It’s Over 400: Cooperative reinforcement learning through self-play

Team Members:
Hadi Elzayn (PennKey: hads; Email: hads@sas.upenn.edu)
Mohammad Fereydounian (PennKey: mferey; Email: mferey@seas.upenn.edu)
Mikhail Hayhoe (PennKey: mhayhoe; Email: mhayhoe@seas.upenn.edu)
Harshat Kumar (PennKey: harshat; Email: harshat@seas.upenn.edu)

Assigned Project Mentor:
Simeng Sun

Team Member Contributions:

<table>
<thead>
<tr>
<th>Team Member</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadi Elzayn</td>
<td>Game Simulator</td>
</tr>
<tr>
<td></td>
<td>NN Architecture</td>
</tr>
<tr>
<td></td>
<td>Heuristic Algorithm</td>
</tr>
<tr>
<td>Mohammad Fereydounian</td>
<td>Literature review</td>
</tr>
<tr>
<td></td>
<td>Related work</td>
</tr>
<tr>
<td></td>
<td>Problem Formulation</td>
</tr>
<tr>
<td></td>
<td>Report write up</td>
</tr>
<tr>
<td>Mikhail Hayhoe</td>
<td>Game Simulator</td>
</tr>
<tr>
<td></td>
<td>NN Architecture</td>
</tr>
<tr>
<td></td>
<td>Custom Objects</td>
</tr>
<tr>
<td>Harshat Kumar</td>
<td>Related work</td>
</tr>
<tr>
<td></td>
<td>Problem Formulation</td>
</tr>
<tr>
<td></td>
<td>Report write up</td>
</tr>
</tbody>
</table>

Code Submission:
We submit our code via a github repository: https://github.com/MHayhoe/400.git

Abstract
Inspired by the successes of AlphaGo, AlphaGo Zero, and Libratus, which have resulted in super-human performance in the games Go and Poker, we study the game Four Hundred. Four Hundred is a team-based trick-taking card game similar to Spades, and has not been widely studied to our knowledge. The game contains many interesting aspects. Notably, the game has hidden information (since hands are not revealed and communication is forbidden), is cooperative (players are split into two teams which compete), and players must bet, ex ante, the tricks they win expect to win (and are punished if they fail to take these tricks).

In this work, we discuss an AI, Over400, which has learned how to play the game via self-play. Over400 uses deep reinforcement learning techniques to identify optimal actions in playing rounds and supervised learning to make optimal bets given its strategy. Training over 100,000 games, we find that Over400 defeats heuristic algorithms based on human knowledge and performs competitively against human players.
1 Introduction and Related Work

During the last two decades, games have been used as marked milestones of progress in artificial intelligence (AI). AIs have beaten top human players at backgammon [14], checkers [7], chess [9], Jeopardy [10], Atari games [6], and Go [4]. These are all included in a category called perfect-information games where the information about the current state of the game is identically shared between all players. On the other side, card games in which players’ private cards give them asymmetric information about the state of the game, are included in imperfect-information games. Heads-up no-limit Texas hold’em (HUNL) poker is an example of such a category for which the AI player Libratus [1] was designed and successfully defeated top human players with statistical significance. HUNL has approximately $10^{160}$ decision points which is comparable to Go (with $10^{170}$). However, it turns out that imperfect-information games fundamentally need more complex reasoning than perfect-information games of similar size [5]. Recent advances in reinforcement learning (RL) such as DeepStack algorithm [5] introduce a general platform to deal with imperfect-information games. DeepStack, used in the AI for HUNL, considers each particular state as it arrives through playing, but not in isolation. It also replaces the reasoning about the huge remainder of the game with a fast approximation and improves this estimation during the game via deep learning [8] using examples generated from random poker situations. DeepStack locally searches for a strategy in an approximate Nash equilibrium or equivalently a low-exploitability strategy. This kind of strategy seeking is typically studied under subgame solving. The subgame solving strategy for Libratus which is described in [11] can be used in many similar settings.

AlphaGo, [3] the first AI with superhuman performance in the game of Go, used two deep neural networks: One policy network, trained by Supervised Learning from labeled data obtained by experts, which estimated the distribution over the states and one value network, trained through self-play by RL, which predicted the winner following the policy network. AlphaGo Zero [4] was the next attempt, whose key difference is that instead of using Supervised Learning , it started from random play and used RL to improve its performance by playing against itself.

RL programs often use Markov Decision Processes (MDP) framework and seek to find an optimal action-value function $Q^*(s,a)$ based on a future discount return function. These action-value functions can be approximated by a non-linear function such as a neural network namely Q-network and the resulted algorithm is then called Q-learning [2]. In finding an optimal action-value function, generally there are two methods to be used: Dynamic Programming (DP) and Monte Carlo (MC) which used by the AI of Atari games and Go respectively. Further information about these two methods can be found in the 5th chapter of [13].

Inspired by the above successes in designing AIs which have resulted in superhuman performance, we study the game Four Hundred, which is a team-based trick-taking card game similar to Spades. The novelty in this present work lies in that this is, to our knowledge, the first time that deep reinforcement learning techniques (like those applied to Go and Poker) have been applied to cooperative-competitive multi-agent learning with hidden information. Full rules and common terms of the game are described in the Appendix.

2 Problem Formulation

Following our project proposal, we are interested in the following problem:

**Learning Problem** Find an optimal pair $\beta^*, \pi^*$ to maximize expected reward for player $i$ given each other and the play of the others:

$$\beta^{i,*} = \arg \max_{\beta \in \mathcal{B}} E \left[ \sum_{t=0}^{14} R \left( \beta(H_i^t), \pi^*_t(H_i^t) \right) \right], \quad \pi^{i,*}_t = \arg \max_{c \in \mathcal{H}_i^t} E \left[ R(\beta^{i,*}(H_i^t), c) \mid S_i^t \right].$$

The full details of this problem can be found in the Appendix. We now discuss the problem formulation for betting and card play.

**BETTING:** While in principle both betting and playing can be viewed from a reinforcement learning perspective, we chose to view betting as a supervised learning problem. That is, given some particular strategy and initial hand, the player can expect to win some number of tricks (where randomness comes
from the distribution of cards across opponents’ hands as well as variation in player strategies, stochastic and otherwise). Given observed games under a given strategy and initial hand compositions, a model that predicts tricks taken by the end of the round may serve as a good bet model.

Thus, the data we generate during games in the form of initial hands can serve as input data, while the total number of tricks won that round functions as the label. We implement a neural network for regression using this data. The input is a 4 \times 13 binary matrix with exactly 13 non-zero elements representing which cards are in the player’s hand. Each column of the matrix represents a value (e.g. 7 or Queen), while each row represents a suit. The first suit is reserved for the trump suit. The output of the neural network is a real number, which we then map to the closest integer between 2 and 13 (since bets of 0 and 1 are not allowed).

We define the loss as the squared difference between the score received and the best possible:

$$\ell_{\text{bet}}(y, \hat{y}) = (y - \text{sign}(\hat{y} - y) \cdot \hat{y})^2 = \begin{cases} (y - \hat{y})^2, & \text{if } y \geq \hat{y}, \\ (y + \hat{y})^2, & \text{otherwise.} \end{cases}$$

In our case, \(\hat{y}\) is the bet, and \(y\) is the tricks won. Notice that this loss function is asymmetric - it penalizes more if the bet is higher than the tricks obtained. This is motivated by the game’s scoring rules. The loss function also penalizes conservative betting, albeit less severely. Therefore our goal in this supervised learning problem is to learn a relationship between the initial cards dealt to each player and the number of tricks that player won at the end of the round. In this regard, the actual play of the game greatly affects the number of tricks expected to win.

**CARD PLAY:** The second component of the game is the actual card play. Here we consider a reinforcement learning approach to solve the problem. We capture the state of the game by three 4 \times 13 matrices and two 1 \times 4 vectors in the following way. The first matrix represents the order history of the game, which begins as a zero matrix. Each card played is represented in the order history by an integer between 1 and 52. For example, in the 5th trick of the game, the card played by the third player will have a 23 in the corresponding location (5th trick \times 4 cards per trick + 3rd card played = 23). The second matrix is for player history, which is also initially a zero matrix. As each card is played, its location will be filled by a number denoting the ID of the player. Continuing our example, if the card above was played by player 2, then a 2 will be put in the corresponding location. Representing order and player history as a matrix in this fashion was inspired by the state representation of AlphaGo. The final matrix is the player hand, which is similar to the input for the betting Neural Network, and indeed is identical at the beginning of the game. As the game continues, whenever a card is played, the 1 indicating the presence of a card in the hand becomes a zero. Returning to our example, the matrix after the fifth round will have 8 (13 initial cards - 5 cards played) non-zero elements. The first 1 \times 4 vector contains the bets that each player made at the beginning of the round. The second vector contains the tricks that each player has won so far in the round.

Given the states, we define a reward at the end of each round by

$$\text{Reward} = \begin{cases} \text{bet}, & \text{if } \text{tricks} \geq \text{bet}, \\ -\text{bet}, & \text{otherwise.} \end{cases}$$

The state size is combinatorially large, and hence we do not consider tabular reinforcement learning solution methods but rather approximate solution methods via function approximation with neural nets. Ultimately, we want to assign a value to each state. Then, given the possible actions available to the player, we will evaluate the function at each of the potential states and choose the action which enters the state with the highest value with probability 1 - \(\epsilon\). With probability \(\epsilon\), we choose a random action to control the exploitation-exploration trade-off. Our approach is informed by the classical Q-learning approach, where the value function is updated by

$$V^{t+1}(s) = \max_a \sum_{s'} p(s'|s,a) (r(s,a,s') + \gamma V^t(s')),$$

where \(s\) is the state, \(a\) is an action, \(s'\) is the state after taking action \(a\) from state \(s\), and \(\gamma \in [0,1)\) is the discount factor. We consider a mild adaptation, similar to [12], in which the reward is provided as a label to
each observed state, and the neural net Q-update occurs in batches. Given that reward is observed at the
end of the round, for trick $t \in \{1, \ldots, 13\}$, we assign a reward to that state $s$ by

$$\text{Value}(s) = \gamma^{13-t} \cdot (\text{Reward}_\text{Team member 1} + \text{Reward}_\text{Team member 2}) + 1\{\text{Team won the trick}\}.$$  

We include the 1 term for reward shaping [6] as it is a favorable outcome which should help increase
convergence. Once we have assigned a label to each of the states in terms of the value defined above, we use
a neural network for regression to map each state to the appropriate value.

**BASELINES:** Due to the nature of the project, we have generated all of our data from a game
simulator we developed from scratch. Before any learning, we created three baselines. The first baseline
is **Random Play-Random Bet.** Random bet selects a random value from $\{2, 3, 4, 5\}$ during the betting
stage, and in each round chooses a random card to play from the set of valid cards. The decision to limit
the bet to 5 allows the strategy to be competitive. The second baseline is what we define as **Greedy Play-
Model Bet.** During play, greedy simply selects the highest card in its hand from the set of valid cards,
without consideration of its partner. Betting for greedy uses a neural net trained on 100,000 games of 4
greedy players with the same architecture as *Over400*. The training input are the hands each player was
dealt, and the labels are the number of tricks that player took at the end of the round. For the final baseline,
a heuristic strategy was defined based on human knowledge of the game. This heuristic takes into account
knowledge of the team member’s actions as well as opponent’s actions. While a betting heuristic was also
defined, the betting for the heuristic baseline is also given by a model which was trained, in the same way
as greedy, on 100,000 games of 4 heuristic players. Hence our final baseline is **Heuristic Play-Model Bet.**

**PERFORMANCE MEASURES:** We use several metrics to evaluate the performance of *Over400*,
with the most natural being percentage of games won against the baselines. To test the performance of
the betting, we observe the average loss, which should decrease as the training iterations increase. Secondly, we
consider the distributions of the betting and tricks. We expect the betting to closely match tricks won, and
therefore we consider the difference between the tricks and bets. Finally, to test the performance of the play
strategy component, we consider the average score of each player. In a self-play scenario, where everyone is
using the RL to decide their actions, we expect to see that the average score of each player should increase
at the same rate.

### 3 Experimental Design

Due to the nature of our project, we had to develop a simulator from scratch before we could implement
any learning algorithms. We chose to develop the simulator with Python so that we could use the high-level
neural network API Keras ran on top of TensorFlow.

![Figure 1: Betting Architecture](image)

The architecture of the Betting Neural Network and Playing Neural Network are fully detailed in
Figures 1 and 3a respectively. This specific architecture was chosen to emulate the input state of AlphaGo,
as described in Section 2. Please see the Appendix for an in-depth discussion on the architecture.

To train our program to learn how to bet and play, we use Algorithm 1. As mentioned before, the
betting model depends greatly on the playing strategy. Similarly, the value of each state depends on the
bet made at the beginning of the round, as seen through the reward function. Therefore, we decided to
train the betting and playing neural networks on different sets of games. For our experiments, we chose to
generate and train from $N = 100,000$ rounds, with exploration parameter $\epsilon = 0.05$. We decay the exploration
parameter exponentially as we update the play neural network. We have $T_{\text{int}} = 1000$ and $T_{\text{off}} = \frac{T_{\text{int}}}{2}$. This
means that we train the betting and playing neural networks at an interval of 1000 rounds with an offset of 500. The offset means that 500 of the training samples for the Bet NN will be from the old Play NN, and the remaining 500 will be the same for the new, and vice versa. We chose to include this overlap to improve sample and computational complexity. After each round, the inputs and their corresponding labels are saved and will be deleted after the corresponding model has been updated.

Algorithm 1 Training Pipeline

1: \( k = 1 \)
2: \( \text{for } t = 1, \ldots, N \text{ do} \)
3: \( \text{Simulate Game with actions coming from Play NN with probability } 1 - \epsilon^k \)
4: \( \text{Save Play Data and Bet Data of player chosen at random} \)
5: \( \text{if } t \equiv 0 \text{ (mod } T_{int} \text{) then} \)
6: \( \text{Update Betting Neural Network} \)
7: \( \text{Delete Bet Data} \)
8: \( \text{if } t + T_{off} \equiv 0 \text{ (mod } T_{int} \text{) then} \)
9: \( \text{Update Playing Neural Network} \)
10: \( \text{Delete Play Data} \)
11: \( \text{if trained } 10 \text{ times then} \)
12: \( k \leftarrow k + 1 \)

4 Results and Summary

RESULTS: To understand the performance of the betting model, we consider the metrics of loss and Trick-bet difference. In Figure 2a, we see the loss as defined in Section 2. We see a very sharp decrease in the loss function within the first few iterations before it levels out and remains low for the remainder of the iterations. The noise observed is due to the betting and playing strategies being constantly updated in tandem. Figure 2b shows the Trick-bet difference metric. As we can see, the highest peaks of the histogram occur at Tricks - Bet \( \in [0, 2] \). This means that most of the bets are slightly less than the number of actual tricks taken, indicating that the players are betting conservatively.

![Figure 2: Betting Network Performance](image)

To understand the performance of the card play network, we consider the average score shown in Figure 3b. As expected, the average score of all four players increases as the number of training iterations increase. It is important to note that the game is learning how to play relative to the other players, in that it understands the relationship between its team member and its opponents relative to its position. Finally, we
consider Table 1 which shows the performance of our Program against the baselines. Our Heuristic algorithm was able to defeat Random and Greedy Baselines 100 - 0, as did Over400. Even more impressively, Over400 beat the heuristic baseline 89.5 % of the time.

Human players also squared off against the network to get a sense as to how Over400 was playing, although the lack of a clean interface made playing games time-consuming and hence few rounds were played. Nevertheless, some interesting trends were observed. The AI understood to win tricks by playing high cards of the lead suit and also to play a trump card when the lead suit had run out in its hand. Interestingly, it seemed that one of the AI players was consistently betting higher and performing better than the other team member. However, there were also moments in the game when it would have made sense to play a low card but Over400 threw away a relatively high one when it was not a lead or trump suit. We imagine that this is due to the difficulty in learning the concept of the lead suit, which may have been accomplished with more samples of training data or with a deeper neural net structure.

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>Greedy</th>
<th>Heuristic</th>
<th>NN + RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heuristic Win %</td>
<td>100</td>
<td>100</td>
<td>-</td>
<td>10.5</td>
</tr>
<tr>
<td>NN + RL Win %</td>
<td>100</td>
<td>100</td>
<td>89.5</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: Percentage of Heuristic wins and our Learned model via self play against the baselines over 200 games.

SUMMARY: After developing a game simulator and baseline strategies from scratch, we created the AI Over400 using principles from supervised learning, neural networks, and reinforcement learning. Trained only through self-play, Over400 beats every baseline, and in practice against humans shows cooperative techniques with its partner.

FUTURE WORK We plan to continue training Over400 for more iterations, and document whether it makes further progress in understanding the game mechanics. In addition, we would like to explore related questions and approaches, including:

- Testing the network against a further-improved heuristic AI, as well as more human trials
- Creating a parameterized heuristic AI that can be optimized via genetic algorithms, and comparing Over400 to this AI
- Optimizing the NN architecture using cross-validation and testing various network architectures
- Testing whether AIs can be trained first individually and then learn to cooperate together for faster convergence and transferable skills
- Testing Over400 against neural networks trained against heuristic AI
- Exploring hybrid approaches wherein human-provided strategic objectives are optimized using machine learning techniques
5 Appendix

Acknowledgments

We would like to thank Heejin Jeong and Steven Chen for volunteering their time to help us with the reinforcement learning portion of our project. We would also like to thank our project mentor Simeng Sun for her helpful suggestions on the Neural Network architecture and regularization for regression. Finally we would like to thank Hadi’s dad for providing helpful hints on how to improve the Heuristic algorithm.

5.1 Rules of Four Hundred

We briefly define some common terms here which we will use to describe our problem and solutions methods. To begin, the game consists of two parts, betting and playing. Before the playing round begins, bet on the number of tricks they will take that round. A trick is a sequence of four cards - if no trump is played, the winning card is the played card of the lead suit with the highest value; otherwise, it is the trump card of the highest value. A round is the collection of 13 tricks. After each round, the cards are shuffled and dealt again. The game is the collection of all the rounds until there is a winner (see below for win conditions).

Each round proceeds as follows: a shuffled deck of cards is distributed evenly (face down) to four players. Each player sits across from their teammate. No communication of hands or desired cards to be played is allowed. Before beginning play, players must bet an expected number of ‘tricks’ they plan to take over the 13-card hand. In each round after bets are placed, players take turns choosing a card to play from their hand in order, beginning with the player to take the previous trick (in the first round, the opening player is determined randomly). The suit of the first card played determines the ‘lead suit’, and all other players must play cards from that suit if they have any. If they do not have any of the lead suit, players may play trump or non-lead, non-trump cards. The winner of the trick is the player with the highest card of the ‘lead suit’, or the highest card of the ‘trump suit’ if any were played (non-lead, non-trump cards cannot win the trick). At the end of 13 rounds, each player’s score is increased by their bet if they meet or exceed it, but decreased by their bet if they fail to meet it. The game is over once one team has a player with 41 or more points, and the other player has positive points.

5.2 Problem Statement

The state of the game at the beginning of each round includes the time $t \in [13]$, where 0 is the round of betting wherein no cards are played; the four players’ hands $H^i_0, ..., H^i_{13} \in \{2, ..., 14\}^{13-1}$ (where 11, 12, 13, 14 represent Jack, Queen, King and Ace respectively); the history of played cards $h^i_0, ..., h^i_{13} \in \{2, ..., 14\}^{13}$; and the history of trick wins $T^i_1, ..., T^i_{13}$ in $[t-1]$. An action is a (feasible) choice of a card from the player’s hand $a^i_t \in H^i_t$. The reward is only accrued at the end, so we define the reward of an action under bet $b^i$ as $R(b^i, a^i_t) = b^i(1 - 2 \times 1 (T^i_t < b^i))$ at time $t = 14$, 0 otherwise, and the goal of the AI is to maximize $\sum_{i=0}^{14} R(b^i, a^i_t)$ over feasible actions at every time step. Importantly, for each player not all state variables are observable; player $i$ in fact observes only $S^i_t = \{H^i_t, b^i_0, ..., b^i_t, T^i_t, h^i_0, ..., h^i_t\}$. Hence from the perspective of player $i$, the state is entirely $S^i_t$, and the other variables are treated as fundamentally random (whose distribution may be learnable).

The first decision problem occurs at the initial node; after seeing their cards, each player must predict how many tricks they expect to win, given their strategy. We frame this as picking a mapping $\beta^i : H^\emptyset_0 \rightarrow [13]$, that maximizes the expected reward. The second decision problem is that in each round the player must decide what action to play given their team’s current trick surplus/deficit from the bets chosen previously, the cards played so far, the cards remaining in their hand, and so on (i.e., based on $S^i_t$).

5.3 Heuristic Algorithm

In order to test the performance of the neural network approach, we compare against a baseline heuristic algorithm programmed in accordance with advice given to new players of Four Hundred. While the algorithm
we use does not contain all knowledge or strategy that a human might possess, the fact that it defeats
greedy and random strategies in full games 100-0 suggests that it does at least play somewhat reasonably
and represents a good initial benchmark.

Below we provide detailed pseudocode for the heuristic algorithm and the most important subroutines.
In words, though, the algorithm is simple: players should first decide whether they want to attempt to take
a trick, or not, and then given this decision, which of the valid cards in their hand to play. Notice that first
decision is not trivial - it may not be optimal to always attempt to take a trick even when success can be
guaranteed, since for example, a player may be taking a trick from her teammate, and this may decrease the
overall score. If a player does not want to attempt to take the current trick, then the heuristic choice will
be to play a low-value card (ideally, this would move the player towards running out of a non-trump suit, in
order to expedite trumping opportunities, but this is not implemented in the current heuristic algorithm). On
the other hand, if a player does want to attempt to take the trick, a simple heuristic would be to determine
the smallest valid card that will guarantee taking the trick - for example, if a 10 of clubs will suffice, there is
no need to play the Ace of clubs. What is the smallest to guarantee, or likely guarantee, taking the card will
depend on the player’s position in the trick order: moving last will mean that any card of higher value than
the currently played cards will guarantee taking the trick. On the other hand, playing earlier will require a
judgment about whether higher cards have been eliminated and how likely the suit is to be trumped.

Thus, there are three main subroutines corresponding to either opening a trick, moving last, or in
between, and a subroutine for determining the safety of particular choices of cards.

Algorithm 2 Heuristic Algorithm

```
Inputs: Cards played in current trick $c_t$, Valid cards in hand $H_v$, vector of bets $b$, vector of tricks taken so far $\tau_t$

1: Determine position $p$ in order based on $|c_t|$
2: if $p == 1$ then
3:   Choice = MoveFirst(inputs)
4: if $p == 4$ then
5:   Choice = MoveLast(inputs)
6: else
7:   Choice = MoveMid(inputs)

return Choice
```

Algorithm 3 MoveFirst

```
Inputs: Same as inputs to Heuristic Algorithm

return max safe non-trump card, or smallest valid card if no safe non-trump cards are available
```

Algorithm 4 MoveLast

```
Inputs: Same as inputs to Heuristic Algorithm

Set min_winning_card as MinWinnable(inputs)
Set min_losing_card as MinLosable(inputs)

1: if partner is not winning or partner bet deficit is less than own bet deficit and min_winning_card is not none then
return min_winning_card
2: else return min_losing_card
```
**Algorithm 5** MoveMid

Inputs: Same as inputs to Heuristic Algorithm
Set \( \text{min}_\text{winning}_\text{card} \) as MinWinnable(inputs)
Set safe as DetermineSafe(min\_winning\_card)
Set \( \text{min}_\text{losing}_\text{card} \) as MinLosable(inputs)

1: if partner is not winning or partner bet deficit is less than own bet deficit and \( \text{min}_\text{winning}_\text{card} \) is not none and safe is True then return \( \text{min}_\text{winning}_\text{card} \)
2: else return \( \text{min}_\text{losing}_\text{card} \)

**Algorithm 6** DetermineSafe

Inputs: Same as inputs to Heuristic Algorithm, plus a particular card \( c \)
Set possible\_higher\_cards to be cards with suit \( c.\text{suit} \) and higher value
Set trumpers to be list of players who have trumped \( c.\text{suit} \)

1: if trumpers includes opponents then return False
2: else
3: for card in possible\_higher\_cards do
4: if card has not been played then return False

return True

5.4 NN Architecture Discussion

In this Section, we will discuss what each block means and their importance. In general, we use Convolutional Neural Networks to evaluate the \( 4 \times 13 \) representation of the deck because of its known success in analyzing images. Representing the deck of cards as a matrix encodes relative information with respect to suits in each row and value in each column. We use 2D Convolution Neural Networks with one filter and kernel size \( 1 \times 1 \) which emulates the initial state evaluation branch of AlphaGo and also minimizes parameters for computational efficiency. The kernel size and number of filters is very tunable, but not explored further due to the time constraints of this project. After passing the input through the 2D-CNN, we then apply batch normalization and a leaky linear rectifier, which we found common in practice when using CNNs. Batch normalization has been shown to speed up training. The leaky ReLu is similar to ReLu in terms of its function, however it has a small positive slope rather than 0 to allow for the gradient to never vanish. This is used in practice to avoid the “dying ReLu” problem. The last layer of the Neural network architecture is a dense layer, where every node is connected to the output.

5.5 More Results

In Figure 4, the orange line shows the bet while the blue line shows the number of tricks won. There were no bets greater than 6. In these plots, we can further see the conservative plays that the Network is predicting. Except for the case of betting 2, all of the trick curves have majority of their area to the right of the orange line indicating the bet. This means that most of the time, the bets are less than the number of tricks taken. This is not the case for betting 2 mainly because of the fact that hands which would warrant 0 or 1 tricks fall under betting 2 since that is the minimum bet.

5.6 Further Discussion

There are many further improvements to be made to our framework, as well as interesting questions to explore:

- Our method does not take the current scores into the state - it treats each round the same way. This is not ideal because toward the end of the game, only a few points are needed to win. Therefore
betting high, even with a good hand, is risky. The game is performing well by treating each round the same way, but it could be better if the information about the scores was included into the state. Such strategic considerations would likely result in improved performance against human players, and testing whether it is possible to learn long-term strategy without the use of explicit techniques like recurrent neural networks would have important applications to other problems.

- A central question is whether team-based learning must be accomplished via team-based training from the beginning, or whether an AI can be trained to play to maximize individual scores, and later trained to cooperate in teams. This latter method is more along the lines of how humans learn, and would have widespread application to coordinated and distributed learning tasks. Moreover, such an approach may speed convergence time, in addition to mirroring how human players approach learning optimal strategies.

- Currently, there is no memory of previous actions, nor long-term planning, besides what is implicitly included in considering the evolution of the game state. To be more concrete: human players often play runs - for example, a human player having opened with the Ace of Spades and holding the King of Spades is likely to play the King of Spades next, as opposed to moving to another suit. It is not clear whether this is optimal, but implementing recurrent neural networks or other memory-like structures would give our AI the ability to plan for such tactics without forcing it to.

- **Over400** represents an interesting initial pass and proof-of-concept that reinforcement learning and supervised learning can be used to achieve strong performance at this game, but there are many choices to explore in terms of architecture, tuning and regularization parameters, reward shaping, and training regiments that may improve overall performance or convergence speed.

![Figure 4: Trick Distribution given Bet](image)
References


