Policy-Based Search for Trick-Taking Card Games

Nate Gruver **Victor Cheung** **Atish Sawant**

Abstract

Trick-taking card games are a class of multiplayer games that can be naturally modeled using a POMDP framework. In this work, we examine an online solving framework similar to POMCP augmented with deep policy networks. Policy gradient methods are used to train these policy networks through a mixture of self-play and direct training against benchmarks that proves to aid convergence. Experimental results are promising, showing significant outperformance of benchmarks and performance competitive with or superior to a human player.

Introduction

Trick-Taking Card Games

Trick-taking card games are a class of multiplayer, nondeterministic, partial-information games based on rounds of "tricks" consisting of one card chosen by each player. The challenge of these games compared to other multiplayer games comes from the stochastic nature of the agent's initial cards and the partial information about the location of other cards in the hands of other agents.

These elements combined make these games more challenging in some respects than deterministic, perfect knowledge games like chess or Go, deterministic, imperfect knowledge games like Phantom Go, or non-deterministic perfect knowledge games like Fischer Random Chess (the level of non-determinism is nowhere near as high as in trick-based card games with 960 possible starting positions versus $52!$ for Hearts).

While non-determinism and imperfect information pose significant challenges to any agent, trick-based card games also have the advantage of short episodes (always 13 in Hearts vs. mean of 40 with long tail in chess), small branching factor (≤ 2197 in Hearts vs. ≤ 129960 in Go), and various game state symmetries.

All trick-taking card games have two phases. For Hearts, these phases are trading cards (strategically change one's starting hand by giving cards away) and card play (playing a policy based on a given hand). For Bridge, Skat, and Napolean, these phase are bidding (a phase that determines the value of the cards in play) and card play. The basic structure of card play is shared between all trick-taking card games and is thus the foundation of any viable AI for these games.

In this work we first construct an agent for card play in Hearts–though the algorithm is plug and play for the card play phase of any trick-taking card game. We begin with an agent with fully observations of the game state to sanity check the algorithm and then expand the agent to the true partially observable setting.

Related Work

Most research in the application of AI to trick-taking card games has focused on Bridge, perhaps the most complex trick-taking card game and a game for which the best human players still far outperform the most advanced AI (Ventos et al., 2017). The biggest successes in this field have resulted from using a "double dummy" framework and perfect information Monte Carlo sampling (Smith et al., 1998) (Ventos et al., 2017) (Long et al., 2010)–a technique similar to POMCP– as well as optimization of MCTS using a "partition search" that exploits symmetries in the state and action spaces (Ginsberg, 2001). Significant work has also gone into improving the proficiency of AI in the bidding phase of the game (Amit & Markovitch, 2006) (Yegnanarayana et al., 1996). Recently this facet of the game has become a target for research in deep reinforcement learning (Yeh & Lin, 2016). Skat is a game with structure very similar to Bridge, though with a much smaller state and action space. Many studies have also focused on the bidding phase of Skat (Keller & Kupferschmid, 2008) (Long & Buro, 2011) and AI has been developed using MCTS that is competitive with the best human players (Kupferschmid & Helmert, 2006) (Buro et al., 2009).

Past research into Hearts has seen the most contributions from Sturtevant, primarily focusing on reinforcement learning with linear function approximation (Sturtevant & White, 2006a) (Sturtevant & White, 2006b) and tree search (Sturtevant & Korf, 2003). MCTS and deep learning offer room for massive improvement over these early works.

The primary contribution of this work is to combine the early breakthroughs in Bridge card play–namely partition search and perfect information Monte Carlo sampling–with the recent advances in policy network augmented MCTS from DeepMind's AlphaGo (Silver et al., 2016). Though policy gradient techniques have been applied to card games outside of peer-reviewed journals (Grobler & Schmetz, 2016), this combination is largely unseen in the literature.

Rules of Hearts

To begin the game, all cards in the deck are dealt out between the 3-5 players. Card play proceeds through rounds of play called tricks, in which each player plays one card from his/her hand in clockwise order starting from a leading player. The first player is free to play whatever card they choose, whereas those that follow must play cards of the same suit that was lead if they have any in their hand. The player who plays the highest valued card in the suit that is lead takes the trick and thus earns the points associated with this trick, as well as gaining the opportunity to lead the first card in the next trick. Points are determined according to three criteria: the presence of the Q♠, the presence of the J♢, the number of hearts in the trick. The Q♠ is -13 points, the J♢ is +10 points, and each heart counts for -1 point. The goal of the game is to avoid taking cards with negative value–with the exception of taking all of them in which case the player has "shot the moon" and obtains +26 points. Card play proceeds until all the cards in the players' hands have been played out.

Approach

Fully Observable Case

MDP Representation

The state of the Hearts game is given by the cards in each player's hand in addition to the cards in tricks already taken (referred to as the discard pile) and the cards in the current trick being played. This representation does introduction a small loss of information as we do not record which player has played cards in the trick or discard pile, but as this information would only be used for opponent policy inference or state inference in the partially observable setting, this loss of information is not important in the fully observable case.

An action is simply a card and rewards are based on the values of completed tricks as described above. Transitions in the MDP of one player involve the actions of the other players that happen between the choice of the agent's n-th card and $n+1$-th card and rewards are based on any tricks that conclude from these actions. There are strict action constraints based on the cards available in the player's hand and the cards played so far in the trick such that $|A(s)|$ is often quite small. This underlying MDP is managed by the equivalent of an AI gym environment depicted in Figure 1. The most interesting facet of this architecture is that the "Rules Model" is plug and play for different card play phases of trick-taking card games [1].

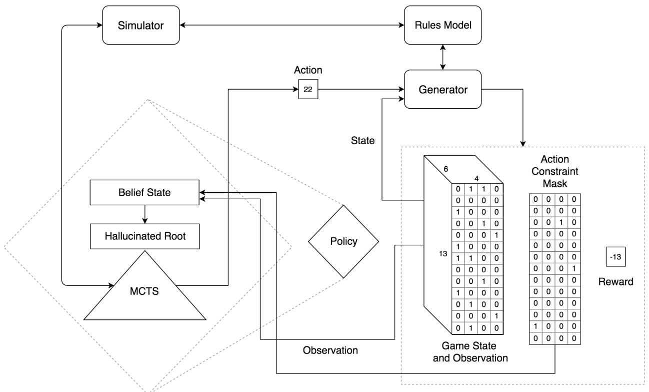


Figure 1. The architecture used for tree search and training policy networks

Tensor Representation

The state of the fully-observable game was represented by a tensor of shape $(P+2, V, S)$ where P is the number of players, S the number of suits in play and V the number of card values in each suit. The first P $V \times S$ matrices in this tensor represent the hand of each player and the last two represent the discard pile and the cards in the trick respectively. Each $V \times S$ matrix represents the presence or absence of a card with a boolean value so that for example

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

maps to

$$[9♣, 3♢, 10♡, 6♠, \text{K}♠]$$

To retain a constant frame of reference for each player, the first $V \times S$ matrix in the tensor is always the current player with the other players at increasing indices in clockwise direction from the perspective of a real table.

[1]Code is available at `https://github.com/ngruver/cs234_project`

Action Equivalence Classes

One facet of trick-taking card games that allows for optimization in tree search is the existence of action set equivalence classes. Given the following hand

$$[5\clubsuit, \mathrm{J}\diamondsuit, 9\heartsuit, 10\heartsuit, 6\spadesuit]$$

the $9\heartsuit$ and $10\heartsuit$ are equivalent, as they have the same chance of winning a trick for the player. Thus we can treat this hand as

$$[5\clubsuit, \mathrm{J}\diamondsuit, (9/10)\heartsuit, 6\spadesuit]$$

and only search over this smaller set of actions. In some cases this significantly reduces the branching factor and allows directed exploration.

Policy Gradient

Vanilla Policy Gradient with Constraints

We used a vanilla policy gradient algorithm to train a softmax policy over the action space, $\pi_\theta(a|s)$,

1. A simple feedforward network with 2 fully-connected layers and 100 neurons per layer
2. A more complex convolutional network with one convolutional layer using a kernel size of [3,3] and 12 filters and 3 fully-connected layers of 100 neurons.

Given the output logits of the policy network, σ_θ, probabilities over the possible actions were calculated via

$$\pi_\theta(a|s) = \frac{e^{(\sigma_\theta(s,a) - \beta\mathcal{C}(s,a))}}{\sum_{a'} e^{(\sigma_\theta(s,a') - \beta\mathcal{C}(s,a'))}}$$

where $\mathcal{C}(s)$ is a binary mask of the same size as σ_θ with $\mathcal{C}(s,a) = 1$ if action a is illegal in state s, and β is an extremely large number (here 10^{10}). This effectively drops the probability of any constrained actions to zero.

Policy Optimization

The policy gradient theorem states

$$\nabla_\theta V(\theta) = E_\theta[Q^{\pi_\theta}(s,a) \cdot \nabla_\theta \log_{\pi_\theta}(a|s)]$$

Using the policy defined above and L2 regularization yields the following update rule

$$\theta := \theta + \frac{1}{m}\sum_{i=1}^{m} \bar{Q}^{\pi_\theta}(s_i,a_i) \cdot \nabla_\theta \log_{\pi_\theta}(a_i|s_i) - \lambda\theta$$

For sampling we used a Dirichlet prior to encourage exploration through optimistic initialization:

$$P(a|s) = \frac{\pi_\theta(a,s) + \alpha}{\pi_\theta(s) + \alpha\, A(s)}$$

The policy network was trained through a combination of self-play and play against a handcrafted finite state machine. The specification of this FSM is included in the appendix along with parameter values used for training. In particular, we used play against the FSM to "hot-spot" the policy network. This effect would normally be achieve through imitation learning on a database of expert trajectories. As no such database is available for Hearts, however, this acted as a reasonable substitute.

Stabilizing Training

Initial training through play of the current iteration of the policy network against itself was for the most part non-convergent. We suspect that the failure to improve through self-play is due to a moving target, similar to the issues encountered in DQN training for Atari. Thus we took extra measures to give the policy network more stable targets for reinforcement learning. These took primarily two forms: first, decreasing non-determinism through random seeding and increasing stability with a target network or round-robin self-play. By seeding both opponent policies and the initial state, we gave the agent an opportunity to train more thoroughly on each random situation it encountered–playing each seeded environment 3-5 times. The target network was inspired by DeepMind's DQN algorithm (Mnih et al., 2013) and the round-robin play against older versions of itself is reminiscent of AlphaZero's style of self-play (Silver et al., 2017). All networks were hot-spotted on the FSM as described above.

MCTS with Deep Policy Network

The core of our Hearts agent is slightly modified implementation of Monte Carlo Tree search. The three basic steps of MCTS are

1. **Selection**, in which the algorithm uses an Upper Confidence Bound to decide which part of the tree to search more thoroughly.
2. **Expansion**, in which simulations of the selected portion of the tree are used to estimate the value of resulting states.
3. **Back-propagation**, in which the results of these simulations are propagated up the tree to impact the next exploration decision and final choice of action.

We used the same technique as Deepmind's AlphaGo for incorporating a policy network into MCTS (Silver et al.,

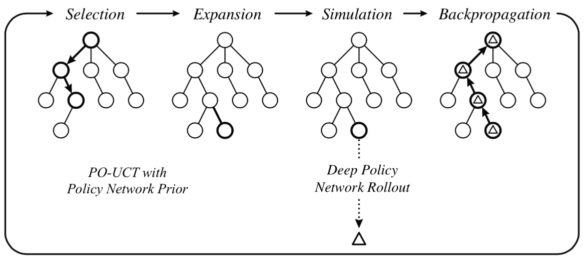


Figure 2. An iteration of MCTS using a deep policy network for selection and simulation. Adapted from (Browne et al., 2012)

2016). Namely, we used the policy network for two purposes (Figure 2):

1. As a prior in the UCT selection phase of MCTS. An action is selected according to

$$a = \arg\max_{a'} \left(Q(s,a) + c\frac{\pi_\theta(a'|s)}{1 + N(s,a)} \right)$$

   This is an amalgam of the partially-observable UCT from POMCP and UCT with a policy network prior.

2. As a rollout policy for the simulation phase of MCTS.

Unlike AlphaGo, we used the same network architecture for both of these purposes.

Direct Search

For low card counts remaining in the agent's hand, the program switches to direct expectimax or minimax search over the possible outcomes. The total possible outcomes starting at a hand with n cards when playing with p players is upper bounded by $(n!)^p$. Thus, as the current agent plays with two opponents if we take $(n!)^p < 1000$ this gives $n = 3$. Thus for all hands of 3 cards or less we do direct search.

Partially Observable Case

POMDP Representation

The POMDP model is identical to the MDP model with two additional parts: observations and belief states. Here states are unknown by the agent, who only obtain information about the underlying state through observations. The agent maintains a belief state which is a probability distribution over the possible underlying states and this belief state is updated with new observations using Bayes rule.

The observation space is simply a subset of the state space including all the information about the player's own sub-state and the trick and discard elements of the other player's sub-states. This results in a $(P+2, V, S)$ tensor. The observation

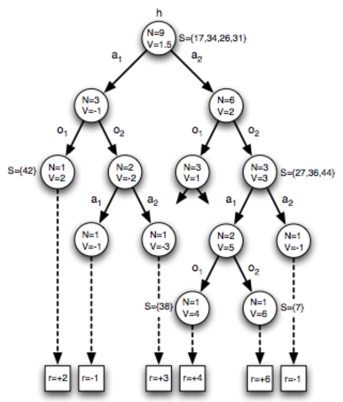


Figure 3. Graphical representation of the POMCP algorithm

model is taken to be a simply $P(o|s) = 0$ if observation o includes an observed action (card played on trick) of player$_i$ that is not in $A(s)$ given s from player$_i$'s perspective and uniform otherwise. This model thus reduces sampling from the belief state to a simple constraint satisfaction problem given the observed actions in the history. We use a least constrained variable approach, randomly filling the most constrained players' hands first.

POMCP

POMCP is an online algorithm for solving POMDPs with large state and observation spaces (Silver & Veness, 2010). It uses a partially-observable variant of MCTS to explore the value of an action. In order to actually perform tree search and rollouts using a black-box generative model, $G(s,a) \mapsto (s', r, o)$, the algorithm samples a state from the belief state at the root of the tree and hallucinates the rest of the MCTS process from this starting state (Figure 3). Value estimates are maintained for histories–which are strings of observations and actions taken–rather than states. For number of iterations $n \to \infty$, the estimated value of an action will approach the expected value of that action given the belief state. In the original paper, an unweighted particle filter is used to be as general-purpose as possible. Here, however, we can use domain knowledge to sample states as described above.

Results

For all of our experiments we used games of 3 players and a deck of 48 cards, as this is divisible by 3 with 4 even suits. We measured success by average reward per game of the agent versus expected reward from completely random actions–i.e. how many points the agent would get if the expected reward per game was 0. We call this metric differential reward. In a game of 3 players the expected reward per player for random play is -5. Additionally as the games are nondeterministic, we tested statistical significance with a t-test:

$$t = \frac{\bar{x} - \mu}{\frac{\sigma}{\sqrt{n}}} = \frac{\bar{x} + 5}{\frac{\sigma}{\sqrt{n}}}$$

Policy Network Training

Hot-spotting with FSM

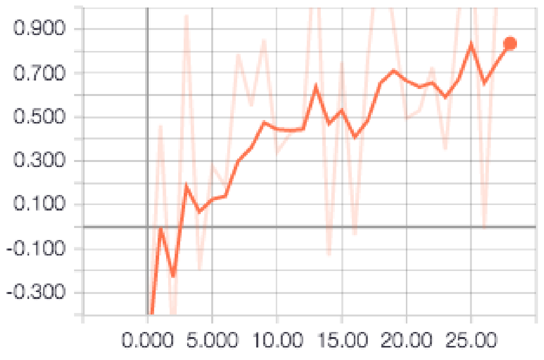
Figure 4 shows training curves for policy networks of different sizes trained directly against the FSM policy. The reward curves (4a, 4b) show stable improvement and the loss plot (4c) suggests that training approaches a deterministic policy. This is acceptable given that we are simply using the resulting policy as a starting point. Otherwise, a higher entropy policy is more desirable.

Training through Self-Play

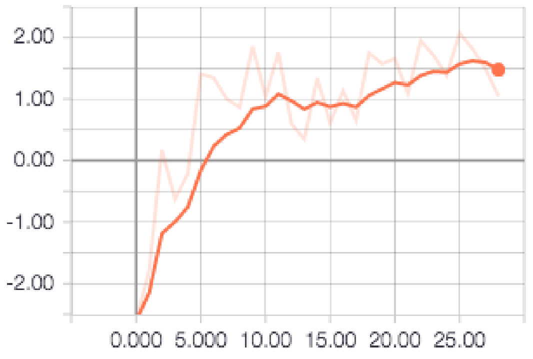
Figure 5 shows training curves for a CNN in self-play against a target network. The target network is initialized to the network resulting from the training in Figure 4. Starting 5 epochs into training the policy network is saved and then loaded back in as an opponent policy 5 epochs later. This allows the network to train against a stable but gradually improving policy. There are two promising take-aways from these results: first, the resultant maximal rewards (5a, 5b) are higher than in direct training against the FSM[2], and, second, the loss (5c) begins to drop back down below 0, implying that the policy network is learning a more general policy–as converging to non-zero is only possible if a policy with more uniform probabilities is found to minimize loss.

The effect of training against old policies is isolated in figure 6. Here, the first convolutional policy network is trained directly against the FSM. We then trained another convolutional policy network against the first. We repeat this procedure twice more and refer to this process as chaining. Figure 6a shows that despite training against policies multiple times removed from the FSM, the policy network is successfully learning against the FSM policy simultaneously. This implies a degree of generalization in the stochastic policy learned, i.e. the network is learning to play the game, and not naively memorizing strategies (overfitting) against one particular policy. Figure 6b shows gradual improvement against the random policy as well with each step removed from the original network trained against FSM.

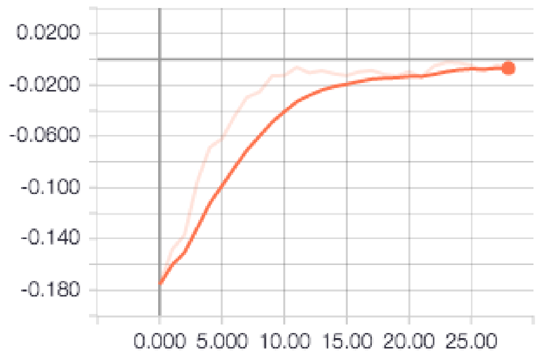
[2]It is possible to get very high differential rewards against the FSM by over-fitting a very expressive CNN against it, but this is not generalizable like the policies generate here in self-play.



(a) **Rewards vs. Random Policy**: The training curve depicts a differential reward per games of the policy network when benchmarked against a random policy

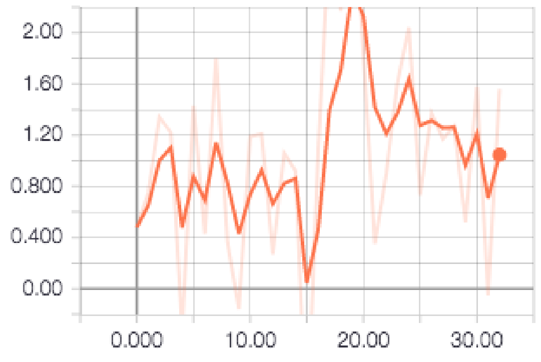


(b) **Rewards vs. FSM**: This training curve depicts differential reward per game of policy network when benchmarked against the FSM policy. It makes sense that the y-values should be higher in this plot as the policy network is being trained directly against this policy.

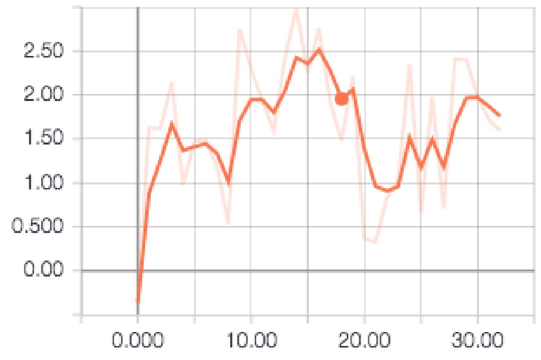


(c) **Loss**: This curve depicts the value of the loss function after every epoch on the y-axis.

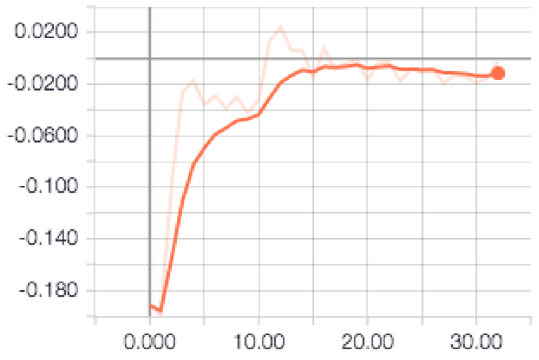
Figure 4. All training curves from training directly against the FSM policy. The x-axis is over 30 epochs, each with 5000 games and 150,000 games in total.



(a) **Rewards vs. Random Policy**: This training curve depicts differential reward per game of the policy network when benchmarked against a random policy

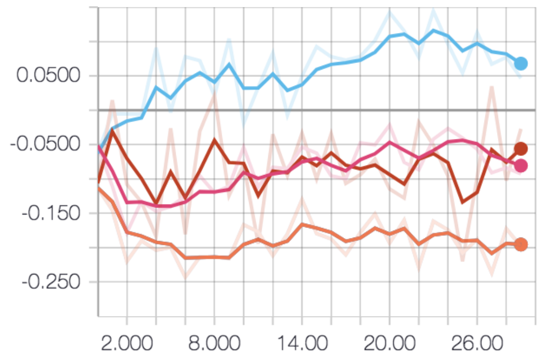


(b) **Rewards vs. FSM**: This training curve depicts differential reward per game of the policy network when benchmarked against the FSM policy.

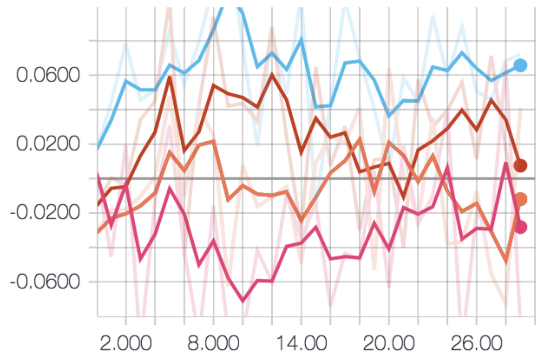


(c) **Loss**: This curve depicts the value of the loss function after every epoch on the y-axis.

Figure 5. Three training curves from training in self-play. All plots have epochs on the x-axis where on epoch is 1,000 games or approximately 20,000 state-action pairs



(a) **Rewards vs. FSM:** these curves show improvement in differential reward over time against the FSM policy by chaining the training of convolutional policy networks.



(b) **Reward vs. Random Policy:** these curves show improvement in differential reward over time against the random policy by chaining the training of convolutional policy networks.

Figure 6. Reward curves isolating the stabilizing effect of target network training. Epochs on x-axis are over batch sizes of 2,500 games and thus 75,000 in total. Networks are trained from zero-initialization against previously trained policy networks (chaining). Rewards are averaged over the batch. The order of training is given by the orange, red, pink, then blue curves.

Policy 1	Policy 2	Diff. Reward	t
Vanilla MCTS	Random	6.01	6.35
Vanilla MCTS	FSM	2.99	4.57
Deep MCTS	Random	7.27	6.81
Deep MCTS	FSM	4.8	3.98
Deep MCTS	Vanilla MCTS	1.2	2.31

Figure 7. Statistical comparison of vanilla MCTS and deep MCTS policies (using $n = 200$ and $c = 100$) against benchmark policies and each other. The differential reward displayed is that achieved by the policy listed in column 1 when played against two opponents using the policy in column 2.

Opponent Policy	Diff. Reward	t
Random	6.39	7.04
FSM	3.8	3.23
Vanilla PO-MCTS	2.22	1.92
PO Human	0.85	0.845
Vanilla FO-MCTS	-0.68	0.92
Deep FO-MCTS	-2.00	3.06

Figure 8. Statistical comparison of policy network partially observable MCTS with various benchmarks. The displayed differential reward is the reward obtained by the deep PO-MCTS agent (using $n = 400$ and $c = 100$) playing against two instances of the listed policy–with the exception of human, in which case there is one human and two AIs."PO" denotes agents with partial observability whereas "FO" denotes full observability.

Vanilla and Deep MCTS

To test the complete agent for the fully observable case, we ran trials of 100 games, pitting the vanilla MCTS and the policy network augmented MCTS against the benchmarks and each other. The results are show in Figure 7 along with measures of statistical significance. In general, results show the ability of MCTS alone to strongly outperform benchmarks but also show marked improvement with incorporation of the deep policy network. Such improvements are also evident in the results of the partially observable case.

Partially Observable Case

PO-MCTS with Policy Network

Using the incorporated agent, we performed trials of 10-100 games against various benchmarks and other tree-search policies. The results are shown in Figure 8. We can conclude that deep PO-MCTS is able to significantly outperform most other policy, with the exception of the human policy and policies that have more knowledge of the game state.

Qualitative Results

Having proved with statistical significance that the agent is performing at a high-level, we can examine some particularly nice moves that are in line with canonical strategy and in one case go beyond it.

1. **Flushing**

   Given the hand

   4♣
   3♢4♢5♢7♢10♢
   3♡5♡7♡J♡Q♡A♡
   6♠

   and empty trick, the agent plays 6♠–placing the most value on 6♠, followed by 5♡ and 7♡. This is a strategy known as flushing in which you strategically draw out the Q♠. The value placed on the hearts also reflects the possibility of drawing opponents into taking hearts tricks in a similar manner.

2. **Reverse Flushing**

   Given the hand

   J♣, Q♣
   7♢K♢
   6♠8♠10♠

   and the trick

   5♡

   the agent decides to dump the 7♢ on the trick, an initially unintuitive strategy that is non-canonical. If we consider the motivation, we realize this is actually the reverse of flushing in which we don't want to play low diamonds to draw out the J♢, as this is more likely to benefit other players.

3. **Getting the Jack**

   Given the hand

   7♣A♣
   4♢5♢7♢8♢J♢K♢
   J♡K♡
   5♠6♠Q♠A♠

   and empty trick, the agent plays 4♢. This is a brilliant play because it then draws the next player to aggressively play the A♢ in the hope of taking the J♢. This ultimately makes it more likely that the original player will be able to take the J♢ for itself.

Conclusion

Combining partially observable MCTS methods with deep policy networks yields a policy that significantly outperforms benchmarks and demonstrates play comparable to a skilled human. Perhaps the most promising facet of these results is that this level of play was achieved without the best policy network, which had $3\times$ the differential reward but was trained after the comparison trials were run.

The most obvious way in which the current implementation is lacking is that it does not play the card passing phase of the game. Incorporation of this phase will be part of a more general effort to work on the initial (such as bidding) phases of trick-taking card games. This work sets up future effort into a general trick-taking game AI.

In terms of direct algorithmic improvements, efficiency and stability of training are two areas of weakness. A much more efficient implementation of MCTS would allow for more iterations per thinking time and using a policy gradient algorithm like TRPO would remove the need for some of the hyperparameter tuning and improve the policy search.

Contributions

Nate wrote the majority of the final paper and poster, coded up the card game "gym" equivalent as well as the vanilla policy gradient framework, MCTS and PO-MCTS algorithms. He additionally gathered results for policy network training and all results for the integrated agents. Atish tested the vanilla policy gradient implementation and gathered policy gradient results, and coded TRPO, but was unable to troubleshoot it completely in time. Victor implemented the CNN and target network training framework and gathered results for these, set up the skeleton for the policy gradient framework, and handled poster logistics.

References

Amit, Asaf and Markovitch, Shaul. Learning to bid in bridge. *Machine Learning*, 63(3):287–327, 2006.

Browne, Cameron B, Powley, Edward, Whitehouse, Daniel, Lucas, Simon M, Cowling, Peter I, Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Buro, Michael, Long, Jeffrey Richard, Furtak, Timothy, and Sturtevant, Nathan R. Improving state evaluation, inference, and search in trick-based card games. In *IJCAI*, pp. 1407–1413, 2009.

Ginsberg, Matthew L. Gib: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 14:303–358, 2001.

Grobler, Conrad and Schmetz, Jean-Paul. Cs221 project final report learning to play bridge. 2016.

Keller, Thomas and Kupferschmid, Sebastian. Automatic bidding for the game of skat. In *Annual Conference on Artificial Intelligence*, pp. 95–102. Springer, 2008.

Kupferschmid, Sebastian and Helmert, Malte. A skat player based on monte-carlo simulation. In *International Conference on Computers and Games*, pp. 135–147. Springer, 2006.

Long, Jeffrey Richard and Buro, Michael. Real-time opponent modeling in trick-taking card games. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, pp. 617, 2011.

Long, Jeffrey Richard, Sturtevant, Nathan R, Buro, Michael, and Furtak, Timothy. Understanding the success of perfect information monte carlo sampling in game tree search. In *AAAI*, 2010.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Silver, David and Veness, Joel. Monte-carlo planning in large pomdps. In *Advances in neural information processing systems*, pp. 2164–2172, 2010.

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): 484–489, 2016.

Silver, David, Hubert, Thomas, Schrittwieser, Julian, Antonoglou, Ioannis, Lai, Matthew, Guez, Arthur, Lanctot, Marc, Sifre, Laurent, Kumaran, Dharshan, Graepel, Thore, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Smith, Stephen J, Nau, Dana, and Throop, Tom. Computer bridge: A big win for ai planning. *Ai magazine*, 19(2):93, 1998.

Sturtevant, Nathan and White, Adam M. Learning to play hearts. 2006a.

Sturtevant, Nathan R and White, Adam M. Feature construction for reinforcement learning in hearts. In *International Conference on Computers and Games*, pp. 122–134. Springer, 2006b.

Sturtevant, Nathan Reed and Korf, Richard E. *Multi-player games: Algorithms and approaches*. PhD thesis, Citeseer, 2003.

Ventos, Veronique, Costel, Yves, Teytaud, Olivier, and Ventos, Solène Thépaut. Boosting a bridge artificial intelligence. 2017.

Yegnanarayana, B, Khemani, Deepak, and Sarkar, Manish. Neural networks for contract bridge bidding. *Sadhana*, 21(3):395–413, 1996.

Yeh, Chih-Kuan and Lin, Hsuan-Tien. Automatic bridge bidding using deep reinforcement learning. *arXiv preprint arXiv:1607.03290*, 2016.

Appendix

Code

The source code for this project can be viewed at `https://github.com/ngruver/cs234_project`

Finite State Machine Description

```
if not leading and have a card in suit lead:
    if suit lead is diamonds:
        play largest diamond in hand
    else:
        play smallest possible card
elif not leading and don't have card in suit lead:
    if have queen of spades:
        play queen of spades
    elif have spade larger than queen:
        play largest spade
    elif have any hearts:
        play largest heart
    else:
        play largest possible non-diamond
else: #leading
    play smallest card
```

Training Parameters

For our most effective trials we used learning rate $= 10^{-2}$, $\lambda = 10^{-6}$, α (Dirichlet prior) $= 10$ with batch sizes between 1000 and 5000 games and no discounting. The simple fully-connected network described in the methods section was used to train the policy that seed self-play and that was used for the experimental results of the incorporated agents shown here. The CNN architecture was used in later, more successful experiments with self-play and was also tested directly against the FSM and shown to be able to handily outperform this policy.