修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報理工学研究科　情報ネットワーク工学専攻　博士前期課程		
氏　　　名	HE YI	学籍番号	1831036
論 文 題 目	不確実性のあるカードゲームのソリティアにおける強化学習の効率に関するいくつかの事例研究		

要　　旨

近年、人工知能研究の分野で強化学習とニューラルネットワーク（NN）を組合せた手法が多くの成果を上げてきた。さらに、ゲーム領域では囲碁などいくつかのゲーム人工知能の性能が人間トップレベルの実力に達した。このような方法の他の様々なゲームに対する効果は未だ知られていなくて、これを明らかにすることは、現在のゲーム領域における重要な課題であると考えられる。

本研究では、二つの不確実性を伴うカードソリティアゲーム「TriPeaks」と「Russian Solitaire」を研究対象として選んだ。そして、二つの強化学習方法「Monte-Carlo 法」と「Q-learning」を用いて人工知能を訓練した。計算機実験を行って勝率を計測し、二つの強化学習方法それぞれが、これら二つのソリティアゲームにおいてもたらす結果を比較することが本研究の目標である。

二つの強化学習の開始点となる初期方策は、ゲーム固有の知識をもたない一様ランダム方策とした。そして、事後状態の価値は NN を用いて近似的に表し、NN は確率的勾配降下法に基づき学習し、学習は Replay Memory を用いて安定化させた。NN の重みの数は百万程度とした。また、Russian Solitaire においては、学習を効率化するために、ゲームプレイの無駄な繰り返しはプレイヤの負けになるということとした。さらに、ソリティアの事後状態から NN の入力列を生成するエンコーディング方法を二種検討した。

TriPeaks のプレイヤを学習する実験においては、MC 法に基づき 100 万回ゲームをプレイすると勝率は約 0.27（ランダムプレイヤの 500 倍）、Q-learning に基づき同回数ゲームをプレイすると勝率は約 0.52（ランダムプレイヤの 1000 倍）に達することが明らかとなった。

Russian Solitaire のプレイヤを学習する実験においては、MC 法に基づき 150 万回ゲームをプレイすると勝率は約 0.0035（ランダムプレイヤの 1.5 倍）、Q-learning に基づき同回数ゲームをプレイすると勝率は約 0.0047（ランダムプレイヤの 2.1 倍）に達することが明らかとなった。

Some Case Studies on the Efficiency of Reinforcement Learning in Card Game Solitaire with Uncertainty

HE YI

Supervisor: Kunihito Hoki, Takeshi Ito

University of Electro-Communications

Graduate School of Informatics and Engineering

Department of Computer and Network Engineering

March 18, 2021

Contents

Chapter 1

Introduction

Since the field of Artificial Intelligence (AI) research was founded as an academic discipline in 1956 [1], game AI is continuously used as a measure of progress of AI throughout its history.

In the middle 50s and early 60s, Arthur Samuel used alpha-beta pruning on his checkers learning program [2]. This program is widely regarded to be the world's first self-learning game-playing program.

Since the development of Arthur Samuel's checkers program, scientists have produced multiple examples of game AI programs that have approached or even surpassed expert human players. A list of some well-known achievements are described below:

- **Chinook** is a computer checkers program developed during the years from 1989 to 2007 at the University of Alberta by a team led by Jonathan Schaeffer. Chinook is the first computer program to win the world champion title in a competition against humans in 1994 [3]. By using a parallel alpha-beta search algorithm with all the latest enhancements, Chinook won the title after Marion Tinsley withdrawal due to health concerns.

- **TD-Gammon** is a computer backgammon program reached the level of expert human player developed in 1992 by Gerald Tesauro. Backgammon is a stochastic game with dice. TD-Gammon is developed by using an artificial Neural Network (NN) trained by a form of temporal-difference (TD) learning, specifically TD-(λ) [4].

- **Deep Blue** is a computer chess program developed by IBM. The developer team is led by Feng-hsiung Hsu and Murray Campbell. Deep Blue is developed by using the alpha-beta search algorithm in parallel. In 1997, Deep Blue played against chess world champion Garry Kasparov in an exhibition match, winning the six-game match 3.5–2.5, becoming the first computer system to defeat a reigning world champion in a match under standard chess tournament time controls [5].

- **Deep Q-network Atari 2600 Programs** are computer programs developed by DeepMind Technologies. As the origin of the programs' name, they are developed by using a variant of Q-learning algorithm with Deep Neural Network (DNNs). The programs play video games in Atari 2600 series (a collection of single player video games), receive only raw pixel value and the game score as input, and have exceeded the level of human in most of games [6].

- **AlphaGo** is a computer Go program also developed by DeepMind Technologies. AlphaGo is trained by Reinforcement Learning (RL) method with NNs. In 2016, AlphaGo played with South Korean professional Go player Lee Sedol (ranked 9-dan, one of the best human experts) and won a match 4-1, drawing public attention [7].

- **Pluribus** is a computer poker program developed by researchers from Facebook's AI Lab and Carnegie Mellon University. The Pluribus's core strategy was computed through self-play, using a form of Monte Carlo Counterfactual Regret (MCCFR) Minimization that samples actions in the game tree. Pluribus is stronger than top human professionals in six-player no-limit Texas hold'em poker. This is the first time an AI program has proven capable of defeating top professionals in any major benchmark game that has more than two players (or two teams)[8].

From the list, it is clear that methods used to build AI programs are different. Nowadays, RL methods with NNs becomes one of the most important trend to solve game problems. However, the efficiency of RL methods with NNs to a wide variety of games is still unclear.

This study researches the efficiency of RL methods through investigating performances of different RL methods in card game solitaire. Solitaire, also called patience or cabale, is any tabletop game which one can play by oneself, usually with cards (card game solitaire). There are hundreds of solitaires

Game	Expected Length of an Episode	Size of Action Set	Win Rate
TriPeaks	47	10^1	0.487
Russian Solitaire	22	10^3	0.028

Table 1.1: The nature of card game solitaires with uncertainty. The expected lengths and win rates were shown on a website [10]. The win rates indicate the winning percentage of casual players. This website is popular, everyone can play multiple card game solitaires and 13 billion gameplays were recorded on the website. Note that this website allows a player multiple cancellations of decisions, therefore the win rate shown on the website may be higher than the win rate of an optimal policy (described in Chapter 2).

with different natures [9], and this study focus on card game solitaires with uncertainty.

This study develops AIs using RL and measures performances in the domain of card game solitaires with uncertainty through analyzing win rates. For the purpose of such comparisons, this study deals with two card game solitaires with different difficulties to win. As a typical example of easy card game solitaire, TriPeaks is chosen. As a typical example of difficult card game solitaire, Russian Solitaire is chosen. Table 1.1 shows the nature of these two solitaires.

In Chapter 2, basic knowledge related to NN and RL are described. In Chapter 3, some related works of existing game AIs trained by RL are introduced. Chapter 4 describes experimental set and results. In the end, a conclusion is drawn in Chapter 5.

Chapter 2

Background Knowledge

The background knowledge of this study includes NN and RL. NNs are algorithms for **function approximation**. RL, as introduced in Chapter 1, is an area of machine learning and nowadays it provides very important and effective methods to build AIs in games. "Reinforcement learning, like many topics whose names end with "ing," such as machine learning and mountaineering, is simultaneously a problem, a class of solution methods that work well on the class of problems, and the field that studies these problems and their solution methods [11]." This section uses the term RL for the second meaning, "a class of solution methods." Subsection 2.1 introduces NN, this subsection is mainly derived from the book [12]. Subsection 2.2 describes Markov Decision Process (MDP) used in RL. Subsection 2.3 shows RL methods that are used in this study. Subsection 2.2 and 2.3 are mainly derived from Sutton's book [11].

2.1 NN

NNs are computing models inspired by the biological neural networks that constitute animal brains. A NN is composed of a large number of connected **units** (or neurons).

2.1.1 Feedforward NN

Feedforward NNs (FNNs) are called feedforward because there are no paths within the networks by which a unit's output can influence its input. Each

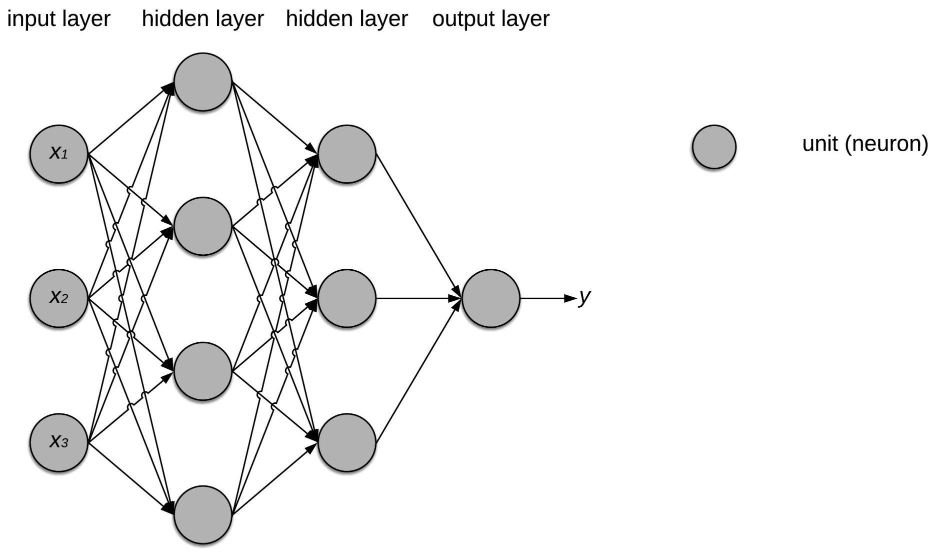

Figure 2.1: Example of a 4-layer FNN with 3 inputs $\{x_1, x_2, x_3\}$, two hidden layers, and an output y

connection between two nodes represents a weighted value $\omega \in \mathbb{R}$ for the signal passing through the connection, which is called **weight**. FNNs are called networks because they are typically represented by composing together many different functions. For example, three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain to form $y = f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. In this case, $\boldsymbol{x} = \{x_1, x_2, \cdots, x_n\}$ is the output of the **input layer** of the network, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer of the network, and so on. The final layer of the network $f^{(3)}$ is called the **output layer**. The length of the chain is called the **depth** of the network. The other layers except input layer and output layer are called **hidden layers**. Figure 2.1 shows an example of FNNs,

The units compute a weighted sum of their input signals and then apply a nonlinear function (called the **activation function**) to the sum, to produce the units' outputs. Figure 2.2 shows an example of a unit.
In the case of Figure 2.2, the output z is computed as,

$$u = x_1\omega_1 + x_2\omega_2 + x_3\omega_3 + x_4\omega_4 + b, \tag{2.1}$$

$$z = f(u). \tag{2.2}$$

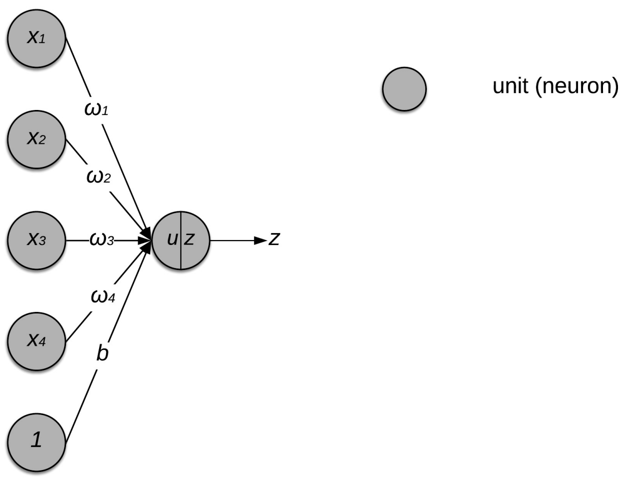

Figure 2.2: Example of a unit with 4 inputs $\{x_1, \cdots, x_4\}$, one bias b, and an output z

Here, $b \in \mathbb{R}$ is called **bias**, and $f(u)$ is an activation function. In this study, I use **Rectified Linear** (ReL) activation function and **sigmoid** activation function. ReL activation function is defined as,

$$f(x) = \max(0, x). \tag{2.3}$$

A sigmoid activation function is applied to the output layer, converting the value of the output to a real value between 0 and 1 to represent the probability of choosing an action. Sigmoid activation function is defined as,

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{2.4}$$

Assume there is an L-layer FNN (depth is L) with each unit in a layer connected to all the units in the next layer. The output of the i-th unit in the l-th layer (input layer is the 0-th layer) is denoted by $z_i^{(l)}$. The weight from the i-th unit in the l-th layer to the j-th unit in the $(l+1)$-th layer is denoted by $\omega_{ji}^{(l+1)}$. The bias of the j-th unit in the l-th layer is denoted by $b_j^{(l)}$. The activation function of the l-th layer is denoted by $f^{(l)}$. The output of the j-th unit in the $(l+1)$-th layer is computed as,

$$u_j^{(l+1)} = \sum_i z_i^{(l)} \omega_{ji}^{(l+1)} + b_j^{(l+1)}, \tag{2.5}$$

$$z_j^{(l+1)} = f^{(l+1)}\left(u_j^{(l+1)}\right). \tag{2.6}$$

Here, $l \in \{0, 1, 2, \cdots, L-1\}$, the input of the FNN is $\boldsymbol{x} = \{x_1, \cdots, x_i, \cdots\}$, and the output of the FNN is $\boldsymbol{y} = \{y_1, \cdots, y_i, \cdots\}$. So that, $z_j^{(0)} = x_j$ and $z_j^{(L)} = y_j$ hold.

The NN layer introduced above is also called **fully-connected layer**. In addtion, there are other kind of layers used in this study. **Convolutional layer** is a kind of layer that employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation,

$$u_{i,j,c}^{(l+1)} = \sum_{n=0}^{N-1} \sum_{p=0}^{K_h-1} \sum_{q=0}^{K_w-1} \omega_{p,q,n,c}^{(l+1)} z_{i+p,j+q,n}^{(l)} + b_c^{(l+1)}. \tag{2.7}$$

Here, c is the index of output channel, (i, j) is the gridpoint of output image of channel c, N is the number of input channels, $K_h \times K_w$ is the size of convolution kernel, and $z_{i,j,n}^{(l)}$ is the value of input image of channel n at gridpoint (i, j). All output images have the same dimension $H \times W$ and the number of images, i.e., the number of channels, is C.

2.1.2 Loss Function

In order to use NNs to approximate a target function, we need to use a training dataset to adjust the weight of NNs. A training dataset D is a set of n samples of a pair of the input $\boldsymbol{x}$ and the expected output $\boldsymbol{d}$, that is,

$$D = \{(\boldsymbol{x}_1, \boldsymbol{d}_1), \cdots, (\boldsymbol{x}_n, \boldsymbol{d}_n)\}. \tag{2.8}$$

For any pair $(\boldsymbol{x}, \boldsymbol{d}) \in D$, the process of updating the weights to approximate $\boldsymbol{d}$ as the output of the NN is called **learning**. We denote the output of the NN as $g(\boldsymbol{x}, \boldsymbol{\omega})$. **Loss function** is used to evaluate loss, in other word, the difference between $g(\boldsymbol{x}, \boldsymbol{\omega})$ and $\boldsymbol{d}$. In this study, we use **euclidean loss** (L2 norm) to compute loss. The euclidean loss F corresponding to training dataset D is defined as,

$$F(D, \boldsymbol{\omega}) = \frac{1}{2} \sum_{(\boldsymbol{x},\boldsymbol{d}) \in D} \|g(\boldsymbol{x}, \boldsymbol{\omega}) - \boldsymbol{d}\|_2^2. \tag{2.9}$$

2.1.3 Stochastic Gradient Descent and Momentum

Stochastic Gradient Descent (SGD) is a widely used optimization algorithm. Above all, we need to understand what is **gradient descent**. Let's take a simple one-dimensional gradient descent as an example. Consider a continuous differenctiable real-valued function $f : \mathbb{R} \to \mathbb{R}$. For a number $\gamma > 0$ that is small enough, by using a Taylor expansion we obtain that [13],

$$f(x+\gamma) = f(x) + \gamma f'(x) + \mathcal{O}(\gamma^2) \approx f(x) + \gamma f'(x), \tag{2.10}$$

for any $x \in \mathbb{R}$. Here, $f'(x)$ is the first derivative of $f(x)$. Then pick a fixed step-size parameter $\eta > 0$ that satisfies $\gamma = -\eta f'(x)$. From equation 2.10 we obtain that,

$$f\left(x - \eta f'(x)\right) \approx f(x) - \eta f'^2(x) \leq f(x), \tag{2.11}$$

for any $x \in \mathbb{R}$. If η is small enough and $f'(x) \neq 0$, we obtain that,

$$f\left(x - \eta f'(x)\right) < f(x). \tag{2.12}$$

Equation 2.12 means that, if we iteratively update x through update rule

$$x \leftarrow x - \eta f'(x), \tag{2.13}$$

the value of $f(x)$ might decline. η is usually called the **learning rate**.

If f is a function of weight vector $\boldsymbol{\omega}$ of size m, the gradient of function f at $\boldsymbol{\omega}$, denote $\nabla_{\boldsymbol{\omega}} f(\boldsymbol{\omega})$ or $\nabla f(\boldsymbol{\omega})$, is defined as,

$$\nabla_{\boldsymbol{\omega}} f(\boldsymbol{\omega}) = \left[\frac{\partial f(\boldsymbol{\omega})}{\partial \omega_1}, \frac{\partial f(\boldsymbol{\omega})}{\partial \omega_2}, \cdots, \frac{\partial f(\boldsymbol{\omega})}{\partial \omega_m}\right]^\top. \tag{2.14}$$

Similarly, we can iteratively update $\boldsymbol{\omega}$ through update rule

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \eta \nabla_{\boldsymbol{\omega}} f(\boldsymbol{\omega}). \tag{2.15}$$

The target function is usually the expectation of the loss functions for each sample of training dataset. Define $f(\boldsymbol{x}, \boldsymbol{d}, \boldsymbol{\omega})$ as a loss function of a training sample, so that the function we need to minimize is,

$$F(D, \boldsymbol{\omega}) = \frac{1}{|D|} \sum_{(\boldsymbol{x},\boldsymbol{d}) \in D} f(\boldsymbol{x}, \boldsymbol{d}, \boldsymbol{\omega}). \tag{2.16}$$

The gradient of $F(D, \boldsymbol{\omega})$ is,

$$\nabla_{\boldsymbol{\omega}} F(D, \boldsymbol{\omega}) = \frac{1}{|D|} \sum_{(\boldsymbol{x},\boldsymbol{d}) \in D} \nabla_{\boldsymbol{\omega}} f(\boldsymbol{x}, \boldsymbol{d}, \boldsymbol{\omega}). \tag{2.17}$$

Obviously, the time complexity for each update by using gradient descent is $\mathcal{O}(|D|)$. Therefore, the computational cost is large if the number of training samples for each iteration is large. SGD is a method to reduce computational cost at each iteration.

At each iteration of SGD, a random sample from the training dataset is picked to update $\boldsymbol{\omega}$ through a rule,

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \eta \nabla_{\boldsymbol{\omega}} f_r(D, \boldsymbol{\omega}). \tag{2.18}$$

Here, $r \in \{1, \cdots, |D|\}$ is randomly chosen. In this way, the time complexity is reduced from $\mathcal{O}(|D|)$ to $\mathcal{O}(1)$.

The method of picking a set of random samples to update the weight is called minibatch SGD. The number of picked samples is called the **batch size**. The set of picked samples at time step t is denoted as D_t. Loss function $F_t^{\mathrm{MB}}(\boldsymbol{\omega})$ is defined as,

$$F_t^{\mathrm{MB}}(\boldsymbol{\omega}) = \frac{1}{|D_t|} \sum_{(\boldsymbol{x},\boldsymbol{d}) \in D_t} f(\boldsymbol{x}, \boldsymbol{d}, \boldsymbol{\omega}). \tag{2.19}$$

The update rule of minibatch SGD is defined as,

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \eta \nabla_{\omega} F_t^{\mathrm{MB}}(\boldsymbol{\omega}). \tag{2.20}$$

The method of **momentum** is designed to accelerate learning [14]. The detailed derivation process is written in the book [12].

Given momentum parameter ζ, it is helpful to think of the momentum hyperparameter in terms of $\frac{1}{1-\zeta}$. For example, given $\zeta = 0.9$, corresponding to multiplying the maximum speed by 10 relative to the gradient descent algorithm.

We often need to compute a large-sized gradient so many times during learning. **Back propagation** algorithm is a method to efficiently calculate gradients, nowadays most of the deep-learning frameworks, including Caffe which used in this study, have back propagation algorithm built in. The detailed algorithm derivation process is written in the book [12].

2.2 MDP

Before introducing MDP, we need to first understand some important terms:

- **Agent** is the learner and decision maker.
- **Environment** is the thing an agent interacts with, comprising everything outside the agent.
- **Actions** are the agent's methods which allow it to interact and change the environment.
- **Rewards** are signals given to an agent by the environment. Rewards are numerical values that the agent tries to maximize over time.
- **State** corresponds to a situation notified by the environment.
- **Policy** is a strategy which is applied to the agent to decide the next action based on the current state through entire process.
- **Value** is the expected sum of long-term rewards.

MDP is named after the Russian mathematician Andrey Markov, it provides a mathematical framework for modeling decision making process in situations where outcomes sometime randomly determined and sometime under the control of the agent.

MDP has a pair of interactive objects, agent and environment, and several elements:

- $\mathcal{S}$ is a set of **non-terminal states** and called **state space**.
- $\mathcal{S}^+$ is a set of all states, including **terminal** state.
- $\mathcal{A}$ is a set of all actions and called **action space**.
- $\mathcal{A}(s)$ is a set of actions based on state $s \in \mathcal{S}$.
- $p(s'|s,a) = \Pr\{S_{t+1} = s'|S_t = s, A_t = a\}$ is the state-transition probability, where $s \in \mathcal{S}$, $s' \in \mathcal{S}^+$ and $a \in \mathcal{A}(s)$. Here, S_t means the state at time t, A_t means an action at time t, and $\Pr\{X = x|Y = y\}$ means conditional probability of the random variable X takes on the value x given Y takes on the value y. This equation means the probability that action a in state s at time step t will lead to state s' at time $t + 1$.

- $r(s, a, s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s']$ is the expected reward received immediately after transitioning from state s to state s' due to action a. Here, $\mathbb{E}[X|Y]$ means expectation of random variable X given Y holds, and $R_t \in \mathcal{R} \subset \mathbb{R}$ means rewards the agent receives at time step t.

- An **episode** is a sequence $S_0A_0, S_1A_1, \cdots, S_t, A_t$ that is generated by the state transition probability and a static initial-state probability.

The state transitions of an MDP should satisfy the **Markov property**, that means, the probability to have a next state depends only on the present state and action.

As we know, the agent's goal is to maximize the cumulative reward over time. The cumulative reward following time t (if it is the terminal time, denote T), in other word the **return** G_t, is defined as,

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.21}$$

$$= \sum_{k=0}^{T-t-1} R_{t+k+1}, \tag{2.22}$$

where $R_{t+1}, R_{t+2}, R_{t+3}, ..., R_T \in \mathcal{R} \subset \mathbb{R}$ mean rewards that the agent receives after time step t, $\mathcal{R}$ is a finite set of rewards, and R_T is the reward given by the environment which is corresponding to a terminal state. The return of terminal time step G_T is equal to 0. Discount rate is omitted in this study.

If the state and action spaces of an MDP are finite, then this MDP is called a **finite MDP**. Finite MDPs are particularly important to the theory of RL. In this study, all MDPs are finite. So that an MDP's state set $\mathcal{S}$, action set $\mathcal{A}(s)$ and reward set $\mathcal{R}$ are finite.

To draw a graph is an efficient way to understand finite MDP. Figure 2.3 shows an example. There is a state node (a large open circle) for each non-terminal state, a state node (a large open square) for the terminal state, and an action node (a small solid circle) for each state-action pair. Taking action a in one of the non-terminal states will lead you along the line from that state node to the action node corresponding to a, then the environment responds a transition to the next state's node by one of the arrows leaving action node (s, a). Each arrow corresponds to a transition probability, $p(s'|s, a)$, which is the left number of the arrow's label, and an expected reward, $r(s, a, s')$, which is the right number of the arrow's label.

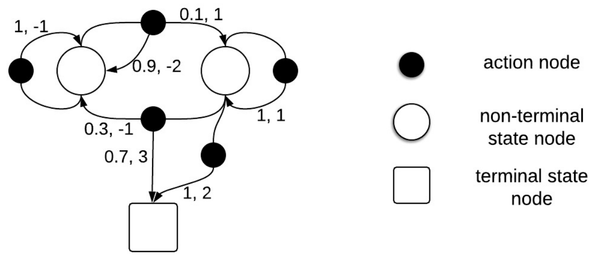

Figure 2.3: Example of a simple finite MDP

The value of a state $s \in \mathcal{S}$ under a policy π, called the **state-value function** for policy π, denoted $v_\pi(s)$, is defined as,

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s], \tag{2.23}$$

for all $s \in \mathcal{S}$. Here, $\mathbb{E}_\pi[X|Y]$ means expectation of random variable X, given Y holds, under policy π. For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states,

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}^+, r \in \mathcal{R}} p(s', r|s, a) \Big[r + v_\pi(s') \Big], \tag{2.24}$$

for all $s \in \mathcal{S}$. Here, $\pi(a|s)$ means probability of taking action a in state s under stochastic policy π. Equation 2.24 is derived from equation 2.23, and it is the **bellman equation** for v_π.

The value of taking action a in state s under a policy π, called the **action-value function** for policy π, denoted $q_\pi(s, a)$, is defined as,

$$q_\pi(s, a) = \mathbb{E}[G_t | S_t = s, A_t = a], \tag{2.25}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. Similarly, the bellman equation for q_π is derived from equation 2.25,

$$q_\pi(s, a) = \sum_{s' \in \mathcal{S}^+, r \in \mathcal{R}} p(s', r|s, a) \Big[r + q_\pi(s', a') \Big], \tag{2.26}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and it is implicit that $a' \in \mathcal{A}(s)$.

Now we want to find **optimal policy** π_*. We say that a policy π is better than or equal to a policy π' if the expected return of π is greater than or equal to that of π' for all states. There is always at least one policy called optimal policy which is better than or equal to all the other policies. We denote all the optimal policies by π_*, they share the same **state-value function** v_*, called the **optimal state-value function**, defined as,

$$v_*(s) = \max_{\pi} v_\pi(s), \tag{2.27}$$

for all $s \in \mathcal{S}$. Optimal policies also share the same **action-value function** q_*, called the **optimal action-value function**, defined as,

$$q_*(s,a) = \max_{\pi} q_\pi(s,a), \tag{2.28}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

In an MDP, v_* or q_* respectively conform to the **bellman optimality equations**,

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + v_*(S_{t+1}) | S_t = s, A_t = a] && (2.29) \\ &= \max_a \sum_{s' \in \mathcal{S}^+, r \in \mathcal{R}} p(s', r | s, a) \Big[r + v_*(s') \Big], && (2.30) \end{aligned}$$

for all $s \in \mathcal{S}$, or,

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] && (2.31) \\ &= \sum_{s' \in \mathcal{S}^+, r \in \mathcal{R}} p(s', r | s, a) \Big[r + \max_{a'} q_*(s', a') \Big], && (2.32) \end{aligned}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. We can easily obtain optimal policies as long as we have found the **optimal value functions** v_* or q_*.

2.3 RL

RL is a framework for solving problems that can be expressed as MDPs. In RL, **Generalized Policy Iteration** (GPI) is a widely used idea to solve the bellman optimality equations. GPI has three steps (**policy evaluation**, **policy improvement** and **policy iteration**) until the policy converges.

2.3.1 Policy Evaluation

Policy evaluation is to predict the state-value function with respect to a given policy.

When solving MDPs, iterative solution methods are suitable for our purpose. For a sequence of approximate state-value functions $v_0, v_1, v_2, \ldots$, each mapping S^+ to $\mathbb{R}$, the initial approximation v_0 is chosen arbitrarily. One method to solve MDPs is to obtain each successive approximation by using the bellman equation 2.24 as an update rule,

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}^+, r \in \mathcal{R}} p(s', r|s, a)\Big[r + v_k(s')\Big], \tag{2.33}$$

for all $s \in \mathcal{S}$. Obviously, $v_k = v_\pi$ is a fixed point, and the sequence $\{v_k\}$ converges to v_π as $k \to \infty$ under the condition of eventual termination is guaranteed from all states under the policy π.

The prerequisite for this method is that the state-transition probability $p(s'|s, a)$ is known, but in this study we don't actually know the probability, therefore we need to use other methods.

Monte Carlo (MC) **Backup** and **TD Backup** are two other methods for learning the state-value function for a given policy without knowing the state-transition probability. As we know, state-value function $v_\pi(s)$ is defined as equation 2.23. For MC backup method in an MDP, when improving the **state-value estimate** of a state $V(S_t)$, G_t is used as a target value. The update rule is

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big], \tag{2.34}$$

where $\alpha \in (0, 1]$ is a step-size parameter. Here, "$\leftarrow$" means assignment. Similarly, we can get the **action-value estimate** of a state-action pair $Q(S_t, A_t)$. The update rule is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[G_t - Q(S_t, A_t)\Big], \tag{2.35}$$

where $\alpha \in (0, 1]$.

As shown in Figure 2.4, in MC method, the action-value estimate of non-terminal state-action pairs can be obtained when the process reaches the terminal state.

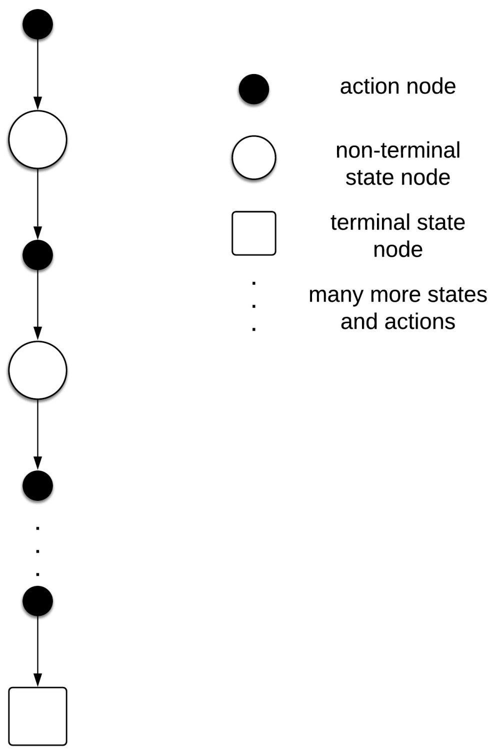

Figure 2.4: Backup diagram of MC method

For TD backup in an MDP, we update the state-value estimate of a state $V(S_t)$ by the observed reward R_{t+1} and estimate $V(S_{t+1})$ at time $t+1$. The update rule of the simplest TD method called TD(0) is

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + V(S_{t+1}) - V(S_t)\Big], \tag{2.36}$$

where $\alpha \in (0, 1]$. Similarly, we can get the action-value estimate of a state-action pair $Q(S_t, A_t)$ of TD(0). The update rule is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big], \tag{2.37}$$

where $\alpha \in (0, 1]$.

Figure 2.5 shows the backup diagram for TD(0). The action-value estimate of the previous state-action pair is updated based on the next state-action pair.

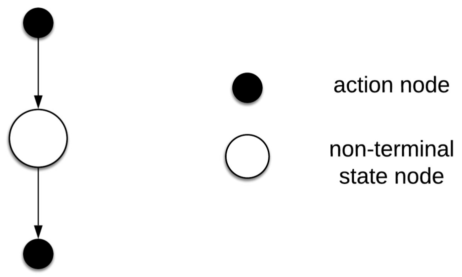

Figure 2.5: Backup diagram of TD(0) method

2.3.2 Policy Improvement

Policy improvement is the process of improving a policy. Policy improvement is actually the process of traversing each action under each state, computing values corresponding to each traversed action, and choosing the action that has the max value as a new policy. The deterministic new policy, π' (mapping from $\mathcal{S}$ to $\mathcal{A}$), is given by

$$\pi'(s) = \arg\max_{a \in \mathcal{A}(s)} q_\pi(s, a), \tag{2.38}$$

for all $s \in \mathcal{S}$. Here, $\arg\max_a f(a)$ means a value of a at which $f(a)$ takes its maximal value. π' is called the **greedy policy**. The greedy policy takes the action that looks best after one time step.

Traverse each state and update the policy as above, then the old policy π is updated to the new policy π'. From Sutton's book we know $v_\pi(s) \leq v_{\pi'}(s)$ for all $s \in \mathcal{S}$ [11]. That is to say, the value of each state of the new policy is greater than or equal to that of the old policy. So that the cumulative value is greater than or equal to before, which means that the policy is improved.

2.3.3 Policy Iteration

When policy π is updated to a better policy π' through policy improvement, we can compute $v_{\pi'}$ through policy evaluation and then update to a better policy π''. The process can be iterated until reaching the optimal policy. This iteration is called policy iteration. We can obtain a sequence of monotonously improving policies and state-value functions,

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_{\pi_*}. \tag{2.39}$$

Here "$\xrightarrow{E}$" means a policy evaluation and "$\xrightarrow{I}$" means a policy improvement. Because an MDP has a finite number of deterministic policies, this iteration must converge to an optimal policy and an optimal state-value function in limit iterations.

However there is a problem with the above GPI idea, that is, many state-action pairs may never be visited. This is very serious because we need to compare all the opportunities and estimate the value of all the actions from each state, otherwise the policy may be trapped in local optima. To avoid this problem, the probability of each action being selected should be greater than 0.

There are two approaches to select each action with a probability greater than 0, we call them **on-policy** methods and **off-policy** methods. As pre-requisite knowledge, we need to figure out what is **target policy** and what is **behavior policy**. The target policy is the policy to be applied after learning. In briefly speaking, the target policy is the policy being learned about. The behavior policy is the policy used to generate behavior, that is, the policy being used during learning. For on-policy methods, target policy is used as behavior policy. For off-policy methods target policy and behavior policy are different.

MC method explained in Subsection 2.3.1 is one of on-policy methods. For MC method, we optimize action-value function q_* (equation 2.32) instead of state-value function, and updated by **ϵ-greedy policy**. The agent follows ϵ-greedy policy takes actions by greedy policy with a probability of $1 - \epsilon$ or randomly with a probability of ϵ. This approach ensures all the states and actions are explored. If the number of episodes that policy evaluated is large enough, MC method is convergent.

Q-learning is an off-policy TD(0) method, the update rule is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]. \tag{2.40}$$

Here $\alpha \in (0, 1]$. Under this condition, the learned action-value function Q is directly approximates the optimal value function q_*. Q-learning is convergent if all state-action pairs continue to be updated. Figure 2.6 shows the backup diagram of Q-learning, and diagram of target policy and behavior policy.

Table 2.1 summarizes RL methods, i.e, MC method, Q-learning, and Sarsa in terms of behavior policies and backups. The popular RL method Sarsa is placed in the table although it is not explained in this subsection.

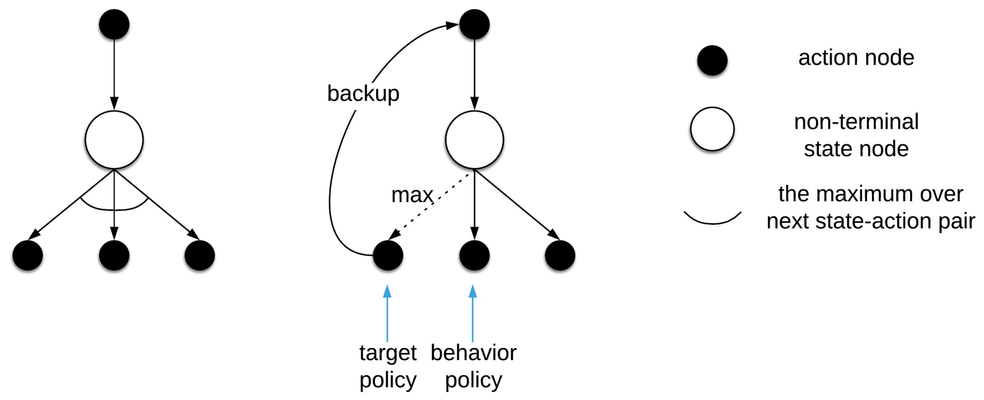

Figure 2.6: Backup diagram of Q-learning, and diagram of target policy and behavior policy

	Backup	
	TD(0)	TD(1)
On-policy	Sarsa	MC method
Off-policy	Q-learning	—

Table 2.1: Table of RL methods

2.3.4 Function Approximation

So far, our discussion of RL are tabular methods, which are using tables to store the value functions. However if state and action spaces are very large, it is difficult to compute and store data. Moreover, in many tasks, "almost every state encountered will never have been seen before. To make sensible decisions in such states it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one [11]." Here we need to rely on function approximation.

Function approximation is an instance of supervised learning, another area of machine learning. The idea of function approximation is to use a proper function (linear function or non-linear function, etc.) to fit the true value of the value function based on a small number of training samples, making it applicable to a large number of test samples. The core task of this idea is to find the optimal parameters of the function.

With function approximation, we write $\hat{v}(s, \boldsymbol{\omega}) \approx v_\pi(s)$ for the approximated value of state s given weight vector $\boldsymbol{\omega}$. $\boldsymbol{\omega}$ is a column vector with a fixed number of real valued components, $\boldsymbol{\omega} = (\omega_1, \omega_2, \cdots, \omega_n)^\top$. Generally, $\hat{v}$ is a function computed by NNs and $\boldsymbol{\omega}$ is the vector of connection weights in all layers. This study is also carried out through NNs.

We summarize MC method and Q-learning in Chapter 2.3.1 and 2.3.3 as a general tabular expression,

$$f(X) \leftarrow f(X) + \alpha \Big[\Theta - f(X) \Big]. \tag{2.41}$$

Here, f is an arbitrary function, $\alpha \in (0, 1]$, and Θ is the target value. Also, X and Θ are random variables, i-th observation of X or Θ respectively gives x_i or θ_i, and $\mathbb{E}[\cdot]$ evaluate an expected value. Let us consider a function approximation $\hat{f}(x, \boldsymbol{\omega}) \approx f(x)$.

Now we would like to update $\hat{f}(X, \boldsymbol{\omega})$ toward target Θ. To achieve this goal, we introduce a loss function to minimize, as,

$$\mathbb{E}\left[\Big[\hat{f}(X, \boldsymbol{\omega}) - \Theta \Big]^2 \right] = \sum_i \Big[\hat{f}(x_i, \boldsymbol{\omega}) - \theta_i \Big]^2. \tag{2.42}$$

So that we can obtain the function approximation of update rule 2.41 with η ($0 < \eta \in \mathbb{R}$ and η is small enough) through the similar derivation process in Subsection 2.1.3, as,

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \eta \Big[\hat{f}(x_i, \boldsymbol{\omega}) - \theta_i \Big] \nabla_{\boldsymbol{\omega}} \hat{f}(x_i, \boldsymbol{\omega}). \tag{2.43}$$

Note that the expected value in equation 2.42 can be defined when the behavior policy is static.

Chapter 3

Related Works

3.1 Backgammon

Backgammon is one of the oldest known board game. It is a two-player turn-based game with a combination of strategy and uncertainty due to rolling dice. Although, the dice affects the outcome of a single game, the experts have good win rate. A lot of reseach on Backgammon have been carried out by computer scientists.

In 1979, BKG 9.8, a backgammon program developed by Hans Berliner, beat the world backgammon human champion in a match by a score of 7-1 [15]. This is the first time in any game a world champion has been defeated by a computer program though, BKG 9.8 received more favorable dice rolls in the match.

In 1995, Gerald Tesauro developed a backgammon program called TD-Gammon by using TD-(λ) method with NNs [4]. Tesauro designed a 3-layer NN to estimate the **afterstate** function. The input of NN is the number of checkers of each point, and the output is the approximation of the value function of two kinds of winning ways for each player. At the beginning of learning, the policy is greedy policy with random initial weights of the NN. TD-Gammon generate a training dataset through self-play, then use the training dataset to compute the loss function, and update the weight through back propagation algorithm. TD-Gammon achieves the level of human experts.

Backgammon and card game solitaires with uncertainty are both turn-based games with uncertainty. The difference is that backgammon is a two-player game, and card game solitaires with uncertainty are single-player games.

3.2 Atari 2600

In 2015, DeepMind Technologies developed deep Q-network Atari 2600 AIs. The AIs is developed by using a variant of Q-learning algorithm with DNNs. A combination of 3 convolutional layers and 2 fully-connected layers is used to approximate the action-value function. This agent receives only raw pixel values and the game score as input. The output is the approximated action values for a set of actions.

The AI is trained by an off-policy Q-learning, whose target policy is greedy policy and behavior policy is ϵ-greedy policy. Moreover, a method called **experience replay** is used to stabilize learning. The method is to store the observed $(s, a, r`, s')$ pair to a **replay memory**, and randomly pick stored pairs from the replay memory for learning.

This AI has exceeded existing RL AIs in 43 games, and has reached the level of professional human players in 29 games among 49 games in Atari 2600 series.

Atari 2600 games and card game solitaires are both single-player games. The expected length of an episode of Atari 2600 games is about 10^3 and the size of action set is about 10^1, either is similar to card game solitaires. Therefore, it is expected that experience replay is also useful for training AIs to play card games solitaires.

3.3 Game 2048

Game 2048 is a stochastic single-player game. In 2018, Naoki Kondo and Kiminori Matsuzaki developed AIs for Game 2048 based on deep CNNs, achieving an average score of 93 830 [16]. Their experiments showed that the AIs that are equipped with CNNs reach a close level of state-of-the-art AIs that are equipped with specialized networks for game 2048.

The researchers incremented the number of convolutional layers from 2 to 9, while keeping the number of weights almost the same. The AIs were trained by supervised learning with a large number of play records from existing strong computer players.

As a result, they found that the AIs trained by 3 or more convolutional layers performed much better than the AIs trained by 2 convolutoinal layers, even they have similar number of weights.

As same as game 2048, card games solitaires are also stochastic single-player games. Therefore, it is expected that CNNs are also performing well for card game solitaires with uncertainty.

Chapter 4

Experiments

This section introduces setup of training AIs by MC method and Q-learning to play TriPeaks and Russian Solitaire, and shows performance analysis on the basis of win rates.

4.1 Setup

Subsection 4.1.1 introduces rules of TriPeaks and Russian Solitaire. Subsection 4.1.2 explains two encoding methods used in this study. Subsection 4.1.3 shows architecture of NNs. Subsection 4.1.4 describes an algorithm of training data sampling.

4.1.1 Rules

Tripeaks and Russian Solitaire are played with a standard 52-card deck, comprising 13 cards each of 4 suits: Spades ♠, Hearts ♡, Diamonds ♢, and Clubs ♣. The cards in each suit are: Ace (A), 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack (J), Queen (Q), and King (K). This order is the basic rank from low (Ace) to high (King). However these two games have very different rules but both with uncertainty.

The rules of TriPeaks are as following:

- **Layout**. Figure 4.1 shows the layout of TriPeaks. 28 cards make up the tableau (3 peaks with 4 tiers). 23 remaining cards build up the stock. The last card is shown up and discarded in the waste pile.

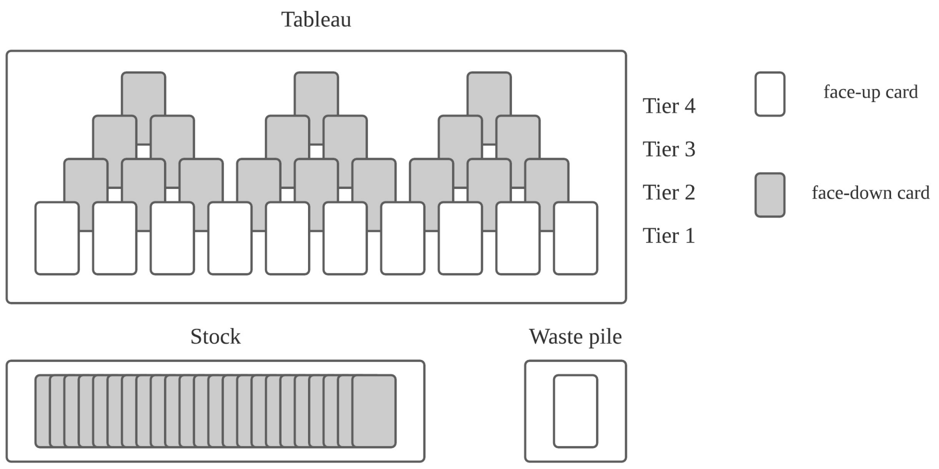

Figure 4.1: Layout of TriPeaks

- **Waste pile** (also known as **foundation**). Build up or down by rank. A King may be played on an Ace and an Ace may be played on a King.
- **Tableau (Peaks)**. A card in the tableau is face up if it's not overlapping, otherwise face down. Face-up cards can be discarded to the waste pile.
- **Stock**. A card picked from the stock can be turned face up and the card is discarded to the waste pile.
- **Winning condition**. Discard all cards in the tableau.
- **Game play**. A card of neighbor rank regardless of suit from the tableau and a card from the stock can be discarded to the waste pile. This card becomes the new top card of the waste pile. The process is repeated multiple times and ends until no action can be taken.

The rules of Russian Solitaire are as following:

- **Layout**. Figure 4.2 shows the layout of Russian Solitaire. 52 cards make up the tableau (7 piles). 4 waste piles are empty at the beginning.

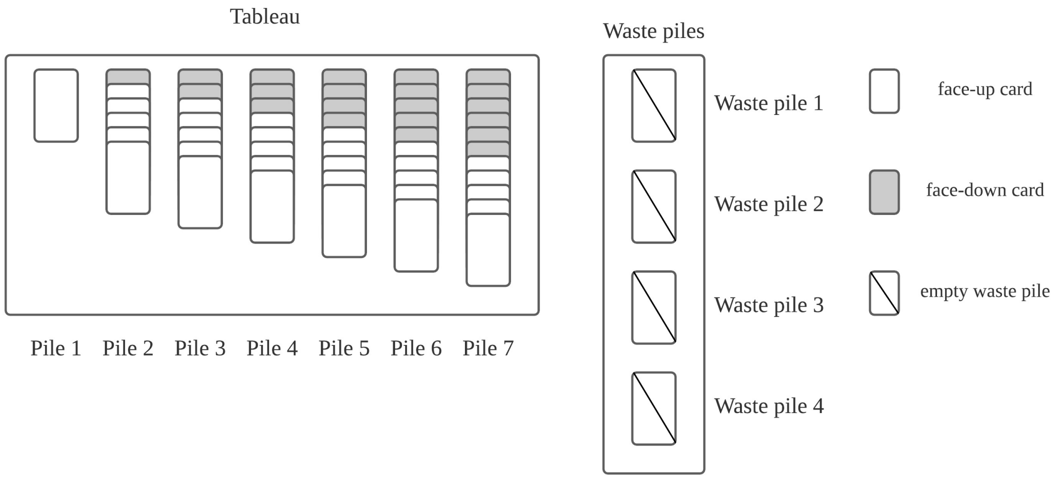

Figure 4.2: Layout of Russian Solitaire

- **Waste piles**. 4 waste piles all build up by rank with same suits. An Ace can be moved to any empty waste pile.
- **Tableau**. A card in the tableau is turned face up if it is not overlapping. A face-up card in the tableau, no matter how deep it is in a pile, may be moved to another pile if it is in the same suit and one rank lower than the top card in the other pile. When a card is moving, all the cards that covering the moving card are also moving with it as a rigid body. A king can be moved to an empty pile. A card from the top of a pile can be discarded to the waste pile.
- **Winning condition**. Discard all cards in the tableau.
- **Game play**. A player can either move cards in the tableau or discard a card from the top of a pile (if the card is in the same suit and one rank higher than the top card of a waste pile). Once a card is discarded, it becomes the new top card of the waste pile. The process is repeated multiple times and ends until no action can be taken.

Note that, for Russian Solitaire in this study, a king is not allowed to be move to an empty pile if the king is the bottom card. This additional rule is to avoid infinite loop of king round trip from one to another pile.

4.1.2 Encoding afterstates

Consider a set $\widetilde{S}$, a map $\Phi : S \times A \to \widetilde{S}$, a map $\widetilde{Q} : \widetilde{S} \to \mathbb{R}$, and a policy π. If $Q_\pi(s,a) = \widetilde{Q}\Big[\Phi(s,a)\Big]$ holds for all $(s,a) \in S \times A$, and $|\widetilde{S}|$ is sufficiently smaller than $|S \times A|$, then learning $\widetilde{Q}$ maybe easier than learning Q_π.

It is known that the concept of afterstate is sometimes useful to compose these $\widetilde{S}$ and Φ [11]. Figure 4.3 shows an example of afterstate. In the case of the figure, a card is moved from one pile to another, this action discloses another card.

Two encoding methods are used to encode afterstate to a binary sequence which is fed into the input layer of NNs. We assume that the standard 52-card deck is indexed from 0 to 51 in accordance with the order of “A♠,$\cdots$, K♠,A♡,$\cdots$,K♡,A♣,$\cdots$,K♣,$\cdots$,A♢,$\cdots$,K♢”.

Encoding method 1 (E1) encodes the game situation focusing on individual cards. In this method, the situation formed by one card is represented by a binary sub-sequence, then the entire situation is represented by combining 52 sub-sequences. i-th sub-sequence represents a location, face up or down, and existence of card i. To indicate a location from U locations, the representation uses U binaries (the binary value corresponding to the location is set to 1, and the others are set to 0). To indicate face up or down, the representation uses one binary (face up, 0; face down, 1). To indicate existence, the representation uses one binary (exist, 0; discarded, 1). For example, if a solitaire uses 4 locations, then “001000” means a card is at the third location, and “000010” means a card is face down.

Encoding method 2 (E2) encodes the game situation focusing on individual locations. In this method, the situation formed by one location is represented by a binary sub-sequence, then the entire situation is represented by combining all the sub-sequences. j-th sub-sequence represents the index of face-up card, the existence of a face-down card, or vacancy of the j-th location. To indicate the index, 52 binaries are used (the binary value corresponding to the index of the card is set to 1, and the others are set to 0). To indicate the existence of a face-down card, one binary is used (not exist, 0; exist, 1). To indicate vacancy, one binary is used (not vacant, 0; vacant, 1). For example, “0100$\cdots$00” means 2♠ exists, and “0000$\cdots$10” means a face-down card exists in the location.

Table 4.1 shows the number of locations and actions of TriPeaks and Russian Solitaire. In the case of TriPeaks, the tableau has 28 locations, the stock

Game	Number of Locations	Number of Actions
TriPeaks	52	29
Russian Solitaire	368	2191

Table 4.1: Size of locations and actions of TriPeaks and Russian Solitaire

has 23 locations, and the waste pile has one location. 28 actions correspond discarding a card from tableau, and one action corresponds discarding a card from the stock. In the case of Russian Solitaire, each pile of the tableau has 52 locations and each waste pile has one location. A card from any location of the tableau may be able to move to the top of any other piles, and the top card of any pile may be able to discard to one of the waste piles.

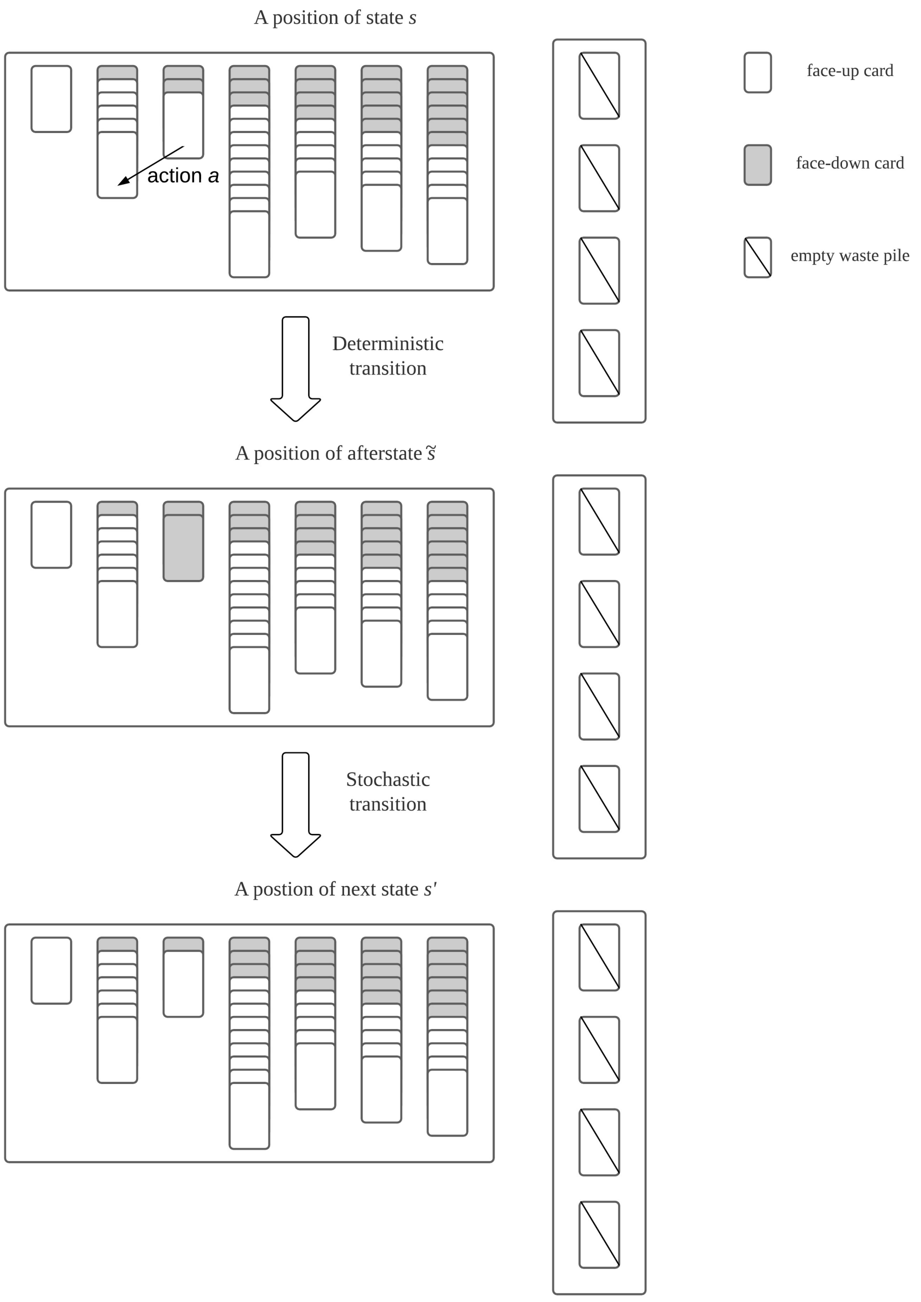

Figure 4.3: Example of afterstate

4.1.3 Architecture of NNs

As shown in Figure 4.4, 4-layer NNs are used for training AIs in this study. The order of the layers is input layer, convolutional layer, and two fully-connected layers. The input is encoded afterstate and the output is a real number between 0 and 1 which represent the probability of choosing an action. Table 4.2∼4.4 shows the details of the layers of each NN. For different NNs for the same game, the number of weights are kept almost the same.

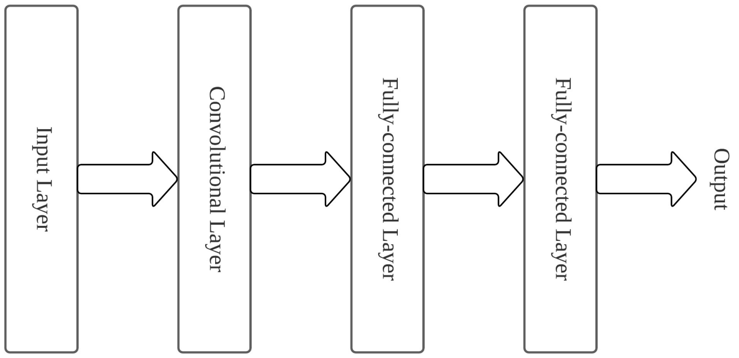

Figure 4.4: Structure of the NN

Layer	Output Dimension	Number of Weights and Biases	Activation Function
Input	52(C)×1(H)×54(W)	—	—
Conv1×1	8(C)×1(H)×54(W)	424	ReL
Fc	100	43300	ReL
Fc	1	101	Sigmoid

Table 4.2: Architecture of NNs with E2 for TriPeaks. Conv1×1 stands for convolution using $1(K_h)\times 1(K_w)$ kernel and Fc stands for fully-connected. The total number of weights and biases is 43 825.

Layer	Output Dimension	Number of Weights and Biases	Activation Function
Input	373(C)×1(H)×52(W)	—	—
Conv1×1	56(C)×1(H)×52(W)	20944	ReL
Fc	600	1747800	ReL
Fc	1	601	Sigmoid

Table 4.3: Architecture of NNs with E1 for Russian Solitaire. Conv1×1 stands for convolution using $1(K_h)\times1(K_w)$ kernel and Fc stands for fully-connected. The total number of weights and biasess is 1 769 345.

Layer	Output Dimension	Number of Weights and Biases	Activation Function
Input	54(C)×1(H)×368(W)	—	—
Conv1×1	8(C)×1(H)×368(W)	440	ReL
Fc	600	1767000	ReL
Fc	1	601	Sigmoid

Table 4.4: Architecture of NNs with E2 for Russian Solitaire. Conv1×1 stands for convolution using $1(K_h)\times1(K_w)$ kernel and Fc stands for fully-connected. The total number of weights and biases is 1 768 041.

4.1.4 Sampling Training Dataset for RL

To use SGD, a method for randomly choosing samples is needed. For the purpose of choosing samples randomly, the experience replay method is used in this study. Algorithm 1 shows such a method for carrying out RL methods using replay memory on the basis of the experience replay method. The algorithm stores a sequence of training samples (line 14), then randomly selects samples from the replay memory for learning (line 17). This method greatly improves the stability of learning.

Algorithm 1: Learning with replay memory

```
1  $B \leftarrow$ batch size;
2  $L \leftarrow$ size of replay memory ($L$ is multiple of $B$);
3  Initialize replay memory $\Upsilon$ to empty;
4  **Function** gen_sample()
5      Initialize gameplay $\Xi$ to empty ($\Xi$ is static and the initialization
         is done only once);
6      **if** $\Xi$ *is empty* **then**
7          Generate and store a gameplay to $\Xi$;
8      **end**
9  Take one oldest sample $\xi$ from $\Xi$;
10 **return** $\xi$;
11
12 **foreach** $t \in \{0, 1, \cdots\}$ **do**
13     **while** $|\Upsilon| < L$ **do**
14         Store gen_sample() to $\Upsilon$;
15     **end**
16     Randomly select $B$ samples from $\Upsilon$ to form $D_t$;
17     Update NN weights using $D_t$ as shown in update rule 2.20;
18     Clear the oldest $B$ samples in $\Upsilon$;
19 **end**
```

To carry out RL using MC method, a sample ξ means (s, a, r_T) as shown in equations 2.35 and 2.43. To carry out RL using Q-learning, a sample ξ means $\left[s, a, r', \max_a \hat{Q}(s', a, \boldsymbol{\omega})\right]$ as shown in equations 2.40 and 2.43.

4.2 Results

Table 4.5 shows 5 AIs developed in this study, and Table 4.6 shows the win rates of these AIs playing TriPeaks and Russian Solitaire.

Table 4.7∼4.12 show hyperparameters used during learning. The parameter ϵ and learning rate are adjusted by hand carefully. Each AI is trained using $\epsilon = 1$ in the beginning so that the AI can explore more afterstates. The win rate improves rapidly in the beginning. When the improvement slows down, the ϵ decreases linearly during a period of iterations. After a certain number of iterations, the ϵ stops decreasing and remains constant, the learning rate descents at the same time.

Figure 4.5 and Figure 4.6 show the training process. Caffe 1.0 is used for these training [17]. B is set to 16 and L is set to 500 000. For TriPeaks, about one million gameplays are used for training each AI. After about 3.4 million iterations, from Figure 4.5, we can see Q2 outperforms MC2. Win rates of MC1 and Q1 are not available in this experiment, because no win game observed. This means that E2 outperformed E1.

For Russian Solitaire, about 1.5 million gameplays are used for training each AI. After about 1 million iterations, from Figure 4.6, we can see Q-learning outperforms MC method. However we cannot decide which encoding method is better.

According to the results, we can say that Q-learning is better than MC method for these two games. But the efficiency of the two encoding methods is still unclear in the case of Russian Solitaire.

Agent	Description
RD	Uniform random policy
MC1	Trained by MC method with E1
MC2	Trained by MC method with E2
Q1	Trained by Q-learning with E1
Q2	Trained by Q-learning with E2

Table 4.5: Five AIs

Agent	Win Rate of TriPeaks	Win Rate of Russian Solitaire
RD	0.00051±0.00014	0.00220±0.00030
MC1	—	0.00340±0.00037
MC2	0.27394±0.00282	0.00352±0.00037
Q1	—	0.00443±0.00042
Q2	0.51865±0.00316	0.00494±0.00044

Table 4.6: Win rates of five AIs playing TriPeaks and Russian Solitaire. The sign ± represent 95% confidence intervals.

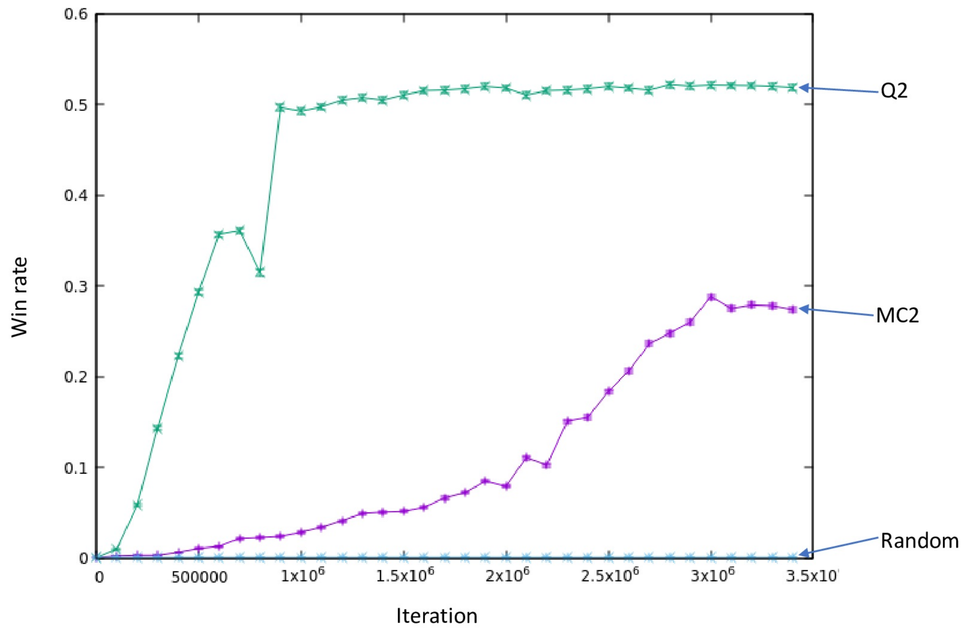

Figure 4.5: Training progress of TriPeaks

Iterations			ϵ	Learning Rate
1	∼	500 000	1	0.1
500 001	∼	700 000	1∼0.2	0.1
700 001	∼	34 000 000	0.2	0.02

Table 4.7: Hyperparameters of MC2 for TriPeaks. The value of ϵ during iterations 500 001 to 700 000 decreases linearly.

Iterations			ϵ	Learning Rate
1	~	600 000	1	1
600 001	~	800 000	1~0.1	1
800 001	~	34 000 000	0.1	0.2

Table 4.8: Hyperparameters of Q2 for TriPeaks. The value of ϵ during iterations 600 001 to 800 000 decreases linearly.

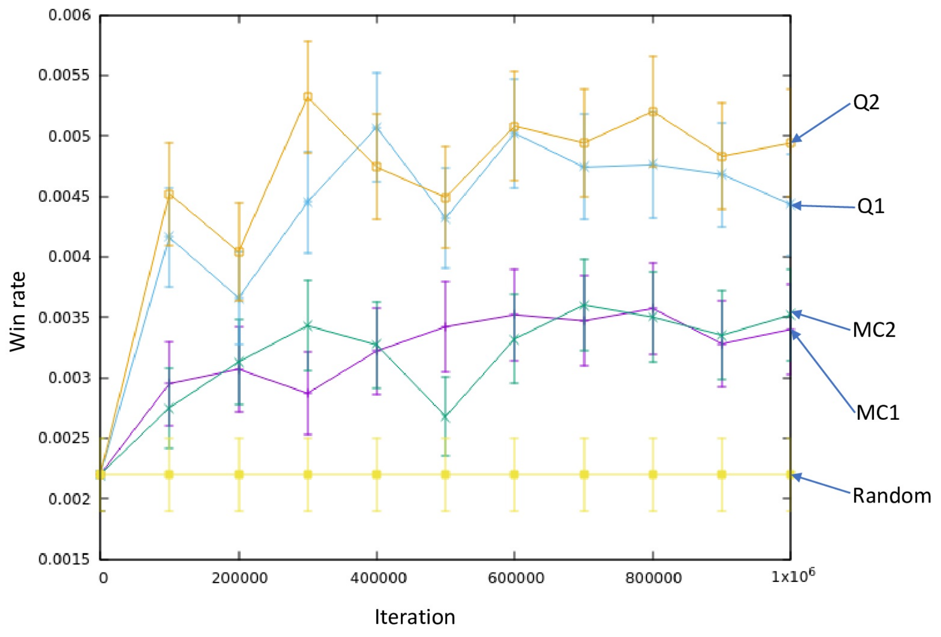

Figure 4.6: Training progress of Russian Solitaire

Iterations			ϵ	Learning Rate
1	~	150 000	1	0.01
150 001	~	250 000	1~0.2	0.01
250 001	~	1 000 000	0.2	0.002

Table 4.9: Hyperparameters of MC1 for Russian Solitaire. The value of ϵ during iterations 150 001 to 250 000 decreases linearly.

Iterations			ϵ	Learning Rate
1	$\sim$	350 000	1	0.01
350 001	$\sim$	550 000	1$\sim$0.2	0.01
550 001	$\sim$	1 000 000	0.2	0.002

Table 4.10: Hyperparameters of MC2 for Russian Solitaire. The value of ϵ during iterations 350 001 to 550 000 decreases linearly.

Iterations			ϵ	Learning Rate
1	$\sim$	500 000	1	0.1
500 001	$\sim$	600 000	1$\sim$0.2	0.1
600 001	$\sim$	1 000 000	0.2	0.01

Table 4.11: Hyperparameters of Q1 for Russian Solitaire. The value of ϵ during iterations 500 001 to 600 000 decreases linearly.

Iterations			ϵ	Learning Rate
1	$\sim$	300 000	1	0.1
300 001	$\sim$	400 000	1$\sim$0.2	0.1
400 001	$\sim$	1 000 000	0.2	0.02

Table 4.12: Hyperparameters of Q2 for Russian Solitaire. The value of ϵ during iterations 300 001 to 400 000 decreases linearly.

Chapter 5

Conclusion

This study developed AIs using RL and measured win rates in the domain of card game solitaires with uncertainty. The solitaires used in this study were TriPeaks and Russian Solitaire, and the RL used were MC method and Q-learning. To represent action-value functions, afterstates were used instead of states, and NNs were used to approximate the functions. Moreover, a method called experience replay was used to stabilize learning.

Experimental results showed that in the case of TriPeaks and Russian Solitaire, off-policy Q-learning outperformed on-policy MC method. In order to confirm the superiority of Q-learning against MC method for card game solitaire, more experiments with a wide spectrum of card game solitaires are required.

In the case of TriPeaks, AI trained by Q-leaning exceeded the level of casual human players that are allowed multiple cancellations of decisions. However, no counter-intuitive behavior taken by the AI was found in the experiments. The reason why the AI exceeded the level is that the AI rarely makes mistakes in TriPeaks.

In the case of Russian solitaire, all the AIs developed in this work did not reach the level of such human players, although the number of weights used in NN is 40 times greater than the number of weights used in the case of TriPeaks. One reason for this inferiority can be that Russian Solitaire has a hundred times larger action set than TriPeaks does. If this is the case, the performance will be improved by enlarging the scale of experiment, e.g., enlarging the size of experience replay or the size of NN. Another reason can be that Russian Solitaire induces casual human players to cancel decisions more than TriPeaks does, so that the win rate of casual human players can be higher than that of an optimal policy.

Bibliography

[1] Kaplan, A., Haenlein, M. Siri, siri, in my hand: Who's the fairest in the land? on the interpretations, illustrations, and implications of artificial intelligence. *Business Horizons*, 62(1):15–25, 2019.

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

[3] Schaeffer, J., Lake, R., Lu, P., Bryant, M. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21–21, 1996.

[4] Tesauro, G. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[5] Campbell, M., Hoane A. J., Hsu, F.-H. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[7] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G., Graepel, T., Hassabis, D. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[8] Brown, N., Sandholm, T. Superhuman AI for multiplayer poker. *Science*, 365(6456):885–890, 2019.

[9] Morehead, A. H. *The complete book of solitaire and patience games.* Read Books Ltd, 2015.

[10] Schultz, R. *World of solitaire.* 2007. `https://worldofsolitaire.com`.

[11] Sutton, R. S., Barto, A. G. *Reinforcement learning: An introduction.* MIT Press, 2018.

[12] Bengio, Y., Goodfellow, I., Courville, A. *Deep learning.* MIT Press, 2017.

[13] Aston, Z., Zachary, C. L., Mu, L., Alexander, J. S. *Dive into deep learning.* 2020. `https://d2l.ai`.

[14] Polyak, B. T. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[15] Berliner, H. J. Backgammon computer program beats world champion. *Artificial Intelligence*, 14(2):205–220, 1980.

[16] Kondo, N., Matsuzaki, K. Playing game 2048 with deep convolutional neural networks trained by supervised learning. *Journal of Information Processing*, 27:340–347, 2019.

[17] Jia, Y.-Q. *Caffe | Deep Learning Framework.* `https://caffe.berkeleyvision.org/`.