Xiangyu Zhao

Deep Reinforcement Learning for Mahjong

Computer Science Tripos – Part II

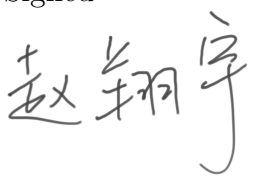
Trinity College

14 May 2021

Declaration of Originality

I, Xiangyu Zhao of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed



Date 14 May 2021

Acknowledgements

Firstly, I would like to thank my project supervisor and Director of Studies, Dr Sean Holden, for his invaluable guidance throughout my work on this project. I would also like to thank my other two Directors of Studies, Professor Frank Stajano and Dr Neel Krishnaswami, for holding weekly project meetings during term time, checking my progress on this project, and kindly providing me with advice on running this project. I am also very grateful to Chenxuan Qu and Yulong Huang for proofreading my dissertation, and the Cambridge High Performance Computing Service (HPCS) support team for kindly and promptly answering my questions about using the HPCS. Finally, I would like to give my special thanks to my girlfriend, Jing Zeng, for her warmest emotional support that accompanied me through my darkest time.

Proforma

Candidate Number:	**2376D**
Project Title:	**Deep Reinforcement Learning for Mahjong**
Examination:	**Computer Science Tripos – Part II, June 2021**
Word Count:	**10,557**[1]
Line Count:	**14,571**
Project Originator:	**2376D**
Supervisor:	**Dr Sean Holden**

Original Aims of the Project

Mahjong is a very popular multiplayer imperfect-information game developed in China in the late 19th-century, with some very challenging features for AI research. In my Part II project, I aimed to build an AI for a reduced, 3-player version of the Japanese Riichi Mahjong, using deep reinforcement learning, and sought to defeat an agent that plays purely randomly. I planned to pre-train 5 deep convolutional neural networks (CNNs), each corresponding to one type of action in Riichi Mahjong, and enhance the major action's model, namely the discard model, using self-play reinforcement learning.

Work Completed

I trained two Mahjong agents using supervised learning (SL), and another agent by improving one of the SL agents using reinforcement learning (RL), with necessary additional modules including data collection and processing, Mahjong hand calculation and a game simulator, for the agents to be trained and evaluated. All agents have passed my pre-set success criteria. In addition, my CNN models have achieved top-tier test accuracies, and the RL agent performed significantly better than both SL agents, suggesting that my project possesses a strong potential to make a contribution to the relevant topic, pending online evaluations against human players.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Special Difficulties

Due to the COVID-19 pandemic, I was forced to work remotely from home in China for the entire Lent and Easter Terms. The environment at home was not ideal for working, and Internet access was not always reliable. I also had to constantly work in antisocial hours due to a 7/8-hour time difference from the UK. My mental conditions were also seriously affected, because of the significantly reduced opportunities to meet my friends and the strong concerns about not being able to perform up to my true ability academically.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Deep reinforcement learning (DRL) has made remarkable achievements in building *artificial intelligence* (AI) in games in the past decade. AIs in *perfect-information* games are already highly advanced and have reached superhuman level even in the game of Go, one of the most complicated perfect-information games, in DeepMind's AlphaGo series [1, 2]. In 2017, DeepMind introduced a general RL algorithm, named AlphaZero, that beat the most powerful programs in chess, Go, and Shogi (Japanese chess). Nowadays, research on game AI has gradually evolved from 2-player perfect-information games (e.g., chess, Go, and Shogi) to even more complicated multi-player *imperfect-information* games (e.g., Dota 2 [3], StarCraft II [4] and Mahjong [5, 6, 7, 8]).

Mahjong is a very popular tile-based multi-round multi-player imperfect-information game that was developed in China in the late 19th century, and nowadays has hundreds of millions of players worldwide. It is a game of skill, strategy and calculation, and it involves a degree of chance. A Mahjong game commonly consists of four players. In each round, the players take turns to draw and discard *tiles* until one of them completes a winning *hand* first, or the round ends with a draw after all tiles are drawn and discarded. The players can also claim a discarded tile from another player to complete an open *meld*, which may interrupt the regular playing order. A more detailed overview of Mahjong can be found on its Wikipedia page [9], and I will explain the rules of Mahjong in more detail later in Section 2.2 and Appendix B.

Mahjong is very challenging for AI research due to the following features:

1. **Mahjong has a huge set of hidden information:** in each round, each player can only see their own hand, open melds of all players, and the tiles discarded by all players so far. The player cannot see the other players' private hands, or the tiles that are not yet drawn. This gives an average of more than 10^{48} hidden states indistinguishable to a player on each decision point [7]. Since the value of an action for a player depends not only on their own hand, but also on the hands of the other players and the remaining tiles, it is hard for an AI to derive a reward system solely from the observed information.

2. **Mahjong has very complex playing rules:**

   (a) There are various types of actions, including discard, Chii, Pon, Kan, etc.;

   (b) The regular playing order can be interrupted by certain actions (Pon, Kan, and win).

   Since there are 13 tiles in each player's hand in most of the common variations of Mahjong, it is hard to predict those interruptions, and therefore it would be very unrealistic to try to build a game tree for Mahjong, which makes tree-search based techniques for games much less helpful to Mahjong.

3. **Mahjong has a huge number of possible winning hands in various patterns**, and a more difficult hand is worth a higher score. Therefore, a professional Mahjong player needs to carefully choose what kind of winning hand to complete, in order to make a trade-off between the winning probability and the winning score of each round.

4. **Mahjong has complicated scoring rules:** a full game of Mahjong contains multiple rounds, and the final ranking of the game is determined by the accumulated round scores of all the rounds. Therefore the loss of one round does not necessarily mean that the round is played poorly: for example, a player may decide to always discard the safe tile in order not to feed other players, or to tactically lose the last round to secure top rank if they have already gained a big advantage in the previous rounds. This makes the reward system of a Mahjong AI even harder to build.

A very popular variation of Mahjong is the *Japanese Riichi Mahjong*, which is played by four players with 136 tiles of 34 different kinds. Each player's private hand consists of 13 tiles. While the basic rules of Mahjong still apply to Riichi Mahjong, it also features some additional rules. The full Riichi Mahjong rules published by the European Mahjong Association can be found on their official website [10].

Sanma is a reduced, 3-player variant of the Riichi Mahjong, with one less action (Chii) and about a suit of tiles removed. Apart from those changes, the three players in Sanma essentially play the same as in the regular 4-player game. Due to having one less suit and fewer tiles, the number of possible states and the size of the hidden information are both dramatically reduced, and hands tend to develop much faster. Consequently, players tend to play more aggressively, and valuable winning hands become more frequent.

My Part II project builds a DRL-based AI for Sanma, named Meowjong. I downloaded game logs among top human players from Tenhou.net [11], a popular global online Riichi Mahjong platform with more than 350,000 active users, and used them to pre-train 5 *convolutional neural networks* (CNNs), each corresponding to one action in Sanma (discard, Pon, Kan, Kita, and Riichi), and improved the discard model using *reinforcement learning* (RL). Instead of trying to optimise the entire multi-round game outcome like Suphx [7], Meowjong only focuses on optimising the outcome of every single round. This naturally eliminates the fourth challenge mentioned above, though the overall performance of Meowjong in the full game may be slightly compromised. I will describe the details of the design, implementation, testing and evaluation of Meowjong in the following chapters.

1.2 Related Work

Various *machine learning* (ML) approaches have been tried in works on Mahjong AI, including both *deep learning* (DL)-based and non DL-based approaches. Non DL-based Mahjong AI projects include Bakuuchi by Mizukami et al. [5] from the University of Tokyo in 2015, which was based on *Monte-Carlo simulation* and *opponent modelling*. Bakuuchi has achieved a rating of 1718 on Tenhou, which was significantly higher than that of an average human player. Another non DL-based Mahjong AI project is built by Kurita et al. [8] in 2019, which was based on multiple *Markov decision processes* as abstractions of Mahjong, and also reached the top level of Mahjong AI, demonstrated by playing directly against Bakuuchi.

There are two DL-based Mahjong AI projects that I have mainly consulted when designing and implementing Meowjong. In 2018, Gao et al. from the University of Tokyo built a Mahjong AI using CNNs [6]. They sampled 600,000 round situation data and trained the CNNs on an NVIDIA Tesla K10 GPU with 32GB memory size, and achieved a test accuracy of 68.8% for the discard action, which is 6.7% higher than the previous best test accuracy of 62.1% reported by Bakuuchi. They also evaluated the AI's strength on Tenhou, reaching a rating of 1822 after 76 matches. They did not use RL for their AI. In 2019, Li et al. from Microsoft Research Asia built a Mahjong AI called Suphx, based on DRL [7]. Suphx used CNNs for its model structures, and improved its discard model through distributed RL, with the models as policy, using the *policy gradient method*. In order to tackle the challenges of Mahjong mentioned in the previous subsection, Suphx introduced *global reward prediction* based on the use of *gated recurrent units* (GRUs), to predict the final reward of an entire game based on the information of the current and previous rounds. It also introduced *oracle guiding* to speed up the agent's improvement in RL. During online playing, Suphx also employed run-time *parametric Monte-Carlo policy adaptation* to leverage the new observations on the current round in order to perform even better. The training of each Suphx agent cost 44 GPUs (4 Titan XP for the parameter server and 40 Tesla K80 for self-play workers) and 2 days. Finally, Suphx reached a stable rank of 8.74 dan on Tenhou, which is about 2 dan higher than Bakuuchi, and is higher than 99.99% of all the officially ranked human players on Tenhou.

Chapter 2

Preparation

2.1 Starting Point

The relevant skills and knowledge I have acquired prior to this project are:

- Self-taught Python, NumPy and LaTeX skills with various project experience, including the *Part IA Scientific Computing* practical and the *Part IB Group Project*;
- relevant evaluation methods taught in the *Part IA Machine Learning and Real-world Data* course;
- basic theoretical knowledge of neural networks (NNs) taught in the *Part IB Artificial Intelligence* course;
- basic theoretical knowledge of DL and CNNs, with practical experience on the Scikit-learn and TensorFlow libraries, taught in the *Part II Data Science: principles and practice* unit.

The skills and knowledge I learnt from the Part II courses during this project are:

- Advanced ML concepts from the *Machine Learning and Bayesian Inference* course;
- advanced DL concepts from the *Deep Neural Networks* unit.

Apart from the above, I have had to learn web crawling techniques (for downloading the online game logs) and RL concepts by myself for this project. The books I mainly consulted are [12] and [13].

2.2 Basic Rules and Terminology of Riichi Mahjong

This section covers only the most relevant Sanma rules regarding this project. For more detailed Riichi Mahjong rules, please refer to Appendix B and the official rules published by European Mahjong Association [10].

2.2.1 Setup

Riichi Mahjong has 136 tiles of 34 different kinds, with four identical tiles in each kind. These 34 kinds are:

1. Three *suits*, each has tiles numbered from 1 to 9:
   - *Manzu*: 1–9m (Man), with traditional Chinese character patterns;
   - *Pinzu*: 1–9p (Pin), with dot patterns;
   - *Souzu*: 1–9s (Sou), with bamboo patterns (1s is often decorated with a bird).

   The 1's and 9's are called *terminal* tiles, and the 2–8's are called *simple* tiles.
2. Seven *honour* tiles, including four *winds* and three *dragons*:
   - Winds: East, South, West, North
   - Dragons: Haku (white), Hatsu (green), Chun (red)

Table 2.1 shows the appearances of the 34 kinds of tiles. In Sanma, 2–8m are removed, leaving only 108 tiles of 27 different kinds. At the start of each round, the tiles are shuffled and each player receives 13 tiles as their starting hand. The seating order of each player is represented by East, South or West, which is called the player's *seat wind*. The East player becomes the dealer for that round.

	Numbers								
	1	2	3	4	5	6	7	8	9
Manzu									
Pinzu									
Souzu									
Honours	Winds				Dragons				
	East	South	West	North	Haku	Hatsu	Chun		
									

Table 2.1: Mahjong tiles (graphical resources adapted from a GitHub repository [14]).

2.2.2 Actions

Other than the normal discard action, a player can perform special actions in legal situations by making *tile calls*. Knowing when to appropriately make a tile call is one of the central strategies in Riichi Mahjong. The available tile calls in Sanma are as follows:

- *Pon*: a player can claim an immediately discarded tile from any other player to form a *triplet* of the same kind (also called *Koutsu*). This action may cause a player's turn to be skipped. The player needs to discard a tile after calling Pon.

- *Kan*: a player can call Kan to form a *quad* of the same kind (also called *Kantsu*). The player draws another tile and continues their turn after a Kan, and a *Dora indicator* is also revealed.
- *Kita*: a player can put a North tile to the side of their hand, and it would be counted as *Dora*, called the *Nukidora*. Also, the player is awarded an extra draw and can continue their turn. Kita is a Sanma-only tile call.
- *Riichi*: a player can pay a 1,000 point deposit and declare a ready hand, which means they only wait for one more tile to win. After declaring Riichi, the player's hand cannot be changed anymore, and the player must discard any tile they have drawn until the round ends. The Riichi player bares certain risks, but has a big chance to win and can gain a larger score if they win.

2.2.3 End of a Round

Winning

There are three types of winning tile combinations:

- The most common standard combination: $x(\text{ABC}) + (4 - x)(\text{DDD}) + \text{EE}$ where $0 \leq x \leq 4$. Here ABC represents *sequences* (also called *Shuntsu*), DDD represents triplets, and EE represents a pair of identical tiles. Melded quads count as triplets.
- *Seven Pairs*, also known as *Chiitoitsu*: a concealed hand with seven different pairs.
- *Thirteen Orphans*, also known as *Kokushi Musou*: concealed hand with one of each of the 13 different terminal and honour tiles plus one extra terminal or honour tile.

There are two types of winning: winning from another player (*Ron*), or winning from self-draw (*Tsumo*). In the case of Ron, the feeding player pays the winning player by the amount of the winning player's hand score. In the case of Tsumo, the other two players split the winning player's hand score and pay the winning player together. The dealer wins 50% more, but when a non-dealer player wins by Tsumo, the dealer also pays twice compared to the other paying player.

Exhaustive Draws

A draw is also called *Ryuukyoku* in Riichi Mahjong's term. When all the tiles from the live wall are drawn with no player wins, the round ends with in an exhaustive draw. The player(s) in *Tenpai* (a ready hand) receive points from those who are not.

Abortive Draws

At the beginning of a round, if a player has 9 or more different kinds of terminal or honour tiles, the player can choose to abort and restart the round, called *Kyuushu Kyuuhai*. This is to prevent a player from being in a very disadvantaged start by receiving a very bad start hand. If the player wishes, they can also go for Kokushi Musou instead of calling Kyuushu Kyuuhai. There are also other types of abortive draw situations, which will be covered in Appendix B.

2.2.4 Scoring

Yaku

Yakus are specific hand patterns or conditions needed for a hand, in order for it to count as a winning hand. Every winning hand must include at least one Yaku. Each Yaku is worth a specific *Han* value, and various Yaku may be combined together in one hand. The full list of Yakus in Riichi Mahjong can be found in Appendix B.

Han and Fu

Riichi Mahjong features a very complex scoring system. Han and *Fu* are the two scoring factors of a winning hand. A hand's Han value is calculated by summing together the Han values of all of its Yakus, plus the Han value for each Dora in the hand. Fu takes into consideration the hand's meld tiles, waiting patterns and winning method.

Dora

A Dora is a bonus tile that adds 1 Han to a winning hand. However, Doras in a hand cannot be counted as Yakus, and the hand still requires at least one Yaku. There are different types of Doras: *Red Dora* (also known as *Akadora*), Dora from indicator, *Kandora*, *Uradora*, and Nukidora, which will be further explained in Appendix B.

2.3 Deep Neural Networks

2.3.1 Overview

Before diving into the DNNs, it is crucial to understand the basics of ML, in particular, supervised learning (SL) for classification. SL is a subset of ML which involves learning from a set of labelled training data. Consider a *training sequence*

$$\mathbf{s}^T = \begin{bmatrix} (\mathbf{x}^{(1)}, y^{(1)}) & (\mathbf{x}^{(2)}, y^{(2)}) & \cdots & (\mathbf{x}^{(m)}, y^{(m)}) \end{bmatrix}$$

where $\mathbf{x}^{(i)}$ is a vector of *features*, and $y^{(i)} \in \mathcal{Y}$ is the *label* of $\mathbf{x}^{(i)}$, the known output for each input feature vector, a SL algorithm tries to learn a *hypothesis* $h : \mathbb{R}^n \to \mathcal{Y}$, which can be used to predict the result $\hat{y}$ from a given input *feature vector* $\mathbf{x}$.

If $\mathcal{Y} = \mathbb{R}$ or some other set such that the outputs are continuous, then this becomes a *regression* problem; whereas if $\mathcal{Y} = \{c_1, c_2, \cdots, c_K\}$, which means the outputs are discrete, then this becomes a *classification* problem. For building models that predict actions for Mahjong, we are interested in classification.

2.3.2 Perceptron

Perceptrons are elementary units in NNs. A perceptron receives an input feature vector $\mathbf{x}$ and applies it to a linear function defined by a *weight vector* $\mathbf{w}$ and a *bias* b. The result is then modified by a non-linear *activation function* σ, as described by the following formula:

$$a = \sigma(z) = \sigma(\mathbf{w}^T\mathbf{x} + b) = \sigma\left(b + \sum_{i=1}^{n} w_i x_i\right)$$

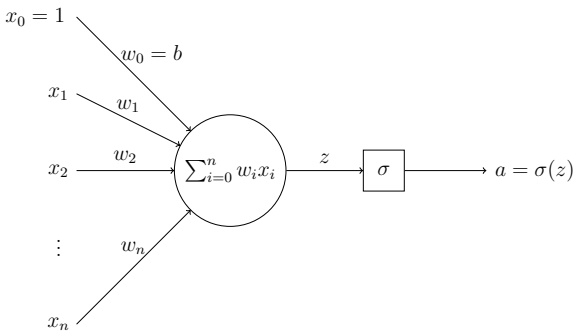
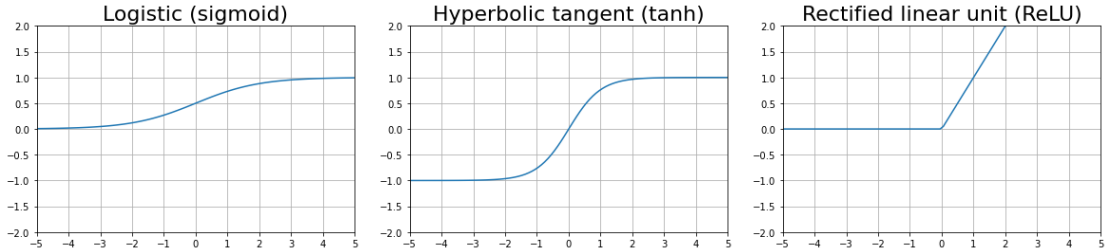


Figure 2.1: Structure of a perceptron

Figure 2.1 illustrates the structure of a perceptron. The non-linear activation function is used to allow an NN to compute non-trivial problems using only a small number of perceptrons or layers of perceptrons, as otherwise the composition of two linear functions is still linear. Some common activation functions are defined and plotted in Figure 2.2.



(a) $\sigma(x) = \dfrac{1}{1+e^{-x}}$ (b) $\sigma(x) = \dfrac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ (c) $\sigma(x) = \max(0, x)$

Figure 2.2: Common activation functions

2.3.3 Multilayer Perceptron

A *multilayer perceptron* (MLP) is a simple example of a DNN. It is a *feedforward* NN with multiple *hidden layers* between the input and output layers. MLPs are called feedforward because information flows in only one direction: from the input perceptrons, through the hidden perceptrons, and finally to the output perceptrons. This process is also called *forward propagation*. There are no cycles in an MLP, and hence the information never goes backward. An example MLP structure is illustrated in Figure 2.3.

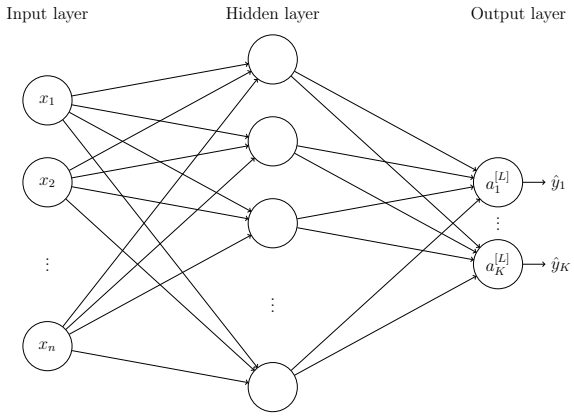


Figure 2.3: Example structure of a multilayer perceptron

2.3.4 Training

The decision making problems of Mahjong actions can be abstracted as classification problems of two or multiple classes. In these cases, in order to obtain $\mathbf{w}_{\text{opt}}$, the NNs should adjust their weights and aim to minimise the *categorical cross-entropy loss function*:

$$L(\mathbf{w}) = -\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \cdot \log \hat{y}_k^{(i)}$$

The NNs can do so by using *gradient descent*: start with a random $\mathbf{w}_0$, and iteratively update it by a small amount in the direction of the negative gradient of $L(\mathbf{w})$, as described by the following formula:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left.\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}}\right|_{\mathbf{w}_t}$$

where

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial L(\mathbf{w})}{\partial w_1} & \frac{\partial L(\mathbf{w})}{\partial w_2} & \cdots & \frac{\partial L(\mathbf{w})}{\partial w_W} \end{bmatrix}^T$$

and η is a small positive value, known as the *learning rate*. The learning rate must be chosen carefully: if η is too small, the NNs will be extremely slow at converging, and may become stuck at some local minimum; if η is too large, the NNs will overshoot and keep stepping over the minimum.

The derivative of the loss function with respect to the weights, $\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}}$, can be calculated using the *back-propagation algorithm*. The reason that the categorical cross-entropy loss function is used as the loss function, as well as the detailed explanation of back-propagation, are elaborated in Appendix C.

2.4 Convolutional Neural Networks

A CNN is a type of DNN that uses *convolution* in place of general matrix multiplication in at least one of their layers. A typical CNN often includes convolutional layers, pooling layers and fully-connected layers.

2.4.1 Convolutional Layers

The convolutional layer is the core building block of a CNN. The layer's weights consist of a set of trainable *kernels*, each of which receives input from only a restricted area if the previous layer, called the *receptive field*. During the forward pass, each kernel is convolved across the width and height of the input, using a convolution-related function called *cross-correlation*:

$$(\mathbf{I} * \mathbf{K})_{i,j} = \sum_m \sum_n I_{i+m,j+n} K_{m,n}$$

This effectively computes the dot product between the kernel entries and the input, producing a 2D *feature map* of that kernel. As a result, the CNN learns kernels that activate when they detect specific types of feature at some spatial position in the input.

Another two common hyperparameters in CNNs practice are *stride* and *padding*, which are used to control the size of the output: the stride size controls how many units the kernels are translated at each time per output. and the padding size controls how many rows/columns of zeros should be padded on the border of the input. Figure 2.4 shows an example convolutional layer with stride and padding.

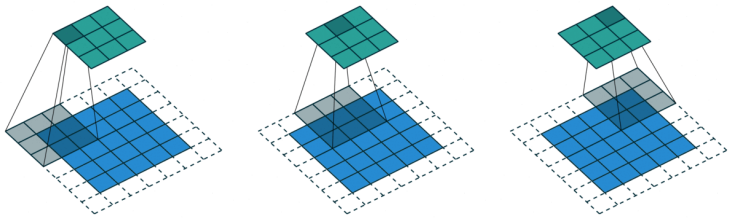


Figure 2.4: Cross-correlation with $N = (5, 5)$, $K = (3, 3)$, $S = (2, 2)$, $P = (1, 1)$

2.4.2 Pooling Layers

Another important concept of CNNs is *pooling*. Pooling layers reduce the size of data by combining the output of perceptron clusters at one layer into a single perceptron in the next layer. *Max pooling* is the most popular implementation of pooling, which partitions the input data into clusters and outputs the maximum for each cluster, as shown in Figure 2.5. Pooling exploits the property that the exact location of a feature is less important than its approximate relative location, and can reduce the number of parameters, memory usage and the amount of computation in the CNNs, and hence to also control *overfitting*.

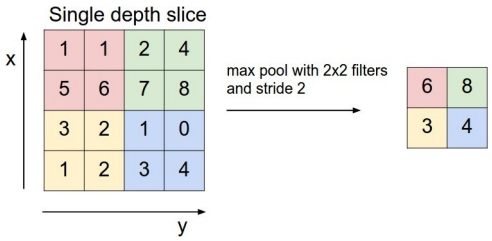


Figure 2.5: Max pooling with filter size $= 2 \times 2$ and stride $= 2$

2.4.3 Fully-Connected Layers

After several convolutional and pooling layers, the final feature map is flatten, and the final classification is done via fully-connected layers. In a fully-connected layer, every perceptron is connected to every perceptron in the previous layer, which is the same as a traditional MLP. Figure 2.6 shows a typical CNN structure used for image recognition.

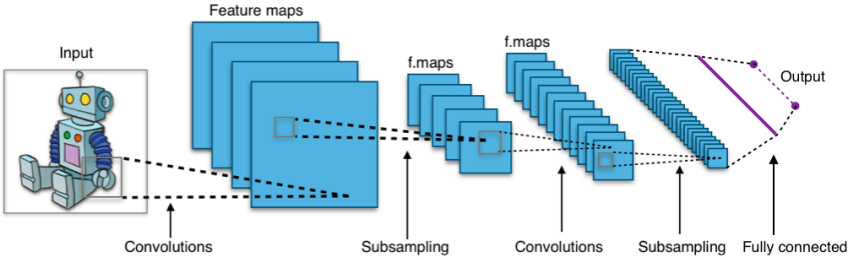


Figure 2.6: A Typical CNN Structure, from Wikipedia [15]

2.5 Reinforcement Learning

2.5.1 Markov Decision Processes

RL is a process where the *agent* learns by interacting with the *environment* in order to maximise a cumulative *reward*. *Finite Markov decision processes* (MDPs) are a classical formalisation of sequential decision making, where actions affect not just immediate rewards, but also subsequent *states*, and through those, the future rewards. In a MDP, the agent and the environment interact at a sequence of discrete time steps $t = 0, 1, 2, \cdots$. At each time step t, the agent receives some representation of the environment's state $S_t \in \mathcal{S}$, and on that basis, selects an *action* $A_t \in \mathcal{A}$. One time step later, as a consequence of its action, the agent receives a numerical reward $R_{t+1} \in \mathcal{R}$, and finds itself in a new state S_{t+1}, as shown in Figure 2.7:

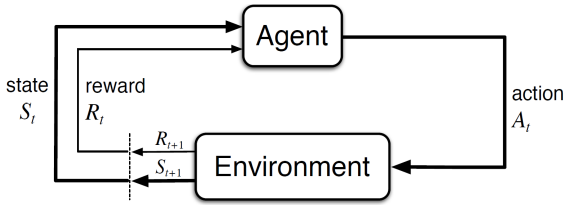


Figure 2.7: The agent-environment interaction in a MDP [13]

In most cases, the agent-environment interaction can be naturally broken down into subsequences, called *episodes*, such as the rounds of a game in Mahjong. Each episode ends in a special state called the *terminal state*, followed by a reset to a standard starting state, or a sample from a standard distribution of starting states. The agent's goal is to maximise the cumulative reward. More specifically, given the sequence of rewards the agent received from time t to the time of termination T, $R_{t+1}, R_{t+2}, \cdots, R_T$, the agent should seek to maximise the *expected return*, G_t, which is a function of the reward sequence. The MDPs use a *discounted return* to trade off the immediate and delayed reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

where $0 \leq \gamma \leq 1$ is a hyperparameter called the *discount rate*. In a round of Mahjong, the reward of each action is not immediately received; instead, the cumulative reward is received one-off at the end of the round, as the score of the agent. Therefore, we need a way to estimate the reward for each action, and the returns at successive time steps hence become important:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T = R_{t+1} + \gamma G_{t+1}$$

The expected return at the termination state, G_T, can be defined as 0.

A *policy* $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a mapping from states to the probabilities of selecting each possible action, which defines the behaviour of the agent:

$$\pi(a|s) = \Pr(A_t = a | S_t = s)$$

We can also define the *state-value function for policy* π, $v_\pi(s)$, and the *action-value function for policy* π, $q_\pi(s,a)$, as measurements of "how good" a state or an action is:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$
$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

2.5.2 Policy Gradient Methods

Now that a hypothesis $h_\mathbf{w}$ has been pre-trained through DL, it can serve as a *parametrised policy* to be improved through RL. Thus the parametrised policy can be written as:

$$\pi(a|s,\mathbf{w}) = \Pr(A_t = a|S_t = s, \mathbf{w}_t = \mathbf{w})$$

Policy gradient methods are methods for learning the policy parameters based on the gradient of some scalar performance measure $J(\mathbf{w})$ with respect to the policy parameters. These methods seek to maximise $J(\mathbf{w})$, so they update $\mathbf{w}$ by *gradient ascent* in J:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta\nabla_\mathbf{w} J(\mathbf{w}_t)$$

In the episodic case, we can define the performance measure J as the value of the start state of the episode. Assuming the episode starts in some initial state s_0, we can define $J(\mathbf{w})$ and calculate its gradient:

$$\begin{aligned}
J(\mathbf{w}) = v_{\pi_\mathbf{w}}(s_0) &\propto \mathbb{E}_\pi\left[\sum_a q_\pi(S_t,a)\pi(a|S_t,\mathbf{w})\right] \\
\nabla_\mathbf{w} J(\mathbf{w}) &\propto \mathbb{E}_\pi\left[\sum_a \pi(a|S_t,\mathbf{w}) q_\pi(S_t,a)\frac{\nabla_\mathbf{w}\pi(a|S_t,\mathbf{w})}{\pi(a|S_t,\mathbf{w})}\right] \\
&= \mathbb{E}_\pi\left[q_\pi(S_t,A_t)\frac{\nabla_\mathbf{w}\pi(A_t|S_t,\mathbf{w})}{\pi(A_t|S_t,\mathbf{w})}\right] \\
&= \mathbb{E}_\pi\left[G_t\nabla_\mathbf{w}\log\pi(A_t|S_t,\mathbf{w})\right]
\end{aligned}$$

This yields the *Monte-Carlo Policy Gradient* update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta\gamma^t G_t\nabla_\mathbf{w}\log\pi(A_t|S_t,\mathbf{w}_t)$$

2.6 Software Engineering Practices

2.6.1 Requirements Analysis

The deliverables required to meet the success criteria, along with their dependencies, importances and difficulties, are summarised in the Table 2.2.

Category	Number	Deliverable	Dependency	Difficulty
Data processing	1	Donwload game logs from Tenhou	None	Medium
	2	Convert game logs to dataset tensors	1	Hard
	3	Prepare standardised dataset	2	Simple
Training	4	Build CNN models	None	Medium
	5	Pre-train and tune the models	2, 3, 4	Hard
	6	Improve discard model using RL	5, 8	Hard
Evaluation	7	Hand calculator	None	Hard
	8	Game simulator	7	Hard
	9	Random agent	None	Simple
	10	SL and SL-scaled agents	5	Medium
	11	RL agent	6	Medium
	12	Random agents vs. Random agents	8, 9	Simple
	13	SL/SL-scaled agents vs. Random agents	8, 9, 10	Medium
	14	RL agents vs. SL agents	8, 9, 11	Medium

Table 2.2: Deliverables required for this project

2.6.2 Choice of Tools

The programming language used for this project is Python due to its complete support for ML, and Prolog was used for simple logical implementation for hand calculation. The ML libraries used for Meowjong were Scikit-learn [16] and TensorFlow [17]. NumPy [18] and Pandas [19] were used for data manipulation. PySwip [20] was used for running SWI-Prolog queries in Python. Requests [21] was used for building the web crawler to download game logs from Tenhou. Pillow [22] was used for converting the examples in the dataset into images for intermediate storage, and Joblib [23] was used for serialising large dataset tensors into files. Last but not least, SciPy [24] and Matplotlib [25] were used for evaluation and visualisations, and this dissertation was written using LaTeX. The versions and licences of the used libraries are listed in Table 2.3, and care has been taken to make sure the licence requirements are met. The code base of Meowjong, including the dissertation, is version controlled using GitHub and has two copies, one in my personal laptop, and the other in my personal directory at the Cambridge Service for Data Driven Discovery (CSD3). The image-encoded datasets are stored in both my personal laptop and CSD3, and the tensors files are stored only on CSD3, where there is enough storage.

2.6.3 Development Practices

I followed the *Iterative Development Model* throughout the implementation of Meowjong, which enabled me to divide the whole project into modules as listed in Section 2.6.1, and build and test the modules individually before integrating them together. This allowed easier management of the workload, and helped me to make sure each module functioned properly before integrating all the modules. The modules were implemented in a order following their dependencies, and flexibilities were allowed for the independent modules.

Library	Version	Licence
Joblib	1.0.0	BSD 3-Clause Licence
Matplotlib	3.4.1	PSF Licence
NumPy	1.18.5	BSD 3-Clause Licence
Pandas	1.2.0	BSD 3-Clause Licence
Pillow	8.1.0	HPND Licence
PySwip	0.2.10	MIT Licence
Requests	2.25.1	Apache-2.0 licence
Scikit-learn	0.24.0	BSD 3-Clause Licence
SciPy	1.6.3	BSD 3-Clause Licence
TensorFlow	2.3.0	Apache-2.0 licence

Table 2.3: Main libraries used for this project

Development and unit testing were carried out on my personal laptop (Microsoft Surface Book 2, with Intel Core i7-8650U CPU at 1.90GHz, NVIDIA GeForce GTX 1060 GPU with 6GB memory, 16GB RAM, 256GB SSD, and Windows 10 with Ubuntu on Windows Subsystem for Linux (WSL)). Large-scale training of the models was carried out on CSD3 Wilkes2 GPU clusters (each node containing 1x Intel Xeon E5-2650 v4 2.2GHz 12-core processor, 4x Nvidia P100 GPUs with 16GB memory, and 96GB RAM, with theoretical peak performance at 19.61 TFlops/s) and Peta4-Skylake CPU clusters (each node containing 2x Intel Xeon Skylake 6142 2.6GHz 16-core processors, and 384GB RAM, with theoretical peak performance at 2.662 TFlop/s).

2.7 Machine Learning Practices

2.7.1 Splitting Data

Once a model is trained, it is crucial to assess its performance on data that the model has never seen during training. A generally accepted practice is to split the training sequence $\mathbf{s}$ into a *training set* $\mathbf{s}_{\text{train}}$ for training the models, a *development set* (also known as a *validation set*) $\mathbf{s}_{\text{dev}}$ for continuous evaluation of the models and hyperparameter tuning, and a *test set* $\mathbf{s}_{\text{test}}$ for realistic evaluation once the training and hyperparameter tuning is finished, as shown in Figure 2.8:

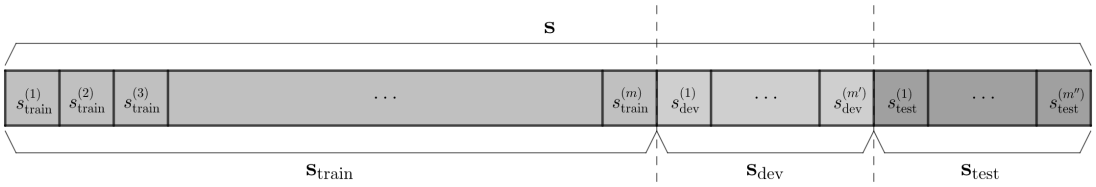


Figure 2.8: Train-dev-test split of the dataset

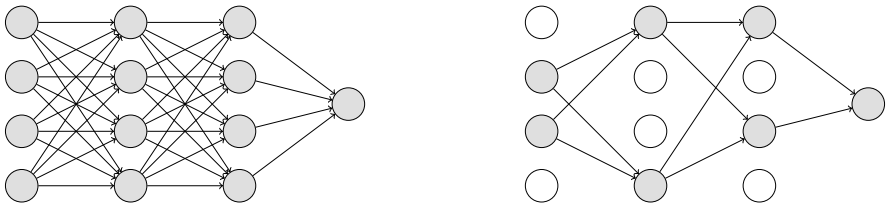
In the case of *unbalanced data* (i.e., the distribution of the classes within $\mathbf{s}$ is not uniform), *stratified sampling* should be applied to make sure the distribution of the classes is kept the same in every split, and is equal to the original distribution of the classes in $\mathbf{s}$.

2.7.2 Regularisations and Optimisations

Regularisations are techniques that prevent overfitting, and *optimisations* are strategies for speeding up convergence and improve the NNs' training performance. These include:

Dropout

Dropout [26] provides a computationally inexpensive but powerful regularisation. In each pass during training, it randomly ignores a proportion of perceptrons in a NN, as shown in Figure 2.9. This not only prevents complex co-adaptations between perceptrons on the training data, but also trains an ensemble of sub-NNs of the original NN. Dropout is only used during the training of a model, and the entire model should be used for evaluation.

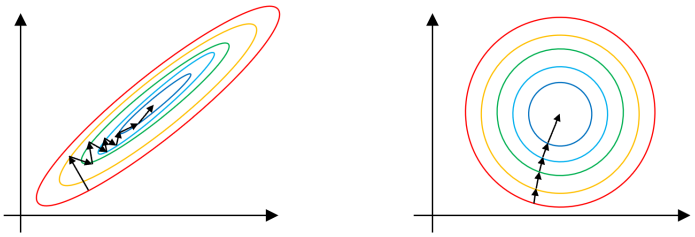


(a) Standard NN without dropout (b) After applying dropout

Figure 2.9: A dropout NN model with dropout rate = 0.5

Data standardisation

Since the range of values or raw data may vary widely, the loss function can be dominated by some large-valued features, and hence not work properly. A good practice is to standardise the values of each feature in the data to have zero-mean and unit-variance. The means and standard deviations of each feature collected during training time can be directly used at test time, under the assumption that both the training set and the test set are sampled from the same distribution. Data standardisation can also make gradient descent converge much faster by normalising the parameters in all axes, as shown in Figure 2.10.



(a) Unstandardised (b) Standardised

Figure 2.10: Gradient descent with vs. without data standardisation

Batch normalisation

Batch normalisation [27] makes the training of DNNs faster and more stable through normalisation of minibatches of activations of the layers by re-centering and re-scaling. Its arithmetic is identical to the aforementioned data standardisation, except that in this case, minibatches of activations of the layers are normalised instead of the input.

Chapter 3

Implementation

3.1 Game Environment

3.1.1 Hand Calculation

Hand calculation can be done in the following 3 steps:

Step 1: Divide the hand into a legal combination

As described in Section 2.2, there are 3 types of winning combinations: standard $x(\text{ABC}) + (4-x)(\text{DDD}) + \text{EE}$ combination, Seven Pairs and Kokushi Musou. The Seven Pairs and Kokushi Musou combinations can be pattern-matched by counting the tile indices and checking if the relevant conditions are met. The standard combinations can be pattern-matched in a recursive *generate-and-test* style, which can be implemented efficiently using Prolog: firstly, a pair is matched and removed from the hand, making the number of remaining tiles a multiple of 3; then, match and remove sequences/triplets one by one, until the entire hand is pattern matched or found to be illegal. Since tiles in a pair/sequence/triplet must all be in the same suit, I can make the generate-and-testing more efficient by first splitting the hand into different suits and then matching each suit separately. I also used a dynamic predicate `traversed/1` to mark the tested hand, in order to prevent excessive searching. This leads to the following Prolog code:

```
% Define sequences and triplets
koutsu([A,A,A]).  % Koutsu means triplet
shuntsu([A,B,C]) :- B is A + 1, C is A + 2.  % Shuntsu means sequence
... % Permutations omitted
% Mentsu is the general term for sequences and triplets
mentsu(Triple) :- koutsu(Triple).
mentsu(Triple) :- shuntsu(Triple).

:- dynamic traversed/1.

% Find all legal combinations
mentsu_combination([], []).          % Base case
```

```
mentsu_combination(Hand, [Triple|Rest]) :-
    subset(3, Hand, Triple),          % Find a sublist of size 3 from Hand
    mentsu(Triple),                   % Test if Triple is a Mentsu
    remove_list(Triple, Hand, Rem),   % Remove Triple from Hand
    mentsu_combination(Rem, Rest),    % Match the remaining hand
    not(traversed([Triple|Rest])),    % Update traversed/1
    assertz(traversed([Triple|Rest])).
```

According to Mahjong rules, if there are multiple ways to divide the hand, it should be divided in the way that maximises the score. Prolog queries are run in Python modules using PySwip, as shown in the code block below. Note that `retractall(traversed(_))` must be used after each query to clear the stored facts in `traversed/1`.

```
def find_mentsu_combinations(hand):
    combinations = []
    prolog = pyswip.Prolog()
    prolog.consult('hand_divider.pl')
    query = 'mentsu_combination(' + hand + ', Combination)'
    for solution in prolog.query(query):
        combination = sorted(solution['Combination'])
        if combination not in combinations:
            combinations.append(combination)
            prolog.retractall('traversed(_)')
    return combinations
```

Step 2: Calculate Han and Fu values based on the combination

After a hand is divided, it is checked against all Yakus in Riichi Mahjong. If the hand has no Yaku, a `NoYakuError` is raised. I built a class for each Yaku, inherited from a common base class `Yaku`. Each Yaku class has a `set_attributes` method, which sets the Yaku's basic information such as names and Han values, and a `is_condition_met` method that checks whether a hand satisfies the Yaku's requirements. I have also built a `HandConfig` class that deals with Yakus that are unrelated to the hand tiles, such as Riichi. After Yakus are checked, the hand's Han value is calculated as the sum of Han values of all Yakus, plus the number of Doras in the hand. I integrated the Han calculation inside a `Han` class, and built a `Fu` class that calculates the Fu value of a hand, based on the rules described in Appendix B.

Step 3: Calculate hand score from the Han and Fu values

The score of a hand is calculated by combining the Han and Fu values into a specific formula, as described in Appendix B. In practice, players tend to refer to a scoring table, instead of doing the tedious calculations at every time. However, in code implementation, it is much easier to apply the scoring formula, rather than hard-coding the entire scoring table. I implemented the score calculation in the `Score` class.

3.1.2 The Game Simulator

Unlike the originally proposed procedural, agent-oriented decision flow in my project proposal and in Li et al. [7], I finally adopted a functional agent design which only performs model predictions when called, so that the game flow could be centralised to the game simulator, which could be implemented more easily. The game simulator also oversees the dealing of the starting hands and tiles, and controls the agents' turns as well as calling the correct model predictions at the appropriate situations. The workflow of the game simulator is demonstrated by the flow diagram in Figure 3.1. After a round ends, the game simulator is also responsible for calculating the round scores/penalties of each agent, by calling the hand calculation module described in Section 3.1.1.

One crucial functionality that the game simulator relies upon is Riichi checking. While the win checking can be performed by pattern matching as described in Section 3.1.1, now with a contributing tile substituted by a non-contributing tile in the Riichi case, the number of possible patterns is significantly increased:

- Standard 4 Mentsu + 1 pair combination: a Riichi hand (including the tile to be discarded to accomplish Riichi) may have either
  - 4 Mentsu and 2 separate singles;
  - 4 Mentsu and a *Taatsu* (2 tiles forming a potential sequence);
  - 3 Mentsu, 2 pairs, and a single; or
  - 3 Mentsu, a pair, a Taatsu, and a single.
- Seven Pairs combination: a Riichi hand may have either
  - 6 pairs and 2 singles; or
  - 5 pairs, a triplet and a single.
- Kokushi Musou combination: a Riichi hand may have either
  - 13 different terminal/honour tiles and a non-terminal/honour tile;
  - 11 single terminal/honour tiles, a pair of terminal/honour tiles, and a single non-terminal/honour tile;
  - 11 single terminal/honour tiles and a triplet of terminal/honour tiles; or
  - 10 single terminal/honour tiles and 2 pairs of terminal/honour tiles.

Similar to the win checking, the Seven Pairs and Kokushi Musou patterns can be matched by tile indices counting, and the standard patterns can be matched by splitting into suits and applying recursive generate-and-testing aided by Prolog. However, due to the existence of doubles and singles, the base cases of the pattern matching become much more sophisticated: for a suit with n tiles,

1. Case $n = 0$ or 1: these trivially hold;
2. Case $n = 2$: this can be either a pair (2), or two singles (1 + 1);

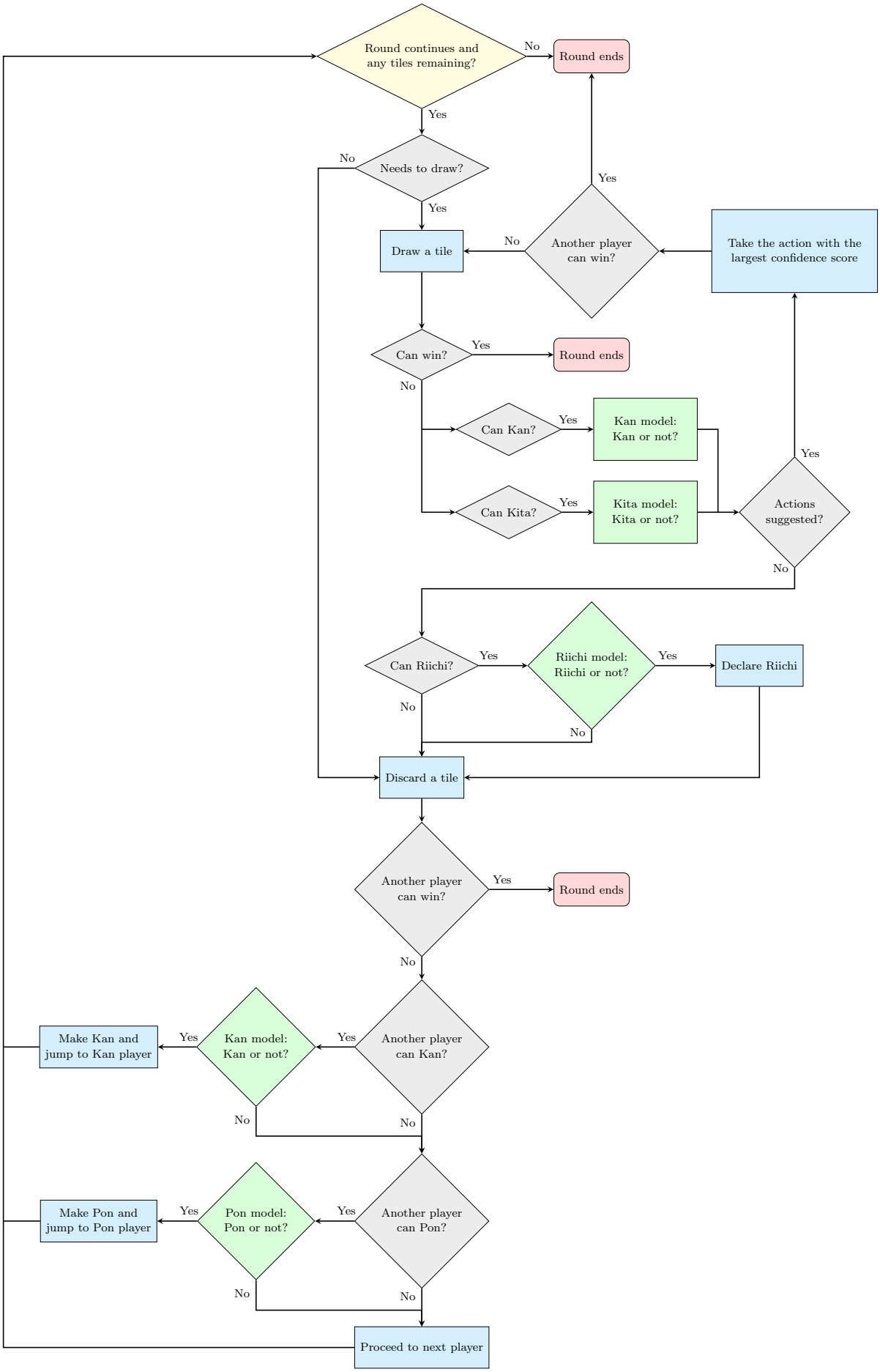


Figure 3.1: Workflow of the game simulator

3. Case $n = 3$: this can be either a Mentsu (3), or a double plus a single (2 + 1);

4. Case $n = 4$: this can be either a Mentsu plus a single (3 + 1), or two doubles including at least one pair (2 + 2);

5. Case $n = 5$: this can be either a Mentsu plus a double (3 + 2), or the case $n = 4$ plus a single (4 + 1, also known as 3 + 1 + 1 and 2 + 2 + 1);

6. Case $n \geq 6$: this can be pattern-matched by matching Mentsu recursively until n is reduced to the above 5 base cases.

This leads to the following Prolog code:

```
% Define pairs and Taatsu
pair([A,A]).
taatsu([A,B]) :- B < 27, B is A + 1.  % Honours cannot form Taatsu
taatsu([A,C]) :- C < 27, C is A + 2.
... % Permutations omitted

% A different dynamic predicate is used to prevent interference
:- dynamic visited/1.

% 0 = 0, 1 = 1
riichi_combination([],[]) :- !.
riichi_combination([X],[[X]]) :- !.

% 2 = 2 or 1 + 1
riichi_combination([A,A],[[A,A]]) :- !.
riichi_combination([A,B],[[A,B]]) :- taatsu([A,B]),!.
riichi_combination([X,Y],[[X],[Y]]) :- !.

% 3 = 3 or 2 + 1 (1 + 1 + 1 is impossible)
riichi_combination([A,B,C],[[A,B,C]]) :- mentsu([A,B,C]),!.
riichi_combination([A,B,X],[[A,B],[X]]) :- pair([A,B]),!.
... % Permutations omitted
riichi_combination([A,B,X],[[A,B],[X]]) :- taatsu([A,B]),!.
... % Permutations omitted

% 4 = 3 + 1 or 2 + 2 (2 + 1 + 1 is impossible)
riichi_combination([A,B,C,X],[[A,B,C],[X]]) :- mentsu([A,B,C]),!.
... % Permutations omitted
% in the 2 + 2 case, there must be at least one pair
riichi_combination([A,B,C,D],[[A,B],[C,D]]):-pair([A,B]),pair([C,D]),!.
... % Permutations omitted
riichi_combination([A,B,C,D],[[A,B],[C,D]]):-pair([A,B]),taatsu([C,D]),!.
... % Permutations omitted
```

```
% 5 = 3 + 2 or 4 + 1
riichi_combination([A,B,C,X,Y],[[A,B,C],[X,Y]]) :-
    mentsu([A,B,C]), pair([X,Y]), !.
... % Permutations omitted
riichi_combination([A,B,C,X,Y],[[A,B,C],[X,Y]]) :-
    mentsu([A,B,C]), taatsu([X,Y]), !.
... % Permutations omitted
riichi_combination([A,B,C,D,X],[[X]|Quad]) :-
    riichi_combination([A,B,C,D],Quad), !.
... % Permutations omitted

% General case
riichi_combination(Hand, [Triple|Rest]) :-
    subset(3, Hand, Triple),
    mentsu(Triple),
    remove_list(Triple, Hand, Rem),
    riichi_combination(Rem, Rest),
    not(visited([Triple|Rest])),
    assertz(visited([Triple|Rest])).
```

3.2 Data Preparation

3.2.1 Features and Data Structure Design

Unlike other board games such as Chess and Go, the information available to each player in Mahjong, as shown in Figure 3.2, is not naturally in the format of images. Therefore, the observable information must be carefully encoded in order to be digested by the CNNs.

Figure 3.2: Example of a state

Since there are 34 different kinds of tiles in Mahjong, I used multiple 34×1 channels to represent a state. Although 2m–8m are not included in Sanma, which leaves only 27 of the 34 different kinds of tiles to be used, I still included all the tiles and left the excluded tiles blank, in preparation for the possible future extension of transferring this project to the full, 4-player Mahjong. The mapping between the tiles and their corresponding indices is shown in Table 3.1. The three red Doras 5m, 5p, and 5s are given separate index representations. Those indices are used when there is a need to distinguish them from the ordinary 5's, in representing the tiles as a list of indices. The red Doras are not given separate spaces in the channels, because they are, after all, still the same tiles as the ordinary 5's, except that they carry an extra Han in the score calculation.

Tiles	Corresponding indices
Manzu, i.e., 1m–9m	0–8
Pinzu, i.e., 1p–9p	9–17
Souzu, i.e., 1s–9s	18–26
Winds, i.e., East, South, West, and North	27–30
Honours, i.e., Haku, Hatsu, and Chun	31–33
Red Doras 5m, 5p, 5s	34, 35, 36

Table 3.1: Tiles and their corresponding index representations

To simulate the real in-game environment and maximise Meowjong's performance, all observable information should be taken into account. Therefore, I used 366 channels in total to represent 22 features, as listed in Table 3.2. I did not include player ratings into the input features, because it is only a platform-specific feature and is not originated in Mahjong. I also did not include the number of remaining tiles, because it can be calculated from the number of discarded tiles. For melds and Riichi status, I included not only the melded tiles and the Riichi status, but also the turn numbers of the meld/Riichi calls. Besides, although the spaces for the fourth player are not needed for Sanma, they are still created for the possible future extension to the full, 4-player Mahjong.

Feature	Number of channels
Target tile	1
Self private tiles	4
Red Doras in hand	1
Self melds	$(4 + 5) \times 4 = 36$
Self Kitas	4
Self discards	30
Dora indicators	5
Other players' Riichi status	$(1 + 5) \times 3 = 18$
Scores	$11 \times 4 = 44$
Round (Kyoku) number	4
Repeats count (i.e., Honba number)	4
Deposit count	4
Self wind	1
Other players' melds, Kitas, and discards	$(36 + 4 + 30) \times 3 = 210$
Total	366

Table 3.2: Input features for the models

The included features can be divided into two categories:

1. Tile features: these are features involving sets or sequences of tiles, such as private tiles, melds, etc. Tile features can be encoded by setting the corresponding tile indices to 1 and leaving the rest to 0, as, for example, shown in Figure 3.3:

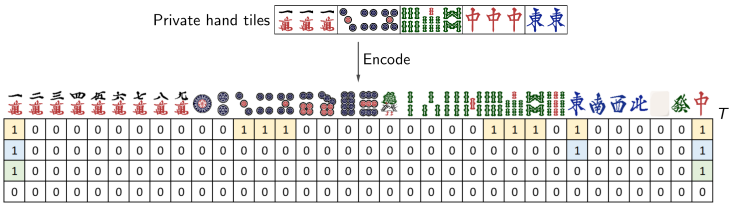
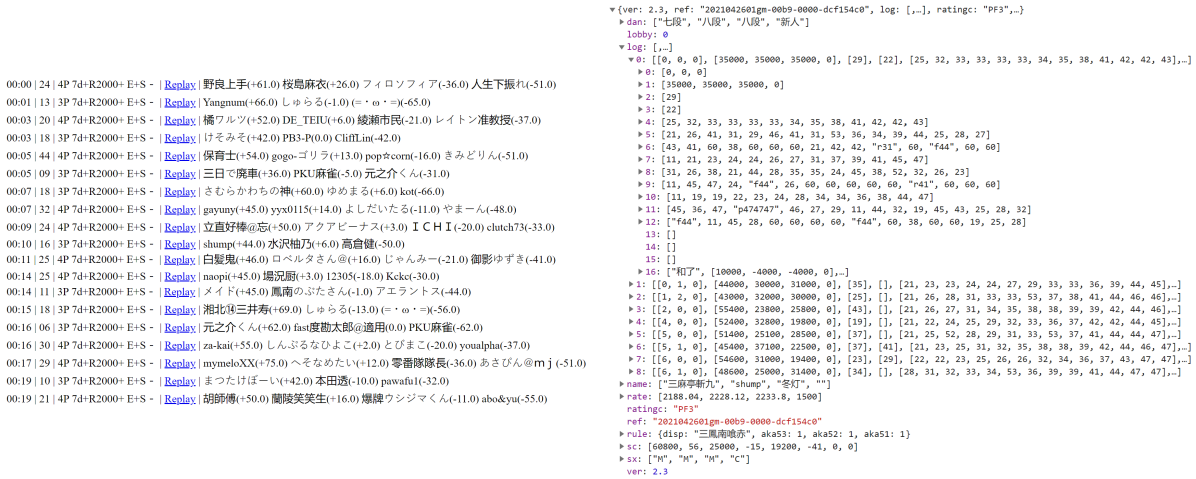


Figure 3.3: Encoding of a private hand using 4 channels

2. Numerical features: these are numerical-based, such as player scores. Numerical features can be encoded into multiple channels, each being either all 0's or all 1's.

3.2.2 Online Game Logs Collection

Tenhou game logs are archived by year in ZIP files on its website.[1] Each ZIP file contains multiple HTML files, each with a number of game records, including URLs to the game logs, as shown in Figure 3.4a. I downloaded the ZIP files from the 'Houou' (pheonix in Japanese) table from 2009 to 2020, collecting 4,241 HTML files in total. The 'Houou' table is only open for the top 0.1% ranked player, so its game records can be considered as of high quality. From those HTML files, I extracted the URLs of 603,416 Sanma games between February 2009 and September 2020. Each URL contains a JSON object of the log of a game, which then may consists of several rounds, as shown in Figure 3.4b. I wrote a web crawler to download all the 603,416 JSON files. Since it took about 1 second to download each JSON object, I used multi-processing and ran 8 processes in parallel, so that the entire downloading was finished in approximately 22 hours. I further divided the downloaded JSON objects and obtained logs for 5,434,685 rounds of Sanma game.



(a) HTML file (b) JSON object

Figure 3.4: Structures of Tenhou game logs

[1] `https://tenhou.net/sc/raw/`

3.2.3 Dataset Generation

After downloading and extracting all the game records, I sampled 50,000 rounds of Sanma games in 2019 to form stratified training and validation sets for each action. To test the generalisability of my models, I sampled another 5,000 rounds in 2020 to form the test sets for the actions. The sizes of the datasets are shown in table 3.3.

Action	Dataset	Examples	Positive	Negative
Discard	Training set	797,285	—	—
	Development set	88,588	—	—
	Test set	147,444	—	—
Pon	Training set	151,348	42,842	108,506
	Development set	16,817	4,760	12,057
	Test set	16,887	4,788	12,099
Kan	Training set	34,319	5,217	29,102
	Development set	3,814	580	3,234
	Test set	3,548	590	2,958
Kita	Training set	136,924	114,863	22,061
	Development set	15,214	12,763	2,451
	Test set	15,498	12,777	2,721
Riichi	Training set	109,804	33,078	76,726
	Development set	12,201	3,675	8,526
	Test set	11,944	3,741	8,203

Table 3.3: Sizes of the datasets

The workflow of dataset generation is similar to the game simulator as mentioned in Section 3.1.2, with the following differences:

- In a game log, all states and actions are fixed, and the go-through of the round is more like a reverse engineering process, with no model evaluation or decision making required;
- A round ends immediately when a player declares win, so the tests of whether a player wins should be replaced by testing whether the round ends;
- Whenever a new state-action pair emerges, it should be extracted, encoded and added to the corresponding dataset.

Therefore the workflow of dataset generation can be illustrated by the flow diagram in Figure 3.5.

Data standardisation

The standardisation of the datasets can be done by simply applying `np.mean` and `np.std` to the training sets, and using the results to update the training, development and test sets, as shown in the following code:

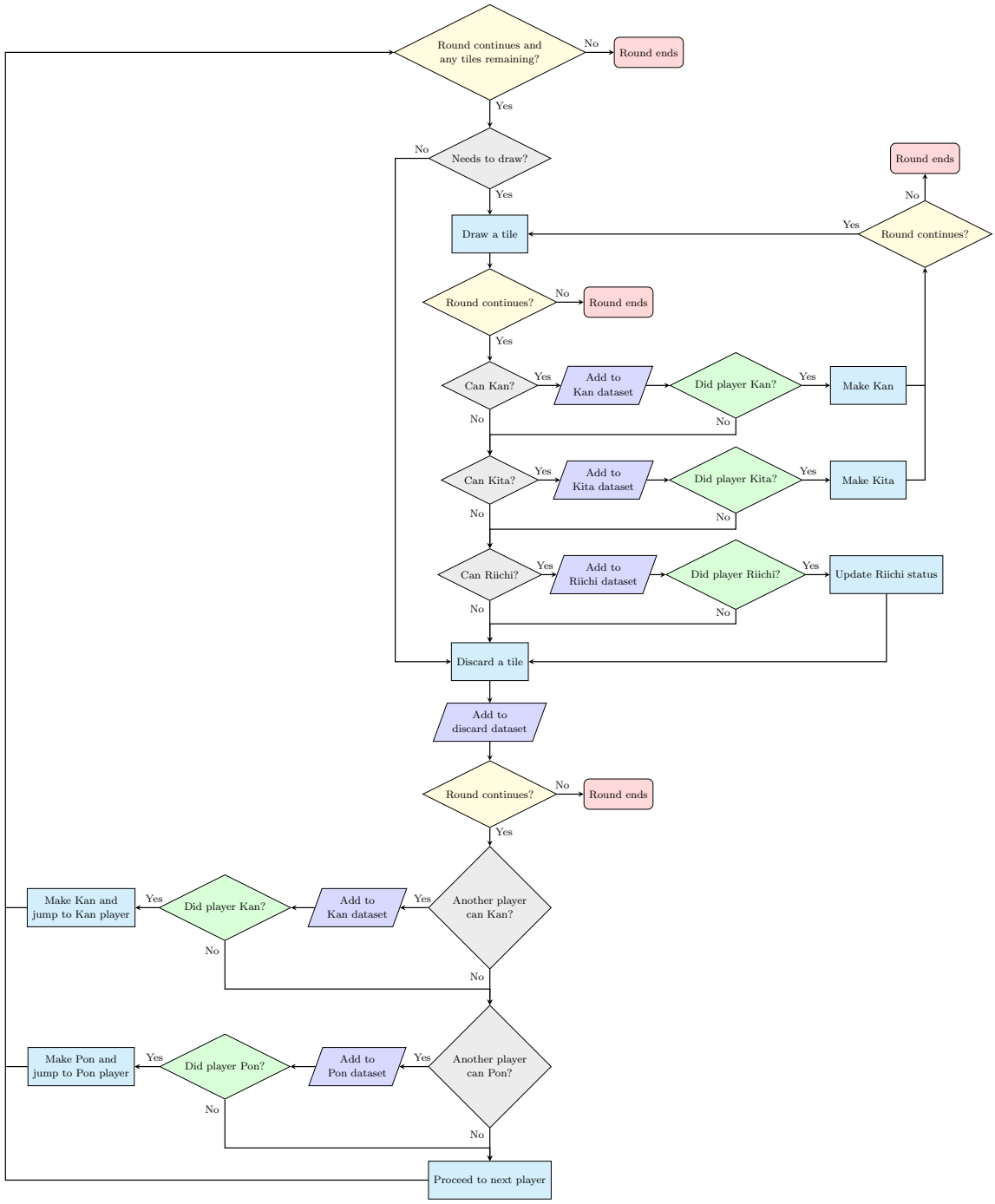


Figure 3.5: Workflow of dataset generation

```
X_mean = np.mean(X_train)
X_std = np.std(X_train)
if X_std == 0:
    X_std = 1e-7  # To prevent division by 0
X_train_scaled = (X_train - X_mean) / X_std
X_dev_scaled = (X_dev - X_mean) / X_std
X_test_scaled = (X_test - X_mean) / X_std
```

This worked in the Pon, Kan, Kita, and Riichi datasets. However, for the discard datasets, since the NumPy implementation of `std` involves creating a copy of the entire array, there was not enough spare memory to create another copy of the discard training set which was 37.0GB in size, and hence the standard deviation could not be calculated using `np.std`. This problem was solved by exploiting the following property of the datasets: since the datasets contain only 0's and 1's, the element-wise squares of the datasets are equal to themselves, and the standard deviation can be derived from the mean:

$$\sigma = \sqrt{\mathbb{E}[\mathbf{X}^2] - (\mathbb{E}[\mathbf{X}])^2} = \sqrt{\mathbb{E}[\mathbf{X}] - (\mathbb{E}[\mathbf{X}])^2} = \sqrt{\mu - \mu^2}$$

This result was also confirmed numerically for the other datasets:

Dataset	$\mu =$ `np.mean`	$\sigma =$ `np.std`	$\sigma = \sqrt{\mu - \mu^2}$	Relative error
Discard	0.050877545	—	0.21974763	—
Pon	0.050338324	0.21864033	0.218642121	8.19218×10^{-6}
Kan	0.053289082	0.2246095	0.22460934	-7.11115×10^{-7}
Kita	0.048485067	0.21479078	0.214788885	-8.82082×10^{-6}
Riichi	0.052583974	0.22320393	0.223201478	-1.09865×10^{-5}

Table 3.4: Means and standard deviations of the datasets

3.3 Supervised Learning

3.3.1 Model Implementation

For the action models of Meowjong, I referred Gao et al.'s CNN structure [6] and adopted a similar CNN structure with 4 convolutional layers followed by a fully-connected layer. Each of the first 3 convolutional layers has 64 kernels, and the last convolutional layer has 32 kernels. Having more convolutional layers or more filters in each layer could result in an oversized model under my memory constraint, and having fewer convolutional layers would result in a huge number of parameters in the flatten operation before the fully-connected layer, also leading to an oversized model. All kernels in those 4 layers share the same kernel size, which is a hyperparameter to be tuned individually for each action model. The fully-connected layer contains 256 perceptrons. A batch normalisation layer and a dropout layer with dropout rate 0.5 are added after each convolutional and fully-connected layer, in order to prevent over-fitting. All action models share roughly the same structure, except the kernel sizes and the shape of the output layer (34 for the discard network, and 2 for the rest). ReLU is used for the activation function of all the convolutional and fully-connected layers, and softmax is used for the activation function of the output layers. The CNN structure for the action models is shown in Figure 3.6.

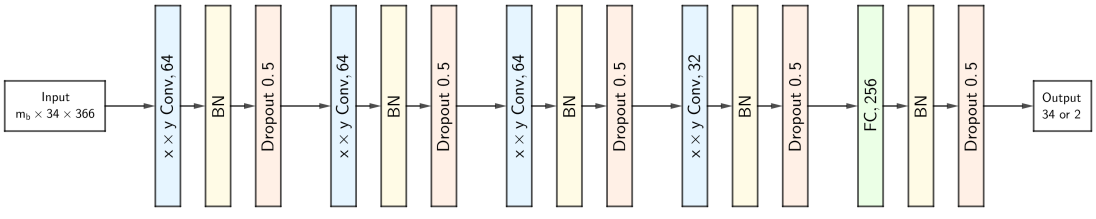


Figure 3.6: CNN structure of the action models

The input and output dimensions of the models are shown in the following table:

	Discard	Pon	Kan	Kita	Riichi
Input	34×366				
Output	34	2			

Table 3.5: Input and output dimensions of the action models

Although pooling is recognised as an efficient down-sampling tool for CNNs in computer vision tasks due to the shift-invariance property of image recognition, in Meowjong's case, the data structure is not an "image", but a compact encoding of discrete feature data, and using pooling here will lose too much information, leading to a lower accuracy. Therefore, pooling is not used in Meowjong's CNN structure. Besides, no padding was used in any convolutional layer, which means the padding style was set to `'VALID'` in TensorFlow. During training, Adam with learning rate 10^{-3} was used as the models' optimiser, and *sparse* categorical cross-entropy loss was used as the loss function, since the labels were provided as integers rather than one-hot vectors.

Discard model

The discard problem can be interpreted as a 34-class classification problem, since there are 34 kinds of tile in total, or a 14-class classification problem, since each player can have at most 14 tiles in their private hand in any given state. As recommended by both Li et al. [7] and Gao et al. [6], I adopted the 34-class classification interpretation, and hence used 34 perceptrons in the output layer. The only potential risk of the 34-class method is the illegal discards, where the discard model outputs a tile that does not exist in the player's private hand. However, as will also be mentioned later in Section 4.3, I found that all discard choices made by my discard model are legal, showing strong learning ability of my CNN and confirming the referenced papers' findings.

Pon, Kan, Kita, and Riichi models

For all the rest of the actions, a player can either declare or skip that action in available situations, therefore all those actions can be interpreted as binary classification problems. Hence, I put 2 perceptrons in the output layer of each action model.

3.3.2 Hyperparameter Tuning

Hyperparameter tuning in the pre-training of the CNN models was focused on the kernel size for each action model. As there was no guarantee that the same kernel size can work for every action, the kernel sizes were tuned individually for each action model. *Grid search* was used for hyperparameter tuning: for each action, all candidate kernel sizes (x, y) ranging from $2 \leq x, y \leq 5$ were tried, making 16 candidate models in total. All 16 candidate models were then trial-trained for 20 epochs, and their performances on the development set in the 20 epochs were compared to select the best kernel size. The model with the best-performing kernel size is chosen as the final model for the full-length pre-training. The selected kernel sizes for the action models are shown in Table 3.6. During trial-training of the Kan models and the Riichi models, two indistinguishably best kernel

sizes emerged for each action, namely $(2,3)$, $(2,4)$ for Kan, and $(3,3)$, $(3,4)$ for Riichi. Both kernel sizes were retained for the full-length pre-training, and the final development set accuracy was used for selecting the better kernel size.

	Discard	Pon	Kan	Kita	Riichi
Kernel size	$(4,5)$	$(5,4)$	$(2,3)$	$(3,2)$	$(3,4)$

Table 3.6: Kernel sizes of the action models

3.4 Reinforcement Learning

Based on the discussion of Monte-Carlo Policy Gradient in Section 2.5.2, I implemented the RL training for Meowjong's discard model based on the following algorithm:

Algorithm 1 REINFORCE: Monte-Carlo Policy Gradient (episodic)

Require: Policy parametrisation $\pi(a|s,\mathbf{w})$, learning rate $\eta > 0$, discount rate $0 \leq \gamma \leq 1$
  Initialise policy parameter $\mathbf{w}$ from the pre-trained model
  while stopping criterion not met **do**
    Generate an episode trajectory $S_0, A_0, R_1, \cdots, S_{T-1}, A_{T-1}, R_T$ under $\pi(\cdot|\cdot,\mathbf{w})$
    $R_T \leftarrow$ Round score/penalty, $G_T \leftarrow 0$
    for each time step of the episode $t = T-1$ downto 0 **do**
      $G_t \leftarrow R_{t+1} + \gamma G_{t+1}$
      $\mathbf{w} \leftarrow \mathbf{w} + \eta\gamma^t G_t \nabla_{\mathbf{w}} \log \pi(A_t|S_t,\mathbf{w})$
    end for
  end while

For each state S_t, an action A_t is sampled at random from the distribution defined by $\pi(A_t|S_t,\mathbf{w})$. However, this does introduce the risk of illegal discards: while $\arg\max_{A_t}\pi(A_t|S_t,\mathbf{w})$, which is used for actual predictions, is indeed always in a player's private hand, the random sampling from $\pi(A_t|S_t,\mathbf{w})$ may still generate a choice that does not exist in the hand. Fortunately, this problem is very rare and only occurred twice during the entire training, and can be bypassed by simply skipping the problematic seeds. This leads to the following code implementation:

```
1 def eval_discard(self, state):
2     # Using the output of the policy network, pick action stochastically
3     policy = self.discard_model.predict(state).flatten()
4     action = np.random.choice(TILES_COUNT, p=policy)
5     # Save the state and action, and output the chosen action
6     self.states.append(state)
7     self.actions.append(action)
8     return action
```

As shown in lines 6–7, the states and actions are added to the trajectory once the actions are sampled. Also, since no intermediate reward is received until a round ends, there is no need to store the rewards $R_1, R_2, \cdots, R_{T-1}$ in the trajectory, since they are

all 0. During training, η was set to 10^{-3}, and γ was set to 0.99. I used the following trick to evaluate the gradient: since the categorical cross-entropy loss is defined as

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \cdot \log \hat{y}_k$$

ignoring the constants γ^t, by setting $y_{A_t} = G_t$, $y_k = 0$ for $k \neq A_t$, and $\hat{\mathbf{y}} = \boldsymbol{\pi}(\cdot|S_t, \mathbf{w})$:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -G_t \log \pi(A_t|S_t, \mathbf{w})$$
$$\nabla_{\mathbf{w}} L(\mathbf{y}, \hat{\mathbf{y}}) = -G_t \nabla_{\mathbf{w}} \log \pi(A_t|S_t, \mathbf{w})$$

Hence the gradient ascent is converted to a gradient descent of the categorical cross-entropy loss function, by setting appropriate values to the labels. By applying this trick, I could even use the Adam optimiser again here in the RL training. Therefore, the REINFORCE update on $\mathbf{w}$ can be implemented using the following code:

```
self.discard_model.fit(x=self.states, y=G_t, epochs=1)
```

3.5 Agents

After training the required models, I integrated them and built the following agents:

- SL agent: the supervised learning agent with all the 5 models trained in the supervised learning way, as described in Section 3.3;
- SL-scaled agent: the supervised learning agent using the data-standardised models;
- RL agent: the reinforcement learning agent with the Pon, Kan, Kita and Riichi models inherited from the SL agent, but its discard model initialised as the SL discard model and enhanced through RL, as described in Section 3.4.

In order to deal with the Kyuushu Kyuuhai case that may happen at the start of a round (as described in Section 2.2.3), I employed the following heuristics: if the start hand contains 11 or more kinds of terminal and honour tiles, the agent should continue the round and go for Kokushi Musou; otherwise, the agent should abort and restart the round. This is based on the statistical results amongst human players:

Start hand	Exhaustive draw rate	Win rate	Deal-in rate	Tsumo-ed rate
13 terminal/honours, 13 kinds	0.000%	99.812%	0.057%	0.048%
13 terminal/honours, 12 kinds	2.834%	66.542%	10.637%	9.859%
12 terminal/honours, 12 kinds	3.797%	55.718%	14.052%	13.162%
12 terminal/honours, 11 kinds	11.049%	20.960%	23.977%	22.067%
11 terminal/honours, 11 kinds	11.074%	19.749%	24.525%	22.381%
11 terminal/honours, 10 kinds	15.024%	6.464%	27.919%	25.453%

Table 3.7: Success rates of Kokushi Musou according to the start hands

Besides, for evaluation, I also built a randomly-playing agent (also called a Random agent), based on the definition in Section 4.1. All agents are a subclass of an abstract class `Agent`.

Chapter 4

Evaluation

4.1 Success Criteria

As stated in my Project Proposal (please refer to Appendix C.2), Meowjong should be deemed *complete* if **it can make decisions of which action to take, based on the observable information in the game at a given time, within an acceptable amount of time**. The minimum time limit for each turn in the Tenhou ranking system is 5 seconds (or 3 seconds in fast play mode), so a suitable choice of time limit for Meowjong would be between 3–5 seconds.

Once Meowjong is complete, it can be deemed *successful* if **it can outperform an agent that plays randomly**. A randomly-playing agent, also named the Random agent, can be defined to have the following behaviours:

- In the discard action, the Random agent shall always randomly select a tile from its hand and discard it;
- In the Pon, Kan, Kita and Riichi actions, as well as the Kyuushu Kyuuhai decision on the first turn, the Random agent shall always decide whether or not to take the action based on random choices, with a probability of 0.5.

4.2 Evaluation Metrics

4.2.1 Model Evaluation

In order to evaluate the performance of SL, the train-dev-test accuracies of all my SL-trained models (i.e., the discard, Pon, Kan, Kita and Riichi models with and without data standardisation) should be examined. Since the RL training of a model is based on agent-environment interactions rather than the training data, the train-dev-test accuracies of the RL-improved discard model are irrelevant, and hence will not be considered for evaluation.

In terms of speed, since the action making process of an agent consists only of model prediction, the time taken for my CNN models to predict one single example can be measured as an approximation of the agent's decision time.

4.2.2 Agent Evaluation

Evaluation of two agents by directly playing against each other

In order to show that Meowjong can outperforms a Random agent, I let each of the SL agent, the SL-scaled agent, and the RL agent play Sanma games against 2 copies of Random agents, using the game results amongst 3 Random agents as the baseline for comparisons. Since the initial private tiles have large randomness and will greatly affect the win/loss of a game, the agents must play a substantial number of rounds, in order to reduce the variance caused by the initial private tiles. In addition, to evaluate the improvement to Meowjong by RL, I let the SL and RL agents play a substantial number of games against each other, in the following 3 categories:

- SL vs. SL: 3 SL agents playing amongst themselves, whose results served as the baseline for the experiment;
- SL vs. 2 RL: 1 SL agent playing against 2 RL agents ;
- RL vs. 2 SL: 1 RL agent playing against 2 SL agents.

To test whether an agent performs significantly better than another, I carried out the following one-tailed significance testing on the results:

Null Hypothesis: There is no significant difference between Agent 1 and Agent 2.
Alternative Hypothesis: Agent 1 performs significantly better than Agent 2.
Significance Level: $\alpha = 0.05$

Since there is no suitable standard significance test for my case, I designed my own version of significance test. A round of Sanma can have 4 different outcomes: 1st place, 2nd place, 3rd place and draw. Therefore, I assume that the game results amongst 3 identical agents can be modelled by a multinomial distribution Multinomial$(n, \mathbf{p})$, where:

- n is the total number of rounds played;
- $\mathbf{p} = \begin{bmatrix} p_1 & p_2 & p_3 & p_{\text{draw}} \end{bmatrix}$ denotes the probabilities of each outcome, with 1st place rate p_1, 2nd place rate p_2, 3rd place rate p_3, and draw rate p_{draw}.

The probability mass function for Multinomial$(n, \mathbf{p})$ is:

$$\Pr(\mathbf{X} = \mathbf{x}) = \frac{n!}{p_1!p_2!p_3!p_{\text{draw}}!} p_1^{x_1} p_2^{x_2} p_3^{x_3} p_{\text{draw}}^{x_{\text{draw}}}$$

where $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 & x_{\text{draw}} \end{bmatrix}$ denotes the frequencies of the outcome, with 1st place frequency being x_1, 2nd place frequency being x_2, 3rd place frequency being x_3, and draw frequency being x_{draw}.

Before conducting the significance testing, it is necessary to define what is "better". Here I define "better" to be having both a higher 1st place rate and a lower 3rd place rate. If both the 1st place rates and the 3rd place rates are equal, the agent should prioritise on improving the draw rate, because 2nd place often comes from another player winning by

Tsumo or by Ron from the third player, resulting the agent themself in a zero or negative round score, and a guaranteed negative score difference. Two outcomes with one having higher values on both the 1st place rate and the 3rd place rate, or lower values on both the 1st place rate and the 3rd place rate, are not directly comparable. Therefore, for two outcomes $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 & x_{\text{draw}} \end{bmatrix}$ and $\mathbf{x}' = \begin{bmatrix} x'_1 & x'_2 & x'_3 & x'_{\text{draw}} \end{bmatrix}$, I define the following strict partial order relation $\succ$, interpreted as "an outcome is better than another", to be

$$\forall \mathbf{x}, \mathbf{x}'.\ \mathbf{x} \succ \mathbf{x}' \iff (x_1 > x'_1 \wedge x_3 < x'_3) \vee (x_1 = x'_1 \wedge x_3 = x'_3 \wedge x_{\text{draw}} > x'_{\text{draw}})$$

The same "better" relation also applies to the probabilities. Therefore, with the baseline probabilities $\mathbf{p}_0 = \begin{bmatrix} p_1 & p_2 & p_3 & p_{\text{draw}} \end{bmatrix}$ observed from the baseline game results, I can set up the following hypotheses:

Null Hypothesis H_0**:** $\mathbf{p} = \mathbf{p}_0$
Alternative Hypothesis H_a**:** $\mathbf{p} \succ \mathbf{p_0}$

The cumulative distribution function of Multinomial$(n, \mathbf{p})$ can be used to test H_0, which can be calculated by the following formula:

$$\begin{aligned}
\Pr(\mathbf{X} \succeq \mathbf{x}) &= \sum_{\mathbf{x}' \succeq \mathbf{x}} \Pr(\mathbf{X} = \mathbf{x}') \\
&= \sum_{x'_1 > x_1} \sum_{x'_2} \sum_{x'_3 < x_3} \Pr\left(\mathbf{X} = \begin{bmatrix} x'_1 & x'_2 & x'_3 & (n - x'_1 - x'_2 - x'_3) \end{bmatrix}\right) \\
&\quad + \sum_{x'_2 < x_2} \Pr\left(\mathbf{X} = \begin{bmatrix} x_1 & x'_2 & x_3 & (n - x_1 - x'_2 - x_3) \end{bmatrix}\right) \\
&\quad + \Pr\left(\mathbf{X} = \begin{bmatrix} x_1 & x_2 & x_3 & x_{\text{draw}} \end{bmatrix}\right)
\end{aligned}$$

where $\succeq$ is the non-strict version of $\succ$. If $\Pr(\mathbf{X} \succeq \mathbf{x}) < \alpha$, then H_0 can be rejected.

Evaluation of two agents by playing against a third agent

In order to test whether data standardisation has made a significant difference to the SL, I compared the difference in the Sanma game results between the SL agent and the SL-scaled agent against the Random agents, and conducted a two-tailed significance test:

Null Hypothesis: There is no significant difference between the SL agent and the SL-scaled agent.
Alternative Hypothesis: There exists a significant difference between the SL agent and the SL-scaled agent.
Significance Level: $\alpha = 0.05$

Since all the aforementioned game simulations share the same range of random seeds, it is reasonable to pair up the game results and use the standard *sign test*: for a total of n pairs of round outcomes, each pair of which share the same random seed and hence the same initial configuration, I counted the following quantities:

- The number of cases where the SL-scaled agent finished at a higher rank than the SL agent, called *Plus*;

- the number of cases where the SL-scaled agent finished at a lower rank than the SL agent, called *Minus*;
- the number of cases where the SL-scaled agent finished at the same rank as the SL agent, called *Null*. Since it is common to get a large number of ties, in order to maintain statical power, I distribute the ties evenly between *Plus* and *Minus*, and rounded up the figures if necessary.

Then, I can use the cumulative distribution function of a binomial distribution Binomial(n, q) to find the probability that the outcome of one agent versus Random agents is better than that of the other:

$$\Pr(\text{Agent 1} \succeq \text{Agent 2}) = \sum_{i=0}^{k} \binom{n}{i} q^i (1-q)^{n-i}$$

where

- q is the probability of success;
- $n = 2\lceil \frac{Null}{2} \rceil + Plus + Minus$ is the total number of cases;
- $k = \lceil \frac{Null}{2} \rceil + \min(Plus, Minus)$ is the number of cases with the less common sign.

The hypotheses can hence be set up as:

Null Hypothesis H_0: $q = 0.5$
Alternative Hypothesis H_a: $q \neq 0.5$

This simplifies the probability formula to:

$$p = \Pr(\text{Agent 1} \succeq \text{Agent 2}) = \sum_{i=0}^{k} \binom{n}{i} 0.5^n$$

If $p < \frac{\alpha}{2}$, then H_0 can be rejected.

4.3 Model Evaluation

4.3.1 Supervised Learning

Training results

The discard models, both with and without data standardisation, were trained in mini-batches of size 64 for 200 epochs, and the rest of the models were trained in mini-batches of size 32 for 500 epochs. Their learning curves in Figure 4.1, by means of validation accuracy, show a unanimously fast and satisfactory convergence. Compared to the discard models, the other models converged much more quickly, which is likely to be due to smaller dataset sizes. According to the learning curves, none of the scaled (data standardised) and unscaled models develop any notable difference. Though one model may converge more stably than the other in some of the actions, this is still not a general characteristic across all actions for either model.

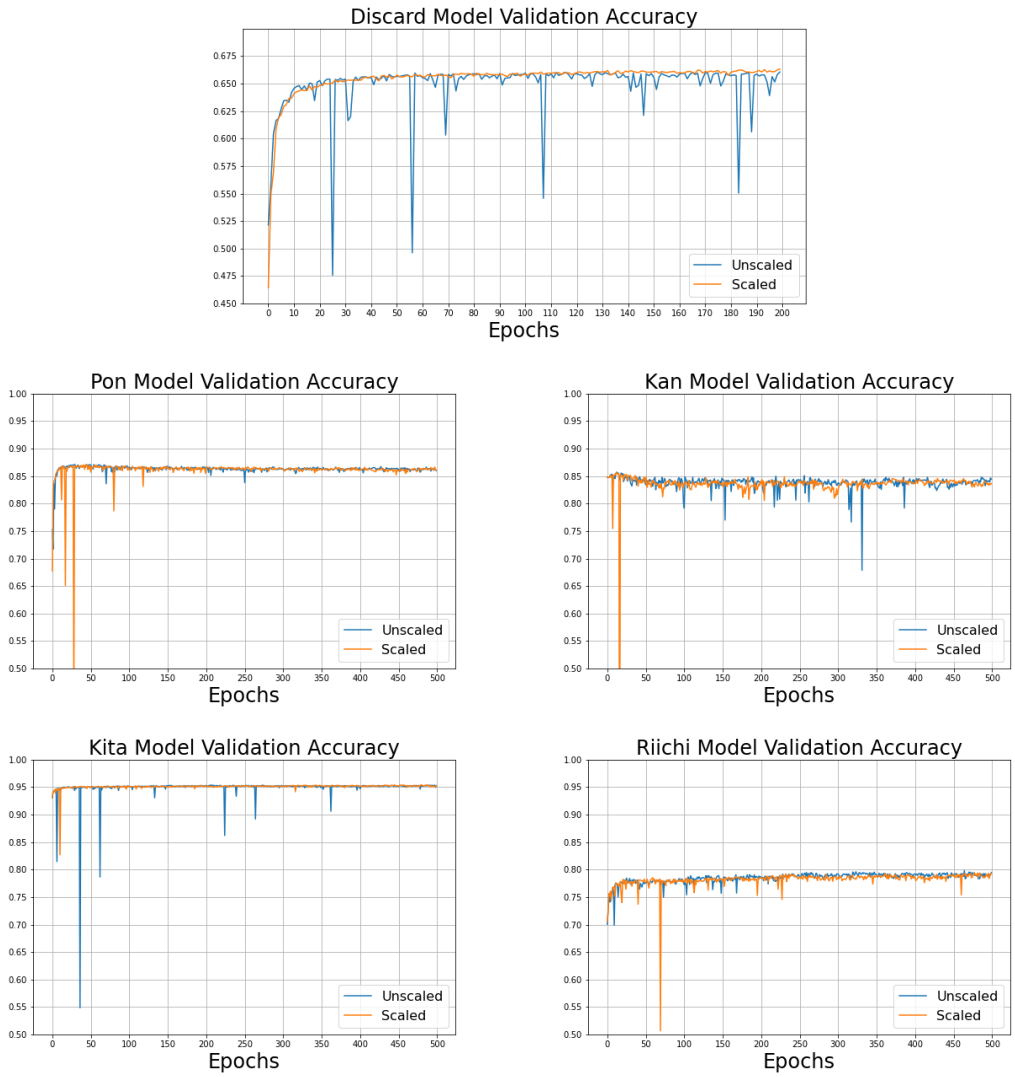


Figure 4.1: Learning curve for the action models in supervised learning

Test results

The test accuracies of the models are reported in Table 4.1. I also included the test accuracies achieved by previous works as a rough reference, though they are not directly comparable due to the different training/validation/test data sources and structures.

Model	Test Accuracy			
	Meowjong	Meowjong (scaled)	Gao et al. [6]	Suphx [7]
Discard	65.81%	65.54%	68.8%	76.7%
Pon	70.95%	72.10%	88.2%	91.9%
Kan	92.45%	88.92%	—	94.0%
Kita	94.26%	94.44%	—	—
Riichi	62.63%	64.29%	—	85.7%

Table 4.1: Test accuracies for the action models in SL

The test accuracies of Meowjong are very satisfactory, achieving Gao et al.'s level [6]. Although there is still a gap between Meowjong and Suphx [7], Suphx used very large, 102/104-layer residual CNN structures and required much more computational power and time to train, whereas Meowjong adopted a much simpler CNN structure and was much cheaper and faster to train. There is still no significant difference in the test accuracies between the unscaled and scaled models.

4.3.2 Reinforcement Learning

The discard model without data standardisation was improved through self-play RL, under the REINFORCE (Monte-Carlo policy gradient) algorithm, for 400 episodes. Evaluations against 2 Random agents were carried out and recorded after every 10 episodes, each playing 500 rounds as East, South, and West. The curves of the win rates are plotted in Figure 4.2, showing a very successful improvement on all winds. The sudden drops on the win rates between Episodes 190/200 and Episodes 370/380 are probably due to the variance in policy updates, which are performed after every episode.

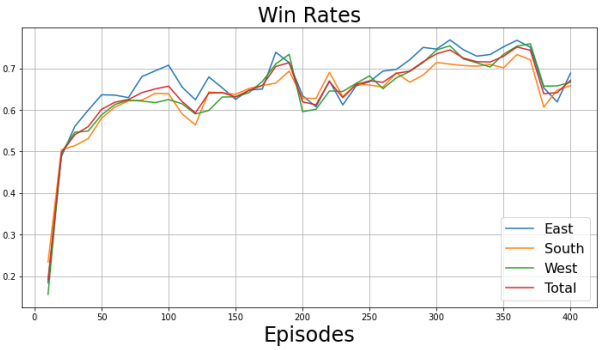


Figure 4.2: Win rates of Meowjong in RL training

Fig 4.3 shows the loss rates, which also gradually decrease over episodes, after sharp increases in the first 100 episodes. However, the final loss rate is still higher than the SL agent. The RL agent may have learned an aggressive style, and finessed its skill through self-play, lowering the loss rate.

Figure 4.3: Loss rates of the RL agent during training

Fig 4.4 shows the average scores of the checkpoint evaluations. The rescaled plots on the right remove the effect caused by the East player (the dealer) winning 50% more,

showing a consistent performance with no preference on any wind. The average scores are also stable across episodes, which is in accordance with the training target for Meowjong: maximising the winning probabilities rather than the round scores.

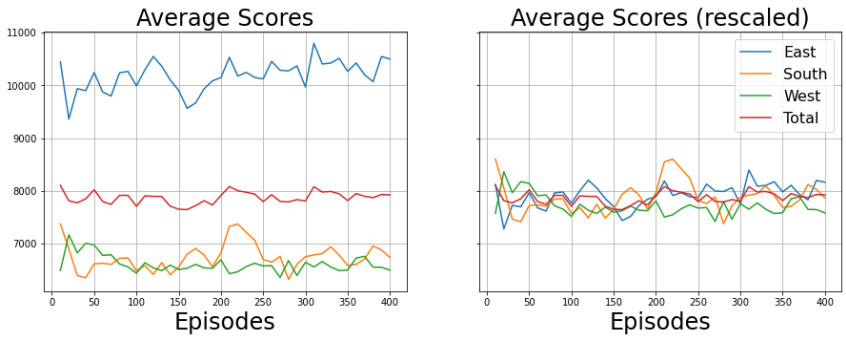


Figure 4.4: Average scores of the RL agent during training

However, the RL training introduced a side-effect: the discard model may predict a tile to discard that does not exist in the hand. However, this is still very rare in evaluations, occurring at most 6 times in 1,500 rounds. According to the bar plot in Figure 4.5, I believe the rate of illegal discards is independent of the episode.

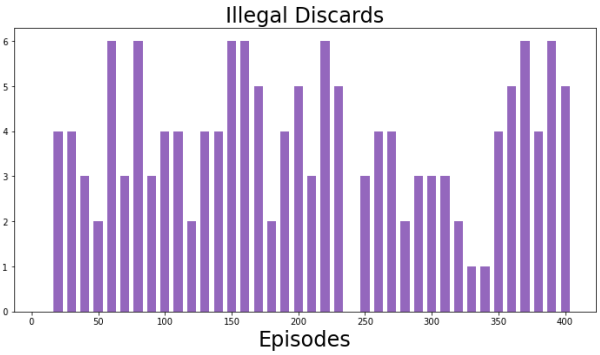


Figure 4.5: Illegal discards of the RL agent

4.4 Agent Evaluation

4.4.1 Evaluation Against Random Agents

I simulated 5,000 rounds of Sanma games amongst 3 Random agents, and 5,000 rounds for each of the SL, SL-scaled, and RL agents against 2 Random agents, in each wind position. Their results, in terms of 1st/2nd/3rd-place and draw rates, are reported in Table 4.2. The results clearly show that all Meowjong's trained agents can outperform a Random agent. There was no significant difference in any of the rates between the SL and SL-scaled agents, and the RL agent's 1st place rates were also much larger than both SL agents. The RL agent also had higher 3rd place rates, which suggests that it learned an aggressive style through self-play. For rigour, I still carried out the significance test, with the aid of `scipy.stats.multinomial`. Since `scipy.stats.multinomial` does not offer a `cdf` function, I manually added up the required probability mass functions based on the previously defined partial order, as shown in the following code:

Agents	Wind	1st Place Rate	2nd Place Rate	3rd Place Rate	Draw Rate
Random vs. Random	—	0.02%	0.02%	0.02%	99.94%
SL vs. Random	East	22.00%	0.06%	0.08%	77.86%
	South	22.68%	0.06%	0.02%	77.24%
	West	20.72%	0.16%	0.04%	79.08%
	Total	21.80%	0.09%	0.05%	78.06%
SL-scaled vs. Random	East	21.40%	0.04%	0.14%	78.42%
	South	22.72%	0.08%	0.00%	77.20%
	West	20.02%	0.20%	0.06%	79.72%
	Total	21.38%	0.11%	0.06%	78.45%
RL vs. Random	East	73.59%	0.02%	3.27%	23.12%
	South	71.93%	0.08%	3.46%	24.53%
	West	71.61%	0.06%	2.85%	25.48%
	Total	72.38%	0.05%	3.19%	24.38%

Table 4.2: Comparisons between the trained agents against Random agents

```
cdf = 0
for x'_1 in range(x_1+1, n+1):
    for x'_3 in range(0, min(n-x'_1+1, x_3)):
        for x'_2 in range(0, n-x'_1-x'_3+1):
            cdf += multinomial.pmf([x'_1,x'_2,x'_3,n-x'_1-x'_2-x'_3],n,[p_1,p_2,p_3,p_draw])
for x'_2 in range(0, x_2):
    cdf += multinomial.pmf([x_1,x'_2,x_3,n-x_1-x'_2-x_3],n,[p_1,p_2,p_3,p_draw])
cdf += multinomial.pmf([x_1,x_2,x_3,x_draw],n,[p_1,p_2,p_3,p_draw])
```

The results of the significance test are reported in Table 4.3, which reject all H_0 and prove the significant difference. This is also confirmed by the box plots of all the scores shown in Figure 4.6. Besides, it took on average 4.2 hours to simulate 5,000 rounds, which is about 3 seconds per round. This means Meowjong meets the success criterion for speed.

Agents	Wind	n	x_1	x_2	x_3	x_{draw}	$\Pr(\mathbf{X} \succeq \mathbf{x})$
SL vs. Random	East	5,000	1,100	3	4	3,893	`0.0` ($< O(\epsilon)$)[1]
	South	5,000	1,134	3	1	3,862	`0.0` ($< O(\epsilon)$)
	West	5,000	1,036	8	2	3,954	`0.0` ($< O(\epsilon)$)
	Total	15,000	3,270	14	7	11,709	`0.0` ($< O(\epsilon)$)
SL-scaled vs. Random	East	5,000	1,070	2	7	3,921	`0.0` ($< O(\epsilon)$)
	South	5,000	1,136	4	0	3,860	`0.0` ($< O(\epsilon)$)
	West	5,000	1,001	10	3	3,986	`0.0` ($< O(\epsilon)$)
	Total	15,000	3,207	16	10	11,767	`0.0` ($< O(\epsilon)$)
RL vs. Random	East	4,949	3,642	1	162	1,144	`0.0` ($< O(\epsilon)$)
	South	4,949	3,560	4	171	1,214	`0.0` ($< O(\epsilon)$)
	West	4,949	3,544	3	141	1,261	`0.0` ($< O(\epsilon)$)
	Total	14,847	10,746	8	474	3,619	`0.0` ($< O(\epsilon)$)

Table 4.3: Significance test results for the trained agents against Random agents

[1] ϵ is the smallest representable positive `float64` value in Python, which equals $2^{-1074} \approx 4.94 \times 10^{-324}$, and the constant factor is at most 15000^3, so the cumulative probability cannot exceed 1.67×10^{-311}.

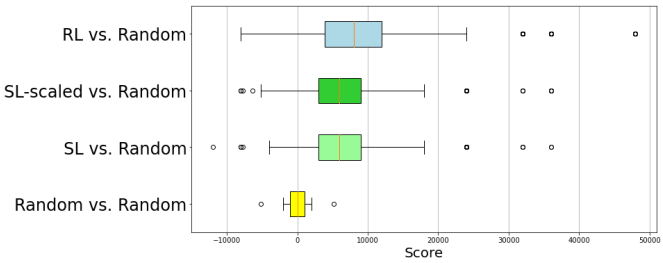


Figure 4.6: Scores of the trained agents against Random agents

4.4.2 Comparison Between SL and RL Agents

In this series of evaluations, I simulated 5,000 rounds amongst 3 SL agents, and 5,000 rounds in each wind positions in both SL vs. 2RL and RL vs. 2SL. The results are reported in table 4.4:

Agents	Wind	1st Place Rate	2nd Place Rate	3rd Place Rate	Draw Rate
SL vs. SL	East	18.70%	11.90%	24.84%	44.56%
	South	19.74%	24.32%	11.38%	44.56%
	West	17.00%	19.22%	19.22%	44.56%
	Total	18.48%	18.48%	18.48%	44.56%
SL vs. RL	East	5.76%	7.09%	81.53%	5.62%
	South	6.10%	38.09%	49.36%	6.45%
	West	4.90%	37.69%	51.68%	5.72%
	Total	5.59%	27.62%	60.86%	5.93%
RL vs. SL	East	57.46%	7.75%	13.92%	20.87%
	South	57.90%	16.34%	4.93%	20.83%
	West	55.68%	18.18%	5.25%	20.89%
	Total	57.02%	14.09%	8.03%	20.86%

Table 4.4: Comparisons between SL and RL agents

It is clear from the outcomes that an RL agent can outperform an SL agent. The higher 3rd place rates at the East position are due to the rule that the East player pays twice as much as the other player when a non-East player wins by Tsumo. The significant difference can also be confirmed by the significance test results in Table 4.5, and the box plots of the scores in the simulation in Figure 4.7. Besides, according to the outcomes, both SL agents seem to perform better at the South position.

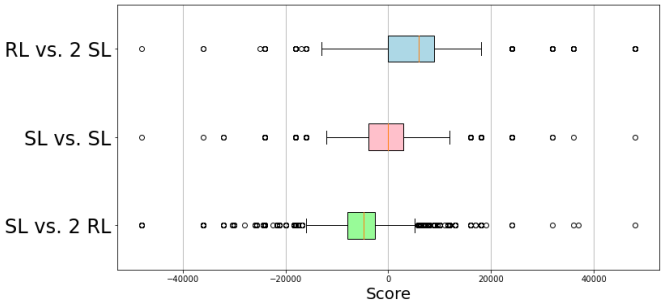


Figure 4.7: Scores of the trained agents against Random agents

Agents	Wind	n	x_1	x_2	x_3	x_{draw}	$\Pr(\mathbf{X} \succeq \mathbf{x})$
RL vs. SL	East	4,993	2,869	387	695	1,042	0.0 ($< O(\epsilon)$)
	South	4,993	2,891	816	246	1,040	0.0 ($< O(\epsilon)$)
	West	4,993	2,780	908	262	1,043	0.0 ($< O(\epsilon)$)
	Total	14,979	8,540	2,111	1,203	3,125	0.0 ($< O(\epsilon)$)
SL vs. RL	East	4,996	288	354	4,073	281	1.0 ($> 1 - O(\epsilon)$)
	South	4,996	305	1,903	2,466	322	1.0 ($> 1 - O(\epsilon)$)
	West	4,996	245	1,883	2,582	286	1.0 ($> 1 - O(\epsilon)$)
	Total	14,988	838	4,140	9,121	889	1.0 ($> 1 - O(\epsilon)$)

Table 4.5: Significance test results for the RL agent against the SL agent

4.4.3 Comparison Between SL and SL-scaled Agents

The sign test as explained in Section 4.2.2 to compare the SL and SL-scaled agents was carried out by using `scipy.stats.binom.cdf`:

```
for seed in range(5000):
    sl_scores, sl_scaled_scores = get_round_scores(seed)
    for wind in 'East', 'South', 'West':
        if get_rank(sl_scaled_scores, wind) > get_rank(sl_scores, wind):
            plus[wind] += 1
            plus['Total'] += 1
        elif get_rank(sl_scaled_scores, wind) < get_rank(sl_scores, wind):
            minus[wind] += 1
            minus['Total'] += 1
        else:
            tie[wind] += 1
            tie['Total'] += 1
for wind in 'East', 'South', 'West', 'Total':
    n = 2 * ((tie[wind] + 1) // 2) + plus[wind] + minus[wind]
    k = (tie[wind] + 1) // 2 + min(plus[wind], minus[wind])
    prob = scipy.stats.binom.cdf(k, n, 0.5)
```

which produced the following results, confirming with the previous observations that there is indeed no significant difference between the SL agent and a SL-scaled agent.

Wind	*Plus*	*Minus*	*Null*	n	k	$\Pr(\text{SL}_1 \succeq \text{SL}_2)$
East	513	545	3,942	5,000	2,484	0.3305
South	529	528	3,943	5,001	2,500	0.5000
West	488	523	3,989	5,001	2,483	0.3153
Total	1,530	1,596	11,874	15,000	7,467	0.2978

Table 4.6: Sign test results for the SL and SL-scaled agents

Chapter 5

Conclusions

5.1 Results

In this project, I built an efficient CNN structure that solves the decision making problem in Sanma, gathered the training data and designed the dataset structure, built from scratch a fully functional Sanma simulator, and trained agents using both DL and DRL. All trained agents for Meowjong have passed all the success criteria, and I can hence conclude that my project is successful. In addition, being the first ever project on Sanma, with top-tier test accuracies from SL and a significant further enhancement from RL, Meowjong stands as a state-of-the-art in Sanma and possesses a strong potential to make a contribution to the relevant field, pending online evaluations against human players.

5.2 Lessons Learned

In building a ML project, the preparatory work, including data collection and processing, as well as building necessary helper environments, is much more demanding than the ML itself. Attention to every detail is crucial, in order to avoid errors and the extra workload required to fix them. Besides, DRL is a very resource-heavy task in terms of both space and time. It would be unwise to overlook the resource constraints in the design stage, and even the smallest resource optimisation is desirable, in order to save resources to achieve better ML performance. Furthermore, In terms of writing up, it is better to write the dissertation along with the work, rather than after the work is completed. Without sufficient written records, by the time the entire project is finished, some subtle details of the early work are likely to have been forgotten.

5.3 Further Work

Firstly, I shall train my CNNs on larger datasets to improve their accuracies. I can also conduct hyperparameter tuning on RL, and apply off-policy distributed RL with entropy regularisation, in order to improve RL performance. As suggested by Suphx [7], I can introduce a GRU-based global reward prediction to Meowjong, so that it can maximise the full-game results, in addition to the single-round performance. Finally, I can extend Meowjong to the full, 4-player Mahjong, and evaluate it online against human players.

Bibliography

[1] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[2] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, et al. Dota 2 with large scale deep reinforcement learning. *arXiv*, 1912.06680, 2019.

[4] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[5] Naoki Mizukami and Yoshimasa Tsuruoka. Building a computer Mahjong player based on Monte Carlo simulation and opponent models. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 275–283. IEEE, 2015.

[6] Shiqi Gao, Fuminori Okuya, Yoshihiro Kawahara, and Yoshimasa Tsuruoka. Supervised learning of imperfect information data in the game of Mahjong via deep convolutional neural networks. *Information Processing Society of Japan*, 2018.

[7] Junjie Li, Sotetsu Koyamada, Qiwei Ye, Guoqing Liu, et al. Suphx: Mastering Mahjong with deep reinforcement learning. *arXiv*, 2003.13590, 2020.

[8] Moyuru Kurita and Kunihito Hoki. Method for constructing artificial intelligence player with abstractions to Markov decision processes in multiplayer game of Mahjong. *IEEE Transactions on Games*, 13(1):99–110, 2021.

[9] Wikipedia contributors. Mahjong – Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Mahjong`.

[10] European Mahjong Association. Rules for Japanese Mahjong. `http://mahjong-europe.org/portal/images/docs/Riichi-rules-2016-EN.pdf`.

[11] Shingo Tsunoda. Tenhou.net. `https://tenhou.net/`.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, second edition, 2018.

[14] FluffyStuff (GitHub user). riichi-mahjong-tiles. *GitHub repository*, `https://github.com/FluffyStuff/riichi-mahjong-tiles`, 2017.

[15] Aphex34 (Wikipedia contributor). Typical CNN architecture. `https://commons.wikimedia.org/w/index.php?curid=45679374`. CC BY-SA 4.0.

[16] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[17] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from `https://www.tensorflow.org/`.

[18] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, et al. Array programming with NumPy. *Nature*, 585:357–362, 2020.

[19] Wes McKinney. Data structures for statistical computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61, 2010.

[20] Yüce Tekol and Contributors. PySwip v0.2.10. `https://github.com/yuce/pyswip`, 2020.

[21] Kenneth Reitz and Contributors. Requests v2.25.1. `https://github.com/psf/requests`, 2020.

[22] Alex Clark and Contributors. Pillow v8.1.0. `https://github.com/python-pillow/Pillow`, 2021.

[23] Gael Varoquaux and Contributors. Joblib v1.0.0. `https://github.com/joblib/joblib`, 2020.

[24] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17:261–272, 2020.

[25] John D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science Engineering*, 9(3):90–95, 2007.

[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, et al. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[27] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456. PMLR, 2015.

[28] Mliu92 (Wikipedia contributor). Mahjong table. `https://commons.wikimedia.org/w/index.php?curid=71930245`. CC BY-SA 4.0.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

Appendix A

Notations

Numbers and Arrays

a	A scalar (integer or real)
$\mathbf{a}$	A vector
$\mathbf{A}$	A matrix
A	A scalar random variable

Indexing

a_i	Element i of vector $\mathbf{a}$, with indexing starting at 1
$A_{i,j}$	Element i, j of matrix $\mathbf{A}$
A_t	Random variable A at time t
$\mathbf{x}^{(i)}$	The i-th example from a dataset
$y^{(i)}$ or $\mathbf{y}^{(i)}$	The label for the i-th example
$a_i^{[l]}$	The activation of perceptron i in the l-th layer

Sets and Functions

$\mathcal{A}$	A set
$\mathbb{R}$	The set of real numbers
$f : \mathcal{A} \to \mathcal{B}$	The function f with domain $\mathcal{A}$ and image $\mathcal{B}$
$\arg\max_x f(x)$	Argument x of the maximum of $f(x)$
$\arg\min_x f(x)$	Argument x of the minimum of $f(x)$
$\lfloor x \rfloor$	The floor function of x
$\lceil x \rceil$	The ceiling function of x
$1_{\text{condition}}$	The indicator function: 1 if the condition is true, and 0 otherwise

Sometimes a function f whose argument is a scalar can be applied to a vector or matrix: $f(\mathbf{x})$ or $f(\mathbf{X})$. This denotes the application of f to the vector/matrix element-wise. For example, if $\mathbf{Y} = f(\mathbf{X})$, then $Y_{i,j} = f(X_{i,j})$ for all valid values of i and j.

Calculus and Linear Algebra Operations

$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$f'(x)$	Derivative of $f(x)$ with respect to x
$\nabla_{\mathbf{x}} y$	Gradient of y with respect to x
$\mathbf{A}^T$	Transpose of matrix $\mathbf{A}$

Probability

$\Pr(A)$	A probability distribution over a discrete variable
$p(A)$	A probability distribution over a continuous variable
$\mathbb{E}_X[f(X)]$ or $\mathbb{E}[f(X)]$	Expectation of $f(X)$ with respect to X
$\binom{n}{k}$	k-combination of n: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
Binomial(n, p)	Binomial distribution with n independent experiments and probability p
Multinomial$(n, \mathbf{p})$	Multinomial distribution with n independent experiments and probability vector $\mathbf{p}$

Appendix B

Rules of Riichi Mahjong

This appendix provides a more detailed Riichi Mahjong rules as a supplement to Section 2.2. The full Riichi Mahjong rules published by the European Mahjong Association can be found in their official website [10].

B.1 Setup: Additional Information

The players' start positions at the table are determined by drawing lots, unless pre-arranged otherwise. The seating order of each player is represented by East, South or West, which is called the player's *seat wind*. The East player becomes the dealer for that round. The round also has a wind, called the *round wind* or the *prevalent wind*, which is initialised as East at the beginning of the game and changes after a full cycle of rounds.

At the start of each round, the tiles are thoroughly mixed and arranged into *walls* of face-down, two-tier-high tiles, as shown in Figure B.1. The dealer (the East player) rolls two dice to determine where to start drawing from the wall. The last 14 tiles form the *dead wall*, which will not be used throughout this round except in some specific cases, and the rest of the walls with available tiles form the *live wall*.

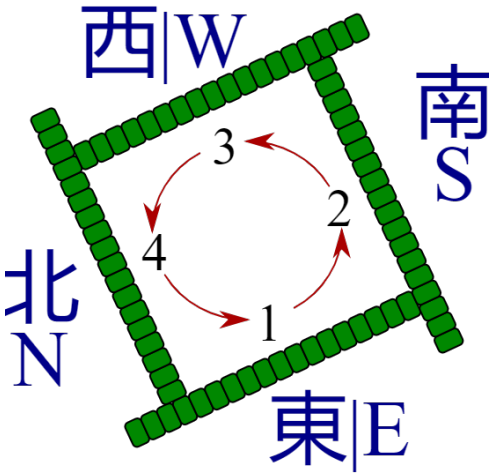


Figure B.1: Table setup for 4-player Mahjong, from Wikipedia [28]

B.2 Tile Calls: Additional Information

Chii

By calling Chii, a player can claim an immediately discarded tile from the player on the left, to form a sequence of three consecutive tiles of the same suit (also called Shuntsu). The player needs to discard a tile after calling Chii. Chii is disabled in Sanma.

Kan

There are 3 types of Kan:

- *Open Kan* (also called *Daiminkan*): the player possesses three tiles of the same kind, and claims the fourth tile discarded by another player to form a quad;
- *Add Kan* (also called *Shouminkan* or *Kakan*): the player has an open triplet from calling Pon previously, and adds the fourth tile in their hand to that triplet to upgrade it to a quad. This must be done in a turn after the player has drawn a tile from the wall (i.e., not in a turn where a tile was claimed for Chii or Pon), but not necessarily at the first opportunity, and can be delayed to any later turn as the players' wish.
- *Concealed Kan* (also called *Ankan*): the player possesses all four tiles of the same kind, and simply separates them from their private hand to form a quad. Concealed Kan only counts as a closed meld, and does not break the player's Manzen status.

The additional tile after the Kan call is drawn from the dead wall. The dead wall always comprises 14 tiles, so the last tiile of the live wall becomes part of the dead wall. The Kan action is prone to a Yaku called *Robbing the Kan* (also called *Chankan*): if another player is in Tenpai and the Kan tile completes their hand, the player may call Ron immediately after the Kan declaration and wins the round.

Kita

The additional tile after the Kita call is drawn from the dead wall. The Nukidora does not count as a meld, and hence does not break the player's Menzen status. The Kita call can be Ronned, but this does not award Chankan.

Riichi

A player may declare concealed Kan and Kita even after Riichi, as long as this does not change the waiting pattern.

B.2.1 End of a Round: Additional Information

Abortive Draws

Apart from Kyuushu Kyuuhai, the following situations can also lead to an abortive draw while tiles are still available:

- *Suufon Renda*: on the first turn without any tile call, if all four players discard the same wind tile, the round is aborted in a draw. Apparently this cannot happen in Sanma.
- *Suucha Riichi*: if all four players declare Riichi, the round is aborted in a draw. Apparently this cannot happen in Sanma either.
- *Suukan Sanra*: if four quads are called by different players, the round is aborted in a draw. If all four quads are called by one player, the round continues and no other players are allowed to declare Kan in that round.

Dealer Rotation

A complete round gives each player the opportunity to be the dealer for at least once. After the end of a round, the East player stays East in the next round if they win or are in Tenpai when there is an exhaustive draw. If there are multiple winners, the East player stays East in the next round if they were one of the winners. Otherwise, the dealer rotates to the next player, i.e., the South player becomes East in the next round. In the case where the dealer repeats (i.e. the East player staying East), a *Honba* counter is added, which will be cleared when the dealer rotates.

Nagashi Mangan

After an exhaustive draw, if a player has a concealed hand (which does not have to be Tenpai) and has discarded only terminal and honour tiles throughout the round, and non of the discards has been claimed, then the player can receive a round score equivalent to a self-draw Mangan. The dealer rotates the same way as an exhaustive draw.

B.3 Scoring: Dora

Red Dora

Red Doras are also known as Akadoras. One of each of the 5's in Manzu, Pinzu and Souzu are marked red and count as Doras, as shown in the following table:

5 Man	5 Pin	5 Sou
		

Dora from indicator

A Dora indicator is revealed from the dead wall at the start of each round. The next tile of the Dora indicator becomes the Dora in that round. The Dora order is circular:

- For each suit, the order is 1-2-3-4-5-6-7-8-9-1;
- For wind tiles, the order is East-South-West-North-East;
- For dragon tiles, the order is Haku-Hatsu-Chun-Haku.

The Han value of each Dora tile can accumulate in the event of repeated Dora indicators.

Kandora

After a Kan call, an additional Dora indicator from the dead wall is revealed, indicating the Kandora. Depending on the rules and/or formality of the game, the timing for revealing the Kandora for each type of Kan may vary.

Uradora

When a player wins after declaring Riichi, they can reveal the tiles underneath the Dora indicators and any Kandora indicators. These tiles indicate the Uradoras under the same order mentioned above, which can only be claimed by the winners who declared Riichi.

Nukidora

Nukidoras are North tiles put aside by Kita calls in Sanma, as described in Section 2.2.

B.4 Scoring: Yaku List

1-Han Yakus

Riichi *Riichi*
Riichi itself counts as a 1-Han Yaku. An extra Yaku, *Ippatsu*, is awarded for winning before and including the next draw of the Riichi player, without being interrupted by tile calls. Another extra Yaku, *Double Riichi*, is awarded for declaring Riichi in the first turn of the round, with tile calls before the Riichi call.

Fully Concealed Hand *Menzen Tsumo*
Winning from self-draw on a concealed hand (no open melds).

Pinfu *Pinfu*
Concealed hand with 4 sequences and a valueless pair, winning by a two-sided wait on a sequence. Such hand is by rules worth no extra Fu, except the base 20 Fu plus 10 Fu if winning by Ron.

Pure Double Sequences *Iipeikou*
Concealed hand with two identical sequences, i.e. the same values in the same suit.

All Simples *Tanyao*
Hand with no terminals and honours.

Seat Wind *Yakuhai*
Triplet/quad of the player's seat wind.

Prevalent Wind *Yakuhai*
Triplet/quad of the prevalent wind.

Dragon Triplet *Yakuhai*
Triplet/quad of a dragon.

Robbing the Kan *Chankan*
Winning on another player's Add Kan. Robbing the Kan also applies to a concealed Kan in the case of winning on Thirteen Orphans.

After a Kan/Kita *Rinshan Kaihou*
Winning on an extra draw after declaring a Kan or Kita. Counts as self-draw.

Under the Sea *Haitei Raoyue*
Winning on self-draw on the last tile in the wall.

Under the River *Houtei Raoyui*
Winning on the last discarded tile.

2-Han Yakus

Outside Hand *Chanta*
All Mentsu contain terminals or honours, and the pair is terminals or honours. The hand must contain at least one honour and one sequence. Deducts 1 Han if the hand is not concealed.

Pure Straight *Ikkitsuukan*
Hand with three consecutive sequences in the same suit, e.g.:

Deducts 1 Han if the hand is not concealed.

Mixed Triple Sequences *Sanshoku Doujun*
Hand with three sequences with the same numbers across three different suits, e.g.:

Deducts 1 Han if the hand is not concealed. This Yaku is not possible in Sanma.

Mixed Triple Triplets *Sanshoku Doukou*
Hand with three triplets of the same number across three different suits, e.g.:

This Yaku is only possible in 1's or 9's in Sanma.

Three Concealed Triplets *Sanankou*
Hand with three concealed triplets/quads. Note that the entire hand is not required to be concealed.

Three Quads *Sankantsu*
Hand with three quads.

All Triplets *Toitoihou*
Hand with four triplets/quads and a pair.

Little Three Dragons *Shousangen*
Hand with two triplets/quads of dragons and a pair of dragons.

All Terminals and Honours *Honroutou*
Hand containing only terminals and honours.

Seven Pairs *Chiitoitsu*
Concealed hand with seven different pairs. Seven Pairs always scores exactly 25 Fu.

3-Han Yakus

Terminals in All Sets *Junchan*
All Mentsu contain terminals, and the pair is terminals. The hand must contain at least one sequence. Deducts 1 Han if the hand is not concealed.

Half Flush *Honitsu*
Hand with tiles from only one of the three suits, in combination with honours. Deducts 1 Han if the hand is not concealed.

Twice Pure Double Sequences *Ryanpeikou*
Concealed hand with four sequences forming two sets of Pure Double Sequences, e.g.:

Does not combine with Pure Double Sequences or Seven Pairs.

6-Han Yakus

Full Flush *Chinitsu*
Hand composed entirely of tiles from only one of the three suits. No honours allowed. Deducts 1 Han if the hand is not concealed.

Yakuman

Blessing of Heaven *Tenhou*
The East player winning on their initial deal. Concealed Kan or Kita are not allowed.

Blessing of Earth *Chiihou*
A non-East player winning on self-draw in the very first uninterrupted turn. Concealed Kan or Kita are not allowed.

Big Three Dragons *Daisangen*
Hand with three triplets/quads of dragons. In the case of three melded dragon triplets/quads, the player feeding the third meld of dragons must pay the entire hand score if the winner wins by Tsumo, and split the payment with the feeding player if the winner wins by Ron.

Four Concealed Triplets *Suuankou*
Concealed hand with four concealed triplets/quads. Winning by Ron is only allowed in the case of single wait on the pair, which is called *Suuankou Tanki* and can be awarded double Yakuman in some rules.

All Honours *Tsuuiisou*
Hand composed entirely of honour tiles.

All Green *Ryuuiisou*
Hand composed entirely of green tiles, which include 2s, 3s, 4s, 6s, 8s and Hats, e.g.:

All Terminals *Chinroutou*
Hand composed entirely of terminal tiles.

Thirteen Orphans *Kokushi Musou*
Concealed hand with one of each of the 13 different terminal and honour tiles, plus one extra terminal or honour tile, e.g.:

Some rules award double Yakuman for the Kokushi Musou that waits for all 13 terminal or honour tiles, which is called *Kokushi Musou 13-Men Machi*.

Little Four Winds *Shousuushii*
Hand with three triplets/quads of winds and a pair of winds.

Big Four Winds *Daisuushii*
Hand with four triplets/quads of winds. In the case of four melded wind triplets/quads, the player feeding the fourth meld of winds must pay the entire hand score if the winner wins by Tsumo, and split the payment with the feeding player if the winner wins by Ron. Some rules award double Yakuman for Big Four Winds.

Four Quads *Suukantsu*
Hand with four quads. If fourth quad is an Open Kan, the player feeding that Open Kan must pay the entire hand score if the winner wins by Tsumo, and split the payment with the feeding player if the winner wins by Ron.

Nine Gates *Chuuren Poutou*
Concealed hand consisting of the tiles 1112345678999 in the same suit, plus any one extra tile in that suit, e.g.:

Some rules award double Yakuman for the Nine Gates that is already 1112345678999 in the private hand and waits for any of the nine tiles, which is called *Pure Nine Gates* (*Junsei Chuuren Poutou*).

B.5 Scoring: Fu and Hand Score Calculation

Fu

Apart from a Seven Pairs hand that always scores a constant 25 Fu, a winning hand can get 20 base Fu, and another 10 Fu if the hand is concealed and wins by Ron. Triplets and quads in a hand also gets Fu, as shown in the following table:

	Simple tiles	Terminal/honour tiles
Open triplet	2 Fu	4 Fu
Concealed triplet	4 Fu	8 Fu
Open/Add Kan	8 Fu	16 Fu
Concealed Kan	16 Fu	32 Fu

In addition, 2 Fu are added for each of the following:

- Pair of the seat wind
- Pair of the prevalent wind
- Pair of dragons
- Winning on one-sided, middle, or pair wait
- Winning on self-draw, except in the case of Pinfu
- Open hand Pinfu

The Fu value is then rounded up to the nearest ten.

Score calculation

The basic points of a hand is calculated as follows:

$$\text{Basic points} = \begin{cases} \text{Fu} \times 2^{(2+\text{Han})} & \text{if Han} < 5 \text{ and Basic points} < 2000 \\ 2000 & \text{if Han} = 5 \text{ or Basic points} \geq 2000 \text{ (Mangan)} \\ 3000 & \text{if Han} = 6\text{–}7 \text{ (Haneman)} \\ 4000 & \text{if Han} = 8\text{–}10 \text{ (Baiman)} \\ 6000 & \text{if Han} = 11\text{–}12 \text{ (Sanbaiman)} \\ 8000 & \text{if Yakuman or Han} \geq 13 \text{ (Kazoe Yakuman)} \\ 8000n & \text{if } n \text{ Yakuman conditions are met} \end{cases}$$

- When the dealer wins by Ron, the feeding player pays the winner $6 \times$ Basic points,
- when the dealer wins by Tsumo, each of the non-dealer players pays the winner $2 \times$ Basic points;
- when a non-dealer wins by Ron, the feeding player pays the winner $4 \times$ Basic points;

- when a non-dealer wins by Tsumo, the dealer pays the winner 2 × Basic points, and each of the other non-dealer players pays the winner 1 × Basic points.

This is then rounded up to the nearest hundred. The winner also gets a bonus 200 × Honba counter points, which is split between the paying players. This gives the following scoring tables:

Dealer					Non-dealer			
4 Han	3 Han	2 Han	1 Han		1 Han	2 Han	3 Han	4 Han
— (2600)	— (1300)	— (700)	—	20 Fu	—	— (400/700)	— (700/1300)	— (1300/2600)
9600 (3200)	4800 (1600)	2400 (—)	—	25 Fu	—	1600 (—)	3200 (800/1600)	6400 (1600/3200)
11600 (3900)	5800 (2000)	2900 (1000)	1500 (500)	30 Fu	1000 (300/500)	2000 (500/1000)	3900 (1000/2000)	7700 (2000/3900)
Mangan	7700 (2600)	3900 (1300)	2000 (700)	40 Fu	1300 (400/700)	2600 (700/1300)	5200 (1300/2600)	Mangan
Mangan	9600 (3200)	4800 (1600)	2400 (800)	50 Fu	1600 (400/800)	3200 (800/1600)	6400 (1600/3200)	Mangan
Mangan	11600 (3900)	5800 (2000)	2900 (1000)	60 Fu	2000 (500/1000)	3900 (1000/2000)	7700 (2000/3900)	Mangan
Mangan	Mangan	6800 (2300)	3400 (1200)	70 Fu	2300 (600/1200)	4500 (1200/2300)	Mangan	Mangan
Mangan	Mangan	7700 (2600)	3900 (1300)	80 Fu	2600 (700/1300)	5200 (1300/2600)	Mangan	Mangan
Mangan	Mangan	8700 (2900)	4400 (1500)	90 Fu	2900 (800/1500)	5800 (1500/2900)	Mangan	Mangan
Mangan	Mangan	9600 (3200)	4800 (1600)	100 Fu	3200 (800/1600)	6400 (1600/3200)	Mangan	Mangan
Mangan	Mangan	10600 (3600)	5300 (1800)	110 Fu	3600 (900/1800)	7100 (1800/3600)	Mangan	Mangan

Han/Fu value	Name	Dealer	Non-dealer
3 Han, ≥ 70 Fu or 4 Han, ≥ 40 Fu or 5 Han	Mangan	12000 (4000)	8000 (2000/4000)
6–7 Han	Haneman	18000 (6000)	12000 (3000/6000)
8–10 Han	Baiman	24000 (8000)	16000 (4000/8000)
11–12 Han	Sanbaiman	36000 (12000)	24000 (6000/12000)
Yakuman or ≥ 13 Han	Yakuman	48000 (16000)	32000 (8000/16000)

Appendix C

Gradient-Based Learning of Neural Networks

C.1 Maximum Likelihood and Loss Function

For an NN, the hypothesis $h_{\mathbf{w}}$ is specified by a vector of weights:[2,3]

$$\mathbf{w}^T = \begin{bmatrix} w_1 & w_2 & \cdots & w_W \end{bmatrix}$$

An NN tries to choose $\mathbf{w}_{\text{opt}}$ such that the *likelihood* for the full data set is maximised; that is, the data $\mathbf{s}$ becomes most probable if $\mathbf{w}_{\text{opt}}$ had been used to generate it. This gives us the *maximum likelihood algorithm*:

$$\mathbf{w}_{\text{opt}} = \arg\max_{\mathbf{w}} p(\mathbf{s}|\mathbf{w})$$

Assume the input feature vectors are independent from the weights. By solving $p(\mathbf{s}|\mathbf{w})$, we get:

$$\begin{aligned}
\mathbf{w}_{\text{opt}} &= \arg\max_{\mathbf{w}} p(\mathbf{s}|\mathbf{w}) \\
&= \arg\max_{\mathbf{w}} \prod_{i=1}^{m} p(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}|\mathbf{w}) \\
&= \arg\max_{\mathbf{w}} \prod_{i=1}^{m} \Pr(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) p(\mathbf{x}^{(i)}|\mathbf{w}) \\
&= \arg\max_{\mathbf{w}} \prod_{i=1}^{m} \Pr(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) p(\mathbf{x}^{(i)})
\end{aligned}$$

Now define the likelihood as a probability distribution

$$\Pr(Y|\mathbf{x}, \mathbf{w}) = \sigma_{\text{out}}(h_{\mathbf{w}}(\mathbf{x}))$$

where σ_{out} is an activation function maintaining the properties of a distribution:[4]

[2]The biases can also be regarded as weights, as if we include an additional constant feature $x_0 = 1$.

[3]When the NN has multiple layers, we can flatten the weight matrix to form a vector.

[4]Here I used σ_{out} to distinguish this output activation function from those used in the hidden layers, as described in Section 2.3.2.

- For all $k = 1, 2, \cdots, K$, $0 \leq \Pr(Y = c_k|\mathbf{x}, \mathbf{w}) \leq 1$;
- $\sum_{k=1}^{K} \Pr(Y = c_k|\mathbf{x}, \mathbf{w}) = 1$.

For example, σ_{out} can be the softmax function:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, 2, \cdots, K \text{ and } \mathbf{z}^T = \begin{bmatrix} z_1 & z_2 & \cdots & z_K \end{bmatrix} \in \mathbb{R}^K$$

Let

$$\left(\mathbf{y}^{(i)}\right)^T = \begin{bmatrix} y_1^{(i)} & y_2^{(i)} & \cdots & y_K^{(i)} \end{bmatrix} = \begin{bmatrix} 1_{\mathbf{x}^{(i)} \in c_1} & 1_{\mathbf{x}^{(i)} \in c_2} & \cdots & 1_{\mathbf{x}^{(i)} \in c_K} \end{bmatrix}$$

be the label encoded by a *one-hot* vector, where $1_{\{\cdot\}}$ denotes the *indicator function*:

$$1_{\mathsf{True}} = 1, \ 1_{\mathsf{False}} = 0$$

and

$$\hat{\mathbf{y}}^T = \begin{bmatrix} \hat{y}_1 & \hat{y}_2 & \cdots & \hat{y}_k \end{bmatrix} = \sigma_{\text{out}}(h_{\mathbf{w}}(\mathbf{x}))$$

be the model's predicted output. Therefore

$$\Pr(Y = c_j|\mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} \hat{y}_k^{1_{k=j}}$$

Substituting this back to $\mathbf{w}_{\text{opt}}$ and then taking logarithms, we can get:

$$\begin{aligned}
\mathbf{w}_{\text{opt}} &= \arg\max_{\mathbf{w}} \prod_{i=1}^{m} \Pr(\mathbf{y}^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) p(\mathbf{x}^{(i)}) \\
&= \arg\max_{\mathbf{w}} \prod_{i=1}^{m} \left(\prod_{k=1}^{K} (\hat{y}_k^{(i)})^{1_{\mathbf{y}^{(i)} = \mathbf{c}_k}} \right) p(\mathbf{x}^{(i)}) \\
&= \arg\max_{\mathbf{w}} \sum_{i=1}^{m} \left(\sum_{k=1}^{K} 1_{\mathbf{y}^{(i)} = \mathbf{c}_k} \cdot \log \hat{y}_k^{(i)} + \log p(\mathbf{x}^{(i)}) \right) \\
&= \arg\max_{\mathbf{w}} \sum_{i=1}^{m} \sum_{k=1}^{K} 1_{\mathbf{y}^{(i)} = \mathbf{c}_k} \cdot \log \hat{y}_k^{(i)} \\
&= \arg\min_{\mathbf{w}} \left(- \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \cdot \log \hat{y}_k^{(i)} \right)
\end{aligned}$$

This is known as the *categorical cross-entropy loss function.*

C.2 Back-propagation

In Chapter 2, I explained forward propagation that feeds the input $\mathbf{x}$ to an NN and propagates it through the hidden layers and finally produces a prediction $\hat{\mathbf{y}}$, the loss function that $\hat{\mathbf{y}}$ should be fed into and the NN should minimise to obtain $\mathbf{w}_{\text{opt}}$, and how the NN can achieve this by using gradient descent. It is clear that the derivative of the

loss function with respect to the weights, $\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}}$, is crucial to the entire learning process. This can be calculated using *back-propagation*: consider a connection from perceptron i to perceptron j, carrying a weight $w_{i\to j}$. Since the $L(\mathbf{w})$ is a sum of contributions by each example in $\mathbf{s}$, we can consider the examples individually:

$$\frac{\partial L(\mathbf{w})}{\partial w_{i\to j}} = \sum_{x=1}^{m} \frac{\partial L_x(\mathbf{w})}{\partial w_{i\to j}}$$

According to the *chain rule*:

$$\frac{\partial L_x(\mathbf{w})}{\partial w_{i\to j}} = \frac{\partial L_x(\mathbf{w})}{\partial a_j}\frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial w_{i\to j}} \tag{C.1}$$

Since we have

$$\frac{\partial a_j}{\partial z_j} = \frac{\partial}{\partial z_j}\sigma(z_j) = \sigma'(z_j)$$

$$\frac{\partial z_j}{\partial w_{i\to j}} = \frac{\partial}{\partial w_{i\to j}}\left(\sum_{k\in \mathrm{pred}(j)} w_{k\to j}a_k\right) = a_i$$

Equation (C.1) can be simplified as:

$$\frac{\partial L_x(\mathbf{w})}{\partial w_{i\to j}} = a_i\sigma'(z_j)\frac{\partial L_x(\mathbf{w})}{\partial a_j} \tag{C.2}$$

If j is an output perceptron, then

$$\frac{\partial L_x(\mathbf{w})}{\partial a_j} = \frac{\partial L_x(\mathbf{w})}{\partial \hat{y}} \tag{C.3}$$

If j is an inner perceptron, then

$$\frac{\partial L_x(\mathbf{w})}{\partial a_j} = \sum_{k\in \mathrm{next}(j)} \frac{\partial L_x(\mathbf{w})}{\partial a_k}\frac{\partial a_k}{\partial z_k}\frac{\partial z_k}{\partial a_j} \tag{C.4}$$

where

$$\frac{\partial a_k}{\partial z_k} = \sigma'(z_k) \tag{C.5}$$

$$\frac{\partial z_k}{\partial a_j} = \frac{\partial}{\partial a_j}\left(\sum_{l\in \mathrm{pred}(k)} w_{l\to k}a_l\right) = w_{j\to k} \tag{C.6}$$

Substituting Equations (C.3), (C.4), (C.5) and (C.6) into Equation (C.2), we can get the following results in summary:

$$\frac{\partial L_x(\mathbf{w})}{\partial w_{i\to j}} = a_i\delta_j$$

where

$$\delta_j = \frac{\partial L_x(\mathbf{w})}{\partial a_j}\frac{\partial a_j}{\partial z_j} = \begin{cases} \sigma'_{\mathrm{out}}(z_j)\frac{\partial L_x(\mathbf{w})}{\partial \hat{y}} & \text{if } j \text{ is an output perceptron} \\ \sigma'(z_j)\sum_k w_{j\to k}\delta_k & \text{if } j \text{ is an inner perceptron} \end{cases}$$

Using this algorithm, the gradient information of $L(\mathbf{w})$ can efficiently flow backwards through the network, and the gradients can be computed and used to update the weights layer by layer.

In Meowjong's case, $L(\mathbf{w})$ is the categorical cross-entropy loss function, σ_{out} is the softmax function, and σ is the ReLU function, and we have:

$$\frac{\partial L_x(\mathbf{w})}{\partial \hat{\mathbf{y}}} = \begin{bmatrix} -\dfrac{y_1}{\hat{y}_1} & -\dfrac{y_2}{\hat{y}_2} & \cdots & -\dfrac{y_K}{\hat{y}_K} \end{bmatrix}^T$$

$$\frac{\partial\, \sigma_{\text{out}}(\mathbf{z})}{\partial z_j} = \begin{bmatrix} \sigma_{\text{out}}(z_1)(1_{j=1} - \sigma_{\text{out}}(z_j)) \\ \sigma_{\text{out}}(z_2)(1_{j=2} - \sigma_{\text{out}}(z_j)) \\ \vdots \\ \sigma_{\text{out}}(z_K)(1_{j=K} - \sigma_{\text{out}}(z_j)) \end{bmatrix} = \begin{bmatrix} \hat{y}_1(1_{j=1} - \hat{y}_j) \\ \hat{y}_2(1_{j=2} - \hat{y}_j) \\ \vdots \\ \hat{y}_K(1_{j=K} - \hat{y}_j) \end{bmatrix}$$

$$\sigma'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \\ \text{undefined} & \text{if } z = 0 \end{cases}$$

Therefore

$$\delta_j = \begin{cases} \hat{y}_j - y_j & \text{if } j \text{ is an output perceptron} \\ \sum_k w_{j\to k}\delta_k & \text{if } j \text{ is an inner perceptron, and } z_j > 0 \\ 0 & \text{if } j \text{ is an inner perceptron, and } z_j < 0 \\ \text{undefined} & \text{if } j \text{ is an inner perceptron, and } z_j = 0 \end{cases}$$

In practice, the non-differentiability of the ReLU function at 0 rarely occurs, and can be avoided by arbitrarily choosing one of the one-sided derivatives. Therefore, ReLU is still considered to be a very powerful activation function, with many advantages including computational efficiency, the ability to avoid the *vanishing gradient problem*, and better convergence performance [29].

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Deep Reinforcement Learning for Mahjong

22 October 2020

Project Originator: 2376D

Project Supervisor: Dr Sean Holden

Directors of Studies: Dr Sean Holden, Dr Neel Krishnaswami, and Prof Frank Stajano

Project Overseers: Dr Jamie Vicary and Prof Andrew Rice

D.1 Introduction

Mahjong is a very popular tile-based imperfect-information game that was developed in China in the late 19th century, and has hundreds of millions of players worldwide nowadays. It is a game of skill, strategy and calculation, and it involves a degree of chance. Mahjong is commonly played with four players, and in each round of a Mahjong game, the players compete against each other towards the first completion of a winning hand. The players take turns to draw and discard tiles until one of them completes a winning hand first or all tiles are drawn and discarded, and they can also claim a discarded tile from another player to complete an open meld, which may interrupt the regular playing order. A more detailed overview of Mahjong can be found on its Wikipedia page [9]. Mahjong is very challenging for AI research due to its following features:

1. Mahjong has a huge set of hidden information: in each round, each player can only see their own hand, open melds of all players, and the tiles discarded by all players by far. The player cannot see the other players' private hands, or the tiles that are not yet drawn. This gives an average of more than 10^{48} hidden states indistinguishable to a player on each decision point [7]. Since the value of an action for a player depends not only on their own hand, but also on the hands of the other players and the remaining tiles, it is hard for an AI to derive a reward system solely from the observed information.

2. Mahjong has very complicated playing rules:

   (a) There are various types of actions, including discard, Chii, Pon, Kan, etc.;

   (b) The regular playing order can be interrupted by certain actions.

   Since there are 13 tiles in each player's hand in most of the common variations of Mahjong, it is hard to predict those interruptions, and therefore it would be very unrealistic to try to build a game tree for Mahjong, which makes tree-search based techniques for games not applicable to Mahjong.

3. Mahjong has a huge number of possible winning hands, which can be very different from each other, and a more difficult hand is worth a higher score. Therefore, a professional Mahjong player needs to carefully choose what kind of winning hand to complete, in order to make a trade-off between the winning probability and the winning score of each round.

4. Mahjong has complicated scoring rules: a full game of Mahjong contains multiple rounds, and the final ranking of the game is determined by the accumulated round scores of all the rounds. Therefore the loss of one round does not necessarily mean that the round is played poorly: for example, a player may decide to always discard the safe tile in order not to feed other players, or to tactically lose the last round to secure top rank if they have already gained a big advantage in the previous rounds. This makes the reward system of a Mahjong AI even harder to build.

A very popular variation of Mahjong is the Japanese Riichi Mahjong, which is played by 4 players with 136 tiles of 34 different kinds. Each player's private hand consists of 13 tiles. While the basic rules of Mahjong still apply to Riichi Mahjong, it also features some additional rules, including Riichi (an extra action in addition to discard, Chii, Pon and Kan, meaning declaring a ready hand) and Dora (bonus tiles). The full Riichi Mahjong rules published by the European Mahjong Association can be found in their official website [10].

During 2019–2020, Microsoft developed a Mahjong AI called Suphx, using deep reinforcement learning [7]. Suphx has demonstrated a stronger performance than most top human players in terms of stable rank on Tenhou, a popular global online Riichi Mahjong platform with more than 350,000 active users, and is rated above 99.99% of all the officially ranked human players in the Tenhou platform. This is the first time that a computer program outperforms most top human players in Mahjong. Prior to Suphx, there were also several attempts on Mahjong with various techniques [5, 6, 8], but they were all far behind top human players.

In my project, I propose to develop an AI for a reduced, 3-player version of Mahjong (also named Sanma in Japanese), using deep reinforcement learning. The reduction of 3-player Mahjong compared to the full, 4-player Riichi Mahjong mainly consists of the following aspects:

1. 7 kinds of tiles (28 tiles in total) are removed, effectively removing almost an entire suite;

2. Because of the above reduction in tiles, the number of winning patterns are also reduced;

3. The Chii action is disabled;

4. Although there's a new action called Kita, its decision flow is much simpler than Chii, and the number of possible occurrences of Kita in a round (maximum four times per round) is much smaller than that of Chii.

The reason that I choose 3-player Riichi Mahjong for my project are as follows:

1. Riichi Mahjong, no matter the 4-player version or the 3-player version, has a standardised, clearly documented set of rules, which provides a clear reference when implementing the game simulator and the score calculator;

2. There are a lot of popular online Mahjong platforms featuring Riichi Mahjong, such as Tenhou and Majsoul, with millions of active players in total, and there are an adequate amount of game logs available to download as training data;

3. As a Riichi Mahjong player myself, I am much more familiar with the rules of this variation, and can be more fluent in implementing its rules, and the decision flow of my AI;

4. Compared to 4-player Riichi Mahjong, 3-player Riichi Mahjong is easier to implement, and is easier for an AI to learn, therefore I can expect a better result eventually;

5. Even though the amount of game logs for 3-player Riichi Mahjong is much less than that of 4-player Riichi Mahjong, there should still be enough data to train my AI.

Also, instead of trying to optimise the entire multi-round game ranking like Suphx, which may introduce a lot of extra difficulties, in my project, I propose to only focus on optimising the outcome of each single round. This would naturally eliminate the fourth challenge as mentioned before, though the overall performance of my Mahjong AI in the full multi-round game may be inevitably compromised.

In 3-player Riichi Mahjong, other than declaring a win, there are 5 different types of actions in total that a player can take: discard, Pon, Kan, Kita and Riichi. The discard action takes place at every turn, where a player needs to choose a tile from their private hand and discard it; the rest of the actions are, in contrast, only triggered under certain conditions (for example, when another player discards a specific tile), and the player needs to decide whether to declare or skip that action. Therefore, I plan to train 5 separate neural networks, each corresponding to one action, and implement a decision flow to properly place those 5 neural networks. Then, I would use self-play reinforcement learning to enhance the discard model only, as that is the most common action in a round of Mahjong game. I shall implement my own script to download game logs from the online Mahjong platforms (possibly also a web crawler to fetch real-time game logs), and I shall implement my own simulator for the self-play, and the evaluation later on. I will explain in more detail about the substance and structure of my project in Section D.3 below.

D.2 Starting Point

The *Part IA Machine Learning and Real-world Data* course provides a foundation of statistical testing which would be used in my evaluation, and the *Part IB Artificial Intelligence* and *Part II Machine Learning and Bayesian Inference* courses provides essential knowledge in machine learning. Since the later course takes place in the Lent Term, at this stage, I shall read last year's lecture notes, alongside with other literature, to understand the theory covered in that course. Unfortunately, the *Part II Deep Neural Networks* unit of assessment in the Lent Term does not have any available materials from previous years, and therefore cannot serve as my starting point.

Other than the Tripos courses, the Microsoft paper on Suphx [7] provides a concrete reference to start with when designing my own machine learning models.

D.3 Substance and Structure of the Project

D.3.1 Decision flow

As mentioned before, in a 3-player Riichi Mahjong, there are 5 possible actions in total that a player can take, other than declaring a winning hand: discard, Pon, Kan, Kita and Riichi. The discard action takes place at every turn, where a player needs to choose a tile from their private hand and discard it. A player can Pon by claiming a tile that has just been discarded by another player to complete a triplet meld. A player can Kan by claiming a tile that has just been discarded by another player to complete a quad meld, or using a tile drawn by the player themselves to complete a quad meld. A player can Kita after drawing a tile, by replacing a North tile in their hand with another tile, and the North tile would be counted as Dora. A player can declare Riichi after drawing a tile, meaning declaring a ready hand. Therefore, it seems natural that 5 separate neural networks should be trained for each action. When a player draws a tile, there are 5 possible actions that the player can take: win, Riichi, Kan, Kita or discard. When another player discards a tile, there are 4 possible actions that the current player can take: win, Pon, Kan, or skip (do nothing). The win decision for my AI is trivial: the AI should always declare and win when it can do so, unless it is the final round and the AI's total score with the winning hand score added up is still the lowest among the 3 players. For the other possible actions, a decision flow can be made:

- When the AI draws a tile:
  - If the AI can Riichi, then it should go through the Riichi model to decide whether to declare Riichi or not. After that, it should go to the discard step to discard a tile.
  - If the AI can Kan/Kita, then it should go through the Kan/Kita model to decide whether to Kan/Kita or not. If the Kan/Kita model decides to Kan/Kita, the AI makes the Kan/Kita and waits for other players' actions (other players may win by robbing this Kan/robbing this North tile). If another player wins

in this way, the round is over; otherwise, the AI draws another tile and return to the beginning of the draw situation. If the Kan/Kita model decides not to Kan/Kita, the AI should go to the discard step to discard a tile.

- In the discard step, the AI should go through the discard model to decide which tile to discard. Then, it discards the chosen tile and then it is the other players' actions, or the round ends with no tiles left.
- If multiple actions are suggested, the AI should choose the action with the highest confidence score outputted by the models.

- When another player discards a tile:
  - If the AI can Pon, it should go through the Pon model to decide whether to Pon or not. If the Pon model decides to Pon, the AI waits for other players' actions (other players may win and win has the highest priority) before making the Pon, and then go to the discard step to discard a tile. Otherwise, the AI does nothing.
  - If the AI can Kan, it should go through the Kan model to decide whether to Kan or not. If the Kan model decides to Kan, the AI waits for other players' actions (other players may win and win has the highest priority) before making the Kan, and then draw a tile and go to the draw situation. Otherwise, the AI does nothing.
  - The discard step here is identical to the discard step described in the draw situation.
  - If multiple actions are suggested, the AI should choose the action with the highest confidence score outputted by the models.

D.3.2 Supervised learning phase

Since deep convolutional neural networks (CNNs) have demonstrated powerful representation capability and been verified in playing games like chess, shogi and Go, and is the architecture used for Suphx [7], I would also use CNNs as the architecture of my 5 models. I would fist adopt Suphx's structure to my models as a starting point, and then make changes when necessary.

I plan to implement my own scripts to download massive game logs among top ranked players in Tenhou and Majsoul, and use them as the training data for my neural networks. If necessary, I would also implement my own web crawler to download real-time game logs updated on Tenhou and Majsoul, to increase the size of my training data. Since those downloaded game logs need to be converted to the $(state, action)$ pairs format before being fed to the neural networks, I shall also implement my own format converter for this purpose.

D.3.3 Reinforcement learning phase

Once all the neural networks are fully pre-trained, I shall improve them further through self-play reinforcement learning, with the models as policy. According to Microsoft's experience in training Suphx, though the Riichi, Chii, Pon and Kan models can also be improved by self-play reinforcement learning, their improvements are not as significant as the discard model and may not be worth the time and computation power [7]. Therefore, my AI shall follow Suphx's practice in the reinforcement learning phase and directly inherit the Riichi, Pon, Kan and Kita models trained by supervised learning, while only aiming at enhancing the discard model. My AI shall initialise its discard model as the discard model pre-trained in the supervised learning phase, and improve it using the popular policy gradient algorithm. I shall implement my own Mahjong simulator for the self-play process.

D.4 Success Criteria

D.4.1 Core Goals

First and foremost, this project shall only be deemed *complete* if *it can make decisions of which action to take, based on its own hand and the previous actions of all players, within an acceptable amount of time.* The time limit for each turn in Tenhou ranking system is 5+10 seconds (or 3+5 seconds if in fast play mode), and is 5+20 seconds in Majsoul ranking system, so a suitable choice of time limit for this project would be between 3–25 seconds.

Once the project is complete, it can be deemed *successful* if *it can outperform an agent that plays randomly.* A randomly-playing agent can be defined with the following behaviours:

1. The agent shall always declare and win a round immediately when it can win;
2. In the discard action, the agent shall always randomly select a tile from its hand and discard it;
3. In the Pon, Kan, Kita and Riichi actions, the agent shall always have a randomly-generated 50/50 chance of whether or not to take the action.

D.4.2 Possible Extensions

Once all the core goals are achieved, the target for my AI's performance can be raised as optional extensions, if the remaining time permits:

1. The extended target for my AI can be set to be outperforming bots in the player versus computer mode on the online Mahjong platforms, which are believed to be not machine learning based and below the level of an average human Mahjong player;
2. Furthermore, the performance of my AI can be set to match against human level, aiming for as high as possible, with no pre-set lower or upper limit.

Another possible extension for this project is to migrate my AI to the full, 4-player Riichi Mahjong, through curriculum learning.

D.5 Evaluation Metrics

D.5.1 Evaluation Metrics for the Core Goals

After the supervised learning phase, all 5 models (the discard, Pon, Kan, Kita and Riichi models) should end up achieving satisfiable train-dev-test accuracies in order to proceed to the reinforcement learning phase.

After both the supervised learning phase and the reinforcement learning phase are completed, to show that the trained agent can outperform a randomly-playing agent, I shall build a randomly-playing agent that satisfies its definition in Section D.4.1, and let my trained agent play Mahjong games against 2 copies of that agent, under the same simulator I have built for the reinforcement learning procedure, as described in Section D.3.3. Then, I shall conduct the following statistical testing:

Null Hypothesis: There is no significant difference between my trained agent and a randomly-playing agent.

Alternative Hypothesis: My trained agent performs significantly better than a randomly-playing agent.

Significance Level: $\alpha = 0.05$

Since the initial private tiles have large randomness and will greatly affect the win/loss of a game, the agents must play a substantial number of games in order to reduce the variance caused by the initial private tiles.

In addition, to evaluate the enhancement from reinforcement learning, I shall include both the following agents in the above statistical testing:

- SL agent: the supervised learning agent with all the 5 models trained in the supervised learning way, as described in Section D.3.2;
- RL agent: the reinforcement learning agent with the, Pon, Kan, Kita and Riichi models inherited from the SL agent, but its discard model is initialised as the SL discard model and enhanced through reinforcement learning, as described in Section D.3.3.

Firstly, both the SL and RL agents should play against the randomly-playing agent for a substantial amount of games, and their 1st/2nd/3rd place rates, hand win rates, and deal in rates against the randomly-playing agent should be compared to show the performance gained by reinforcement learning; then, the SL and RL agents should also play a substantial number of games against each other, and I shall conduct a similar statistical testing to show their difference:

Null Hypothesis: There is no significant difference between the RL agent and the SL agent.

Alternative Hypothesis: The RL agent performs significantly better than the SL agent.

Significance Level: $\alpha = 0.05$

As the RL agent is equivalent to the final trained agent of my project, the result of the RL agent against the randomly-playing agent shall be used as the indicator of my success criteria.

D.5.2 Evaluation Metrics for the Extensions

To show that my AI can outperform bots of the online Mahjong platforms as an optional extension, I shall let my AI play Mahjong against the bots in the player versus computer mode, both on Tenhou and Majsoul, for a substantial number of games, and then conduct a similar statistical testing to show their difference:

Null Hypothesis: There is no significant difference between my AI and a Tenhou/-Majsoul bot.

Alternative Hypothesis: My AI performs significantly better than a Tenhou/Majsoul bot.

Significance Level: $\alpha = 0.05$

To evaluate the performance of my AI versus human level as another optional extension, a suitable indicator would be the stable ranking, or the ELO rating of my AI, after a substantial number of online ranked games versus human players. In this evaluation, I shall use Tenhou as my sole ranking platform. Tenhou is chosen over Majsoul or other online Mahjong platforms as it the most popular and competitive platform for Riichi Mahjong, with a very professional rating system. The completed project should play ranked, 3-player Riichi Mahjong games against other human players in the highest level game lobby it is allowed in Tenhou, without any human intervention, starting from a new account, until it has achieved a stable rating, or the time allowed for me to carry out this optional evaluation has exceeded. Since this project is set to focus only on optimising the outcome of each single round, the most suitable option for the online evaluation should be only playing ranked games with minimum number of rounds, namely the East-only games, which consists of one full cycle and approximately 4 rounds. The final record rank, stable rank and player statistics generated by Tenhou will be used as reference to its final performance versus humans. I shall implement my own scripts to enable my AI to automatically play ranked games on Tenhou.

D.6 Timetable and Milestones

Planned starting date is 10 October 2020. All weeks in this timetable start on Saturday. There will be an 1-hour project supervision with my project supervisor, and a 10-minute DoS meeting to report my project progress to my DoSes, every week during term time.

Weeks 1 – 2 (10 October 2020 – 23 October 2020)

Work:

- Complete the project proposal.
- Set up a project logbook for documenting project work done on every day, and all communications between I and my project supervisor, overseers and DoSes.
- Set up a GitHub repository.

Milestone: Completed project proposal submitted by 12 noon on 23 October 2020.

Weeks 3 – 4 (24 October 2020 – 6 November 2020)

Work:

- Implement a Riichi Mahjong hand score calculator.
- Implement scripts that can automatically download massive game logs from Tenhou and Majsoul.
- Run the scripts to download game logs from Tenhou and Majsoul.

Milestones:

- Implementation of the hand score calculator and the scripts completed;
- At least some game logs downloaded from Tenhou and Majsoul.

Weeks 5 – 6 (7 November 2020 – 20 November 2020)

Work:

- Finalise the data structure to be used for representing the training data.
- Implement a module that converts the game logs into $(state, action)$ pairs format which can be used for training later on.
- Run the module to convert all game logs into $(state, action)$ pairs format.
- Implement the logic flow.

Milestones:

- Implementation of the format converter completed;
- Implementation of the logic flow completed;
- Training data ready to be used to pre-train the 5 neural networks.

Weeks 7 – 8 (21 November 2020 – 4 December 2020)

Work:

- Familiarise with PyTorch.
- Begin to implement the discard, Pon, Kan, Kita and Riichi neural networks with PyTorch.

Milestone: Implementation of the first version of the 5 neural networks completed.

Weeks 9 – 10 (5 December 2020 – 18 December 2020)

End of the Michaelmas Term. This is a buffer time frame at the start of the Christmas Vacation, in case any of the previous work take longer than anticipated.

Work: Pre-train, test and tune the neural networks.

Milestone: The 5 neural networks are improved compared to previous time frame.

Weeks 11 – 12 (19 December 2020 – 1 January 2021)

Work:

- Continue pre-training, testing and tuning the neural networks.
- Read relevant materials on reinforcement learning.

Milestone: The supervised learning phase is completed, and all 5 neural networks can achieve satisfiable train-dev-test accuracies.

Weeks 13 – 14 (2 January 2021 – 15 January 2021)

Work:

- Implement a Mahjong simulator for self-play.
- Implement the randomly-playing opponent agent.
- Begin training my AI through self-play reinforcement learning.
- Evaluate the performance of my AI versus the randomly-playing agents throughout the self-play process.

Milestones:

- Implementation of the Mahjong simulator completed;
- Implementation of the randomly-playing agent completed.

Weeks 15 – 16 (16 January 2021 – 29 January 2021)

Start of the Lent Term.

Work:

- Continue training my AI through self-play reinforcement learning.
- Continue evaluating the performance of my AI versus the randomly-playing agents throughout the self-play process.
- Produce a draft progress report and send it to my project supervisor for feedback.

Milestones:

- The performance of my AI should continue to be improving, if not having already passed the success criteria;
- A completed draft progress report sent to my project supervisor, and feedback obtained from him.

Weeks 17 – 18 (30 January 2021 – 12 February 2021)

Work:

- Continue training my AI through self-play reinforcement learning.
- Continue evaluating the performance of my AI versus the randomly-playing agents throughout the self-play process.
- Complete the progress report.
- Prepare for the progress report presentations.

Milestones:

- My AI is now successfully trained to pass the success criteria;
- Completed progress report submitted by 12 noon on 5 February 2021.

Weeks 19 – 20 (13 February 2021 – 26 February 2021)

Work:

- Begin dissertation planning, starting from gathering materials from my project logbook, source code and documents, and outlining what to include in each chapter in bullet points.
- *Extension:* Implement scripts for my AI to automatically play online games on Tenhou and Majsoul.
- *Extension:* Migrate my AI from 3-player Riichi Mahjong to 4-player Riichi Mahjong through curriculum learning.

Milestones:

- Progress report presentation delivered (if not have done already in the previous time frame).
- *Optional:* Curriculum learning process started.

Weeks 21 – 22 (27 February 2021 – 12 March 2021)

Work:

- Begin writing the dissertation, starting from expanding the bullet points.
- *Extension:* Evaluate the performance of my AI versus Tenhou and Majsoul bots by playing player versus computer mode games.
- *Extension:* Evaluate the performance of my AI versus human by playing ranked games on Tenhou.
- *Extension:* Continue migrating my AI from 3-player Riichi Mahjong to 4-player Riichi Mahjong through curriculum learning.
- *Extension:* Evaluate the performance of my migrated AI on 4-player Riichi Mahjong.

Milestones:

- Finalised dissertation outline (in bullet points) sent to project supervisor for feedback.
- *Optional:* Documented evaluation for extensions.

Weeks 23 – 24 (13 March 2021 – 26 March 2021)

End of the Lent Term. The project supervision on Week 23 should be about the dissertation, in case my project supervisor is unreachable during the Easter Vacation. This is a buffer time frame at the start of the Easter Vacation, in case any of the previous work take longer than anticipated, or there is any outstanding extension work unfinished.

Work:

- Write the Introduction and Preparation chapters of the dissertation.
- Finish the optional extensions if started.

Milestones:

- Source code finalised.
- First draft of the Introduction chapter completed.

Weeks 25 – 26 (27 Mar 2021 – 9 April 2021)

Work:

- Write the Preparation, Implementation and Evaluation chapters of the dissertation.
- Send my draft dissertation chapter by chapter to my project supervisor for feedback.
- Amend my draft dissertation based on the feedback from my project supervisor.

Milestone: First draft of the Preparation and Implementation chapters completed.

Weeks 27 – 28 (10 April 2021 – 23 April 2021)

Work:

- Write the Evaluation and Conclusions chapters of the dissertation.
- Send my draft dissertation chapter by chapter to my project supervisor for feedback.
- Amend my draft dissertation based on the feedback from my project supervisor.

Milestone: First draft of the full dissertation completed and sent to the project supervisor for discussion at the start of the Easter Term.

Weeks 29 – 30 (24 April 2021 – 7 May 2021)

Start of the Easter Term.

Work:

- Send my draft dissertation in full to my project supervisor for feedback.
- Amend my draft dissertation based on the feedback from my project supervisor.

Milestone: Final draft of the dissertation completed or almost completed.

Weeks 31 – 32 (8 May 2021 – 19 May 2021)

This is a buffer time frame for the final review and refinements of the dissertation.

Work:

- Finalise the dissertation.
- Complete the supervisors' report form with my project supervisor and DoSes.

Milestones:

- Completed dissertation submitted by 12 noon on 14 May 2021, and the source code submitted by 4:00 pm on the same day.
- Completed supervisors' report form submitted by 4:00 pm on 19 May 2021.

D.7 Resources Declaration

My personal laptop

Microsoft Surface Book 2, with Intel Core i7-8650U CPU @ 1.90GHz, 16GB RAM, 256GB SSD, and Windows 10 with Ubuntu on WSL. I plan to use this device for code development and dissertation writing. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

My contingency plans against data loss are that all source code shall be held under GitHub, and everything shall be regularly uploaded to my Google Drive and OneDrive, and copied to my USB kept only for project purposes.

My contingency plans against hardware/software failure are that in the case of unrepairable hardware/software failure on my laptop, I shall make an emergency purchase of an economical, possibly second-handed, laptop that meets the minimum hardware requirements, and retrieve all data from my various backup sources, so that I can resume working on the project as quickly as possible, and reduce the impact of the hardware/-software failure down to a minimum level.

High performance computing services from the Department / University

According to the Suphx paper, the training of each Suphx agent took 44 GPUs (4 Titan XP for the parameter server and 40 Tesla K80 for self-play workers) and 2 days, and the evaluation of each Suphx agent took another 20 Tesla K80 GPUs and 2 days [7], which is quite resource heavy. Since my project tackles a reduced version of Riichi Mahjong than Suphx, and aims at a much lower goal (i.e. outperforming a randomly-playing agent, rather than a top human Mahjong player), I shall expect a much lower resource demand than Suphx, but it is still likely that I may need to train my neural networks using the Department's high performance computing services or the Cambridge HPCS.

Extra disc space on the MCS

It is likely that this may be required for storing the downloaded game logs, and the re-formatted training data converted from them.