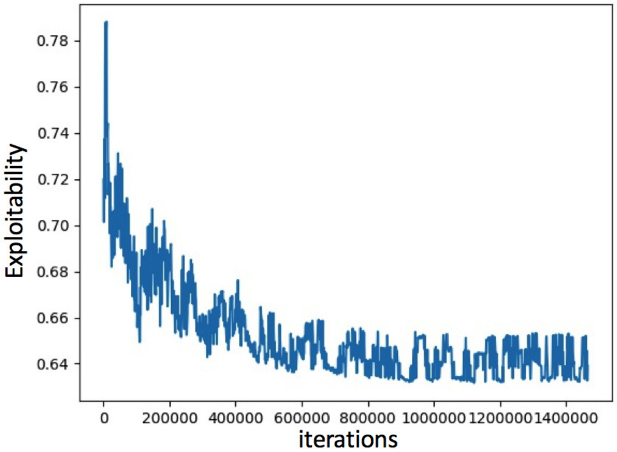


Figure 5: Exploitability of ANFSP in modified Leduc Hold'em

4.2.2 Comparison with NFSP

The architecture of neural network in NFSP is the same with ANFSP's. We set the size of memory to 20w and 200w for M_{RL} and M_{SL} respectively. The reinforcement learning and supervised learning rate were set 0.01 and 0.005. Players performed gradient updates of mini-batch size 128 per network for every 128 actions. The target network of

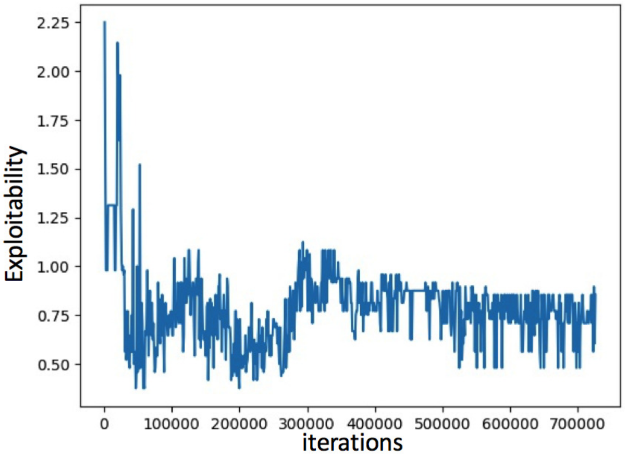


Figure 6: Exploitability of NFSP in modified Leduc Hold'em

best response network was refitted every 300 training. NFSP's anticipatory parameter was set to $\eta = 0.1$. The ϵ-greedy policy' exploration rate started at 0.06 and decayed to 0, proportionally to the inverse square root of the number of iterations.

Figure 6 shows the learning performance of NFSP. The exploitability of strategy fluctuated and appeared to stablize at around 0.75 after 70w episodes of play. The training also costed about 2 hours. It means in same training time (2 hours), ANFSP can complete more episodes and achieve better results (lower exploitability).

5 Evaluation in First Player Shooting Game

5.1 Experiment Setting

In order to evaluate the effectiveness of our algorithm in a complex imperfect-information game, we tried to train it in an FPS game and make it combats with human-bings. The FPS platform used in this experiment is designed by our research team. The game scene is an offensive and defensive confrontation of two teams (10 VS 10). In training, one side is the MC-NFSP, the other side is a memory trained by thousands of human plays (SL-Human). The experiment was performed in a fixed closed 255 x 255 square map. The entire map was divided into 12 x 12 areas each with a 20 x 20 square. The detail of the scene is shown below (Figure. 7). All the green areas in Figure. 7 are passible regions, and the gray areas are obstacles that cannot be crossed (rock or fence). Figure 7) is marked with two points A, B, which are the birth points of the two teams (used only during experimental testing). In addition, the nine checkered areas marked with red in Figure. 7 represent the candidate areas of the experimental decision layer AI transfer agents. The coordinates of the center points of the nine grids are (125, 175), (185, 155), (205, 95), (185, 35), (125, 15), (55, 35), (45, 95).), (55, 155), (125, 95), (125, 135), (165, 95), (125, 55), (85, 95). The five inner areas in the figure are surrounded by a wall in a 100 x 100 area, and the centers of the four walls respectively correspond to four doors. The size of the doors is limited to 2-3 people at the same time. The team outside has a mission to break into the walls and kill all of the inside ones, and the inside team is to defense.

5.2 Experiment

In training, each team is represented as a player. The states is a dictionary of the form $\{L, C_{T,L,t}, B_{T\prime,L,t}\}$, where L is the location block in game map, $C_{T,L,t}$ means the number of current trained team T in L in time t, and $B_{T\prime,L,t}$ is the believed number of team $T\prime$ in location L. The actions of a team is the force assignment of number of fighters to different locations like $< n_f, L_1, L_2 >$, which means to assign n_f fighters from L_1 to L_2. For reward, each fighter in team has a health of 100, so the reward of team T is $LostHealth_T - LostHealth_{T\prime}$. Different with our previous works in this paper, the two networks are built and trained for both the outside team and inside team. Figure 8 shows the training result of the outside team (results of inner team is similar). We can see the training converges very fast (in no more than 150 episode, each episode has 5 games). The win rate of the outside team against the SL-Human gets higher than 80%, and the loss of training gets near to zero.

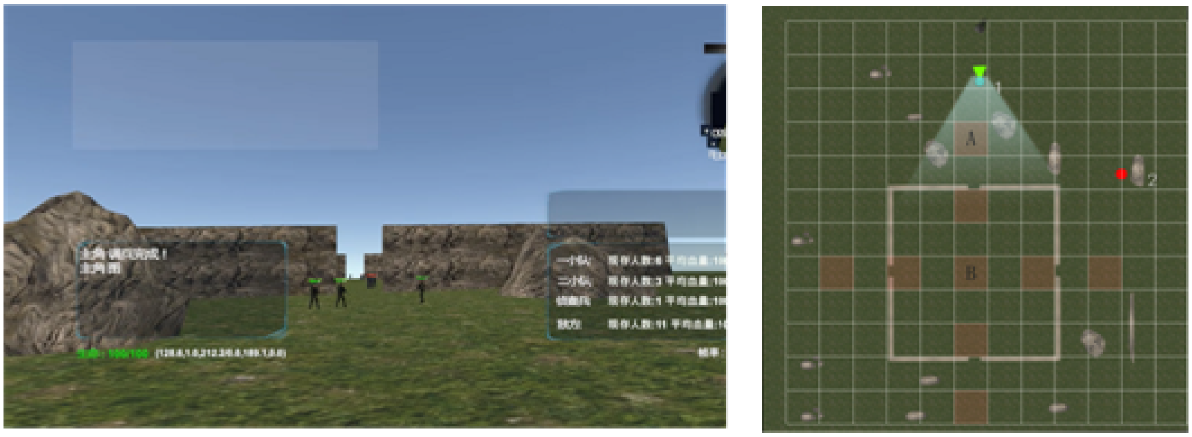
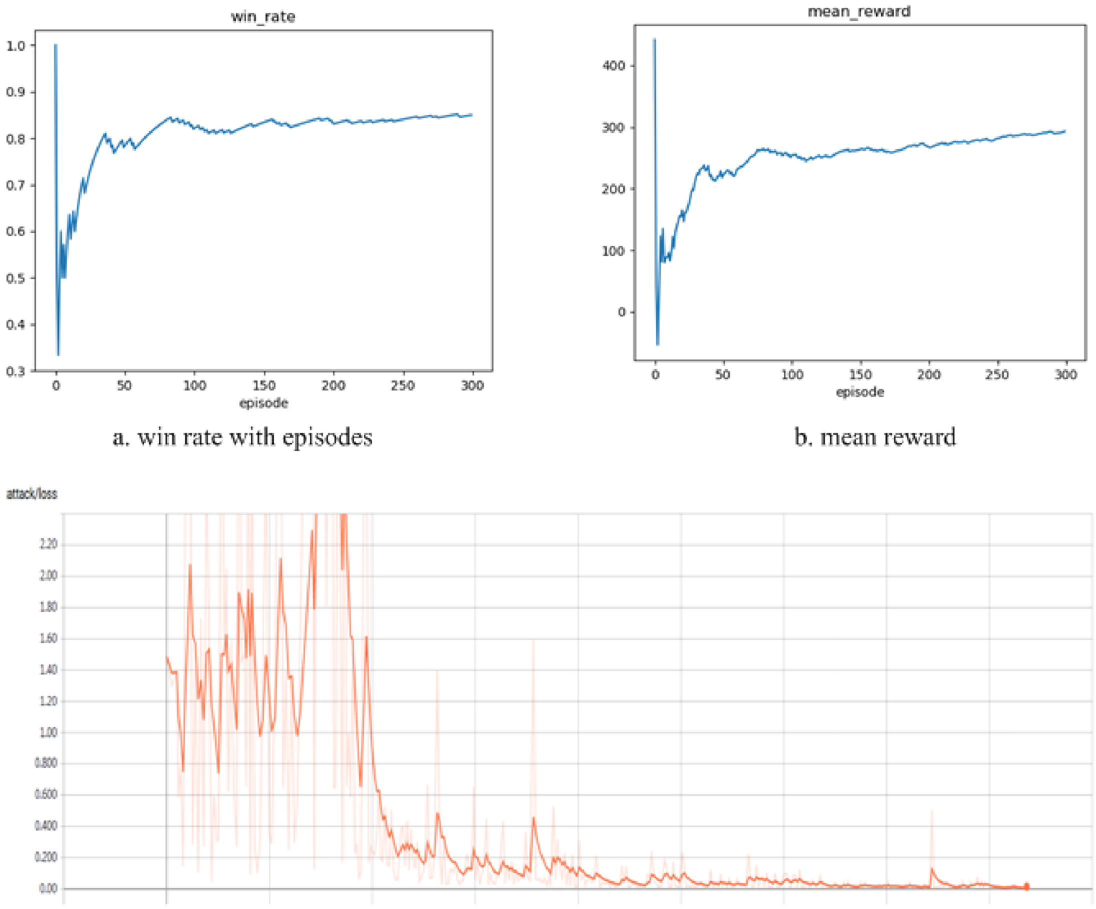


Figure 7: FPS Game Environment

After training, we make this algorithm to play the game with pure 10 college students, it plays with one human for 10 rounds (after five rounds, they change the location), totally 100 games are played, in which our trained algorithm achieved 75 victories, so it is a superhuman result.



a. win rate with episodes

b. mean reward

c. training loss

Figure 8: Evaluation in FPS Game

6 Conclusion

We propose MC-NFSP and ANFSP in this paper. They are improvements on NFSP and can learn approximate Nash equilibrium in two-player games. MC-NFSP improve the performance of the original algorithm in two-player zero-sum games with large-scale search space and search depth. Experiment on Othello shows players' strategy converges to approximate Nash equilibrium with the improved algorithm running a certain round, but the original algorithm cannot converge. ANFSP uses asynchronous and parallel architecture to collect game experiences efficiently and reduces converging time and memory NFSP needed. Experiment on modified Leduc Hold'em shows ANFSP can converge in a shorter time and the convergence is more stable compared with NFSP. Finally we tested the algorithm in FPS games, and it achieved superhuman results in short time, it shows combination Monte Carlo Tree Search and NFSP is a practical way to solve imperfect-information game problems.

References

[1] J Nash. Non-cooperative games. *Annals of mathematics*, pages 286–295, 1951.

[2] T. Sanholm. The state of solving large incomplete-information games, and application to poker. *AI Magazine*, 31(4):13–32, 2010.

[3] B. Bosansky, C. Kiekintveld, V. Lisy, and M. Pechoucek. An exact double-oracle algorithm for zero-sum extensive-form games with imperfect information. *Journal of Artificial Intelligence Research*, pages 829–866, 2014.

[4] G. W. Brown. Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 1951.

[5] J. Heinrich, M. Lanctot, and D. Silver. Fictitious self-play in extensive-form games. In *Proceedings of the 32nd International Conference on Machine Learning.*, 2015.

[6] J. Heinrich and D. Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.

[7] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[8] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. 1998.

[9] R. B. Myerson. *Game Theory:Analysis of Conflict*. Harvard University Press, 1991.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, pages 529–533, 2015.

[11] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of Machine Learning Research*, 2016.

[12] J. Robinson. An iterative method of solving a game. *Annals of Mathematics*, pages 296–301, 1951.

[13] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 1985.