Mastering Cribbage using Deep Reinforcement Learning

Marc-Antoine Bélanger (260905049) Alexis Tremblay (260922703) Nicholas Vachon (260905671) Joseph Viviano (20115694)

Abstract

Cribbage is a card game of incomplete information and with a high degree of chance. We establish that standard deep reinforcement learning (DRL) algorithms can learn how to play Cribbage against baseline deterministic agents with a performance similar to human players. We explore the impact the architecture of the function approximator, learning algorithm (Monte Carlo, SARSA, Expected SARSA, and Q-Learning), learning rate, and reward definition. Our contributions include an open-source OpenAI Gym (Brockman et al., 2016) compatible Cribbage environment, a greedy deterministic agent that serves as our non-DRL baseline, a human baseline established against this deterministic agent, and benchmark results of standard DRL agents trained to play cribbage via self-play. We demonstrate that human-level performance is obtainable if the correct design decisions are made.

Figure 1. Typical Cribbage board displaying the highest possible hand (29 points).

1. Introduction

1.1. The Rules of Cribbage

Cribbage can be played according to multiple rules. We follow the Masters Traditional Games rules (MTG) and focus on the 2 players version of the game.

The goal of Cribbage is to be the first player who obtains 121 points. It is played as a series of hands, where the deck is shuffled after each. At the beginning of a hard, the dealer is chosen randomly and six cards are dealt to each player. Each hand progresses through three distinct phases: **the Deal**, **the Play**, and **the Show**. Players employ distinct strategies during the first two phases and no strategy during the third phase, where the players simply receive points for their actions during the Deal. During the Deal, each player discards 2 cards to *the Crib*. The dealer can score points from this crib following the Play, so each player will employ different discard strategies depending on whether they are the dealer. During the Play, players alternate placing cards on the table, scoring points for completing particular sequences. The total "value" of all cards currently on the table can never exceed 31, so players may only be able to play some, or none, of their cards at any given time. If no players can play their remaining cards, the table is cleared and players continue as before, until all players have no remaining cards in their hand. After the Play, players earn additional points during *the Show*, based on the cards used during the Deal in conjunction with a randomly-selected "starter" card. The dealer additionally earns points from the Crib. Following this, the hand is over and the deck is shuffled. The average cribbage game has 10 hands.

Points are primarily awarded in three ways: pairs (2 points), sequences of $n \geq 3$ or more (n points) and combinations of cards that add to a value of 15 (2 points). For sequences, suits are irrelevant. Card values are given by the number on the cards (Aces are worth 1 point; jacks, queens and kings are worth 10 points).

1.2. Previous Work

Very few researchers have tackled the game of Cribbage in the past and we could only find researches who had studied the *Show*. Since this is a one-step game, the problem reduces to a contextual bandit problem. One previous school project used a small feedforward network as a function approximator, in combination with TD learning, to learn the value of each possible legal discard (from a hand of 6 cards) (O'Connor). The trained agent outperform a random discard agent, and was less that 1 point from the mean performance of the "cheating" deterministic agents that discarded cards based on knowledge of all the cards in play (including the hidden "starter" card). These results were averaged over 20 000 hands. A second study (Kendall & Shaw, 2002) used evolutionary strategies to learn how to discard, where two agents were trained to play against each other, learning the values of each set of hands by exploring them randomly. Then the agents were evaluated by playing against a commercially available program. In both cases, the agent learned how to play the game better than chance, but had trouble besting the best deterministic agent, though the nature of the deterministic agents were quite different between these experiments, so it is hard to compare their results directly.

However, a general strategy for learning how to play games much more complicated than Cribbage, **Deep Q learning**, has emerged in the past few years (Mnih et al., 2015). They used a neural network function approximator to approximate the true value of taking some action in a state, $Q^*(s,a)$, and then applied Q-learning to update this value function approximator with the following update:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\ U(D)}\big[\big(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a,\theta_i)\big)^2\big], \tag{1}$$

where s is the current state, a is the current action, r is the reward from the environment for taking that action, and s' is the state that the agent will end up in after taking that action. Crucially, these updates are applied to the neural network gradient descent updates computed from randomly-sampled minibatches of previous experience (s,a,r,s'). This is known as **experience replay**, which stabilizes training such that the function approximator does not only remember recently events and ensures that gradient descent updates are done using IID samples. The semi-gradient update applied is:

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}_{s,a,r,s'}\big[\big(r + \gamma \max_{a'} Q(s',a',\theta_i^-) - Q(s,a;\theta_i)\big)\nabla_{\theta_i} Q(s,a;\theta_i)\big]. \tag{2}$$

This approach was used to train an algorithm to play a suite of Atari 2600 games, in many cases matching human performance. We aimed to test whether a similar approach would be able to improve in the case of cribbage.

2. Environment

One of our contributions is an open-source implementation of the Cribbage game as an Open AI gym environment (Bélanger et al.). In the Gym framework the environment can be stepped to produce the next state. Typically this is straightforward in a single-agent environment, but cribbage can be played by up to 4 players, player's turns can be skipped, and the game follows three distinct phases (*the Deal*, *the Play*, and *The Show*), during each of which a step action takes a different meaning. Therefore, our approach was to return the state to the player in the following format:

- A hand of legally-playable cards for the next agent to play,
- The ID of the next agent,
- The reward received by the last agent to act,
- The ID of the last agent to act,
- The total score (out of 121) of the next agent,
- A list of all opponent scores (out of 121).

This allows the agents outside of the environment to receive rewards, select the next card to either play or discard according to the phase of the game, and for the current player to keep track of their progress in the game relative to their opponents. During the Deal, when a player submits a card to the environment as an `env.step()`, this card is selected to be discarded to the crib. During the Play, cards submitted using `env.step()` are instead played on the table. During the Show, the agent can submit any card, even a nonsense card, as the only point of this `env.step()` is to retrieve the total points owed to that agent.

When no cards remain to be played (the game has passed through all three phases), the deck is shuffled and a new round is started. The game continues as such until one of the agent's total score goes over 121 at any point, at which point the environment will return `done == True`. The environment will no longer accept input at this point, and requires an `env.reset()` to begin a new game. The game automatically advances through the parts of the game as the agents interact with the environment. The current phase of the game can always be accessed via `env.phase`. Finally, a rendering of the current state of the board can be seen in one's terminal by executing `env.render()`. The environment is released under the MIT license.

3. Methods

We define 1 episode as a single hand (one round of the *Deal* and one round of the *Play*). Since the number of steps is small in an episode (≤ 6), we used a **discount factor** $\gamma = 1$ for all experiments.

Agents need to learn one independent strategy for each of the Deal and the Play. These were learned as two Q function-approximators.

3.1. Card Representation

One can represent cards as one-hot encoded vectors $x \in \mathbb{R}^{52}$. Multiple cards can either be represented as concatenated or summed one-hot vectors. This encodes everything there is to know about a card, its rank (ace to king) and its suit (heart, diamond, etc). Since at no point during Cribbage does the strategy rely on both card ranks and suits, we do no represent the cards in this way. Instead, we encode the rank of a card as a one-hot vector $x_r \in \mathbb{R}^{13}$ and it's suit as one-hot vector $x_s \in \mathbb{R}^4$, alternatively this can be represented as a matrix of $\mathbf{x}_d s \in \mathbb{R}^{13\times 4}$. This representation of the card was inspired by previous work employing convolutional neural networks (CNNs) to analyze poker hands (Yakovenko et al., 2016). This representation allows for our model to learn common patterns in cards. We refer to this as the "compact representation".

More specifically, pairs of cards can be detected by a filter that detects two elements in the matrix at the same rank index (all pairs are worth the same amount), sequences are present elements with no breaks along the rank axis (ignoring suits; 2-3-4 is worth as much as 9-10-J), sums of 15 can be detected with only a few filters, as can cards of the same suit. By utilizing weight sharing with a CNN, we allowed the agent to learn features that generalize across card combinations over less games.

3.2. Deep Q Learning With Experience Replay

DRL using Q-function approximation relies on experience replay to stabilize training, as the semi-gradient updates used when updating the Q-function must be IID (Hessel et al., 2017; Lin, 1992; Silver et al., 2016). We built a buffer of experiences (S,A,R,S') using the generalized policy iteration framework (Sutton & Barto, 2018, p. 86), iterating between *playing* games to refresh the replay buffer, and *training* the Q-function approximator. We define a loop of a *playing* part and *training* part as a *replay iteration.*

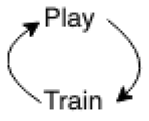


Figure 2. Training Methodology

While *playing* games, the agent uses the latest Q-function approximators and plays N games against itself (i.e., self-play as in AlphaGo Zero or TD Gammon (Silver et al., 2017; Tesauro, 1995)). Therefore, the two competitive agents share the same value functions while *playing*, so each game yields 2 games-worth of *training* data (one for each player). The number of games N is a constant hyperparameter. This set of N games we call an epoch of playing, and the buffer is configured to store experiences from P epochs, for a total buffer size of $N \times P$, where P is a second hyperparameter. The experience tuples (S,A,R,S') collected from playing the game are saved to disk to be used during training. We chose $P = 5$ and $N = 200$ to encourage training to make maximum use of the computed experiences.

Training is done using randomly-sampled experiences from the buffer. A duplicate network is copied before training that serves as the evaluation network for bootstrap values ($\hat{Q}(S,A)$ in equation (4))(Mnih et al., 2013).

Following *training*, the updated Q-value function approximators are used to play a set of N games, and the loop continues.

3.3. Agent Evaluation

Previous works (O'Connor) and (Kendall & Shaw, 2002) defined different deterministic agents to compare with their trained agents during the *Deal*. In the first project, the author defined both a cheating agent (has access to information a player does not normally have) and a random agent. In the second project, the baseline was a programmed computer game following pre-specified rules. For evaluation, these metrics were not useful to us.

We therefore defined a greedy agent, who evaluates the maximum number of points it can receive in a single move and plays that card with no long-term planning. This *Deterministic Agent* wins approximately 98% of the time against a random agent. We did not train our DRL agent against this deterministic agent, to prevent it from learning these patterns directly.

At each step of evaluation, we played our DRL agent against the Deterministic Agent for 5000 games. Because the environment is stochastic and both agents explore, we view each game as a random variable following a Bernoulli distribution, similar to a biased coin. Assuming this, we can calculate a 95% confidence interval for the real performance of the DRL agent with $\hat{p} \pm 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$ where $\hat{p}$ is the win ratio and n is the number of games played. Table 1 shows the confidence intervals for different numbers of games.

	n		
$\hat{p}$	**1000**	**3000**	**5000**
0.6	±0.0304	±0.0175	±0.0136

Table 1. Bernoulli Confidence Interval (95%)

With 5000 games against the deterministic agent we have high confidence in the results shown in figure 9 and 10.

3.4. Human Baseline

In order to compare the results of our DRL agents with human level performance, we had 5 good Cribbage players play 27 games against our Deterministic agent. Humans won 16 games while the Deterministic agent won 11. This gives us a Human Baseline of 59.3% wins against the Deterministic agent. Of course, 27 games is little and more games should be played, especially because of the randomness in the game. If we work we the same modeling assumption as above where each game is a random variable following a Bernoulli distribution, then the confidence interval is huge at ± 0.1853. We therefore make no strong claims about human-level performance.

3.5. Agent Behaviour Analysis: Expert Strategies

Expert human players tend to employ the following strategies (cri). We present 3 strategies from the Play phase, which we evaluate the agents on qualitatively.

1. The first card played should not be a 5 or 10, as the opponent can easily make 15 with this,
2. Bait your opponent to help you create a 3+ card run. E.g., if one leads with a 7, the opponent would be tempted to play 8 to create 15 for two points. This allows one to score three points via a 7-8-9 sequence,
3. If one has a pair in-hand, playing the first card of that pair is a good idea. If the opponent has the matching card, they will score 2 points for the pair, but you will score 6 points for a "pair royale" (since 3 cards of the same rank can be paired in 3 ways).

3.6. Q-Function Approximator: The Deal

3.6.1. Reward

Decisions made during the deal affect the number of points that can be obtained during the Play, and the Show. However, the agent only knows the cards in it's own hand when making these decisions. The reward for the Deal was therefore defined as the total number of points earned during the Play and the Show. In this way, the decisions made by this Q-function approximate does not inadvertently hamper the performance of the agent during the Play.

3.6.2. Update rule

Discarding two cards out of six is a very short RL problem. Once done, you are in a terminal state S' with $Q(S', A') = 0$. This turns the traditional update rule into a simple supervised learning problem. The weights of the Q-function approximator for the Deal was therefore

$$w_{t+1} \leftarrow w_t + \alpha[R - Q(S,A)]\nabla_{w_t} Q(S,A). \qquad (3)$$

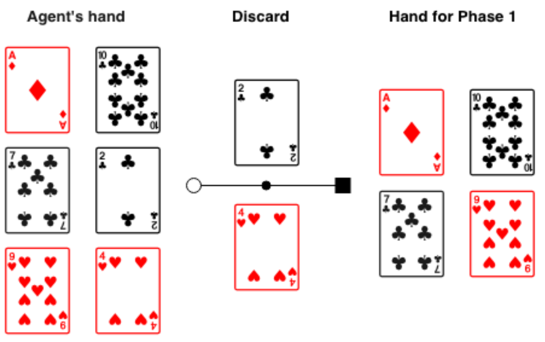


Figure 3. Backup diagram for the *Show*

See figure 3 for the backup diagram of this update.

3.6.3. Architecture

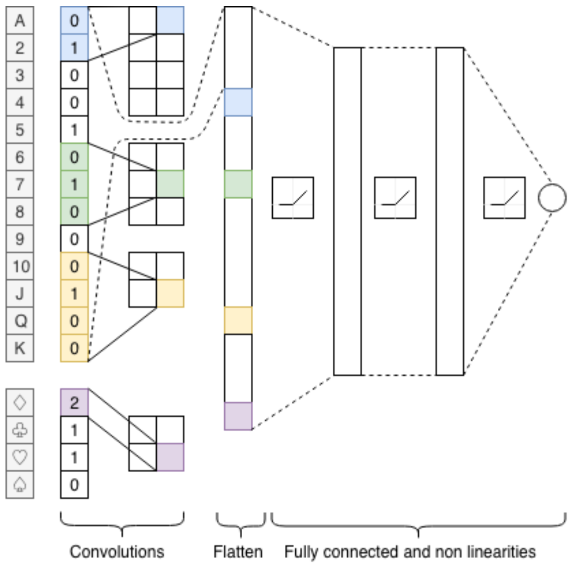
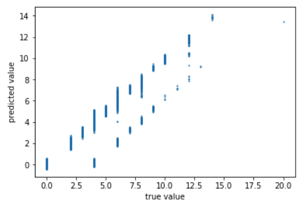
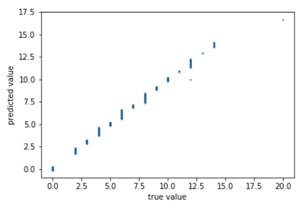


Figure 4. Architecture of value function for the *Deal*

During the *Deal*, the task is to choose 2 cards from 6 to place in the crib. The inputs to the value function (figure 4) are the sum of all one-hot encoded ranks and one-hot encoded suits. Running convolutions of size 1×4 over the summed rank matrix $\mathbf{x}_r$ will highlight pairs while larger convolutions will highlight sequences of 3 or more cards. Having a convolution of size 1×4 over the suit matrix $\mathbf{x}_s$ has a similar effect, allowing the network to detect whether the player holds four cards of the same suit. We denote the convolution operations $c(x)$. These convolution outputs $c(x_r)$ and $c(x_s)$ are flattened and concatenated with a fully connected layer over the rank matrix $\mathbf{x}_r$ that learns combinations of cards that sum to 15. We additionally included whether the player was the dealer or now as a binary input variable, as this information determines whether the cards discarded will count towards this agent's points, or the opponent's points. This then goes through a few fully connected layers with ReLU activations to finally output a value. For an overview of the architecture, see figure 4.



(a) Invalid value function



(b) Valid value function

Figure 5. The deal Q-function approximator without (left) and with (right) appropriate convolutions.

As an example of how convolutions helped our model learn good value predictions, see figure 5. The left figure (5a) depicts the value predictions made by a model that did not understand that cards of the same suit give extra points. Adapting the model with convolutions over suits solved this problem (right, 5b), where predicted values closely track true values.

3.6.4. Pretraining

For the Q-function approximator of the *Deal* we experimented with supervised pretraining. Since learning a good strategy for *the Deal* reduces to a supervised learning problem, we hypothesized that we would see performance gains by directly training the Q-function approximator to predict the value of 4 cards according to the rules of Cribbage.

Choosing 4 cards out of 52, $\binom{52}{4}$, produces 270,725 different 4-card combinations. The network was trained to predict the correct value for all of these combinations in minibatches of 64. We define value here as the number of points they provide without considering the impact on the rest of the game. See figure 5b for a demonstration that this training is possible.

This model is trained only for 3 epochs, with an epoch being a sweep of all 4-card combinations. While this initializes the Q-function approximator well, it is not sufficient, because the true reward includes all points generated during the *Play* plus the value of the hand including the starter card. Therefore in a real game of Cribbage, the reward for a given set of 4 cards can vary depending on what the other player has and the starter card.

See figure 6 to see the same value prediction shown in figure 5b, but now including the effect of the unseen random starter card. There are 48 possible starter for any given hand. While there is a trend in the data, it is very hard for the agent to confidently approximate the value of a hand. For example a hand with [3 - 4 - 6 - 6] is worth 2 points without a starter. If the starter is a 5, the resulting hand [3 - 4 - 5 - 6 - 6] is now worth 14 points. This is a clear illustration of the dramatic effect of chance in Cribbage, and this effect is compounded by the additional randomness introduced by *the Play*.

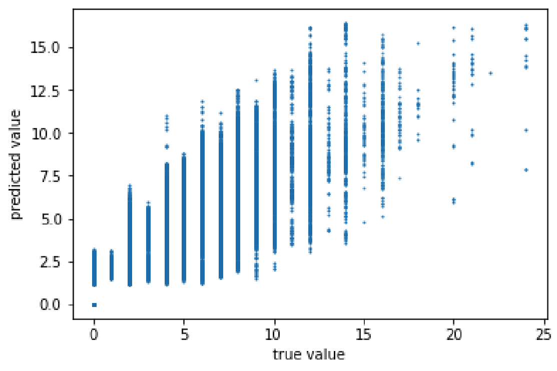


Figure 6. Value of 4 cards with a random starter

3.7. Q-Function Approximator: The Play

3.7.1. Reward

We compared 3 reward formulations for each step of the *Play*:

1. *Offensive*: Points earned by the agent for it's last action,
2. *Offensive-Defensive*: Points earned by the agent for it's last action minus points made by the opponent during the following turn,
3. *Game Reward*: All steps have a reward of 0 except for a reward of 1 given to the player who scores the most points during the entirety of the Play after it plays it's last card. We experimented with giving 0.5 to both agents in the event of a tie.

3.7.2. Update rule

In this work, we analyze one step methods and the Monte-Carlo method. An episode is defined as a complete hand, so the number of steps is limited in length. The update for the Q-function approximator is the semi-gradient update

$$w_{t+1} \leftarrow w_t + \alpha \left[R + U_t - \hat{Q}(S,A) \right] \nabla_{w_t} Q(S,A) \qquad (4)$$

where U_t is either the Monte Carlo, Expected Sarsa, Sarsa, or Q-Learning update. In the Monte Carlo case, $U_t = G_t$ and in Q-learning, $U_t = \max_a Q(S',a)$. See figure 7 for the backup diagram for the one step methods.

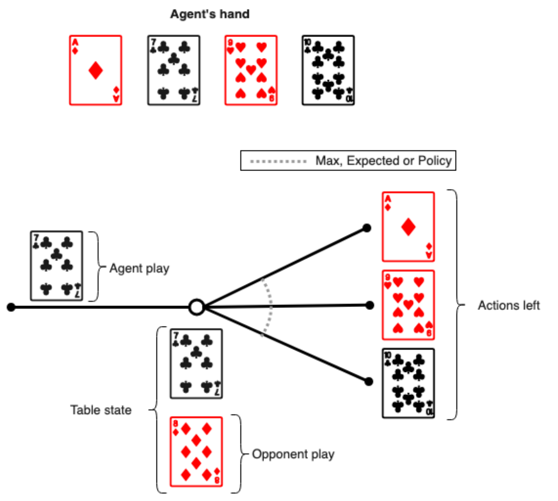


Figure 7. Backup diagram for the *Play* for all one step methods considered.

3.7.3. Architecture

The goal during the *Play* is to generate pairs and sequences of cards: suits have no effect on the points scored. Therefore represented cards using one-hot encoded vectors of the card's rank $x \in \mathbb{R}^{13}$.

The order that cards are played affect the resulting score, so we employed a long short-term memory (LSTM) network in addition to convolutions over sequences of cards to compute the value of sequences of actions. For example, if we are looking at a sequence of 5 cards, we concatenate the card vectors x to form a matrix $\mathbf{x} \in \mathbb{R}^{(5x13)}$. We run (2x2), (3x3) and (4x4) convolutions (with appropriate padding) to find pairs and sequences in the card matrix (similar to the deal), which we call $c(\mathbf{x})$. Unlike the deal, we only care about features if they appear toward the end of the sequence, so we concatenate the flattened versions of both the convolution representation of the card matrix $c(\mathbf{x})$ and the original input $\mathbf{x}$ and run an LSTM over it, again the intention of inputting the non-convolved input will allow the Q-function approximator to identify sets of cards that sum to 15.

Part of the state are also the cards left in your hand $x_h \in \mathbb{R}^{13}$ and whatever has been played before $x_p \in \mathbb{R}^{13}$, both being summed one-hot rank representations. We concatenate the last hidden state of the LSTM with those two vectors and go through fully connected layers with ReLU non-linearities to obtain our estimate of the value.

3.7.4. State Information

We experimented on the impact of the state representation during the Play using two levels of completeness::

- *Basic*: Includes the hand of the player (sorted, since order has no effect on points) as 4 concatenated one-hot encoded vectors of 13 elements, and cards in the table as 7 one-hot encoded vectors of 13 elements (not-placed cards are represented as all-zero vectors) for an input size of 156.

- *Full* include all available information to the player:
  1. The agent's hand.
  2. All cards placed on the table.
  3. All cards played by all players.
  4. All cards placed by the agent into the crib.
  5. The starter.

  Items 1 and 2 are as represented in the *Basic* setup. 3, 4, and 5 are all encoded as summed vectors $x \in R^{13}$.

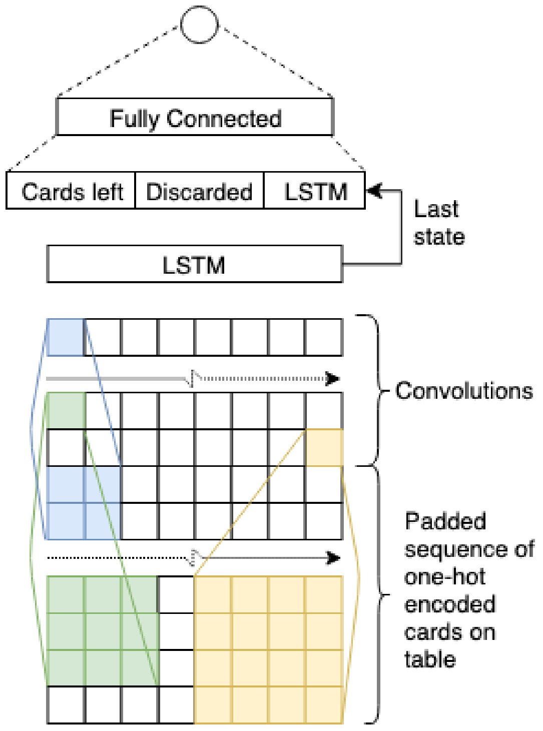


Figure 8. Architecture of value function for the *Play*

4. Results

4.1. Effect of Q-function Approximator Architecture

For the following results, the Q-function approximator for *The Show* was pretrained. The following hyperparameters were used: an *epsilon greedy* policy with exponential decay of 0.995, beginning with $\epsilon = 0.05$. Semi-gradient updates were done with IID minibatches of size 64. The learning rate was $\alpha = 5.0e-5$ for *the Deal* and $\alpha = 7.0e-4$ for *The Play* with exponential decay of 0.99, beginning with = 0.05. All rewards during *The Play* were *Offensive-Defensive*.

	Deep FeedForward	CNN + LSTM
Monte Carlo	6.5	12.5
Sarsa	10.75	33.75
Q-Learning	13.25	34.50
Expected Sarsa	15.5	34.75

Table 2. Computation time, in hours, for 500 *replay iterations*.

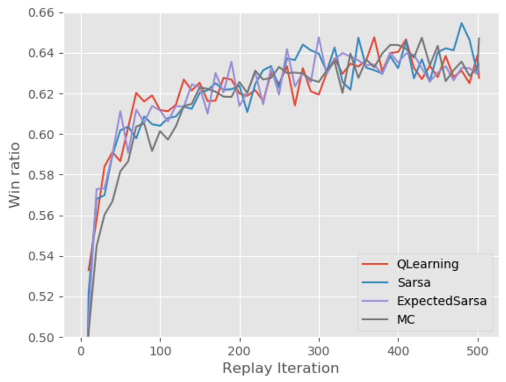


Figure 9. Performance of CNN & LSTM architecture over replay iterations.

Figure 9 and 10 show the performance of the agents during *the Play*. Using the combined CNN and LSTM Q-function approximator to learn good feature representations of the cards gave a small boost in performance, both in reducing the sample complexity required for learning and the final performance after 500 *replay iterations*. However, as can be seen in table 4.1, the CNN+LSTM agent substantially more computationally expensive to train than the feedforward agent. These results hold for all experiments performed.

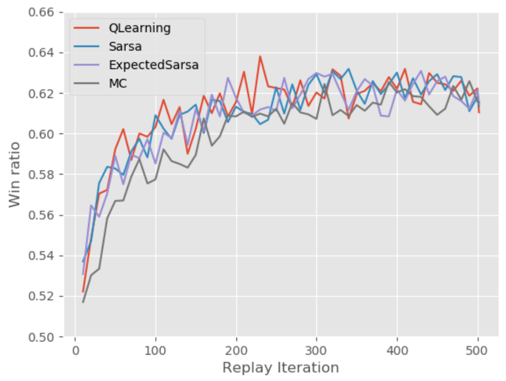


Figure 10. Performance of Feedforward architecture over replay iterations.

4.2. Learning Algorithms

In this work, we compare the basic Q-function approximation algorithms from (Sutton & Barto, 2018) to see if there is an advantage in terms of agent performance and computing time. We compare one step Expected Sarsa, one step Sarsa, one step Q-learning, and Monte Carlo algorithms. We only did one run for each algorithm due to limited computing time.

Figures 9 and 10 showno clear differences in terms of agent performance between algorithms. The Monte Carlo algorithm shows poorer results during early replay iterations due to high variance, but this effect does not persist. The MC algorithms computed approximately $2\times$ as quickly as the other three algorithms (table 4.1). This is due to the cost of interacting with the Q-function approxmator. With Sarsa, Expected Sarsa, and Q-learning, we are required to compute an extra forward pass through the neural network in order to get the bootstrap values. MC does not require such calculations since it does not have to bootstrap. Since we employed batch-processing of all possible actions when evaluating the value of different moves, Sarsa, Expected Sarsa, and Q-learning performed largely similarly (as expected, Sarsa was slightly faster than Q-learning, and Q-learning was slightly faster than Expected Sarsa). If we didn't use batch processing, we would expect Sarsa to be faster than Q-learning and Expected Sarsa, as they evaluate the values of all possible actions.

4.3. Hyperparameter Search for Reward Design Experiments

For each reward function evaluated, we performed a hyperparameter search for the learning rate, learning rate decay, epsilon decay, buffer size, replacement rate of the buffer, for each reward type. Due to time constraints, different models configurations (reward & state configurations) were trained a different number of times, for a total of 318 trained models. We report the average over all trials. Orion was used ((Tsirigotis et al., 2018) with Bayesian optimization to search for optimal hyper-parameters for each reward separately. The hyper-parameters and the range of their study are displayed in table 4 and the results are available in table 3. Learning rate and ϵ decay were defined as $rate_t = rate_{t=0} \times rate\_decay^t$ and were decayed once per replay iteration.

We also observe a relationship between the learning rate and replacement rate of the data in the experience buffer, where faster learning rates correspond to higher replacement rates. It seems like if the model learns faster, we can feed it more new data. A similar relationship is observed between the decay factor is negatively correlated with buffer size. It appears that models that explore less over time benefit from a larger replay buffer, which suggests that these models

Reward	State Information	Nb wins / 1000 games	Lr	Lr Decay	Epsilon Decay	Buffer Size	Replacement Rate of Data	Nb of trials
Offensive	Basic	531	0.0045	0.941	0.927	45k	20%	97
Offensive - Defensive	Basic	560	0.0099	0.956	0.988	45k	50%	141
Offensive - Defensive	**Full**	**567**	0.0008	0.957	0.995	45k	10%	30
Game Reward	Full	456	0.0042	0.951	0.951	45k	20%	26
Game Reward + ties at 0.5	Full	462	0.0011	0.971	0.910	90k	20%	24

Table 3. Performance & Hyper parameter search for different modelizations (the *Play*)

draw on early exploration experience more when is decayed faster. However, further experiments would need to be conducted to draw any conclusions.

4.4. The Effect of Reward Design During the Play

We compared the three rewards proposed (Offensive, Offensive-Defensive and Game Reward) during the Play, defining a win as any hand where a player obtains more points than it's opponent (table 3). All experiments were performed on the *Play* using Sarsa, ϵ-greedy policy (with $\epsilon = 0.1$) and the feedforward architecture, given the similar performance of the two Q-function approximators and substantially reduced compute time.

These experiments show that dense rewards outperform sparse rewards (i.e., *Game Rewards*), and that including information about the agent's performance relative to the opponent (*Offensive-Defensive*) is preferable to simply encouraging the agent to maximize reward (*Offensive*). *Game Rewards* were improved when introducing ties by a small factor. Encouraging the agent to only focus on maximizing it's own reward could cause it to ignore move that give an advantage to the opponent. For example, we would want the agent to avoid move that produce a 2 point but give their opponent the opportunity to score 6 points afterwards. Designing the reward to be the difference of performance between the two players in a turn allows the agent to learn these relationships.

Learning rate	(0.0001, 0.1)
Learning rate decay	(0.9, 1)
ϵ decay	(0.9, 1)
Buffer size	{10k, 45k, 90k)
Replacement rate of data	{10%, 20%, 50%}

Table 4. Hyper-parameters and the range studied

4.5. Effect of State Representation

We observed a marginal boost in performance when using the complete state information available (567 vs. 560; table 3). We suspect the additional information is not useful in the face of the high degree of randomness in the environment.

4.6. Expert strategies

We subjectively evaluated the trained agent's behaviour when provided with a particular hand:

1. When provided (A, 4, 5, 10), the agent prefers to play the cards in this order (10, 4, A, 5), thus leading with a 10 and violating a common expert strategy. An expert would prefer to play 4 since the opponent cannot reach 15. It remains possible that the agent believes it is better to keep 4 (and other small cards) for the end of the round to score exactly 31,

2. When provided (A, 7, 9, 10), the agent prefers to play 7, likely in an attempt to bait the opponent to play 8 and score two points by making 15, and then complete a sequence with a 9, giving the agent 3 points,

3. When provided (8, 8, Q, A), the agent prefers to lead with an 8. If the opponent plays another 8 (pair 2), then the agent can play its second 8 resulting in 6 points.

4.7. Set-Up of the Ideal Agent

Based on our experiments, we believe the ideal agent to play cribbage would use the offensive-defensive reward, the CNN & LSTM Q-function approximation architecture for the *Play* and the CNN architecture for the *Deal*, extended state representation for both phases, -greedy policy with decay, and a replay buffer with around 45,000 experiences. Pretraining the Q-function approximators appears to assist training. From our experiments it does not appear to matter which algorithm is employed for updating the Q-function approximator: SARSA, expected SARSA, Q-learning, and MC all performed similarly given enough training time. We have some evidence that MC takes less time to train due to the less number of passes over the Q-function approximator, but no evidence that this will produce better results.

5. Conclusion

In this work, we contribute an open-source *gym-cribbage* environment, which is a simple card game with a high degree of chance and partial observably. We hope this will inspire future researchers to use Cribbage in their research.

We additionally demonstrate that simple DRL approaches can successfully play Cribbage better than a deterministic greedy agent, and potentially at non-expert human level.

A possible extension of our work would be to combine the dense offensive-defensive rewards with game rewards. This could help the agent learn the importance of winning the game while still taking advantage of the dense feedback available during *the Play*. This could be implemented as our current definition of the offensive-defensive reward, with an additional large reward at the end of the game for the winner.

Another avenue for improvement would be improving the sample complexity required for learning the game. Experiments could be done using prioritized experience replay buffers as in (Zhang & Sutton, 2017). We are also unsure of the exact impact of buffer size as we did not directly analyze the impact of it. Ideas from (Hessel et al., 2017) could also be implemented to improve the agent performance.

Finally, we would propose that future researchers try to use new approaches designed for imperfect information games. A recent approach that learned how to play heads-up no-limit Texas hold'em (Moravčík et al., 2017) uses an approach called counterfactual regret minimization. In imperfect information games, the optimal strategy is one that follows the Nash equilibrium, such that the strategy chosen by the agent is the best counter to the opponent's assumed strategy or strategies, i.e., it will do no worse than a tie no matter the opponent's strategy. It works by evaluating previous games played, and for each move, evaluates how well the agent would have done over the entire game if it only made that move. The intuition is that if the move was good then, it would likely be good in the past and the future as well. A move that would have been better than was actually done has positive regret. The agent plays games and computes it's regret for each move, and then for new games, selects new actions with probabilities proportional to their previous positive regret. It would be interesting to see whether employing this very different approach would outperform the standard DRL approaches employed here.

A. Appendix

References

Masters traditonal games: The rules of cribbage. https://www.mastersofgames.com/rules/cribbage-rules.htm. Accessed: 2019-04-17.

Cribbage strategies. http://cribbagecorner.com/strategy. Accessed: 2019-04-25.

Brockman, Greg, Cheung, Vicki, Pettersson, Ludwig, Schneider, Jonas, Schulman, John, Tang, Jie, and Zaremba, Wojciech. Openai gym, 2016.

Bélanger, Marc-Antoine, Tremblay, Alexis, Vachon, Nicholas, and Viviano, Joseph. Cribbage for open ai gym. https://github.com/atremblay/gym-cribbage. Accessed: 2019-04-17.

Hessel, Matteo, Modayil, Joseph, van Hasselt, Hado, Schaul, Tom, Ostrovski, Georg, Dabney, Will, Horgan, Daniel, Piot, Bilal, Azar, Mohammad Gheshlaghi, and Silver, David. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL http://arxiv.org/abs/1710.02298.

Kendall, Graham and Shaw, Stephen. Investigation of an adaptive cribbage player. In *International Conference on Computers and Games*, pp. 29–41. Springer, 2002.

Lin, Long-Ji. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin A. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Moravčík, Matej, Schmid, Martin, Burch, Neil, Lisỳ, Viliam, Morrill, Dustin, Bard, Nolan, Davis, Trevor, Waugh, Kevin, Johanson, Michael, and Bowling, Michael. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

O'Connor, Russell. Cs 486 project: Temporal difference reinforcement learning applied to cribbage. http://r6.ca/cs486/. Accessed: 2019-04-16.

Silver, David, Huang, Aja, Maddison, Chris J, Guez, Arthur, Sifre, Laurent, Van Den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587): 484, 2016.

Silver, David, Schrittwieser, Julian, Simonyan, Karen, Antonoglou, Ioannis, Huang, Aja, Guez, Arthur, Hubert, Thomas, Baker, Lucas, Lai, Matthew, Bolton, Adrian, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

Sutton, Richard S. and Barto, Andrew G. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 2nd edition, 2018. ISBN 9780262039246.

Tesauro, Gerald. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

Tsirigotis, Christos, Bouthillier, Xavier, Corneau-Tremblay, François, Henderson, Peter, Askari, Reyhane, Lavoie-Marchildon, Samuel, Deleu, Tristan, Suhubdy, Dendi, Noukhovitch, Michael, Bastien, Frédéric, et al. Oríon: Experiment version control for efficient hyperparameter optimization. 2018.

Yakovenko, Nikolai, Cao, Liangliang, Raffel, Colin, and Fan, James. Poker-cnn: A pattern learning strategy for making draws and bets in poker games using convolutional networks. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Zhang, Shangtong and Sutton, Richard S. A deeper look at experience replay. *CoRR*, abs/1712.01275, 2017. URL http://arxiv.org/abs/1712.01275.