Deep Reinforcement Learning for Big Two

CHOW, Hau Cheung Jasper
hcjchow@connect.ust.hk

CHENG, Hoi Chuen
hcchengaa@connect.ust.hk

1. Introduction

Games serve as the foundation of society. Numerous scenarios ranging from political science to project management [1] can often be modelled by a set of agents; each with their own objective, which may or may not overlap with other agents' objectives. These agents operate within an environment defined by some set of rules. The complex nature of both inter-agent interactions in the environment makes the application of artificial intelligence to games a lucrative field of research.

1.1. A History of Reinforcement Learning in Card Games

Mathematically, a game can be defined using a game tree [2], a graph which represents all possible game states within a game, with the edges of the graph denoting the actions taken either by an agent or the environment itself.

Reinforcement learning has made great strides in many games, such as establishing state-of-the-art performance in Chess and Go [3]. However, we focus on **imperfect-information games** since they represent a wider array of real-life situations. An imperfect-information game is one where each agent cannot observe the entire state of the environment in which they operate [4]. As such, recent works have focused on investigating imperfect-information games with large computational action and state spaces, such as No-Limit Heads Up Texas Hold' Em ("HUNLHE") poker [5], [6], [7] and Mahjong [8], among others.

1.2. Limitations of Prior Approaches

A common challenge faced by reinforcement learning agents in imperfect-information games is the large size of the state-action space. However, the state-action space is typically abstracted. For example, in HUNLHE, the continuous nature of bet sizes in each betting round causes the game tree to become exponentially large. (A bet size of \$500 and \$501 form two distinct subtrees of similar size within the overall game tree.) Therefore, [7] applied state abstraction to group similar bet sizes to drastically reduce the game tree size. However, such state abstraction is often game-specific and thus not always possible.

Additionally, an open problem in reinforcement learning attempts to balance collaboration and competition. [9] demonstrated the effectiveness of Deep Monte Carlo methods for training agents to play Dou Dizhu, which requires two 'peasant' players to collaborate against the 'landlord' player in a partially observable environment with limited communication. However, Dou Dizhu features clearly defined objectives once the roles of 'peasant' and 'landlord' are assigned, because both peasants win if either of the two peasants is the first to shed their entire hand. In contrast, [10] showed that temporary collaboration between multiple self-interested agents is ubiquitous in real-life scenarios. Therefore, developing strategies that involve temporary collaboration between agents will prove to be useful for modeling such situations.

1.3. Background of Big Two

This paper attempts to apply reinforcement learning methods to Big Two (also known as Chor Dai Di or deuces), one of the most popular card-shedding games in Southeast Asia. The game is played between 4 players with a standard 52-card deck, and each player is randomly dealt a hand of 13 cards.

Although we could generate game data and model Big Two as a supervised learning problem to predict an action for every state, due to the large size of the state-action space, we would have low generalisability on unseen state-action pairs.

We introduce an abstracted environment to simulate Big Two's rules with a variety of adjustable parameters such as the severity of the penalties imposed upon losing with certain cards remaining in hand, as well as popular variants of the 5-card hand rankings (such as the ordering of straights) which we expect to greatly influence the strategies of the agents. For a list of the full rules and popular variations, please refer to [11].

Although Big Two is commonly compared to the aforementioned Dou Dizhu, it poses challenges to the current reinforcement learning landscape due to the **long-term planning** required to balance the trade-off between flexibility and risk in the late-game. This is because the actions taken or not taken in the early-game are correlated with the set of available actions in the late game. And the early phase of the game is when the agent can observe the most information and has a more accurate picture of what their expected return might be. This is exacerbated by the fact that the penalty for losing increases when there are more unplayed cards the player has. This is unlike Dou Dizhu, where the penalty for losing and winning is fixed once a winner is decided. Therefore, although more cards usually means more manoeuvrability later in the game (when other players are likely to have exhausted their own options), there is a greater risk of incurring a high penalty, however, exhausting one's strong cards early is likely to lead to a guaranteed but smaller loss in the late-game. Finding the balance between these two objectives is a key challenge our reinforcement learning agent must face.

Furthermore, Big Two addresses **the fluid and temporary nature of collaboration** between agents. For instance, it is common for players to "pass" when the opponent before them (upstream-player) has just played in the hopes of preserving higher ranked cards for later and having the opportunity to play second in case the upstream player before them "takes the lead" (gets to play first) on the next round. Conversely, players tend to not "pass" (if they are able) if the player after them (downstream-player) has played because there is a high chance that if the player following them "takes the lead", they will have to play last on the subsequent round. However, agents may not want to give the upstream-player a chance to "pass" if that player is low on cards as they are likely to win if they "take the lead". The nuance in this decision means that each given agent cannot treat all other competing agents equally; each has an inherent advantage or disadvantage in terms of their **relative position** to another agent.

Throughout this work, we will refer to cards as [Rx] where R denotes the rank R $\in$ {3,4,5,6,7,8,9,T,J,Q,K,A,2} and x $\in$ {c,d,h,s} denotes the suits (clubs, diamonds, hearts, and spades.)

2. Data

Using a shuffled array of length 52 to represent the deck of cards, 13-card hands are randomly assigned to each of the 4 players and passed into the `Env` class, which encapsulates all game behaviours in Big Two, such as identifying all legal actions and computing rewards.

During training, data is repeatedly fed to actor processes (see 4.1) whenever an actor process finishes computing the estimated $Q(s, a)$ of all (s, a) in the episode.

During evaluation, data for 15000 Big Two games will be randomly generated.

3. Methodology

There are various algorithms to solve an imperfect-information card game. We chose to implement the **Deep Monte Carlo (DMC)** algorithm, which generalises the Monte Carlo (MC) method to deep neural networks. The Monte Carlo method is a reinforcement learning algorithm based on the principle of conducting repeated random sampling and averaging the sampled returns. It is most effective for episodic tasks, where an **episode** is defined a sequence of states, actions, and rewards experienced between an initial state and a terminal state.

Since each game of Big Two is finitely long, there are no incomplete episodes (i.e. there is always an initial and terminal state), and we can partition the experience to be learned by the agent into a series of episodes, we believe MC methods to be suitable for Big Two.

More formally, the every-visit MC algorithm optimizes a policy π by estimating a Q-table $Q(s, a)$ for all states $s \in S$ and all actions available $a \in A$ by repeatedly executing the following:

1. Generate an episode using π with ϵ-greedy exploration
2. For each (s, a) appearing in the episode, calculate and update $Q(s, a)$ with the return averaged over all the samples concerning (s, a).
3. For each s in the episode, $\pi(s) \leftarrow argmax_a Q(s, a)$

In MC methods, the policy π is updated only after each episode rather than after every action. Instead of using a large, global $Q(s, a)$ table to store these values, which requires a large amount of memory and is not scalable to the large state-action space present in Big Two, DMC builds on MC methods by using a deep Q-network to approximate $Q(s, a)$.

One disadvantage of DMC methods is that they have very high variance [12]. However, by the Law of Large Numbers, the use of parallel actor processes to generate many samples per second and training for a sufficiently large number of frames can help mitigate this problem.

It should be noted that if an agent always acts greedily, (i.e. always selects the action $a_{greedy} = argmax_a Q(s, a)$ that maximises the Q-table column) at every time step, it can easily dismiss non-greedy actions that actually have better future rewards, especially during earlier episodes. This is due to the fact that the agent does not have a full picture of the environment. As a result, such an agent can easily converge to a sub-optimal policy.

In order to discover the optimal policy, the agent needs to be able to refine the estimated $Q(s, a)$ for all (s, a). The agent needs to balance the need to behave optimally based on its current knowledge (exploitation) and the need to explore more of the state-action space to hopefully gain better rewards (exploration). Therefore, we decided to use the ϵ-greedy policy that picks the action $a_{greedy} = argmax_a Q(s, a)$ that maximises that Q-table column with probability $1 - \epsilon$ and picks a random action with probability ϵ. For our experiments, we chose exploration factor $\epsilon = 0.01$.

4. Experiments

At the current stage, we have fully implemented the game environment, including computing the reward at the end of a game and determining the set of legal actions available at each turn. We have also fully implemented the training and evaluation with multiprocessing support based on the TorchBeast paradigm. Our Github repo is available here. A brief overview of how multiprocessing is used is as follows:

4.1. Multiprocessing

On each device, multiple actor processes are created. These actor processes act in parallel to sample trajectories using the `Env` class to compute estimates of the cumulative reward for each state-action (s, a) pair in the current episode.

This approximates the $Q(s_t, a_t) = r_t$ function. Whenever an actor process finishes an episode (one player wins), the cumulative rewards and their corresponding state-action pair (s_t, a_t, r_t) are written to a shared buffer on that device. Then, new 13-card hands are then generated and assigned to that actor process, and the actor process repeats the sampling of trajectories. This process repeats endlessly.

Meanwhile, a learner process on the same device communicates with the actor processes by extracting batches of state-action pairs and their corresponding rewards (s_t, a_t, r_t) from those shared buffers, and computes the MSE loss between the output of the Q-network (i.e. the predicted Q-value $\hat{r_t}$) and the actual outputs r_t, then performs backpropagation to update the Q-network's weights.

The actor and learner processes form the common producer-consumer paradigm and leverage concurrent execution to vastly increase training speed.

4.2. Training settings

All training was conducted on HKUST's High Performance Community Computing Cluster (HPC3) with CPU processors using 5 actors, 50 shared buffers, and 4 learner threads per CPU. Currently, we train with a batch size of 32, an unroll length per episode of $T = 100$, and the RMSProp optimizer with learning rate 0.0001, smoothing constant $\alpha = 0.99$, and epsilon 0.00001 (note this is different from the exploration factor $\epsilon = 0.01$.) We also set discount factor $\gamma = 1$ since Big Two is a game with sparse horizons; i.e. it has a nonzero reward only at the last timestep and early moves are extremely important to overall strategy.

5. Model Architecture

Currently, our model (Q-Network) takes as input a tensor of features and available actions $x \in \mathbb{R}^{559}$ and a tensor of historical actions $z \in \mathbb{R}^{4\times 208}$. (They are a concatenated list of features as seen in 1).

The historical actions z are fed into a unidirectional LSTM with hidden size 128. The output h_z is concatenated with x to serve as the input of the first fully connected (FC) layer. There are 6 such connected FC layers in sequence. The first 5 have an output size of 512 with ReLU as activation function for each layer, and the 6th FC layer has an output size of 1 with no activation function. The output of the final layer is: (i) the Q-value estimates of the current legal action-state pairs, and (ii) the selected action according to the ϵ-greedy policy. With our current abstracted implementation, more complex models can be designed and tested easily by modifying one file.

Since the positions {south, east, north, west} are symmetric, we assume that at all timesteps, i.e. one arbitrary position (e.g. south) uses an agent utilising the trainable DMC network, and the other three positions use copies of a different type of agent (for instance, a random agent, or a DMC agent with different weights.) It is important to note that only the weights for the Q-network used by the DMC agent in the south position are updated during training; the weights for the Q-networks used by the other positions are frozen and are not updated.

This is because if all four positions use agents with identical Q-networks, it is impossible to evaluate performance, as the model's EV over a large number of games is 0 (because the game is symmetric). We can only measure a model's performance if one or more of the positions utilises a different agent from the others.

5.1. Evaluation Metrics

Evaluation data is generated as specified in 2. For evaluating each game, the trained DMC agent will play each game from each of the 4 positions to ensure there is no positional bias, since in certain games, some players' hands may be much stronger than other players' hands. The other three positions (”opponents”) will use identical copies of a different agent type to serve as a benchmark.

The possible opponent agent types are:

1. Random agent:

   The random agent chooses an action from the set of legal actions uniformly (all actions have equal probability to be chosen.)

2. PPO agent:

   The PPO agent uses a pretrained PPO Network (architecture designed by [13]) for predicting the next action to take. The network was trained via proximal policy optimization with state and action abstraction.

(At the current stage, we only compare our trained DMC agent with random agent by self-play.)

Given an agent type A in one position and agent type B in all other positions such that the turn order is $(A, B_1, B_2, B_3, A, ...)$, we compute the following statistics:

1. WP (Winning Percentage): The number of the games won by A divided by the total number of games.

2. EV (Expected Value): The amount of units won/lost by A over the total number of games. $EV > 0$ indicates that A outperforms B, and vice versa. The higher the EV, the better A performs relative to B. We also define $EV_{B_i}[A]$ to be the amount of units won/lost by A solely from agent B_i. This allows us to see the difference in EV caused by positional advantage (we expect that $EV_{B_3}[A] < EV_{B_1}[A]$, for instance.)

If the trained DMC agent has total $EV > 0$ across all 4 positions of the same game, then it outperforms the agent type it was tested against.

5.2. Evaluation of Preliminary Results

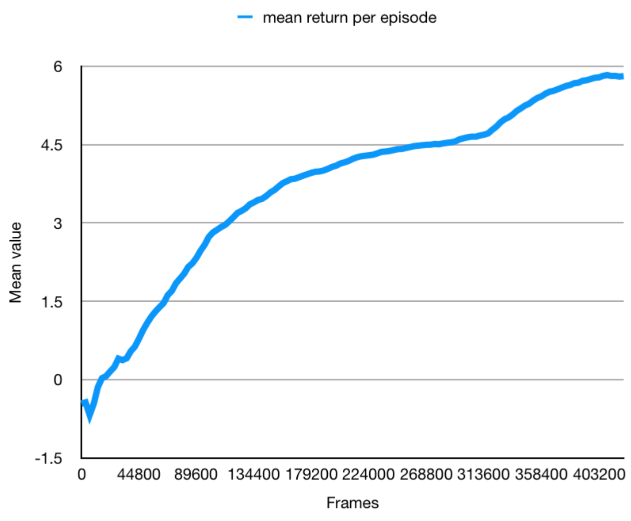

Figure 1. Mean return in each frame

We train the model for around 450,000 frames and show that our DMC agent is gradually improving over time (Fig. 1), as the mean episode return (EV) is consistently improving and is positive, meaning that our agent significantly outperforms the random agent. Of course, this is expected since the random agent is not difficult to beat and the results are only a proof of concept that our implementation is functional.

References

[1] K.H. Bočková, G. Slávikova, and J. Gabrhel. *Game theory as a tool of Project Management*. 2015. URL: https://www.sciencedirect.com/science/article/pii/S1877042815058462.

[2] Wikipedia. *Game tree*. URL: https://en.wikipedia.org/wiki/Game_tree.

[3] D. Silver, A. Huang, and C. et al. Maddison. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–489. DOI: https://doi.org/10.1038/nature16961.

[4] D. Nau. "Introduction to Game Theory: 6. Imperfect-Information Games". URL: https://www.cs.umd.edu/users/nau/game-theory/6%5C%20Imperfect-information%5C%20games.pdf.

[5] M. Zinkevich et al. "Regret minimization in games with incomplete information". In: *Advances in Neural Information Processing* (2008).

[6] M. et al. Moravcik. "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker". In: *Science* 356.6337 (2017).

[7] N. Brown and T. Sandholm. "Libratus: The Superhuman AI for No-Limit Poker". In: (2018). URL: https://www.cs.cmu.edu/~noamb/papers/17-IJCAI-Libratus.pdf.

[8] Li et al. "Suphx: Mastering Mahjong with Deep Reinforcement Learning". In: (2020).

[9] D. Zha et al. "DouZero: Mastering DouDizhu with Self-Play Deep Reinforcement Learning". In: (2021).

[10] H. Konishi and C. Pan. "Endogenous Alliances in Survival Contests". In: (2021). URL: http://fmwww.bc.edu/EC-P/wp974.pdf.

[11] Wikipedia. *Big Two*. URL: https://en.wikipedia.org/wiki/Big_two.

[12] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. The MIT Press, 2020.

[13] H. Charlesworth. "Application of Self-Play Reinforcement Learning to a Four-Player Game of Imperfect Information". In: (2018).

6. Appendix

Table 1. Features fed into the network.

	Feature	**Size**
Action	One-hot vector encoding a prospective action	52
State	One-hot vector encoding hand cards held by current player	52
State	One-hot vector encoding union of opponents' hand cards	52
State	One-hot vector encoding the most recent non-pass move	52
State	One-hot vector encoding downstream-player's most recent action (can include pass)	52
State	One-hot vector encoding across-player's most recent action (can include pass)	52
State	One-hot vector encoding upstream-player's most recent action (can include pass)	52
State	One-hot vector encoding the number of cards left in downstream-player's hand	13
State	One-hot vector encoding the number of cards left in across-player's hand	13
State	One-hot vector encoding the number of cards left in upstream-player's hand	13
State	One-hot vector encoding the set of cards played thus far by downstream-player	52
State	One-hot vector encoding the set of cards played thus far by across-player	52
State	One-hot vector encoding the set of cards played thus far by upstream-player	52
State	Concatenated one-hot matrices describing most recent 16 moves (4 from each player)	4×208