Stanford University: CS230

Multi-Agent Deep Reinforcement Learning in Imperfect Information Games
Category: Reinforcement Learning

Eric Von York
zataomm@stanford.edu

March 25, 2020

1 Problem Statement

Reinforcement Learning has recently made big headlines with the success of AlphaGo and AlphaZero [7]. The fact that a computer can learn to play Go, Chess or Shogi better than any human player by playing itself, only knowing the rules of the game (i.e. *tabula rasa*), has many people interested in this fascinating topic (myself included of course!). Also, the fact that experts believe that AlphaZero is developing new strategies in Chess is also quite impressive [4]. For my final project, I developed a Deep Reinforcement Learning algorithm to play the card game Pedreaux (or Pedro [8]). As a kid growing up, we played a version of Pedro called *Pitch with Fives* or *Catch Five* [3], which is the version of this game we will consider for this project.
Why is this interesting/hard? First, it is a game with four players, in two teams of two. So we will have four agents playing the game, and two will be cooperating with each other. Second, there are two distinct phases of the game: the bidding phase, then the playing phase. To become proficient at playing the game, the agents need to learn to bidbased on their initial hands, and learn how to play the cards after the bidding phase. In addition, it is complex. There are $\binom{52}{9} > 3 \times 10^9$ initial hands to consider per agent, coupled with a complicated action space, and bidding decisions, which makes the state space too large for tabular methods and one well suited for Deep Reinforcement Learning methods such as those presented in [2, 6].

1.1 Rules of the Game

In order to understand that algorithms developed, and how we will evaluate them, we must first understand the game we are going to play. *Catch Five* is a four player game of the *trick taking* variety. There are two players per team, sitting opposite of each other. The goal of the game is to win *tricks* that contain cards that are worth points. There are 9 total points that can be won in a complete hand of *Catch Five* (more on this in a bit). Initially, the dealer deals 9 cards to each player. The player to the dealer's left then makes a *bid* which indicates the number of points they think they can win in that hand based on the 9 starting cards. Players do not have to bid and may pass if they choose. This happens in turn for the next two players, where they either bid (higher than the highest bid so far), or pass until it is the dealer's turn to bid. The dealer then can decide to take the bid for an amount equal to the highest bid so far, or pass. If everyone passes to the dealer, the dealer must make a bid. The winning bidder then gets to pick the *bidders suit* to be played for that hand. For example, after the deal, *Player 1* can bid 3 points, *Player 2* can pass, *Player 3* can bid 4 points, and the *Dealer* can decide to take the bid for 4 points. At this point, the winning bidder chooses the *bidders suit*, and all players discard all cards in their current 9 card hands that are not the *bidder's suit.* Continuing with our example, if the *Dealer* called Hearts as his suit, then all players would discard all cards in their hands that were not Hearts. After this is done, the dealer would replenish all players hands until their current card count equaled 6 cards[1]. At this point, there are six rounds of *tricks*, consisting of each player playing one card. The first *trick* starts with the winning bidder playing one of the 6 cards in their hand. The suit of the first card played in a trick is called the *lead suit.* The players that play after the first card is played, must play the *lead suit* suit if possible. If they cannot follow

[1]There are some subtleties we are glossing over, such as a player having 7 cards of the bidding suit, but there are rules for these situations as well, and there are rules to ensure that all point bearing cards make it into players final hands.

suit, then they can play any card they choose in their hand. Within a given suit, the cards have their normal order, i.e

$$2 < 3 < 4 < \ldots < K < A.$$

However, any card of the *bidder's suit* out ranks all cards of any other suit. So, in our example, the Two of Hearts out ranks the Ace of Spades. The winner of the current trick, takes the cards and plays the first card in the next hand. As stated previously, there are 9 points per hands, where the Two, Ten, Jack, and Ace of the *bidder's suit* are worth 1 point, and the Five of the *bidder's suit* is worth 5 points, hence the name of the game *Catch Five*! Points that each teammate scores are combined into the team score for the set of six *tricks*.
There are several strategic things to consider when playing this game, the first one being bidding. Obviously a player with a good initial set of 9 cards in their hand would like to bid high enough so that they can control the game and dictate the *bidder's suit*. However, after a complete set of 6 tricks are played, if the bidding team does not obtain at least the number of points they bid, then their score for the game is negative the amount they bid. In other words, if the team that won the bid by bidding B points does not obtain at least B points in the tricks that they win, then they receive a score of $-B$ points for that round. So, there is a risk reward trade-off that our Deep Neural Network will need to navigate. In addition to this, there is definitely some strategy to playing the tricks to try and maximize the points your team scores. For example, if you are the winning bidder and your bidding suit was hearts, you may want to play the ace of hearts as your first cards, expecting that any point cards that your partner is holding, will be played this hand so that your team can accumulate more points; in particular if your partner is holding the 5 of the *bidder's suit*. At times, you may also want to win a particular *trick*, even if it contains no points, just to be able to determine the lead suit for the next hand, to elicit information regarding your opponents hands, or force your opponent to play certain cards because they have to follow suit.

1.2 Basic Approach

Our approach to this problem was to use *end-to-end* Deep Learning to teach our agents these distinct aspects of the game using an actor-critic policy gradient method known as Proximal Policy Optimization [6]. We combined the bidding and playing into a single game state and trajectory and tried to capture all of the nuances of the game in this representation. In Section 2 we discuss in detail how we represented the game. Then in Section 3 we will present the evaluation criterion we developed to assess the bidding and playing of the agents as we tested different architectures and hyper-parameters. In Section 4 we will present a brief overview of the Proximal Policy Algorithm as presented in [6] and discuss the details of the implementation of this algorithm. In Section 5 we present the results of our development of the PPO algorithm applied to the game of *Catch Five* and show agents that improve play significantly as training progresses.

2 State and Action Space Representation

The first problem to consider is how to represent the state and action spaces for the agents playing the game. As mentioned above, there are two basic phases of the game: bidding, and game play. In the bidding phase, each player will choose a bid b such that:

$$b \in \mathcal{B} = \{\text{Pass}, 3, 4, 5, 6, 7, 8, 9\}.$$

Next, the winning bidder chooses a suit to be the *bidder's suit* that will out rank all of the other suits, choosing s such that

$$s \in \mathcal{S} = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}.$$

In addition to representing the bidding, we must also track the cards in each agent's hand, as well as all of the cards that are either (1) in play, or (2) have been played and observed by all of the agents. We must also do this from each individual agent's perspective. In other words, agents will have knowledge of *all* the cards they hold, on both the initial deal, as well as the secondary deal after the bidding phase. They will only have knowledge of the players discards after they have been played in the actual game. To represent this we need to track the bids, the suit chosen, the current cards in play, the cards in the agents hand, and the observed discards of the other players, as well as the agent's discards.
After experimenting with several ways to represent this effectively[2], a state representation that was a 1×504 binary vector was used that captured all the elements of the game, and allowed us to determine which phase of the game we were in by how the different sections were populated. The table below describes the vector description, with the size of each section in bits, as well as the maximum number of bits that could be set per-section:

Bids	Suit	Cards In Play	Agent's Live Cards	Agent's Discards	Player 1's Discards	Player 2's Discards	Player 3's Discards
32	4	52x4	52	52	52	52	52
≤ 4	≤ 1	≤ 4	≤ 9	≤ 14	≤ 5	≤ 5	≤ 5

As you can see the vector will be sparsely populated, with at most 47 bits set. For the bids, 8 bits will be used for each player's bid, where one bit of $s_0, s_1, \ldots, s_7$ will be set for the agent's bid choice (i.e. 10000000 indicates Pass, and

[2] I started with an integer representation instead of binary of dimension 42 but could not get this to successfully learn the game. It was useful for easily tracking the game state, but failed to learn.

01000000 indicates a bid of 3). The remaining three sets of 8 will indicate the bids by the remaining players, when they have made them. For the suit, once it is chosen by the winning bidder, one bit represents each suit, i.e. the bits $s_{32}s_{33}s_{34}s_{35}$ will be set as

$$1000 = \clubsuit, 0100 = \diamondsuit, 0010 = \heartsuit, 0001 = \spadesuit.$$

For the cards, there will be one bit set in each of the 52 binary vectors for the cards played by the respective players and agent in the live hand. For the remainder of the 52 long vectors, there will be a bit set for each card that the agent can play, or that was witnessed being discarded by the agent, or other players in the game being played. The vectors representing the decks of cards will be ordered by suit in alphabetical order, and by value from $2, 3, 4, \ldots, K, A$. Each state is oriented to the particular agent's view of the table. For the bids, the first 8 bits represent the agents bid choice, the next 8 bits represent the player to that agents right, and so on counter-clockwise around the table. Each agent will track their own view of the game in this way.
The action space is much more straight forward to represent. For a given state at time t, s_t, here are 64 possible actions a_t total, not all allowed at each phase of the game. These actions are choosing one of 8 possible bids, choosing one of 4 possible suits, and playing one of 52 possible cards, giving 64 actions in total. We represent this as a 1×64 vector in the order of bids, suit choices, and cards with the same order as described above. As already mentioned, not all actions are allowed at different phases of the game. We will discuss how we handled this in Section 4.2.

3 Evaluation Criterion

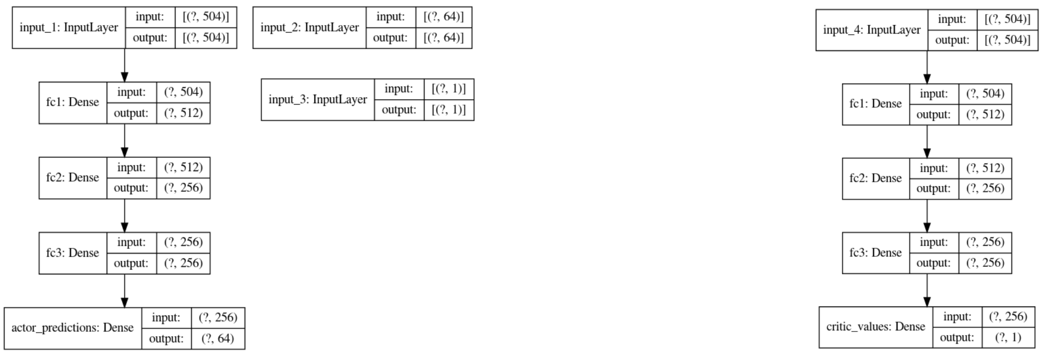
In order to evaluate our agent's ability to learn to play the game of *Catch Five* we need to evaluate the agent's ability to bid, to select a suit, and to play cards once the *trick* phase of the game begins. One obvious way to evaluate our agent's progress as we attempt to optimize a policy for playing this game, is to have the later agents play the earlier agents in a set number of games. This is certainly a good way to evaluate; however, we could have a faulty implementation, and we could simply be getting better at being worse!! So, in addition to seeing if our agent is improving against itself, we will compare it against three simple game playing agents. The first agent will play completely randomly. When the action space is presented for a given state, the agent will choose an action a_t with equal probability (among the legal actions). For example, if the random agent wins the bid, the agent chooses a suit as the *bidder's suit* at random. If we cannot beat this agent, then we have not done too much in the way of training a Deep Neural Net to play this game! The second agent we will use is one that always passes on bidding whenever possible, but still plays the other aspects of the game randomly. Since bidding does involve some risk (reward of $-B$ if points are not $\geq B$), and the expected value of a bid that a random team makes is ≈ 6.78, eliminating bidding for the random team will (1) test if our agents are bidding properly, and (2) if they are choosing the proper suit according to the cards in the players hand. Our final random agent that we test against will bid and play randomly. However, when choosing a suit, the agent will pick the bidder's suit based on the suit that contains the maximum number of cards in their initial deal. This agent will play and bid randomly, but will generally be playing with decent cards when they have won the bidding (which will be often).
In addition to the tournaments mentioned above, we will also track the agent's average bid, the number of bids won, and the average rewards per game when the agent won the bid (i.e. got to decide what suit to play) and when they did not win the bid, and were forced to play with the hand they were dealt. Tracking these metrics will allow us to see if our agents are actually learning the subtleties of bidding and playing as we continue to try to optimize the policy.

4 Choice of Algorithm

There are several algorithms that are potentially well suited for this task; in the end, Proximal Policy Optimization (PPO) as detailed in [6] seemed like the best choice. PPO is an actor-critic policy gradient method that learns online (unlike Deep Q networks). PPO is relatively easy to code, sample efficient, and easy to tune the hyper parameters. In fact, the defaults seemed to work fairly well. PPO was developed as an improvement to Trust Region Policy Optimization methods as detailed in [5]. There are two networks at work in this set-up. The first, an actor that will learn to play the game and choose the optimal action give the current state of the game. The second, a critic network that will learn to estimate the value of the states from the batches of trajectories, and provide the actor network with an evaluation of the value of the current state via a generalized advantage estimate, which is a smoother estimate of the return function from a given state s_t till the end of the episode (see [6] for details). In addition to this, PPO was applied to a multi-agent imperfect information card game known as *Big2* as detailed in [1]. *Big2* does not have bidding, and is more like playing Rummy with poker hands. Although my implementation differed from the one detailed in the paper referenced, and zero code was reused from the git repo referenced in the paper, it was comforting to know that this algorithm was applied to a similar problem successfully.

4.1 Choosing an Architecture and HyperParameters

In the early stages of development, we toyed with basic network structure, both the number of layers and number of neurons per layer as well. In the end we settled on having the base of the Deep Neural Network to consist of three


input_1: InputLayer | input: [(?, 504)] | output: [(?, 504)]
input_2: InputLayer | input: [(?, 64)] | output: [(?, 64)]
fc1: Dense | input: (?, 504) | output: (?, 512)
input_3: InputLayer | input: [(?, 1)] | output: [(?, 1)]
fc2: Dense | input: (?, 512) | output: (?, 256)
fc3: Dense | input: (?, 256) | output: (?, 256)
actor_predictions: Dense | input: (?, 256) | output: (?, 64)
input_4: InputLayer | input: [(?, 504)] | output: [(?, 504)]
fc1: Dense | input: (?, 504) | output: (?, 512)
fc2: Dense | input: (?, 512) | output: (?, 256)
fc3: Dense | input: (?, 256) | output: (?, 256)
critic_values: Dense | input: (?, 256) | output: (?, 1)

(a) Actor Network (b) Critic Network

Figure 4.1: Actor Critic Network Architecture

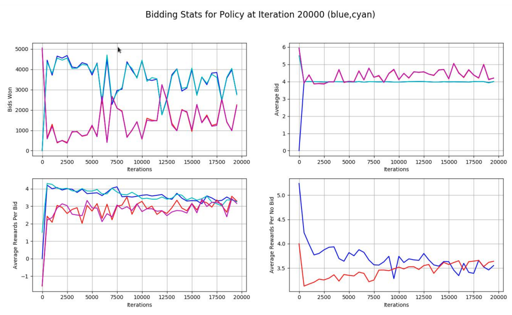
hidden layers of size 512, 256, and 256 respectively, giving us 472,128 trainable parameters. The input layer to both the actor and critic will consist of the 504 dimensions input state, and the output later of the actor network will be a softmax activation on the 64 possible actions. The output of the critic will be a *tanh* giving a normalized estimate of the value function of the given state s_t. A plot of the network models is shown in Figure 4.1.
The main hyper-parameters to choose unique to the PPO algorithm are the *clipping value*, *entropy beta*, and *critic discount*. The clipping value limits how much one adjusts the policy on each gradient ascent step of the algorithm, ensuring no over corrections are done and your policy does not wander to far afield. The entropy beta adjusts how much exploration one does in the beginning of training and is a scaling factor on the standard entropy calculated for the output of the softmax layer of the network. The idea here is that in trying to maximize your policy gradient, the entropy term would cause you to explore more in the beginning of training by trying to make the probabilities closer to equal. The critic discount is used if you are going to combine both the actor and critic network into one base network. For the clipping value we used the default value of 0.2 as presented in [6]. We experimented a bit with the entropy beta but this did not have any noticeable effect on our training and we left this at 0.01. The critic discount was not used since we opted to train the actor and critic networks separately. This was due more to my lack of knowledge of Keras/Tensorflow that anything else, and I plan on revisiting using combined early layers of the network in the near future. Experimentation was also done with different activation functions in this network, and *Thanh* and *Leaky ReLU* activation functions both learned to play the game reasonably well. We used Xavier initialization when using *tanh* activation's (*Glorot normal* in Keras) and *He uniform* initialization with *Leaky ReLU*, with a 0.01 slope on the negative x-axis.

4.2 Implementation

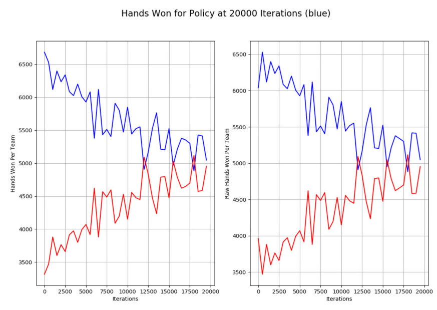
The code for my implementation of the PPO algorithm for learning the game of *Catch Five* can be accessed via GitHub at `https://github.com/Zataomm/cs230_catch5`. The algorithm was implemented in python using Keras and Tensorflow 2.1. The main program is catch5_env.py, which defines the agents environment, and determines how the agent interacts with the environment. The c5ppo.py module defines the PPO network and its parameters. c5utils.py provides utility functions for analyzing hands, detailed printing of states, counting cards of a given suit in a hand, and other useful functions. The class c5_train.py was used to train the multi-agent Deep Neural Network and will generate trajectories to feed to the Actor-Network, as well as trajectories and their returns to feed the critic network. The class c5_simulation.py is used to setup playing tournaments using the drivers c5_play.py or c5_tournament.py.

5 Results

Our initial attempts focused on using ReLU activation functions with the network architecture described above. In doing these initial experiments, the agents tended to determine early on that bidding was dangerous, and that suit selection did not matter (which it does not if you rarely win the bid!). These early agents tended to develop a policy that rarely bidded, and would always select one suit when it did win the bid no matter the cards in hand. In looking at the output probability distributions, there were actions that would always have probability zero after some training time and I assumed this was happening due to neurons dying off in the network. After this, I changed the activation functions to tanh and had more success in training the network to play and bid properly. In addition to this, I also created a data augmentation function (see c5_train.py) that would take any trajectory representing a full game of *Catch Five* and apply a random 4-long permutation to the state that would shuffle the suits, i.e. the relative positions of the binary vectors representing the cards and the choice of bidder's suit. This would allow for more data to be easily generated and analyzed by the network, and allow for the network to learn on a more diverse data set. The initial results for the network given in Figure 4.1 with the following hyper-parameters;

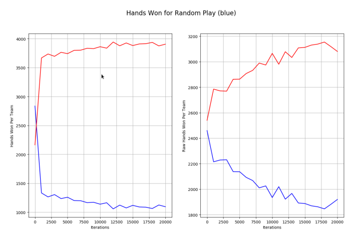


(a) Bidding Analysis

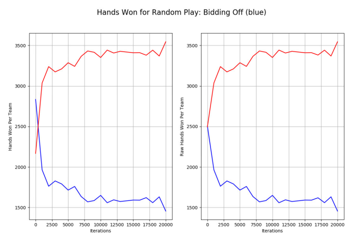


(b) Win Rates

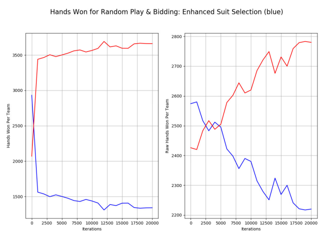
Figure 5.1: Policy at 20K Iterations (Blue,Cyan) vs Earlier Policies (Red,Magenta)



(a) Random



(b) Bidding Off



(c) Good Suit Selection

Figure 5.2: Three Random Agents (Blue) vs Trained Policies at Increasing Iterations (Red)

Activation	# State Action Pairs	Batch Size	Epochs	Learning Rate	Clipping Value	Entropy Beta	γ	λ
tanh	928 (32 full games)	8	5	0.00003	0.2	0.01	0.99	0.95

We trained the network for 20,000 iterations using the above parameters. For testing, we ran the policy after 20K iterations, against policies from earlier phases of training from 0 to 19.5K iterations in steps of 500. 10K games were played for each point on the graphs. Figure 5.1 show the comparison of the policy after 20K iterations against earlier policies. In particular, Figure 5.1a shows the bidding analysis obtained. The top left graph shows the number of bids won by each player (blue, cyan = Policy 20K, red, magenta = Earlier policies). From this graph we see that the 20K policy is winning most bidding contests. The graph in the top right shows the average *winning bid* per player, and from this we see that the 20K Policy is bidding on average around 4 points per hand and the previous graph shows this policy winning more bids with a lower winning bid average, so bidding smarter than previous policies. The lower left graph shows the average rewards per player's team when they won the bidding phase. This graph shows that the 20K policy is consistently winning more points than the earlier policies when they win the bid. Finally, the graph on the lower right of Figure 5.1a shows the average rewards per team when they have not won the bid, and again this graph shows that the 20K policy is playing better than the corresponding earlier policies even when not allowed to choose the *bidder's suit*. The graphs in Figure 5.1b show the number of games won by the Policy after 20K iterations against the earlier policies. The left graph shows the game win rate when you consider the *adjusted true score* for the game, i.e. penalty applied for not meeting your bid. The graph on the right shows the win rates for *unadjusted raw score*. We looked at this in order to get a gauge on impact bidding is having as compared to the way the agents are playing the game. Once again, there is significant improvement as you train longer. The performance against the three random opponents as described in Section 3 above is depicted in Figure 5.2, and here we clearly see as the iterations increase, the trained policy quickly outperforms these basic agents in each case. Finally, I personally analyzed many of the games that the later agents were playing, and observed sophisticated play by the agents, e.g. bidding high when recognizing a strong hand, and recognizing when the opponents are out of cards in the bidder's suit, and the five of that suit can be played and safely won.

6 Next Steps

There is alot more that I want to do with this project. First, I would like to make a more exhaustive search over the hyper-parameter space to fine tune the algorithm. Second, I implemented the actor-critic as separate networks, and will revisit combining these networks to see if this increases the speed of training, and or the quality of the agents at different levels. I also want to explore how data-augmentation mentioned on Section 5 affects training. I did manage to get networks with Leaky ReLU functions to play better than the random agents, but still need to look at improving the training and performance of these networks. In addition to this, I would like to add the ability for training to take place with earlier agents and not simply the latest agent as is currently implemented, since this will improve the later agent's play. I also want to build an interface and have the trained agents play human players as well. As a stretch goal, I may try to integrate the game into the RLCard package [9].

References

[1] Henry Charlesworth. Application of Self-Play Reinforcement Learning to a Four-Player Game of Imperfect Information, 2018.

[2] Johannes Heinrich and David Silver. Deep Reinforcement Learning from Self-Play in Imperfect-Information Games, 2016.

[3] pagat.com. Rules of card games:pedro. https://www.pagat.com/allfours/pedro.html#fives (link verified February 2020), 2020.

[4] Matthew Sadler and Natasha Regan. *Game Changer; AlphaZero's Groundbreaking Chess Strategies and the promise of AI.* New In Chess, The Netherlands, 2019.

[5] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017.

[7] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, 2017.

[8] Wikipedia. Pedro (card game). https://en.wikipedia.org/wiki/Pedro (card game) (link verified January 2020), 2020.

[9] Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: A toolkit for reinforcement learning in card games, 2019.