Deep Reinforcement Learning for Big Two

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

CHENG, Hoi Chuen
hcchengaa@connect.ust.hk

Abstract

Though significant breakthroughs have been made in various perfect and imperfect-information games, there is limited research on decision-making in many complex real-life situations that are modeled by imperfect-information games where multiple agents interact. We choose the popular card game Big Two as an appropriate abstraction for such situations. This is because successful strategies need to consider many actions at each state and require both long-term planning and temporary collaboration between agents. We consider a variety of deep model architectures that take the current game state and historical actions to train an agent via Deep Monte Carlo capable of generalizing well to unseen actions. We evaluate each agent's performance by computing their expected value when playing a batch of games against varying types of agents, including random agents, PPO-based agents, and DMC agents using different model architectures at prior iterations. We show that our final model, a DMC agent with LSTM + MLP and residual connections, outperforms all other agents (including existing policy-gradient methods) in terms of the mean episode return (EV) over a large batch of games, thereby demonstrating the robustness and effectiveness of Monte Carlo methods in generalizing to unseen situations.

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.)

Our results show that a standard DMC agent with LSTM + MLP trained via DMC already significantly outperforms the existing PPO benchmark by [12] by achieving an average EV of 1.204 over 1000 games, significantly better than break even. Additionally, utilising LSTM with MLP and residual connections can significantly improve the EV (to 1.841) during evaluation.

2. Related Work

A variety of reinforcement learning approaches have been proposed for dealing with imperfect-information games. In this section, we outline the works that inspired our research.

Counterfactual regret minimization (CFR) proposed by [5] is a model for solving the Nash equilibrium in two-player imperfect-information games via self-play. It does so by decomposing overall regret into a set of additive regret terms, which can be minimized independently. Regret is a measure of the maximal difference in expected utility: $R_i^T = \frac{1}{T} max_{\sigma_i^* \in \Sigma_i} \sum_{i=1}^{T} (u(\sigma_i^*) - u(\sigma_i^t))$, where $u(\sigma_i^t)$ denotes the utility of using strategy σ_i at round t, and Σ_i denotes the set of strategies available at round t. CFR was implemented for HUNLHE (a game with 10^{12} states) by [7] and achieved excellent results. However, since the usage of the max operation in regret calculation requires many traversals of the game tree, it is not feasible for a game like Big Two where both the depth and branching factor of the tree is large. Given that the number of possible moves available on each player's turn is large, we believe CFR to be too computationally costly with our limited resources.

[9] proposed using Deep Monte Carlo (DMC) methods on episodic data generated by parallel actors. They compare their trained model against a variety of baselines, including heuristic-based agents, actor-critic agents, and supervised learning agents. This approach is similar to ours. However, due to the lack of availability of online hand histories to use as data for a supervised learning approach, and a lack of available works on Big Two, we are unable to evaluate our model against these types of agents. Additionally, the training approach is slightly different in that DouZero uses different models for each position, with one for the landlord and one for each of the two farmers. During training

of DouZero, both the actor and learner models are updated simultaneously so that models are always playing the most recent version of that themselves. However, our approach involves a lag of 1000 games, in that the local actor models in each position are only updated to the most recent iteration of the learner model every 1000 games. Finally, we predominantly focus on the EV (Expected Value) metric as opposed to the Winning Percentage (WP) metric. DouZero's success at using DMC in a similar multi-agent reinforcement learning problem to achieve high performance while outperforming both A3C and DQN makes it an attractive option for our task.

[12] used the proximal policy optimization (PPO) algorithm, a policy-gradient based actor-critic method which uses a neural network to output both a policy $\pi(a|s)$ over the available actions $a \in A$ in any state $s \in S$ alongside $\hat{A}(a|s)$, the estimated advantage of taking an action in any particular state. Four identical copies of the model are initialised with random parameters and agents use this model to produce actions to execute at each decision point. The PPO agent was trained via self-play against their most current iterations for approximately 3 million games. Our approach uses the same rule settings for the games, however, we experiment with different model structures. Additionally, the PPO agent only uses a small subset of the available information at each decision point as input features, such as the most recently played card of each player. In contrast, our DMC agent captures all information available to it at its position, including the historical actions and exact cards played by every other player. Furthermore, unlike the DMC approach we employ, policy gradient methods are sensitive to the size of stepsize, and often have very poor sample efficiency in that it takes a high number of updates to learn simple tasks [13]. Empirically, we demonstrate that our agent trained on prior iterations for a small number of frames (approx. 2 million) already significantly outperforms the PPO agent, as the PPO agents rarely pass when playing singles and give many chances for the other agent type to discard their low-value cards.

3. 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 5.1) whenever an actor process finishes computing the estimated $Q(s, a)$ of all (s, a) in the episode. This is preferred over pre-generating large amounts of data beforehand, as this reduces disk space.

Data for 1000 Big Two games will be randomly generated and serialized in a `.pkl` format. No other preprocessing or filtering is required. During evaluation, these `.pkl` files will be deserialized and used to simulate the DMC agent's performance. The same set of 1000 games will be used for evaluation of the different model structures.

4. Methods

4.1. Alternative Approaches: DQN and NFSP

Deep Q-Learning (DQN) [14] uses Q-learning to estimate $Q(s, a) = r + \gamma max_{a'}[Q(s', a')]$ where r is the current reward and γ is the discount factor. However, this leads to overestimating the values for the optimal action $max_{a'}[Q(s', a')]$ [15] in large action spaces. One potential solution is to use two different function approximators (double Q-learning, using one for selecting the best action and another for calculating the value of this action). Since the approximators are trained on different samples, it is unlikely they overestimate the same action. Nevertheless, as discussed previously, the (s, a) pairs sampled in each game of Big Two do not satisfy the Markov independence assumption because actions taken on earlier in the game will affect what actions are available at the late game, and subsequently the values of the terminal states that can be reached. Therefore, since DQN relies on this assumption [16] for every distinct state-action pair $(s_t, a_t), (s_{t-1}, a_{t-1})$, *even those within the same game*, the approach is not suitable for our task.

Neural fictitious self-play (NFSP) is a deep reinforcement learning algorithm for approximating the Nash equilibrium of imperfect-information games. In a **Nash equilibrium**, no player can deviate from their strategy and expect to improve their expectation. An NFSP agent interacts with other agents and uses two memories to store prior experience. M_{SL} is a memory/dataset used to store the agent's "best response" when playing against other agents, and M_{RL} is a memory/dataset used to store the experience of the agent's own behaviour (i.e. state-action transitions it has taken.) The agent then utilises these memories to train two separate neural networks $\Pi(s, a|\theta^{\Pi})$ and $Q(s, a|\theta^{Q})$ where Π is the network that stores the agent's average historical behaviour and is trained via supervised learning on M_{SL} and Q is the network that predicts action values for the (s, a) pairs it has encountered and stored in the memory M_{RL} [17]. During evaluation, the agent uses a mixture of the two strategies β and Π where $\beta = \epsilon$-greedy(Q). In self-play, NFSP agents can expect to converge to Nash equilibria under certain conditions such as 2-player zero-sum games, however these conditions may not necessarily hold for Big Two. Additionally, since [12] utilised this approach, we will consider an alternative approach so as to provide a comparison of our method with the existing work. However, our approach is quite similar in that it features self-play, but we only train the network $Q(s, a|\theta^{Q})$ to predict $Q(s, a)$ values

for the (s, a) pairs generated in each episode.

4.2. Deep Monte Carlo

Ultimately, 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 in the large state-action space in Big Two, DMC builds on MC methods by training a deep Q-network to approximate $Q(s, a)$.

This is different from Deep Q-Networks (DQN) approaches, which uses **bootstrapping**. Bootstrapping refers to using one or more estimated values in the update step for the same kind of estimated value. For instance, the update rule in DQN [14] is $Q(s, a) \leftarrow Q(s, a) + \alpha(J - Q(s, a))$, where $J = r_{t+1} + \gamma Q(s', a')$ is the estimate for the true value of the optimal future value $Q(s, a)$, and α is the learning rate. Using J, an estimate of $Q(s, a)$ to estimate another $Q(s, a)$ is what causes $Q(s, a)$ to be **biased towards the starting values** of $Q(s', a')$.

In contrast, the disadvantage of DMC methods is that they have very **high variance** [18]. 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 $\epsilon = 0.01$.

4.3. Model Architectures

We consider a variety of model architectures (Q-Networks) to be trained via the DMC algorithm. All models take 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 Table 2).

Although there is an order to the card values, with [3d] being the weakest and [2s] being the strongest, the disparity in strength of the cards is not linear. For example, the difference between the [5h] and [5c] is much less than the difference between the [2s] and the [2h]. The player with the [2s] is guaranteed to be able to take the lead whenever singles is played, and consequently, may be able to play a weak five-card hand to minimise their losses, whereas the player with the [2h] has no such guarantee. However, the [5h] and [5c] are equally weak and unlikely to take control when singles are played. Therefore, we chose one-hot encoding to represent both x and z features as opposed to using label encoding.

4.3.1 Standard MLP

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.

4.3.2 Convolutional

The historical actions z are fed into a series of convolutional layers rather than an LSTM to get output h_z. The rest of the structure is identical to the Standard MLP model. We consider this an alternative to identify high level features in the historical action sequence, since repeated convolutions have been shown to be equally effective as LSTM in a variety of tasks involving feature extraction from sequence data.

4.3.3 Residual

The structure is identical to the Standard MLP Model, with the exception that the first 5 FC layers all utilise residual connections. We chose this method as residual connections allow earlier layers to learn just as fast as later ones and increases the expressive power of the model by using nested function classes [19].

4.3.4 Residual + Convolutional

The structure is identical to the Convolutional model, with the exception that the first 5 FC layers all utilise residual connections.

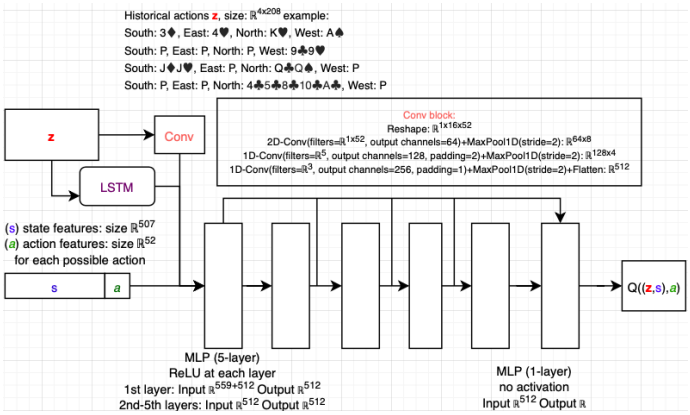

Figure 1. Detailed description of convolutional operations and residual connections

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. Experiments

In this paper, we wish to clarify the following questions. **Q1.** How do our trained models compared to algorithms of existing works? **Q2.** What is the effect of different model structures on performance? **Q3.** What are some common failure modes of the model?

Our implementation of the training and evaluation with multiprocessing support based on the TorchBeast [20] paradigm. Our Github repo (big2-rl) is available here [21]. A brief overview of how multiprocessing is used is as follows:

5.1. Multiprocessing

On each device, multiple actor processes inspired by [9] are created. These actor processes act in parallel to sample trajectories within the game environment 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 7.5 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 7.4 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.

5.2. 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.

For all experiments, we train with the same ruleset used in [12]'s PPO agent to allow for valid comparison of our agent with their PPO agent. However, the model can be easily retrained with a different ruleset by changing `settings.py`.

5.3. Evaluation Metrics

Evaluation data is generated as specified in 3. 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. PPO agent:

   The PPO agent uses a pretrained PPO Network (architecture designed by [12]) for predicting the next action to take. The network was trained via proximal policy optimization with state and action abstraction.

2. 'Prior' (DMC) agent:

   The DMC agent (or 'prior') uses the model trained with DMC as defined in the paper at previous stages during training ('prior iterations'). This allows us to compare how much the agent has improved over a certain amount of training time (e.g. training for 1 million frames vs training for 2 million frames).

3. Random agent:

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

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. 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 $EV > 0$, then since it has played all 4 positions, it outperforms the agent type it was tested against.

5.4. Training Results

We plot the mean EV of the various model architectures at different number of frames trained against the fixed Random and PPO agent benchmarks, showing that the residual model tends to perform best overall, with the standard MLP model being a close second.

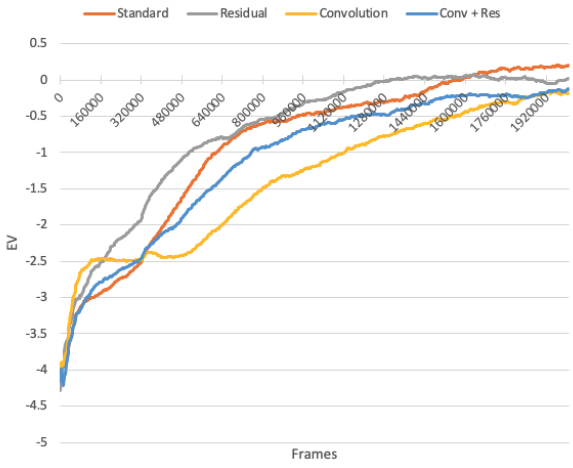

Figure 2. Models training against PPO agent

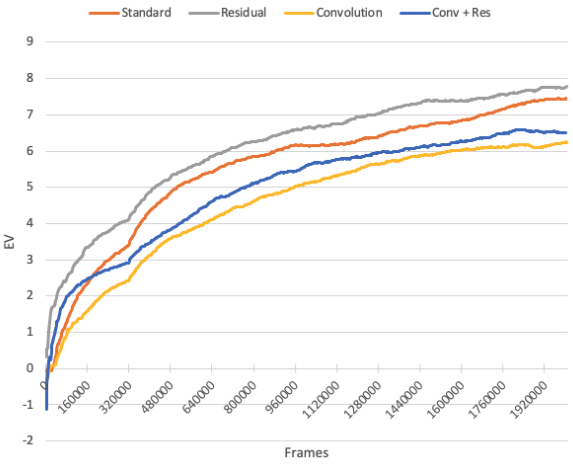

Figure 3. Models training against random agent

It should be noted that the loss function doesn't decrease and stabilise for reinforcement learning. This is because a lower loss implies more accurate predictions of value $Q(s, a)$ for the current policy π, however Deep Monte Carlo methods have a 'moving target' since the policy π is continually being refined by the Q-value estimates, which are constantly being calculated. Therefore, in reinforcement learning, the loss metric is not an effective indicator of performance, and the agent's performance is improving as long as the EV continues to increase steadily.

Finally, there is no fixed rule to decide how long the models can be trained for. In practice, the number of frames F_{max} is very large, and therefore training can take weeks to months. However, due to diminishing returns in performance and limited time constraints, we limit training of each of our models to around 2.5 million frames. As seen by the above plots, the model's performance is continually improving, so we expect that further training can yield even better performance.

5.5. Evaluation Results

From Table 1, different models {Standard, Residual, Convolution, Convolution + Residual} are trained on 3 different approaches {PPO, Prior, Random} and evaluated against the same types of opponents {PPO, Prior, Random} in terms of EV (a total of 36 combinations). However, the evaluation results against Random agents are only included for completeness since they are not realistic reflections of opponent strategies, so we do not consider them for the remainder of the discussion. Each evaluation was conducted over the same set of 1,000 games. From the table, for each training_approach-evaluation_opponent pair, the Residual model architecture has the highest average EV and only fails to have positive EV against a prior iteration of itself when training using a PPO benchmark. The Standard MLP architecture is similar but performs slightly worse overall, and it can be observed that the Convolutional and Convolutional + Residual models perform poorly relative to their Standard and Residual counterparts, respectively. The ranking of the model architectures from best to worst performance is Residual, Standard, Convolutional + Residual, and Convolutional. The slight improvement in average EV of the Convolutional + Residual model compared to the Convolutional model is likely due to the residual connections. This indicates that the LSTM component for extracting features from historical actions is superior to the series of convolutional layers, and the residual connections in MLP significantly improve average EV in all situations and increase the model's expressive power.

It should also be noted that the 'Prior' training approach (having a DMC agent train against prior iterations of itself rather than a fixed benchmark like a Random agent or a PPO agent) is the best, since for most of the model_structure-evaluation_opponent pairs, the average EV training against Prior is the highest.

From Figure 4, we observe that our intuition regarding the advantage offered by relative position ($EV_{B_3}[A] < EV_{B_1}[A]$) is indeed true. Whenever the trained DMC agent in SOUTH is evaluated against a non-Random agent, the EV of EAST is always significantly less than the EV of WEST (the player that goes directly after SOUTH), with the EV of NORTH falling between the two.

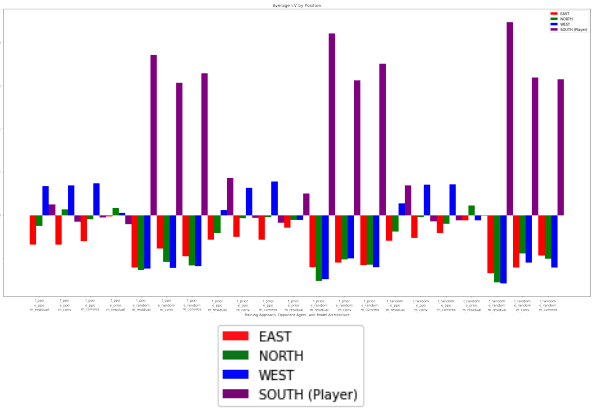

Figure 4. Average EV by position of selected model results in Table 1

5.6. Failure Modes

Here, we highlight some failure modes of our model, where it is believed to make poor decisions. In general, our models seem to gravitate towards playing their high-ranking combinations early. In one game, the WEST player starts the game by playing [3h3dTcThTs], and the SOUTH player is recommended to make the move [7c7h2d2c2h] despite the rest of their hand being relatively weak and lacking in high cards. Intuitively, SOUTH should pass in this situation since the three deuces have at least two, if not three opportunities to allow SOUTH to play low-ranking single cards or combinations. SOUTH also runs the risk of being beaten by a quads or unplayed straight flush, and though rare, this is possible since neither NORTH nor EAST have played any cards yet. However, we expect this behaviour to be mitigated when the DMC agent is trained for more frames due to the training curves not stabilising yet as seen in 5.4.

6. Conclusion and Future Work

This work has produced a robust reinforcement learning agent capable of considering the complex nature of collaboration in Big Two, yielding excellent performance against both the existing PPO benchmark and previous iterations of itself after self-play. This demonstrates the effectiveness of Deep Monte Carlo methods in making decisions for complex real-life situations with large state-action space, imperfect information, and sparse rewards, and shows the effectiveness of residual connections for improving our network's performance.

For future work, we could also explore variations on ruleset with different training approaches. For instance, instead of each game being treated as an independent episode from the previous game, it is common for players to compete to win the most units from the others after a fixed num-

Table 1. Evaluation result (in EV) of the 4 model architectures against different opponents in training and evaluation. All models were trained for approximately 2.5 million frames for fair comparison. Evaluation against 'prior' is conducted on the same model architecture trained for 2 million frames. (* The maximum EV is 39 and the minimum is -39.)

Eval Opponent	Opponent during training: PPO				Opponent during training: Prior				Opponent during training: Random			
	Std.	Res	Conv	Convres	Std.	**Res**	Conv	Convres	Std.	Res	Conv	Convres
PPO	-0.1	0.522	-0.335	-0.204	1.204	**1.841**	-0.326	-0.275	0.779	0.968	-0.288	0.139
Prior	-0.572	-0.41	-1.442	-1.283	0.341	**1.01**	-0.992	-0.661	0.552	0.705	0.343	0.417
Random	6.517	7.10	5.805	5.69	6.14	8.298	5.831	5.844	7.57	8.465	6.435	6.788

(Std. = Standard, Res = Residual, Conv = Convolution, Convres = Convolution + Residual)

ber of games in tournament settings. This may mitigate the effect of relative position on strategy since the ranking of each player on the overall leaderboard is more likely to influence strategy. Deep Monte Carlo methods may not be suited to such a task since the episodes are now incomplete, and other methods such as temporal difference learning may prove superior.

Finally, additional experimentation could be conducted regarding opponent sampling. Currently, the learner process faces copies of the same agent at each of the 3 positions, however, in real life scenarios, it is often the case that different opponents will adopt different strategies. The most common approach to make predictions about properties of modeled agents is via a self-play procedure which trains a *predictor model* concurrently alongside the *decision model* as recommended by [22] . For each position, the *predictor model* is meant to predict the unknown hand cards of the opponents at all other positions, whereas the *decision model* is used to compute what action should be taken for the agent at that position. This is useful because upon the end of the episode, we can observe the contents of all opponents' hand cards (ground truth) and compare them with the outputs of the *predictor model* during that episode. This allows us to define a loss function over the predictor model outputs to improve the predictor model. Finally, opponent modeling can be adopted in such a way that the RL agent considers the learning of other agents in the environment when updating their own policies/values, and this may help improve generalizability when facing different strategies from different positions [22] [23].

References

[1] K.H. Bočková, G. Sláviková, 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] H. Charlesworth. "Application of Self-Play Reinforcement Learning to a Four-Player Game of Imperfect Information". In: (2018).

[13] OpenAI. *Proximal Policy Optimization*. 2017. URL: https://openai.com/blog/openai-baselines-ppo/.

Table 2. Features fed into the network.

	Feature	Size
Action	One-hot vector encoding a prospective action current player is considering	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

[14] V. Mnih, K. Kavukcuoglu, and D. et al Silver. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533.

[15] T. Zahavy et al. "Learn what not to learn: Action elimination with deep reinforcement learning". In: *In Advances in Neural Information Processing Systems* (2018).

[16] Jianqing Fan et al. "A Theoretical Analysis of Deep Q-Learning". In: *Proceedings of the 2nd Conference on Learning for Dynamics and Control*. Ed. by Alexandre M. Bayen et al. Vol. 120. Proceedings of Machine Learning Research. PMLR, Oct. 2020, pp. 486–489. URL: https://proceedings.mlr.press/v120/yang20a.html.

[17] Johannes Heinrich and David Silver. "Deep Reinforcement Learning from Self-Play in Imperfect-Information Games". In: (2016). URL: https://arxiv.org/pdf/1603.01121.pdf.

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

[19] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: (2015). URL: https://arxiv.org/pdf/1512.03385.pdf.

[20] Facebook. *TorchBeast*. URL: https://github.com/facebookresearch/torchbeast.

[21] *Big Two with Deep Reinforcement Learning*. URL: https://github.com/johnnyhoichuen/big2-rl.

[22] Y. Zhao et al. "DouZero+: Improving DouDizhu AI by Opponent Modeling and Coach-guided Learning". In: (2022).

[23] J. N. Foerster et al. "Learning with Opponent-Learning Awareness". In: *AAMAS '18: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems* (2018). URL: https://arxiv.org/pdf/1709.04326.pdf.

7. Appendix

7.1. Total Number of Games

Let the players be $\{A, B, C, D\}$. Then for A we choose 13 cards (out of total 52), for B we choose 13 cards to make up his hand (out of the remaining 39), etc... Then the number of possible hands for a given player is $\binom{52}{13} * \binom{39}{13} * \binom{26}{13} * \binom{13}{13} = 5.36 \times 10^{28}$.

7.2. Total Number of Possible Moves in Big Two

Single: 52

Pair: $\binom{4}{2} \times 13 = 6 \times 13 = 78$

Triple: For each rank, $\binom{4}{3} * \binom{13}{1} = 52$

5-card straight: Enumerating the straight orders: $34567 < 45678 < \ldots < TJQKA < 23456 < A2345$. We notice we can uniquely identify a straight by the lowest card in the straight. There are 10 possible ranks for the lowest card in the straight. For each card in a straight, we can choose one of 4 suits. Then for each rank, there are 4 straights that are also straight flushes. So there are 40 straight flushes total: $10(4^5) - 40 = 10200$

5-card flush: For each of the 4 suits, there are 13 cards, from which we choose 5. Then we need to subtract the 10 straight flushes in that suit, hence $(\binom{13}{5} - 10) * 4 = (1287 - 10) * 4 = 5108$

5-card full house: We choose one of 13 available ranks for the triple. We then choose one of the 4 available

suits which our triple will not contain. Then, we choose one of the 12 remaining ranks to have as the pair, and choose 2 of the 4 available suits which our pair will contain. $\binom{13}{1}\binom{4}{1}\binom{12}{1}\binom{4}{2} = 3744$

5-card quads: We choose one of 13 available ranks for the quads. Then we choose one of the 48 remaining cards possible to use as the kicker. $48 * 13 = 624$

5-card straight flush: 40

Pass: 1

In total, there are 19,899 possible moves. However, as described in [12], there are at most 1695 total moves available to a 13-card hand.

7.3. Feature representation table for Deep MC

(See Table 2)

7.4. Learner process

Input: Shared buffer B_d with B entries, and size S for each entry, batch size M, learning rate ψ
Initialise global Q-networks Q^g_{north}, Q^g_{east}, Q^g_{west}, Q^g_{south}
for $iter = 1$ **to** F_{max} **do**
  if *number of full entries in* $B_d \geq M$ **then**
    Sample batch of $\{s_t, a_t, r_t\}$ with $M \times S$ instances from B_d and move these entries to free queue
    Update Q^g_{south} with MSE loss and learning rate ψ
  end
end

Algorithm 1: Learner process

7.5. Actor process

Input: Shared buffer B_d with B entries, and size S for each entry, exploration hyperparameter ϵ, discount factor γ
Initialise local Q-networks Q_{north}, Q_{east}, Q_{west}, Q_{south} and local buffer D_d
/* F_{max} is the maximum number of frames */

for $iter = 1$ **to** F_{max} **do**
  /* Periodically update weights of opponents if training using Prior */
  if *Q-Network is trained using Prior and* $iter \mod 1000 = 0$ **then**
    Synchronise Q_{north}, Q_{east}, Q_{west} to be identical copies of the most recent checkpoint of Q_{south} and freeze their weights
  end
  Synchronize Q_{south} with learner process
  /* generate episode */
  for $t = 1$ **to** T **do**
    $Q \leftarrow Q_{south}$
    $a_t \leftarrow \begin{cases} \text{argmax}_a Q(s_t, a) & \text{(prob: } 1-\epsilon\text{)} \\ \text{random action} & \text{(prob: } \epsilon\text{)} \end{cases}$
    Perform a_t, observe s_{t+1} and reward r_t
    Store $\{s_t, a_t, r_t\}$ to D_d
  end
  /* obtain cumulative rewards and store them in local buffer */
  for $t = T - 1$ **to** 1 **do**
    $r_t \leftarrow r_t + \gamma r_{t+1}$ and update r_t in D_d
  end
  /* move from local buffer to shared buffer by storing the indices in a full queue */
  if $D_d.length > T$ **then**
    Request and wait for empty entry in B_d
    Move $\{s_t, a_t, r_t\}$ of size L from D_d to B_d
  end
end

Algorithm 2: Actor process