Pokerbots Course Notes

Massachusetts Institute of Technology
6.176: Pokerbots Competition*

IAP 2020

Contents

*Contact: pokerbots@mit.edu

1 Lecture 1: Introduction to Pokerbots

This lecture is taught by David Amirault.[1] All code from lecture can be found at the public github.com repository mitpokerbots/reference-lecture-1. The slides from this lecture are available for download on Stellar.

At this lecture (and all others), we raffle off a pair of Beats Solo 3 Wireless Headphones. Attend lectures in person if you want a chance to win them!

We'd also like to thank our eight Pokerbots 2020 sponsors for making Pokerbots possible. You can find information about our sponsors in the syllabus and on our website, and drop your resume at `pkr.bot/drop` to network with them. Our sponsors will also be able to see your progress on the scrimmage server, giving you a chance to stand out over the course of our competition.

1.1 Class Overview & Logistical Details

There will be six 90 minute lectures running on MWF 1:00 – 2:30 pm from 1/6 to 1/17 in 54–100, and office hours will be held during the first three weeks of IAP.[2] There will also be a live scrimmage server for the first three weeks, on which you can challenge any other team in the class as well as our reference bots. Weekly tournaments will be held on the scrimmage server every Friday night, and there will be prizes for winning teams. The final tournament and event will be held on January 31, 2020, which is where the Pokerbots 2020 winners will be announced. The final event will also feature more prizes, an expert guest talk, winning strategy analysis, a chance to play against the bots, networking with sponsors and more! This year's Pokerbots prize pool is over $32,000, distributed over many different categories—the syllabus lists many of the categories we will be awarding. The six lecture topics will be as follows:

1. Introduction to Pokerbots
2. Poker Strategy + Bot Demo
3. Inference
4. Game Theory
5. Advanced Topics I
6. Advanced Topics II

To receive credit for the class, you must submit bots to the scrimmage server. Your bot for each week has to defeat your bot from the previous week, as well as a random bot in a one-shot tournament. At the end, you must also submit a 3-5 page long strategy report.[3] More guidelines will be announced later in the course.[4] Take special care to read the Rules and Code of Conduct on Stellar.

1.2 Introduction to Poker

For this section, we will be talking about the game known as heads-up no-limit Texas Hold'em ("poker"). The objective of poker is to win as many chips as possible. Players bet into a pot in several rounds, and the pot is won by the player with the better poker hand at the end. In a single betting round, the first player can either bet 0 (check) or any amount between the "big blind" and number of chips they have left. If they check, action passes to the second player, and if they bet, the second player can *fold* (quit the round), *call* (bet the same amount as the first player), or *raise* (bet more than the first player, up to the number of chips they have left). A player's final "poker hand" is determined as the best five-card hand that can be formed out of seven cards: their unique two *hole* cards and five shared *community* cards.

[1] Email: davidja@mit.edu.

[2] The schedule and locations for our lectures and office hours can be found on Piazza in post @8.

[3] You are welcome to include images or code snippets in your final report, as well as discuss strategies you attempted that did not pan out. This is an open ended report, and is for us to gain insight about how you approached the Pokerbots challenge.

[4] All class details are included in the syllabus, available on Stellar at `pkr.bot/stellar`.

The first betting round is special because it begins with blinds, a forced amount players have to bet. The next time around, there is no minimum amount.

The structure of a game is as follows: the game begins with each player receiving two hole cards. The first betting round takes place, and then the "flop" (three community cards are revealed). After another betting round, there is the "turn" (a fourth community card is revealed). Another betting round occurs, and there is the river (a final fifth community card is revealed). The last betting round takes place next, followed by settlement (cards are revealed and the pot given to the winning player).

The possible poker hands are displayed in the slides, in order of best to worst hands starting with the top left hand. The final hand is called "high card," which is none of the displayed ones. Even within each hand, there are tiebreakers if both players have the same hand. Make sure to look up, using one of the provided resources, which hands are better when building your bot.[5]

1.2.1 2020 Variant: "Permutation Hold'em"

This year, our variant of poker is Permutation Hold'em. Permutation Hold'em is based on the popular poker variant Texas Hold'em with a modification that the card values are randomly permuted each game. A poker hand's strength is determined using the permuted card values rather than the card values seen by the players. A random permutation of the card values 2–A is sampled at the start of each game, and the permutation is fixed for all rounds of the game. The sampling process uses a fixed prior distribution. The permutation may be different from one game to the next. Details of the prior distribution used to choose permutations can be found in the variant write-up on the 2020 class Stellar site.

Betting is also different from Texas Hold'em in that players have the same bankroll during every round (200 for this variant). Winners are calculated in terms of their change in bankroll aggregated across rounds.

1.3 Skeleton Bot Setup

1.3.1 Github and Version Control

GitHub is a version control/code management system. Using it, *clone* the public Pokerbots repository mitpokerbots/engine-2020 (available on github.com) to get started. If you've successfully set up Git on your machine, this can be done by navigating to the directory where you'd like to keep the code and running the command

```
$ git clone https://github.com/mitpokerbots/engine-2020.git
```

Skeleton bots for all supported languages are included in this repo.

We recommend that you create a new private repository of your own to code in on github.mit.edu, and then set this up by cloning it into your working directory and copy-pasting the engine files (`engine.py`, `config.py`, `cpp_skeleton`, `python_skeleton`, and `java_skeleton`) into the clone of your own repository. On the Pokerbots GitHub, the folder "python-skeleton-2020" contains the Python 3.7 skeleton bot. There are also Java and C++ skeletons available on our GitHub repository.

To upload code from your machine, you have to create a *commit*. To make a commit, *add* the changed files you want to push, describe it with a commit message, create the *commit*, and *push* the commit online. Your partners can now *pull* your changes to their own desktops.[6] For example, after editing `player.py`, you would push it to Github with the following commands

```
$ git add player.py
$ git commit -m ``made our bot super cool''
$ git push
```

Table 1 lists some common Git commands for your convenience.

[5]The following resource is great for learning more about Texas Hold'em: `pkr.bot/poker-rules`.

[6]You can learn more about this workflow at `http://web.mit.edu/6.031/www/fa19/getting-started/#git`.

Command	Description
git clone **your_link**	Downloads code from remote
git status	Print the current status of your repo
git pull	Pulls latest changes
git add **your_files**	Stages changes for commit
git commit -m "**your_message**"	Commits added changes
git push	Pushes your changes to remote

Table 1: Important Git Commands

1.3.2 Connecting to the Scrimmage Server

To use the scrimmage server, go to `pkr.bot/scrimmage`. There, you can create or join a team with your one to three partners. To upload a bot, go to the "Manage Team" tab. Bots must be submitted as a zipped file, which you can easily do by going to "Clone or download" on your online GitHub repository and downloading your repo as a zipped folder. After you set one of your uploaded bots as your main bot, you can challenge any of the teams on the scrimmage server. If a team has a higher ELO rating than you, your challenge will be automatically accepted—otherwise, they must accept your challenge request.

1.4 Testing Your Bot Locally

To test your bot locally (without using the scrimmage server), you have to download the engine—again using GitHub. The engine consists of two files: `engine.py`, which runs your bot, and `config.py`, which contains parameters for your bot. The default parameters for the game, `BIG_BLIND` and `STARTING_STACK`, are the values we will be using, so don't change those. You should feel free to change `NUM_ROUNDS` and the time-related parameters; however, `STARTING_GAME_CLOCK` is capped at 30 for our tournament, since we do not want bots pondering for a long time. Before running the engine, you must specify which bots you wish to use in your local game. This is done by providing the file path of each bot in `PLAYER_1_PATH` and `PLAYER_2_PATH` (you may use the same file path to pit a bot against itself). To run the engine, we will be using command line. Change the working directory as needed to your engine folder, and then run `python3 engine.py`.[7] You will be greeted by the MIT Pokerbots logo, and your game log(s) will be output.

Looking at the game logs, we can see every action taken in each game and the permuted cards with the corresponding true values. The log shows the action chosen by each bot. You can also see that the log notes the flop, turn and river. The engine does not tell you what type of poker hand each player has—this has to be determined by yourself—but it will do the calculations and tell you who won.

In addition to the game logs, you will get a dump file, and this will be done for each bot. If you put any print statements in your bot, they will show up in the dump file. If you have an error in your code, the error descriptions will be in your dump file as well.

1.5 Overview of Skeleton Bot Architecture

Now, we'll look at the Python skeleton bot itself (`player.py`). The function `__init__` simply initializes the player object when a new game starts, and is useful for initializing variables that you want to access over the course of the game. The function `handle_new_round` is called at the start of every round, and the parameter `game_state` contains some information about the current state of your game. The function `handle_round_over` is called whenever a round ends, and is thus great for updating variables using the information from the last round played.

The `get_action` function determines your bot's move based on the presented information in the function's parameters—it's where the magic happens. Each of the commented-out lines contains important variables and their respective explanations, which you will likely find very useful as you develop your pokerbot.

[7] Depending on your setup, the command used may vary. Please refer to the setup Piazza post.

1.6 Coding Lecture 1 Reference Bot

We will now be implementing a basic inference bot and submitting it to the scrimmage server. The first thing we need is a way to evaluate how good cards are—we will do so by keeping track of how often we or our opponent wins for each card rank we or they, respectively, hold using dictionaries. We initialize two dictionaries in our `__init__` function: a `self.wins_dict` and `self.showdowns_dict`. Note that we map each possible card value to a 1 in the first dictionary and a 2 in the second dictionary; this is to avoid possible divide by 0 errors, as we care about the ratio between the two dictionaries for a card's win frequency. In our `handle_round_over` function, we will take care of modifying these dictionaries using the information from each round played. We first note our bankroll change from the previous round in `my_delta`, as this will tell us who won the round (positive: us, negative: opponent); this logic will help us determine the winning pair of cards for our dictionary modification. We next want to make sure that we only consider cases in which there was a showdown between our cards and our opponents', as we need complete information to ensure that a pair of cards is actually better than another. The remaining code, available in the Lecture 1 Reference Bot code base, modifies the two dictionaries using the delta and showdowns-only condition; each time, we must update the appearance of all four cards by one, and the winning frequency of only the two winning cards by 1 as well.

Finally, we look to the `get_action` function, where we will actually implement our pokerbot's actions. For the Lecture 1 Reference Bot, the only logic we implement is raising (whenever allowed) by the minimum amount when both of our two cards have a win rate above $\frac{1}{2}$. We calculate the winrate of our cards by taking the ratio of their respective values in the two dictionaries we have been updating throughout the game. If we cannot raise, we simply check-call.

We run the engine using this bot against the random bot and analyze the results. This time, we win against the random bot by a much greater margin than before; this is an example of how even basic strategy can help significantly.[8]

[8]Note that this strategy is deterministic, which is actually undesirable—in the next class, we will cover why. If you are playing purely based on how good your hand is, your opponent will be able to tell and then dominate you.

2 Lecture 2: Poker Strategy

This lecture is taught by David Amirault.[1] All code from lecture can be found at the public Github.com repository mitpokerbots/reference-lecture-2. The slides from this lecture are available for download on Stellar.

At this lecture (and all others), we raffle off a pair of Beats Solo 3 Wireless Headphones. For this lecture, we also held a special raffle for an Xbox One S. Attend lectures in person if you want a chance to win them!

We'd also like to thank our eight Pokerbots 2020 sponsors for making Pokerbots possible. You can find information about our sponsors in the syllabus and on our website, and drop your resume at `pkr.bot/drop` to network with them. Our sponsors will also be able to see your progress on the scrimmage server, giving you a chance to stand out over the course of our competition.

2.1 Hand Types

A good way to understand hand types is by looking through examples. Slide 9 shows a board on the turn, with a 2c, Kd, Th, and 2s showing. Regarding hand notation, cards are described by value (2 through K) and suit (clubs, diamond, hearts, and spades). Note that it is impossible to have a flush on this board because regardless the unknown fifth board card, at most two cards on the board could share a suit, which allows for at most four cards sharing a suit if the two private cards also match (five cards sharing a suit are required for a flush). Also, a pair of 2s is showing on the board—a board that has a pair on it should be played differently, as the baseline is stronger (everyone has a pair or better). The hand Jc and Qc in this scenario is called a "drawing hand;" once the fifth and final board card is drawn, the board will either be really good or really bad for us. This provides us with some certainty on the river, which is very good to have in an uncertain game like poker. We could get a straight (with either a 9 or an A), so we have an "open ended straight draw" right now (there are two ways to get a straight with the river). If we only had one way to get a straight, it would be a "closed straight draw", or "gutshot draw." Since 9 and A give some certainty of winning, these are called "outs."

Now, lets compute hand strength. Hand strength is computed on a scale from 0 to 1, and it measures how many hands out there will beat our hand. The graph on slide 10 gives us our hand strength in this scenario as a histogram, taken over all the 46 possible rivers. The likelihood of us having a particular hand strength corresponds to the area of the histogram bucket. If we get a 9 or an A, then our hand strength will be about 1, which is why there is a bump on the graph around 1.0. Similarly, the two small bumps above 0.5 and 0.6 in the graph correspond to us drawing a J or a Q respectively to give us a two pair. Note that this graph is very bimodal—this supports our intuition that a drawing hand provides avenues to either a very strong or very weak hand. The bimodal distribution is something we like to see, as it will settle to a point mass that has clear hand value by the end of the round.

Now, looking at the hand strength graph, you might notice that we have a large probability of having a losing hand with strength $\sim$0.4—this corresponds to us not hitting anything that will improve our hand on the river.

Slide 11 has another example scenario, this time with pocket 3s instead of the J and Q. This hand already gives you a two pair, which is better than nothing, as the board already has a pair; however, this hand loses to a lot of possible hands, such as any higher pocket pair, a K, or a T, since this would make a higher pair with the board. This hand does beat nothing though, which is not too bad. Looking at slide 12, we see the hand strength graph for this pair—note that this graph looks very different than the one on slide 10. In Texas Hold'em, a player has roughly a 50% chance of getting a pair or better on a random draw, explaining why there's a bump on the distribution around 0.5. You might also notice that there is a bump around 1.0—this will happen if the last card is a 3, which would give us a "full house," one of the best possible hands on this board. A problem with this board though, is that even once all 5 cards are revealed on the board, we still don't know if we're winning or not—we don't know what the opponent has. This is different from the drawing hand, where we know exactly whether we'll likely win or not based on the fifth card. Therefore, even though the low pair has more mass to the right than the drawing pair, its uncertainty means that it'll make us lose money more often because of how our opponent can fold if they have a bad hand or keep playing if they have a good hand (we are "adversely selected" against). When you bet high with a low pair, there aren't many opportunities for improvement, and you have a good probability of losing—you

should be skeptical of playing longer when you have a hand with a distribution like that of the low pair.

Next, we'll talk about the "made hand" (slide 13). We'll talk a lot about hands that are good, because when you have a bad hand it's very easy to fold out. That's why the best pokerbots will be the ones that can differentiate good hands from better hands from the best hands, because they will know when to play and how. They're able to bet cleverly to extract the maximum value from their opponent. The made hand is also a two pair, and the K is called the "top pair." Even if the opponent has a T, we'll still win the two pair when it comes to hands, and we also have a good fifth card (the A), which is called the "kicker" or "tiebreaker" since it beats most cards.

Note that this year's variant makes it very difficult to identify when you have a low pair or high pair. The teams that will be able to determine so accurately will have a stark advantage in the early stages of this competition, due to the strategic differences appropriate for each of these two hands.

Notice that even though we still have a two pair, the made hand has a very different hand strength from the low pair (slide 14). No matter what shows up as the fifth card, there are a lot of hands that our hand will strictly dominate, or win, in any scenario. In a game like poker with a lot of uncertainty, this is exactly what you want—and so the made hand is considered a good hand that will make us a lot of money. The reason why there is some mass around 1.0 is because we could get a full house if another K shows up. Note that there are still some hands that could beat ours (pocket As, for example). If you run into a scenario like this you could lose a lot of money thinking you have a great hand but running into an even better hand. However, in the long run betting here is net positive gain.

The last hand we'll be talking about is "the nuts" (slide 15); when we talk about the nuts, we mean that there exists no better hand than ours on the current board. When you have the nuts, you want to extract as much money from your opponent as possible; you want to bet, raise, or call every possible turn. The distribution for the nuts (slide 16) is basically 1: it is a point mass. In this case, by the turn the nuts are 2h 2d. We are simplifying this calculation a bit, as there are exactly two hands that would beat this: the opponent has pocket K's and the fifth card is a K, or the opponent has pocket T's and the fifth card is a T. This is an extremely unlikely scenario, one where both players have a four-of-a-kind but the opponent has a better one. If this scenario occurs and strong hands go against each other, both players will be aggressively betting and the pot will be massive.

Even though there exists a scenario in which our four-of-a-kind loses, that does not change the fact that *on the current board*, this four-of-a-kind is the best possible hand. There is a small possibility that the fifth card will change the board to introduce a better hand, but there are still no two cards we would rather be holding right now than a pair of 2s to match the pair of 2s showing on the board.

Again, remember that you want your pokerbot to focus on the hands that are good, as you'd often be better bluffing with a drawing hand than bluffing with nothing, even if you're confident in your bluffing abilities.

2.1.1 Board Types

In Texas Hold'em, there are also types of *boards* you must consider. A simple example the one we've been using this whole time: a 2, K, T, 2 board. In the first scenario, you have pocket Aces (slide 18). This hand is much better than almost every hand, unless your opponent has a three-of-a-kind or better. You would feel great about your hand strength here. Instead, let's think about the scenario in slide 19. You'd almost certainly feel poorly about your hand strength here, even though you have the highest possible pair. The board has four clubs and several straight, flush, and straight flush possibilities. On a board like this, you can end up losing a lot of money to someone who gets an unlikely hand for an ordinary board, but a more likely hand in this "drawing board."

In Permutation Hold'em, board types still exist due to the fact that suits are not permuted—you can still analyze probabilities of flush draws, and as your bot determines the relative order of cards, straights will also become hidden draws. This should be an important consideration as you develop a starting algorithm.

2.2 Pot Odds

Pot odds are very related to the idea of maximizing expected value. We're going to begin with a claim, which is that for any state of the game, there exists some probability of winning. Even if our opponent

adopts a strategy that incorporates randomized behavior, this probability p still exists. Given that we have some probability of winning, we can calculate an expected value using the equation on slide 23. If we continue to play, the expected amount we'll win is $p \cdot \texttt{pot.grand\_total}$, and the expected amount we'll lose is $(1-p) \cdot \texttt{cost\_to\_continue}$. Note that these amounts aren't symmetric. That's because in poker, you should consider every cost a sunk cost; that is, never worry about money committed to the pot in the past, as it is already gone. The only cost you're considering is the further cost of continuing, which you're using as some stake to win all the money in the pot. When it comes to expected value, we don't want to make a negative expected value decision - in fact, we wish to maximize expected value. Note that if the expected value is 0, we're indifferent when it comes to folding or continuing.

The next step is to perform some algebra to separate out the probability of winning, and this gives us a cutoff for whether or not we stay in the game. We call the right hand side of the new inequality for p the "pot odds:"

$$\frac{\texttt{cost\_to\_continue}}{\texttt{pot.grand\_total} + \texttt{cost\_to\_continue}}.$$

If we know our opponent's strategy well enough to calculate p, we'll never have to worry as we can always calculate pot odds to make a positive expected value decision against every bet.

Now consider an example for calculating pot odds (slide 24). We mentioned this example briefly, but we did not consider pot odds. On average we'd expect our opponent to win this board, as they have a made hand and we're banking on a straight to win. Remembering our distribution, only ∼2 out of 13 cards will complete our straight, but our opponent already has a good hand. Suppose our opponent puts 10 more chips into the pot, and we now have to decide whether or not to continue. Let's calculate the pot odds using the previous formula (slide 26). If we think our probability of winning is greater than 0.1, we should continue, and otherwise we should fold. This explains why drawing hands are so much better than you might otherwise expect; estimating our probability of winning is easy, so it's easy to make good decisions with them. Our opponent gave us pot odds that look good enough for us to stay in with our drawing hand and make money on average, so they "underbet." This is highly undesirable in poker, because it will let your opponent stay in the game when they should've folded a long time ago.

Let's look at this from our opponent's perspective. If our opponent made a higher bet that caused us to fold, they would've won 80 chips 100% of the time. However, with us staying in the game, they will win 90 chips ∼85% of the time, for a win of ∼76.5 chips on average. They are worse off by underbetting!

This is a little unintuitive to people unfamiliar with poker; you might think that you should just bet proportionally to hand strength, but you should generally not underbet as this would give your opponent opportunities that they would've otherwise not had. In addition, it reveals too much about your hand. If your opponent had instead bet more confidently, you would've folded as your pot odds would've looked a lot worse. For example, if your opponent had gone from 40 to 80, your pot odds would be 0.25, which is not good enough to merit a call.

You may think that pot odds are irrelevant for Permutation Hold'em as we cannot tell when we have straights, but the concept still holds for flushes; using pot odds can be very useful for your bot architecture if you implement them successfully. In addition, as bots figure out the permutation based off the showdown results straights start to matter more.

There's also something called "reverse pot odds," which unsurprisingly, are pot odds for your opponent that give your opponent the opportunity to call. When we talk about reverse pot odds, it means that we're considering whether or not our opponent will stay in the game. The way that we can give our opponent opportunities is by "overbetting" relative to the size of our pot, which will give our opponent the possibility to exploit the pot odds and take our money. Overall, when considering pot odds you gain much more control over the size of the pot and maximize expected value.

Now, we'll consider an example bot that you may have seen on the scrimmage server: the "all-in bot." Our opponent goes all-in before the flop. Note that this is easy to beat, simply by check-folding until you have high pot odds (are dealt a high pair), crush them and win big. They'll collect the blinds on all cards we check-fold, but we can win big against them when we wait for great cards. Note that in Permutation Hold'em it can be difficult to tell when you have a high pair, but the saving feature of Permutation Hold'em for this case is that our opponent has the same information in terms of showdows for figuring out the permutation as we do. If they're making better decisions for going all-in, it simply means they're using the showdown information better than we are to figure out what the permutation in.

2.3 Ranges

When we're faced with a bet, everyone knows the pot odds. If we can estimate p better than our opponent, then on average we'll make money. Ranges are the types of hands that you play—when playing poker, you want to be restrictive about the hands that you play. Our opponents *range* is the distribution of hands we expect them to hold. We can estimate this during the course of the game—if our opponent hasn't folded late in the game and has bet a lot of money, we'll expect them to have a better distribution of cards. This means that ranges are key to calculating our probability of winning, which affects pot odds and our decisions. When it comes to poker, the best play style is typically "tight-aggressive," which means folding early and often with bad cards and betting aggressively with good cards. If your strategy is more tight-aggressive than your opponent's strategy, then you will often win more money on average, because you will get into high-stakes scenarios with better cards. For regular poker, the "tight-aggressive" strategy is folding ∼70% of hands preflop and betting frequently with the rest.

2.4 Variant Strategic Considerations

Note that permuting the card ranks doesn't change the relative strength of hand types. For example, a three of a kind still beats a pair, and flush draws remain consistent as suits are not permuted. However, in showdowns where the relative hand strength is the same (e.g. two pair vs two pair), the permutation must be taken into account. Though these are the times when results may deviate from normal hold'em, it's also where bots begin to piece together the permutation.

It's difficult to identify straights in Permutation Hold'em, and doing so will rely on your ability to identify the permutation over the course of the game. If straights are identified early on, they give a ton of information about the permutation order. There is a lot of low hanging fruit, however, as forming straights isn't even necessary to do well. In particular: flush draws, three of a kinds, and pairs are all consistent with Texas Hold'em.

2.5 Coding `reference-lecture-2` Bot

We'll build off our `reference-lecture-1` bot, where we had the dictionary of winning hands to calculate our ranges. We start by taking a look at the `get_action` function. The first thing we implement is a simple optimization: when all in, the engine will still ask for your action. An `if` statement can be added to check when this is the only possible action to allow your bot code to run faster.

In our `RaiseAction` logic, we have the dictionaries calculating the win rate of our hole cards. One of the key aspects of Permutation Hold'em is that cards of the same value agree: if you're holding a 2h and there's a 2d on the board, you will always have a pair; however, its value is likely not actually that of a pair of 2's. Note that last lecture all our logic was done independent of the community cards. Thus, today we implement code to see if any of our hole cards have the same value as one of the board cards and determine how many of each such match we have. We store this information in `agree_counts`.

Once we determine if our hole cards match with any of the board cards, we wish to implement the poker theory described earlier this lecture. For simplicity, we will raise whenever we have a two pair or better (e.g. three-of-a-kind, full house, etc.). Now, we must consider how to behave when we have a one pair, remembering that low pairs are much less valuable than high pairs. Though Permutation Hold'em means that it's hard to determine the actual rank of our pair, we can use our dictionary of winrates to approximate the relative strength of the cards. Using the winrate of our paired cards, we return a raise action with probability equal to the pair's observed winrate through random sampling. Concretely, this means that if K has a 20% winrate, the `if` statement to raise will be activated 20% of the time we have a K one pair. Similar logic can be implemented with the second hole card.

There's one more scenario we must consider: if we have two matching hole cards, or a "pocket pair." Here, we must determine if this is a high or low pocket pair. This can be done again by checking the winrate in our dictionary. Pocket pairs are strong hands because pairing with the board cards gives a "hidden" three-of-a-kind.

When returning a `RaiseAction`, we determine sizing using pot odds. In lecture, we discussed betting with respect to the size of the pot instead of your relative hand strength. In Texas Hold'em, pot sizing ranges from 0.5× to 1.0× of the pot. In our example, we'll just let the sizing be 0.75× pot. Though this

seems arbitrary, it turns out this value is actually pretty effective. Feel free to play around with sizing to see what works the best with your bot. Note that when betting, we need to add a few edge cases to ensure bet sizing is larger than the minimum raise amount and less than or equal to the size of your stack.

Comparing the results of this bot to `reference-lecture-1`, these implemented `if` statements on just pairs yielded a much larger chip lead over the opponent.

Finally, a reminder to always commit code back to git![6] There's nothing worse than ending a marathon coding session with a hard drive failure and losing all your progress.

3 Lecture 3: Inference

This lecture is taught by David Amirault.[1] All code from lecture can be found at the public Github.com repository mitpokerbots/inference-lecture-3. The slides from this lecture are available for download on Stellar.

At this lecture (and all others), we raffle off a pair of Beats Solo 3 Wireless Headphones. Attend lectures in person if you want a chance to win them!

We'd also like to thank our eight Pokerbots 2020 sponsors for making Pokerbots possible. You can find information about our sponsors in the syllabus and on our website, and drop your resume at `pkr.bot/drop` to network with them. Our sponsors will also be able to see your progress on the scrimmage server, giving you a chance to stand out over the course of our competition.

Finally, a reminder that Mini-Tournament 1 is tonight! Be sure to **submit your bot by 11:59pm EST on 01/10/2020** to receive credit for this course.[9] If you're pressed for time, feel free to incorporate some of the code we wrote in lectures 1 or 2, available at our public Github repository.

To start with a brief definition, inference is the theory of deducing properties of probability distributions.

3.1 Probability Distributions

The way we usually talk about probability distributions for math, and the way we write them down, is through a "probability function" which describes the probability of seeing an event. The simplest such one is an unbiased coin flip, which has 0.5 probability of being heads and 0.5 probability of being tails. In this case you can see that the probability of each side is equal, which is what we mean by "unbiased." Note that these probabilities sum up to 1; in fact, for any discrete probability distribution, the probabilities should sum up to 1, as there are a finite number of cases, of which one must occur.[10] By discrete, we mean that we can count the number of possible outcomes. You can also think about a slightly more complicated example, where instead of 2 possible outcomes we have 3: consider a biased rock–paper–scissors strategy as outlined on slide 8. This particular example is very similar to our previous example, but instead of 2 possible outcomes we have 3. Note the probabilities still sum to 1. By "biased," we mean that one case (rock) occurs with higher probability than the other two cases.

When we get into permutations, it sounds a lot scarier because there are so many more cases and it takes more work to write down what a permutation looks like than one of the previous example events. This is truly similar to the previous cases though. When we assign a probability to a permutation, it means the probability of seeing that permutation occur. As a toy example, we have some numbers on slide 9.[11]

We can represent a distribution over permutations in multiple different ways. On the previous slide, we simply listed a permutation and its respective probability; however, since there are 13! or $\sim$6 billion permutations, writing all these out would take way too much memory.[12] Another way to represent this distribution is through a specified probability mass function. An example is the geometric distribution, with the parameter $p = 0.25$.[13] If you are unfamiliar, this means the probability mass function is

$$P(x) = 0.25 \cdot 0.75^x.$$

If we had a nice function for our permutation distribution, we could simply calculate the probability of seeing a particular permutation and this would give us a leg up in the competition; however, we will likely need to use a random sampling process to generate this probability distribution function, as finding a closed form function may be challenging (or perhaps impossible).

If you want an example of a random sampling process, slide 11 presents some pseudocode to do so.[14] This pseudocode describes the process used to sample a permutation that your bot may see when playing a game. When you have a random sampling process to describe a permutation distribution, it makes it very easy to

[9] Bots must be submitted to our scrimmage server at pkr.bot/scrimmage.

[10] We call this the "normalization."

[11] Note that these numbers may not be accurate to the engine, but they are an approximation we can use to think about these permutations' frequencies.

[12] Also, clock speeds on computers are measured in Gigahertz. Simply iterating through this list would take roughly 10 seconds.

[13] The geometric distribution is discussed in more detail within our Stellar materials.

[14] This pseudocode is copied from our Stellar Engine How-to document.

do some things, yet very difficult to do others. For example, if my goal was to calculate the probability of drawing a specific permutation, it would be very difficult, as I would have to solve a math problem; on the other hand, if my goal was to generate a permutation, it would do so in a way very similar to the engine code. If something is hard to do with the probability distribution, it may not be worth doing and instead you should look to do what's easy with the given probability distribution.

We're going to imagine a slightly different version of poker than the variant we're using this year for Pokerbots, just to understand probability distributions. Suppose we choose a new permutation every round instead of every game. In this game, observing showdowns would not help you learn the permutation as it always changes for the next hand. In fact, in this example version of Permutation Hold'em, you cannot do better at approximating the permutation than sampling from the "prior distribution" and then using this permutation to simply play standard Texas Hold'em well. In this variant, a pokerbot that simply played regular Texas Hold'em best would win, which is why we do not change the permutation every round for regular Permutation Hold'em.

3.2 Bayes' Theorem

Imagine the setting where you're playing Permutation Hold'em, and you've just observed a showdown. We use S to represent the outcome of the first showdown we observe, and let M represent a candidate permutation. We're going to bypass thinking about where M came from for now, and just analyze probabilities assuming we're given M somehow. We're going to look at the conditional probability, denoted as $P(M|S)$, which is the probability of permutation M given that S occurred. This concept is visualized with a venn diagram in slide 15 of the lecture. This is the goal of Permutation Hold'em: using information from our showdowns. Thus, this is the natural way to explain the strategy mathematically using probabilities.

Bayes' Theorem gives us a way to calculate conditional probabilities based on the three terms in the right hand side of the below equation.[15]

$$P(M|S) = \frac{P(S|M) \cdot P(M)}{P(S)}$$

The first term is the inverted conditional probability, $P(S|M)$, or the probability of an observed showdown outcome given a permutation M. Note that this is very simple: either 0 or 1, depending on whether or not the permutation is consistent with the observed showdown outcome (i.e. whether or not the outcome would be the same if the permutation used in the showdown was M). We can simply apply the permutation to our hands, run a hand evaluator like eval7 to compute the new showdown, and see if the outcome agrees with the showdown outcome S. We let it be 1 if the outcomes agree and 0 otherwise. We know that we can calculate this value since it is deterministic. The second term is $P(M)$, our prior distribution which we described in pseudocode on an earlier slide. If we didn't know our prior distribution we couldn't calculate this term, but since we do we're going to leverage that to perform inference. The third term $P(S)$ is usually an intractable normalization term; however, we can calculate it using approximation methods of random sampling from $P(M)$ by sampling many permutations, counting the number of permutations consistent with S, then dividing by the total to approximate $P(S)$.

We call $P(M|S)$ the "posterior probability," as we are taking into account the prior given information to calculate our probability. This is standard terminology in probability theory and inference.

3.3 Two Fundamental Approaches to Inference

3.3.1 Non-Bayesian Approach

This sounds scary, but an easy example you've already seen is linear regression. How is this inference? Suppose the variables x and y are related by the following statistical model:

$$y = mx + b + \varepsilon,$$

where ε is a random noise term. Assuming our error term is normally-distributed and has zero-mean, linear regression is then solving the maximum likelihood estimate of m and b.

[15] Proof not provided, as that's for semester–length classes and Pokerbots is about application-based learning. Google "Bayes' Theorem proof" if you're interested in related resources.

Maximum likelihood estimation means given observations y and a statistical model with some sort of parameters θ and a likelihood function $P(y|\theta)$ given our observations, the maximum likelihood estimate of the parameters θ is simply θ^*, the value of θ which maximizes the likelihood function $P(y|\theta)$. Maximum likelihood estimation is one of the cornerstones of inference, but it doesn't really make sense to be used when we have prior beliefs about θ being drawn from a probability distribution, which we do in Permutation Hold'em. The equation on the board doesn't let us incorporate any such prior beliefs about θ, instead focusing on discrete values. Thus, we go back to Bayes' Theorem and Bayesian Inference.

3.3.2 Bayesian Inference

Recalling Bayes' Theorem, the bayesian equivalent of maximum likelihood estimation is "maximum a posteriori estimation".[16]

At this point, we can explain what we really mean by Bayesian inference. Bayesian inference is inference with prior beliefs on distribution parameters. By beliefs, we mean that we have a representation of a prior distribution that describes some parameter of our model, but not discrete values. In our case, we have a prior distribution (sampling process) for the permutations used in our game, aka generating the permutation or card values.

3.4 Model selection with observations

Our goal here is to apply Bayesian inference to estimate a posterior distribution; that is using our information from showdowns to get a posterior definition, and even better get a sense of what the permutation really is.

Now, we're going to apply everything we've talked about Bayesian Inference through a technique called the "particle filter". This may sound scary, but we're just leveraging two good ideas about resources we have available to us. Our first good idea for the particle filter is that it is easy to generate sample permutations, by simply copying the permutation generation code from the engine. The next idea is to quickly eliminate permutations which contradict our given showdowns. We apply each permutation, run a hand evaluator, and check if the outcomes agree. You can think of this as us emulating our own engine, by generating permutations, sampling them and running the hand evaluator.

Putting this into practice with permutation poker, we first generate k candidate permutations of the start of the game. Next, whenever we observe a showdown, we remove permutations that are inconsistent with the outcome we saw in-game. The leftover permutations are representative of the posterior distribution!

What a particle filter might actually look like is we have a long list of permutations (I've listed out the first five in the slides), and a showdown. After a showdown, we learn two of those permutations don't work and we eliminate them. Another showdown happens and another permutation is eliminated, and so on. Eventually we only have a few permutations left, so now we have a good idea of what the permutation must actually look like. So why does this work? To see how, we go back to Bayes' Theorem, and look at our prior distribution. Note that our k particles come from the start of the game.

For the first term $P(M)$, we know that we are sampling our candidate permutations from the prior distribution, meaning that we incorporate this term. For the next term in Bayes' theorem, $P(S|M)$, this is the 0 or 1 binary term we determine by checking consistency with showdowns. Therefore, eliminating inconsistent permutations incorporates this $P(S|M)$ term. Finally, if we select a random candidate permutation, then the normalization term is accounted for, just as we hoped. Thus the end result of the particle filter is that we are sampling from our posterior distribution, just as we hoped.

One problem remains: what if we run out of candidate distributions? This is a big issue for the particle filter, as if we run out we cannot sample new candidate distributions and check them against every showdown reasonably. It's unreasonable for us to generate more and compare them against all of the showdowns we've already seen. Instead, we could stop eliminating when we reach a certain point in the filter and randomly sample from the remaining permutations to find a representative one. Therefore a big part of the candidate filter's success is the ability to randomly sample enough candidate distributions quickly.

Putting it all together, given a particle filter we can sample a random candidate permutation whenever we need an action. Then we can play ordinary Texas Hold'em assuming the candidate permutation. This is a huge advantage because it essentially reduces Permutation Hold'em to normal Texas Hold'em. However,

[16] "Maximum A Posteriori Estimation" is often abbreviated to "MAP."

this can break down since we can't sample all the candidate distributions. Eventually, we need to stop eliminating candidate permutations when we are down to the last few to avoid running out of them, leading to our inference stagnating over time, and as a result depends heavily on how many candidate permutations we can sample at the beginning, again tying into the efficiency of our code. A tip to speed up this process is by pre-computing permutation dictionaries at the start of the game. The reason we say the start is because the engine allows you 10 seconds to connect to the engine, and code located in `__init__` methods run during this first 10 seconds (not eating up your 30 seconds). Finally, using advanced libraries (like eval7) for consistency checking will make your code **blazing fast**.

3.5 Permutation Hold'em Code

We will be coding up a Python particle filter for a python skeleton bot in this section. We create a new repository with just the python skeleton, engine, and config files to benchmark how fast the particle filter is by running the python skeleton against itself. Change your bot paths in `config.py` to make the python skeleton bot play itself.

To do this section, we'll need to look at the code both in `engine.py` and `player.py`. Start by viewing the code in the `engine.py` file, which seems a bit scary but we know what we're looking for and can break it down. If we remember from lecture, one of the first pieces of code we'll need from the engine is to know exactly how the engine samples from the probability distribution: specifically, the prior code. We know the function form of the prior distribution uses a geometric distribution. Searching for the import geometric, this takes us to a function in the engine called `permute_values`, which looks slightly different from the pseudocode due to some performance engineering optimizations. However, this is good for us as it means it's optimizing our code as well!

We copy this and import the geometric distribution to our `player.py` file in order to be able to run this function. Thus write `from numpy.random import geometric` at the top of our player to get access to this distribution. We paste the prior distribution code into our `player.py` file as the first function, and take a quick look at it. It looks like the prior starts with an original permutation which is a list whose entries are integers from 0 to 12—this is one way to represent the permutation, and we must take note of how it is done so that we can use this function later on.[17]

We next consider what we need to do with our permutations for our particle filter. We need to quickly add permutations at the beginning and delete them as they contradict showdowns. A great datatype for doing this is the built-in Python dictionary datatype, as the runtime is linear. We also need a quick way to compare handtypes. We'll do this with the eval7 library, the library the engine is using to evaluate strength of poker hands and specifically calculate what the showdown result should be. We do this by adding the line `import eval7` to the top of our code.

Scrolling down to the `__init__` function, we create lists representing the ranks of poker cards in a list called `values` and similarly initialize suits in another list called `suits`. We construct another list called `self.proposal_perms`, which contains all the permutations which still satisfy the showdowns. `self.proposal_perms` should be initialized using a for loop which repeats once for each particle we wish to sample.

For each iteration of the loop, we need to sample a permutation by calling the function we just copy–pasted from the engine's code. Next, we loop through our `values` and `suits` lists to form all 52 standard cards (in the eval7.Card datatype) and use a dictionary to map the standard cards to their permuted counterparts according to the permutation we sampled. Once finished, this dictionary will represent a particular permutation from the prior, and we add it to our `self.proposal_perms` list.

We now go to the `handle_round_over` function, where we'll implement our particle filter logic. We start by writing a variable to account for the board cards, and then creating an if statement to test if we were in a showdown that round (opponent's cards not empty). If we are in a showdown, we want to loop through our proposal permutations in `self.proposal_perms`, and we wish to check if each permutation is still valid.

To check if it's valid, we need to get the permuted card values of our cards, the board cards, and the opponent's cards, to evaluate the ending showdown. We're going to do so using list comprehensions, which simplifies our code and makes it easy to permute the three lists of card values. Now we have the set of permuted cards for each of the three types of cards described above. We can now go ahead and run eval7 on our hand and our opponent's hand and record the given strengths.

[17] A 0 corresponds to a 2, which is the lowest rank, and a 12 to an Ace, the highest rank.

We now check the showdown result which actually happened, since we wish to remove contradictory permutation candidates from the list of proposed permutations. Something you may have learned is that it is very bad practice to modify a list you are iterating through, so we instead create a new list called `new_perms` which contains permutations which are consistent with the result of our showdown.

If the strengths we calculated are consistent with the result of our showdown, we add that permutation to `new_perms`. Otherwise, we discard the permutation. We can check the result of a showdown by looking at the `my_delta` variable. If this variable is positive, it means that we won money from that round and thus must have won the hand. If it's negative, then we must have lost the round. In the unlikely event that the variable is zero, it means that the round was a tie.

Now, we add a condition limiting minimum size of our `self.proposal_perms` list, since we do not wish to enter a situation in which we have eliminated all proposed permutations. We choose to limit the minimum size to 5 by not updating the attribute when the new list of permutations we have constructed is very small.

Thus if `new_perms` has at least 5 elements in it, we set `self.proposal_perms = new_perms`. Otherwise we keep our old permutation list.

We run the particle filter code, and see that it runs with 29.7 seconds left; it didn't eat up our time at all.

We can now develop new strategies to incorporate these candidate permutations into our pokerbot!

4 Lecture 4: Game Theory

This lecture is taught by David Amirault.[1] The slides from this lecture are available for download on Stellar.

At this lecture (and all others), we raffle off a pair of Beats Solo 3 Wireless Headphones. Attend lectures in person if you want a chance to win them!

We'd also like to thank our eight Pokerbots 2020 sponsors for making Pokerbots possible. You can find information about our sponsors in the syllabus and on our website, and drop your resume at `pkr.bot/drop` to network with them. Our sponsors will also be able to see your progress on the scrimmage server, giving you a chance to stand out over the course of our competition.

Announcement: The Final Tournament logistics have been finalized: the event will take place on Friday, January 31, 2020, from 4:30–7:00pm in Building E51 (Tang Center). Be sure to save the date in your calendars! The Final Tournament event is where we'll announce the winners of this year's large prize categories (including the Pokerbots 2020 Champion), and will feature more raffles, an opportunity to network with our sponsors, food, and the chance to play against the bots you've built this month! More details will be sent out over Piazza during weeks three and four.

4.1 Pre–Lecture Game Rules

Before the lecture formally begins, we have two games for you to play. The first, which will be today's giveaway, is everyone submits a number between 1 and 1000 inclusive, and the person who guesses closest to $\frac{2}{3}$ of the average wins.[18] The second is opt-in or opt-out; you'll wager 20 ELO points, and the winner is the person with the highest last 4 digits of their MIT ID—they'll be receiving everyone else's ELO points.

4.2 What is a game?

There are many kinds of games, but we will only consider two–player zero–sum games. Two–player, very naturally, means there are only 2 players. The technical definition of a zero–sum game is one where any value won by you is lost by your opponent, and vice versa; there is no value created by winning the game. This is very naturally related to the concept of chips in poker.

A *game* between players 1 and 2 consists of a pair of strategy sets S_1 and S_2, and a utility function u. Examples of possible strategy sets are your bot going all in, or only check–calling. The utility function u outputs R, the utility for player 1, so the negative of this is the utility for player 2. Note that strategies are submitted simultaneously. Naturally, player 1 wishes to maximize u, and player 2 seeks to minimize u. This notation works very well for two–player zero–sum games, as each player is simply trying to maximize their own utility function. Examples of these games are: rock, paper, scissors (referred to "RPS" going forward); "my number is bigger than yours" (a game where two players say a positive integer and the one with the largest integer wins, referred to as "MNIBTY" going forward), and even chess or heads–up poker.

4.3 Pure and mixed strategies

RPS has 3 pure strategies: always rock, always scissors, or always paper. A mixed strategy is any that uses randomness to interpolate between pure strategies (e.g. flip a coin, and heads implies scissors and tails implies rock). If you're using a mixed strategy, however, you cannot play a $\frac{1}{2}$ rock and $\frac{1}{2}$ scissors hand at the final showdown; you must choose one of the three pure strategies for a specific round. The reason we analyze mixed strategies is that a lot of fundamental game theory is only truly applicable when we let people use mixed strategies, choosing between various pure strategies. A pure strategy in RPS is not good, as it is easily exploitable if your opponent knows the strategy (e.g. always rock is beaten by always paper).

One of the many ways to represent games is through a matrix. The rows correspond to the pure strategies player 1 is allowed to take, and the columns to the pure strategies player 2 is allowed to take. The payoffs we usually write represent the payoffs to player 1. RPS is also easy to create a matrix out of, because each player has a finite number of possible strategies at every stage of the game—thus, it is a matrix form game. Doing this allows us to compute the utility of a strategy S_1, as we can average payoffs of the pure strategies

[18] This is called the "Beauty Contest" because there was initially a newspaper advertisement where you had to rank people's attractiveness, and the winner was the person with $\frac{2}{3}$ of the average rank.

using the probability of picking such a pure strategy. We use +1 to represent the payoff of winning rock paper scissors, and −1 to represent losing; note that the magnitude does not truly matter as long as we are consistent.

	Rock	Paper	Scissors
Rock	0	−1	1
Paper	+1	0	−1
Scissors	−1	1	0

Another thing about RPS that makes it special is that it is a symmetric game, as switching the players or payouts doesn't affect the strategies.[19] This can be seen by the matrix as its transpose (switching the players) equates its negative (switching the payouts).[20]

4.3.1 The Keynesian Beauty Contest

Now, we're going to talk about the first game that we played. This is also a matrix form game, as each player has 1001 possible strategies; however, it has a multidimensional matrix, as there are multiple players each with their own strategies.

We'll analyze "rational" play for this game. The first thing that you notice about the game is we're trying to guess $\frac{2}{3}$ of the average, and the maximum possible number is 1000; therefore, whatever we play, we shouldn't pick above $\frac{2}{3}$ of the max, or 667. So, if no one is going to play over 667, the average won't be higher than $\frac{2}{3}$ of this, or 445. By the same logic, we shouldn't guess more than $\frac{2}{3}$ of this number. Continuing in this fashion, no rational player should play anything other than 0 or 1, depending on how you round. Notice how "rational" here is in quotes—this is because you're trying to win the Beats headphones, and not necessarily play rationally, as this does not always translate well to practice. This demonstrates why there's a distinction between playing game–theoretic optimally, and playing to win Pokerbots—something interesting to keep in mind while developing your bot's strategy.

4.3.2 Dominance

Next we'll talk about dominance: we say that strategy A "dominates" strategy B if playing A is always a better idea. In the Beauty Contest, submitting 667 dominates the strategy of submitting 1000, as we multiply the average by $\frac{2}{3}$. We can take this idea further by talking about "second–order dominance;" once we've crossed out the first dominant strategy, we can do the same consideration again. This is where we multiplied 667 by $\frac{2}{3}$ again to get 445. You can recursively apply this definition to even get to "infinite-order" dominance, which is when you only play strategies that are not dominated to any order.

This takes us to a point where no strategy can dominate another any more. It's helpful to define the point of an "equilibrium:" an *equilibrium* is a set of strategies, one for each player, such that nobody has an incentive to switch.[21]

4.4 Nash equilibria

You've probably heard the words "Nash Equilibrium" thrown around before. It may be very easy to get confused and think that a Nash Equilibrium refers to a perfect strategy; this is incorrect. A Nash equilibrium simply means a set of strategies such that no player has a point in switching, but this does not always make it the best strategy. You will see an example of this when we go over the beauty contest statistics.

A famous quote, that many of you may have seen in *A Beautiful Mind*, is the guaranteed existence of a Nash equilibrium in every finite game using mixed strategies. This was proven by Nash himself, in 1951, using topology.[22] Note that this theorem only applies to finite games, which MNIBTY is not—for every strategy you can shift yours over to beat your opponent. We are also only refering to games as we defined

[19] Note that being a symmetric game does not imply that its matrix is symmetric.

[20] This is referred to as the matrix being "skew–symmetric."

[21] Note that by "strategies," here, we are talking about mixed strategies. The "incentive to switch" means they cannot increase their expected utility by modifying their mixed strategy in any way.

[22] The specific theorem is called the fixed point theorem, which some of you mathematicians in the room may have heard of.

them, where they are matrix form, you have a set of strategies for each player, and a deterministic utility function which given these strategies assigns a utility for each player.

A Nash equilibrium happens when everyone plays 0 in the beauty contest game (definition on slide 13). Even if you know that everyone else is playing 0, 0 is still $\frac{2}{3}$ the average of 0 so you have no incentive to switch, nor does anyone else. In the game RPS, there's no pure strategy equilibrium, as for every pair of player strategies players will be incentivized to switch to the pure strategy that dominates their opponent.

There is an "equilibrium" for RPS, which is playing each pure strategy $\frac{1}{3}$ of the time (slide 17). Let r, p, s be our probabilities of playing rock, paper, and scissors, respectively. Slide 18 shows the calculation for why both us and our opponent playing this same strategy is an equilibrium, as neither of us has an incentive to switch. Our opponent wants to pick a strategy r, p, s which minimizes our utility, and we want to maximize our utility given that our opponent is attempting to do this. The solution to our min–max relation guarantees us at least 0 value, which you may think isn't very good, but this means that on average we are never losing. If a pokerbot was able to get this sort of performance in a tournament, it would almost guarantee them winning as they would never be losing money on average.

Now we'll analyze a more complicated game, themed around military battle plans (slide 20).[23] I can choose to either charge or sneak attack against my opponent, and they can respond with either a full defense or defend in shifts. The matrix below displays my utility for each case:

	Full Defense	Defend in shifts
Charge	0	+3
Sneak attack	+1	−1

Looking at this matrix, there are mostly positive numbers; thus, before mathematically analyzing, we'll intuitively expect player 1 to have positive utility on average. Also, this matrix is not skew–symmetric, as switching the players and strategies changes the payoffs more than just multiplying by negative 1 (as one can easily tell by the +3 payoff).

Let c, s be the probabilities of me launching a charge and sneak attack, respectively. My expected values for utility as a function of my opponent's strategy then becomes:

$$\mathbb{E}[\text{opp full}] = c \cdot 0 + s \cdot 1 = s,$$

$$\mathbb{E}[\text{opp shifts}] = c \cdot 3 + s \cdot (-1) = 3c - s.$$

Let's suppose we want to find the values of c and s so that we are always guaranteed a certain expected value in this game no matter what our opponent picks—that is, we want to find c, s such that $\mathbb{E}[\text{opp full}] = \mathbb{E}[\text{opp shifts}]$. Calculating, we find that $c = 0.4, s = 0.6$ satisfies this constraint; substituting into either expected value equation tells us that no matter what our opponent picks, we have an expected utility of 0.6. Hence, the "value" of this game is 0.6 for us, or we are claiming value by playing this game.[24,25]

4.5 Applications to poker

I've shown you that Nash equilibriums exist, which may sound very weak for practical applications, but this information is very useful for finding it in different games; now let's look at it for poker. First, we have to prove that poker is a finite game, but then lets talk more about what equilibrium actually means in the context of poker. RPS is very easy to represent as a matrix, but other times its more natural to represent games as a tree. This representation may be more natural to you if you've taken computer science classes using graph theory.

We'll look at the concept of "extensive form games;" we can represent a game that's played sequentially, such as poker, by game trees (slide 23). Consider the game of tic-tac-toe, which has its tree displayed on slide 23. You'll notice that tic-tac-toe is very symmetric, so we've made the tree smaller already—however,

[23] This game is themed around military battle plans, as game theory was actually developed significantly by military strategists at the RAND Corporation, a large think tank, during the 1950s!

[24] Note that if you use this value to guarantee a minimum expected value of utility in other games, you can get nonsensical numbers if dominated strategies are present.

[25] You can always use this computational method, even when more strategies are available, using linear programming for those of you who are familiar with it.

its tree is still very large. An extensive form game can be represented by a finite (possibly large) game tree where the nodes represent game states, levels of nodes are alternating players, and eventually you reach a leaf which is a number representing player 1's utility, rather than another game stage. Poker is an extensive form game, where you're using the information your pokerbot gets from the `GameState` to build your game tree. It's clear that chess is another extensive form game, but with chess the game tree is incomprehensibly large compared to tic-tac-toe's. These game trees can get really big because they grow exponentially, especially permutation poker with its combinatorial explosion; however, they can be dealt with through a technique called "backwards induction."

Looking at tic-tac-toe's game tree, we can start at some point late in the game and consider a specific section of the tree (slide 24). Note that the leaf nodes have their player 1 payoffs marked in blue. Next, note the black number next to certain nodes—this represents from that game stage onward, only the black number is possible as a payoff. We can do this by going upwards from leaf nodes, calculating what a player playing optimally would rationally choose between the game stages available to them at the level below that state. This upwards traversal means that eventually we'll reach the start node of the game, giving the game itself a payoff for each player. This approach has been applied to games as big as Connect Four, which is strongly solved, but not much bigger than that.

Every extensive form game—such as tic-tac-toe—is also a "normal form game," meaning that it can be expressed in matrix form. The problem with writing this matrix, however, is that it grows exponentially in nature, because the number of strategies for each player is equivalent to every possible non-end-stage of the game. This matrix in fact is doubly exponential, as the number of both row and column strategies is exponential. This existence, however, means that tic-tac-toe has a Nash equilibrium in existence.

However, there's a problem–we're still not working with poker! Poker has an unimaginably large doubly exponential matrix, but since all bets are in integral amounts of chips, it has a finite matrix. Therefore we know the existence of a Nash equilibrium for poker. There is one additional caveat for poker, however: the reason we can't explain poker with this type of game tree is because poker has incomplete information. With poker, you can have a situation like the one displayed on slide 27. Nodes in the poker game tree can be reached not only by player actions, but also by "nature:" elements of randomness. When the engine tells you there are three new cards and that's what the flop is, this is randomness, as well as when your opponent flips their cards over. Randomness is the following tree representation of our poker game is represented by player R. The dotted lines on the game tree explain that players don't know which node they're situated at because of what hand their opponent has, which is why the game tree becomes imperfect. R always leads to a node labeled 1, which represents player 1's turn. They then have two options (to fold or to bet), which leads to a player 2 scenario. One of the key aspects of poker is that our opponent does not know our hand—that is our secret knowledge—so this is represented in the game tree by our player 2 nodes being linked by a dashed line. Our opponent cannot tell whether we were dealt the AA or the AK, so the game states are effectively equivalent to our opponent. They can see our game tree movements, but not our particular game state. This is why poker is such a beautiful game to play. The question then becomes whether we can still apply backwards induction to these types of game trees; the answer is we cannot, due to these dashed lined which we refer to as "information sets." These dashed lines throw off the backwards induction process.

Poker is also a matrix game, as for every type of strategy that our opponent takes—betting, calling, or folding, for example—it's possible for us to create a responsive strategy based on the information that we have. This shows that Poker does have a Nash equilibrium after all, since it can be expressed in matrix form! This equilibrium strategy will have average payout 0, which will guarantee that we do not lose on average—that is, we cannot lose; however, this matrix is double exponential in size. Thus, computing the Nash equilibrium is near–infeasible due to computational complexity. Methods do exist to do this using linear programming, however, but even then it's still very difficult due to complexity reasons.[26] The answer is also no for a different reason. If some of you were there for our pre–registration game Blotto, you may realize that game also has a Nash equilibrium; however, in Blotto, the optimal strategy is not to solve for the Nash equilibrium but rather to play against the other players. Similarly, in poker, your goal is to beat the other players, not solve for the Nash equilibrium. If you cannot solve for the Nash equilibrium, that is totally okay; your goal is to take your opponents chips. If you can play strategies that beat multiple opponents and take their chips, then you have a great shot at winning this tournament. If you target specific opponents,

[26]The MIT course 6.853, "Topics in Algorithmic Game Theory," covers many ways to find and approximate Nash equilibria; we highly recommend it.

we call this an "exploitation strategy;" sometimes, these can do better than approximating or solving for the Nash equilibrium.

In poker, one strategy you can attempt is to approximate the Nash; this strategy will do pretty well on average. In practice, playing the Nash equilibrium in RPS is not always the best idea. For example, if you play an opponent who uses rock with probability 0.6, you want to play paper more than the Nash equilibrium suggests you do. The Nash will guarantee that on average you play the value of the game, and in each of the subgames in the game tree it will also have you play optimally in that situation. What this means is that you could definitely come up with a strategy that is not the Nash equilibrium that plays equally against the Nash, but finding it is computationally infeasible. The Nash equilibrium in poker, on the other hand, will always do pretty well, but might not be the best strategy against certain strategies which are really suboptimal that you could crush with a different strategy—it's up to you to determine when to approximate the Nash and when to do something different that makes your bot great rather than good.

Example: Applied Game Theory

YouTube Link: `https://www.youtube.com/watch?v=hRIXXCe0Hi0`

4.6 MIT ID Game Discussion

Just like the beauty contest game, we're going to analyze rational play for the MIT ID game. There's again a cutoff—this time, people would rationally only play if their last four digits are in the 90,00+ range. Again, however, people play to win, not necessarily rationally. People will always behave in ways that are hard to model mathematically, but it will be advantageous for you if you can figure out how to exploit this behavior in some way. This game is certainly one where it makes sense to deviate from the Nash equilibrium.

This game is a great example of "adverse selection," which happens anytime a "buyer" and "seller" have asymmetric information. You likely don't know anyone else's MIT ID, so you're only competing with people when they *want* to compete against you—selecting adversely to your chances of winning the game. There's a game called the "market for lemons"[27]: suppose everyone is trying to sell a car in a lot, where the cars vary wildly in quality. The prices range from \$1000 to \$10,000, and you're trying to determine the fair price for a car on this market. If the fair price is \$5,500, all cars worth above this value will leave the market as their sellers don't want to lose so much money. Now, the average value of cars on this market will drop again, and as a result the fair price will drop as well. This process continues until all cars left on the market are the worst of them all (the "lemons"). This happens because there's so much adverse selection—every seller has private information about the true worth of their car.[28]

There are many sources of adverse selection in the game of poker: e.g. choosing to bet on the first action, since your opponent is much more likely to call when they have good cards, etc.

We said in the last two lectures that we'd make an argument for why deterministic strategies are bad; we'll get to do that now. Suppose we always call when we have a top $X\%$ hand; how can our opponent exploit this against us? Answer: our opponent can adversely select against us, as they'll choose to only bet when they have a top $\frac{X}{2}\%$ hand. This means that on average, they'll blow us out.

[27] The word "lemon" is slang for a car that is discovered to have problems only after it's bought. This refers to the fact that when you drive a new car off the lot it loses 20% of its value instantly. If you try to sell your new car the day after you purchased it, people will wonder why you're selling it since they have less information about it than you.

[28] A common economics joke is that if you find a \$20 bill on the street, you should pick it up; if you find a \$20 bill in the middle of Grand Central, you should leave it be since if all the rational people around it are walking past, there must be some adverse selection to picking it up.

5 Lecture 5: Advanced Topics I

This lecture is taught by David Amirault.[1] The slides from this lecture are available for download on Stellar.

At this lecture (and all others), we raffle off a pair of Beats Solo 3 Wireless Headphones. Attend lectures in person if you want a chance to win them!

We'd also like to thank our eight Pokerbots 2020 sponsors for making Pokerbots possible. You can find information about our sponsors in the syllabus and on our website, and drop your resume at `pkr.bot/drop` to network with them. Our sponsors will also be able to see your progress on the scrimmage server, giving you a chance to stand out over the course of our competition.

Announcement: The Final Tournament logistics have been finalized: the event will take place on Friday, January 31, 2020, from 4:30–7:00pm in Building E51 (Tang Center). Be sure to save the date in your calendars! The Final Tournament event is where we'll announce the winners of this year's large prize categories (including the Pokerbots 2020 Champion), and will feature more raffles, an opportunity to network with our sponsors, food, and the chance to play against the bots you've built this month! More details will be sent out over Piazza during weeks three and four.

5.1 Reinforcement learning

5.1.1 Goals and fundamental challenges

As an overview, reinforcement learning typically frames the problem as talking about "agents" and "actions." The agent is the person who plays the game, or your pokerbot, and the actions are the moves your pokerbot takes. Our agent takes actions to move between states; in this case characterized by the information the engine gives your pokerbot about the game's condition. Your goal is to maximize some sort of reward: in pokerbots, this is maximizing your number of chips. We are typically using the extended game tree (or extensive form) of a game, rather than the game matrix. The agent has multiple attempts to improve, which could be the 1000 hands of your pokerbot's game, or more often will be offline attempts gained from using your pokerbot on the scrimmage server. Reinforcement learning is typically not used for image or speech learning, as these usually require some sort of human–labeled input to tell what the correct answer is (commonly classified as "supervised learning"). What you'll notice is reinforcement learning never even mentions datasets, which makes it different from other types of machine learning. Recommender systems, like when Netflix tells you what to watch next, are also not reinforcement learning, as these systems do not try to maximize some reward.

Reinforcement learning has a lot of success when it comes to two–player games, which makes it natural for us to think about in Pokerbots. If you've never heard of AlphaZero, it's very relevant to Pokerbots. It was a reinforcement learning agent trained from to play a variety of games (chess, go, shogi), and did so with tremendous success. Even more amazingly, it started by learning only from the game tree—there was no human input at all. It could play games even better than its designers could, which would be very helpful in pokerbots. If your bot can play poker even better than you, it would transcend the limits of your if-statement logic bots. Another reinforcement learning agent published by the DeepMind team learned by thinking of something walking along a map of the game tree: you have some sort of cheetah, or simplified human, walking along the game tree with inputs such as the terrain map, angles and angular velocities. What's amazing is with no input, DeepMind learned to move along this terrain like a human would.

Slide 8 shows animated charts of the various games AlphaZero learned to play, showing it move from playing completely randomly to beating the state of the art expert bots in the three respective games it played.[29]

Now we're going to take a look the reinforcement learning agent DeepMind walking on a terrain. Its only inputs are the terrain map and how its legs are currently moving–it has no idea how to walk and has to learn this on its own. It also has no idea what to do with its arms, as that truly does not matter, explaining why it flails around as it moves in the video. Perhaps the most interesting part is that these trained agents can adapt to new environments, like obstacles they've never seen before.

[29] The "four hours" is a statistic the research team liked to publish as an attention–grabber, but it turns out they were using four hours on *all of Google's computing resources*. We will talk more about the computational requirements of reinforcement learning later in this lecture.

As an algorithm designer, there are multiple appealing properties of reinforcement learning. You can train an algorithm to do something you can't even do yourself! There are caveats, however. Reinforcement learning is incredibly hard to train, and it's very easy for these algorithms to fail to escape a local optima. This happens when the algorithm has found something that kind of works, and doesn't have motivation to do better. Researchers have to tweak their parameters very heavily to get good results. The other issue is it's incredibly sample inefficient; it takes tons of samples to get any sort of results, sometimes requiring hours of compute time.

For example, the AlphaZero agent was trained in only four hours, but using 5000 machine cores to perform the necessary computations. DeepMind required 6400 hours of computation time to train its humanoid agents.

On top of the previous issues, reinforcement learning also struggles with multi–agent scenarios. It's very easy for reinforcement learning algorithms, even when you're playing two–player games, to run into a scenario where you continue to make improvements, but aren't really moving forward in a general sense. The reason AlphaZero had great success in chess is because if a bot1 loses to bot2 and bot2 loses to bot3, there is a good chance that bot3 is the best of the bunch. This is because chess and other games AlphaZero trained on are games of complete information, as nothing is hidden from the agent. This gives a clear route to *iterative improvement*; if a bot is beating others it is probably getting better. However, this is far from guaranteed in poker. If bots beat each other, it can simply mean you are updating your parameters in a cyclic manner.

This can happen in even the simplest probabilistic game, rock–paper–scissors. if you train two reinforcement learning agents against each other on RPS, they'll be very good at beating each other, but there is no guarantee they would be good on the equivalent of our Pokerbots scrimmage server. There is a high risk of getting caught in a cycle of deterministic parameter changes, called a "policy cycle," which doesn't make any meaningful improvements to the bot. The issue with poker is you could have this happen to your pokerbot as you train it without even noticing the cycle, then lose on the scrimmage server when you upload your fundamentally bad bot.

What makes reinforcement learning very hard sometimes is designing your reward function. In poker this is easy, as you want to make more chips, so chips should be your reward function. For Tetris, however, when researchers made the mistake of making the reward function staying alive as long as possible in the game, it led to their agent simply pausing the game and staying on that screen for as long as possible when it was about to lose. In Mario Cart there have been instances of improper reward functions leading to agents sitting on the locations of coins and item boxes rather than actually winning the race. If you haven't thought about your reward function, your agent will simply poke chinks in your intentions.

If you've thought about all of these various complications, meaning you have a well–crafted reward function, sufficient computation resources, and multi–agent scenarios that avoid cycles, you're ready to proceed with using reinforcement learning.

5.1.2 Approaches: Q–learning

Let's return to thinking of poker as a multi–step process, or as a game tree, rather than the matrix form we used in the game theory lecture. Let's say we have a big table of game states, where we list the quality of every action we're allowed to take.[30] You can imagine a lot of humans play like this: you're put in a game state with a bunch of actions you're considering, you evaluate the quality of each of these potential actions, and simply take the action with the highest quality. The way you determine quality is probably based on your past experiences, and this is exactly what Q–learning does: learn from past experiences.

We initialize a Q–table of all these possible actions with all 0s, since we have no past experience with any potential action. The specific example on slide 19 is for a transportation problem, where you have a robot trying to move along a map. We start by initializing the table to 0, since we have no idea of the quality for any move. We then update the Q–table after every action, since we now have some experience. Eventually when you're done training, you have an idea of the quality of each possible action, which hopefully works well based on your experiences training.

[30]The name for Q–learning comes from "quality," as it's basically using a table of actions and how high–quality we think each action is.

$$Q^{\text{new}}(s_t, a_t) \longleftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Above is the update rule for Q–learning. Q^{new} is the Q value you're going to assign an action at a particular game state after taking it. α is our learning rate, usually chosen to be very small. Another aspect of this α is it allows you to not rely on previous mistakes, as it decays the old value, and allows you to play dynamic games (where the rules are changing over time) so you can adapt to new scenarios. On the second half of the right hand side, we multiply the learning rate by the learned value. The learned value takes into account both the the *immediate value* you gain from taking an action along with the *future value* you gain from being placed in a resulting game state. In poker, the immediate value of a move is 0, as there is no immediate value until you get your chips at the end of the round. You also incorporate the value of future rewards, which you multiply by a discount factor so you don't over–evaluate it in case you don't end up getting it. It turns out this particular update rule has a lot of nice mathematical properties, which is what makes Q–learning such a powerful tool.

Q–learning can learn a policy to maximize the final reward even if rewards happen incrementally. In Permutation Hold'em, for example, your rewards are incremented over 1000 hands, but Q–learning can learn to maximize your final bankroll at the end of the game. It is simple and intuitive to learn, as you repeatedly play games deterministically take the best action depending on what worked well in the past. Finally, Q–learning is theoretically sound. Using some math jargon, for any finite Markov decision process, Q–learning finds a policy that maximizes the reward at the end. A finite Markov decision process can be one of: perfect information games (chess, go), video games (Tetris), or Monopoly. On the other hand, Q–learning is *not* guaranteed to work for games with hidden information such as: poker, trading on the stock market, or liar's dice. There are modifications we can make which have great empirical success, however, so don't throw out Q–learning as a technique yet. There are some more downsides of this technique, though: it can be slow to converge, is prone to getting stuck in local optima, and is intractable if the state space is too large. In Permutation Hold'em there are to many game states to apply pure Q–learning to since there are 13! permutations; what we do instead is take inspiration from successes in supervised learning to approximate Q–learning.

5.2 CFR

CFR is an algorithm in computer poker which refers to a specific supervised learning strategy. This strategy can be shown to converge to a Nash equilibrium in imperfect information games in general, making it one of the few algorithms to do this. Every single one of the world's pokerbots which have recently beat human experts has used CFR, and it's amazing for computer poker. Those are the only upsides of CFR: everything else about this algorithm is a downside. The CFR algorithm is confusing, very difficult to implement and debug, impossible to evaluate perfectly, and hard to train well (as it's not obvious how many iterations are needed). If your CFR bot is not working, you may have a bug or you may not have trained it enough and it's very hard to tell the difference.

The first aspect of learning CFR is learning what regret is.[31] I have no regret for the action I do play, but I may have regret for other actions which I did not play. In RPS, for example, if my opponent plays paper and I play rock, I have 0 regret for rock, +1 regret for paper, and +2 regret for scissors (as I would have won). We get these regrets by computing the difference between our actual payoff and the payoff we would've had by playing according to those alternate strategies, where the payoff of a win in this case is +1, a tie is 0, and a loss is -1.

The regrets from our first play give us a mixed strategy for the following round, where we play a pure strategy with probability proportionate to our regret for playing that strategy in the previous round. Regret

[31] The "R" in CFR actually stands for regret, as CFR is an acronym for "Counter Factual Regret Minimization." Even this acronym is bad as it leaves out the "M," but no one says "CFRM."

matching means that you're going to do stuff which you regretted not doing before, which is exactly the same concept as reinforcement learning.

Imagine we play according to that mixed strategy, and our opponent plays scissors. Our regret now is $-\frac{1}{3}$ for our mixed strategy. Now imagine we played one of the pure strategies, instead of our mixed strategy: our regret would have been 1 for rock, -1 for paper, and 0 for scissors. This gives us new regrets for our pure strategies, which we add with our old regrets to get our cumulative regret. This cumulative regret gives us a new mixed strategy. The reason we add is we think of this as updating our strategy. Our old strategy corresponded to our old regrets, but our new strategy needs to correspond to both our new and old regrets to incorporate learning into our strategy. We hope that over time, our mixed strategy will correspond to $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$ and give us a Nash equilibrium, but game theory is not that easy. What we have to do is also keep track of our average strategy. The shortened version of CFR is if we keep track of our average strategy and our new mixed strategy over time according to regret matching, the average strategy over time will converge to the Nash equilibrium. The bad thing about CFR's new mixed strategy is it's very impulsive to big wins or big losses—the average strategy, however, does not have this problem.

To compute an average regret minimizing strategy, we begin by computing a regret–matching strategy profile, which is exactly what we started to do in the previous paragraph. We start by keeping track of our regrets, and create a strategy profile which is proportional to the positive ones.[32] What is interesting is during training we are always playing proportionally to our cumulative regrets, using the strategy which we previously described as very irrational and changeable by big wins or big losses. We use this strategy during the learning process, but during the actual competing we only use the average regret profile to create our strategy.

We compute our average strategy as follows. We begin by computing a mixed strategy for each player by matching cumulative regrets. We then select each player's action by sampling from their mixed strategies. Next, we update the cumulative regrets. We repeat this process T times, and return the average mixed strategy across the T iterations. This process we've described would be able to approach the Nash equilibrium for a game like RPS, but it gets much more complicated when you have something like poker with a full game tree.

CFR is even more complicated as it calculates strategies using "counterfactual regrets," which we're not going to go over in this lecture. For now, we're going to think of counterfactual regret as the same as regret since it has the same intuition as the regret we've been talking about so far.[33] Counterfactual regret answers the question: what would node n's value change to if I picked some pure action a? In order to talk about this, we first need to define the value of a node. In RPS we only had value of the game, but in the setting of Hold'em where you have a big game tree it's not obvious what we mean by value of a node in the game tree since the game is not finished yet. We define the value of a node recursively: the value of a node is the value of that node's children multiplied by the corresponding action probabilities (according to n's counterfactual regret–matching strategy). This is sort of like averaging the value of the continuation states weighted by their probabilities, and is thus very similar to calculating an expected value.

What we will talk about is summing counterfactual regrets over information sets, which you can think of as our possible game states in the poker game tree. This brings us to the overall procedure for CFR. We're specifically presenting a version known as "Monte Carlo CFR." The original paper implementation of CFR would need to traverse the entire game tree, which means it would have to traverse every possible hand or board you're dealt and then apply the CFR algorithm. This is unfortunately way too much computation complexity, but thankfully new improvements came to the CFR algorithm which allows you to use CFR as if you are only playing a single hand of poker. We're only going to play one iteration. We're going to construct the game tree we're considering, and assign weights to each node—the weight of each node is the likelihood of us getting there, determined by the products of our current strategy profiles (again, proportional to positive regrets). What this means is we're not weighting by the probability of us getting particular cards, as this is intrinsic and we're going to simulate us getting particular cards many times over. We then compute counterfactual regrets at each node, and this corresponds to our reinforcement learning aspect or updating the strategy profiles. We then run regret-matching to get our updated new strategy profiles. After we're done

[32] You may notice our regrets were negative at some point, but you cannot do an action with probability proportional to a negative number. What we do is throw out the negative numbers and set the probability of that strategy to 0. If all our probabilities are 0, our profile is telling us all our actions are really bad, so we randomly choose a pure strategy.

[33] Just keep in mind that we'd need to tweak this algorithm if we code it up based on differences in the research papers.

many iterations with many samples, we're going to use a weighted average strategy profile to play poker. We weight later iterations heavier, as they've had more of a chance to converge and use more nuanced CFR strategies but that aspect is less important. This resulting algorithm is called Monte Carle CFR (MCCFR).[34]

There are many problems with this. In order to calculate a regret for every node of the game tree, we need to visit every node of the game tree, which is computationally impossible. We decide to instead group together board states we consider similar enough to reduce the size of the game tree, and we call this "bucketing." One way to start this is by grouping together hands you're dealt that look similar—for example, a [5, 2] hand may look like a [6, 2] hand if the cards are different suites, or a bet amount of 17 may look like a bet amount of 18. You can eliminate previous histories to group together board states as well, and only deal with pot odds or other aspects from the poker lecture to bucket together types of game tree nodes. Another issue you may run into is bot size limits: you can only store such a large game tree before running out of the space we allot you. A way to get around this is by using a modified algorithm called CFR+, which converges the average strategy profile with the latest strategy profile.[35] Finally, external sampling can be used to improve your CFR bot. By using external sampling, you can reduce the height of your game tree by a factor of two, without compromising much on convergence speed. You're only sampling the first or second player, so you can eliminate all the back and forth and simply sample one action from the other player's strategy rather than iterating over all possible actions. We'll next look at how CFR was combined with another machine learning strategy called neural networks to make it even better.

5.3 Neural networks

if you take 6.036, they will tell you a neural network is a "multilayer perceptron," but in Pokerbots we only care about how to use them. Neural networks are one of the most empirically successful ways to approximate a function based on a limited number of input–output pairs which constitute our training data. Neural networks can approximate a wide variety of functions with great success in practice. They come in many shapes and sizes (bigger ones are more expressive, or can better approximate complicated functions). What makes them especially good is they can generalize to unseen data—in Pokerbots, if you run into an unseen scenario in a tournament, a neural network will be able to generalize from known scenarios to still play good poker.

They are used frequently in image and speech recognition, recommender systems, AlphaZero and also in the DeepMind Parkour example. The key idea here is that a Q–table is in itself a function, meaning we can approximate it using neural networks. This is an entire field of itself called "deep Q–learning." Finally, another pokerbot called DeepStack uses neural networks, and we'll talk about this pokerbot more next week.

There are lots of functions that could be worth approximating. Some examples are game states leading to a betting strategy. To learn this, however, you'd have to have example logs of an already very strong poker player, and if you have a strong poker playing bot you'd just use that instead. Another function you may want to learn is mapping cards to a final hand strength, which could save you from using very expensive Monte Carlo simulations and is thus very practical. You can also approximate the function from previous showdown results to the permutation. Finally, you map from information sets to buckets, but this has never been done successfully yet.

The biggest problem of neural networks is that even though they need limited training data, they still need quality training data, and this can be hard to obtain. Getting good training data in Permutation Hold'em is very difficult to obtain. If you get your training data from the scrimmage server, you'll simply end up copying your opponents—this will lead to you just being able to impersonate your opponents, not necessarily beat them as you would like to. If you play two neural nets against each other, it's hard to know whether it will converge or get caught in cyclic behavior; thus, this is not the best way to train a neural net.

[34] We highly recommend reading more about CFR here: `http://modelai.gettysburg.edu/2013/cfr/index.html`.
[35] Note that CFR+ is even more complicated than CFR, so proceed at your own risk.

6 Lecture 6: Advanced Topics II

This lecture is taught by David Amirault.[1] The slides from this lecture are available for download on Stellar.

At this lecture (and all others), we raffle off a pair of Beats Solo 3 Wireless Headphones. Attend lectures in person if you want a chance to win them!

We'd like to thank our eight Pokerbots 2020 sponsors for making Pokerbots possible. You can find information about our sponsors in the syllabus and on our website, and drop your resume at `pkr.bot/drop` to network with them. Our sponsors will also be able to see your progress on the scrimmage server, giving you a chance to stand out over the course of our competition.

Additionally, a reminder that Mini-Tournament 2 is tonight! Be sure to **submit your bot by 11:59pm EST on 01/17/2020** to receive credit for this course.[9] Your week two bot must beat your week one bot and a bot that plays randomly. If your week two bot fails to do so, we will contact you over the weekend and you will be expected to submit a half-page double-spaced makeup report on failed improvements that you attempted over this past week. All makeup reports are due by 11:59pm EST on Tuesday, 01/21/2020.

Announcement: The Final Tournament logistics have been finalized: the event will take place on Friday, January 31, 2020, from 4:30–7:00pm in Building E51 (Tang Center). Be sure to save the date in your calendars! The Final Tournament event is where we'll announce the winners of this year's large prize categories (including the Pokerbots 2020 Champion), and will feature more raffles, an opportunity to network with our sponsors, food, and the chance to play against the bots you've built this month! More details will be sent out over Piazza during weeks three and four.

6.1 DeepStack

DeepStack plays the game of poker by using a continual re-solving approach. That is, in the format of Pokerbots, DeepStack would implement the following logic in the `get_action()` method of its `player.py` file: i) run thousands of iterations of CFR starting from the current game state as the root node in the game tree; ii) calculate a mixed strategy for the root node, which is the current game state; iii) finally sample from this mixed strategy to decide which action to take. Ordinary CFR cannot do this, so DeepStack has to go about implementing this algorithm in a very unique way.

DeepStack is the combination of a bunch of good ideas in poker, meant to address many flaws in CFR which we covered in lecture five. The first flaw is that bucketing introduces exploitability. What we mean by "bucketing," to review from the CFR lecture, is we take a bunch of different game states with different information available to us and group them together to create a smaller game tree. An example we gave was treating all bets from 60 to 80 as the same bet, and this is a process that simplifies the game tree. Depending on how good your bucketing function is, your CFR implementation will get different mileage. What DeepStack does is use a completely different version of CFR. Every time you need a decision, it completely re-solves CFR without keeping track of a full-game strategy, only starting from the current game state. You don't need to make compromises about the information you're starting with, as without bucketing you have exact information about your position in the game tree. It's very surprising this even works, as CFR solves for an approximation of the Nash equilibrium. This doesn't depend only on the current game state, but multiple possible game state in the game tree. The Nash depends on how you would be playing other cards had you been dealt other cards, but DeepStack can compress this into a small amount of memory which changes as you proceed through the game. Another flaw of CFR is the game tree is too large, which makes our training slow. How DeepStack addresses this is by using multiple neural networks to approximate the game tree beyond a certain depth, preventing the combinatorial explosion in the size of the game tree. This gives you results as if you're exploring the whole game tree without doing so, which gives you great results for computational complexity.

On slide 8, CFR might treat the listed poker game states as the same with its bucketing function, which loses a lot of information about the current game state. DeepStack, on the other hand, starts from the *exact* current game state, which means we don't need to compromise on which current information to include through bucketing. We can include all the available current information when we solve forward for how to play optimally. As humans, if you have a poker hand and are in a current game state, you can reason about your different actions available to you and proceed based on that. The Nash equilibrium, however, does not depend only on future possibilities—it depends on many alternative game states as well.

DeepStack uses neural networks to simplify the prediction of future game possibilities. Slide 9 depicts a schematic of the poker game tree. What DeepStack does is use a neural network to approximate the result of a traversal on all the nodes below a certain depth of our game tree. For example, we only need to predict up to the flop in our original game tree; we then plug in a flop neural network to approximate exploring the game tree further. At the turn, we use another turn neural network to approximate exploring the game tree to the showdown. DeepStack's CFR only uses one round of betting, which lets us explore further than we could in ordinary CFR. An issue with this game tree though is its high branching factor and high depth. The game tree branches a lot as whenever you're asked for a decision you have lots of choices you could make, and it's deep because you're making a lot of actions and there is a lot of back and forth decision-making between the two players. Both of these numbers are prohibitively large in poker.

We've said DeepStack uses a continual re-solving approach, but we haven't shown how we can do this. The authors of DeepStack prove that you can solve for the Nash equilibrium only with two pieces of information from the current game state, which is very surprising. These two pieces of information are: our hand range and our opponent's counterfactual values. Technically, if you have some sort of bot that can play perfectly given these two pieces of information, you can play a Nash approximation of poker. Our"range" refers to the cards we could be holding and with what probabilities, if our opponent knew our exact strategy but not our private cards. The array of our opponent's counterfactual values refers to how good it would be for our opponent to be holding two hole cards for all possible two-card combinations. You can think of these as the two instance variables in DeepStack's `Player` class in the `player.py` file.

For any given hand h, the counterfactual value of h answers the question: what is the expected number of chips I would win if I were instead holding h? We ignore the hand we are actually holding, and instead compute the expected number of chips we would win *if we were instead* holding a different hand h. Some example counterfactual values are on slide 11.[36] We may be wondering how DeepStack actually computes these counterfactual values, and the general answer is to update it each time we run CFR. This doesn't answer how we have an array of counterfactual values the very first time we run CFR though, and the answer to this is we must simply trust the ability of our neural networks to evaluate poker through the end of the game tree with a good enough approximation.

DeepStack starts from the current game state, and we have our range and opponent's counterfactual values memorized. Next, we simulate all continuations recursively. In the DeepStack paper, they cut things off at recursion depth four; once they reach this depth, they plug the game state into a neural network, which estimates what would happen if we continued all the way down using CFR in the subtree up until leaf nodes.[37] As we traverse the game tree, we update our range using the action probabilities and our opponent's counterfactual values using CFR. What makes DeepStack's CFR special is this CFR is augmented to reconstruct our opponent's range given their counterfactual values.[38] At recursion depth four, we estimate counterfactual values of continuing the game until showdown using one of our neural networks.

Now, let's get back to how DeepStack uses neural networks and how it generates the high quality data needed to train these neural networks. The function DeepStack chooses to approximate is a function with four inputs, which are: our range, our opponent's range, the cards on the board, and number of chips on the board. Given these four inputs, our function will output the counterfactual value of each hand that could be held by each player at the turn. This is a very clever function for DeepStack to approximate, since two of the inputs depend on being dealt cards (specifically for us and our opponent). We also have four fixed board cards and pot size, so this describes a fixed game state. These represent a (smaller) game, just like poker, which means it can be solved with approaches much smaller than the DeepStack approach—such as regular CFR. The reason this game is so much smaller than poker is there is only one card left to be dealt, and there's only one round of back and forth betting remaining. This game is considered tractable—we can solve it relatively easily offline to get high quality data for training our neural network. DeepStack solved 10 million turn games to get data for their turn neural network.

With the turn neural network, DeepStack trained another neural network. This new neural network has a very objective similar function, with the same inputs and outputs, except this time board cards are fixed at the flop. You may think this resulting game is more complicated than the previous game, but it's not—because we have a high quality turn neural network. Instead of solving for the fifth card scenarios this

[36] Note that these are completely hypothetical pre–flop counterfactual values for these hands.
[37] Recursion depth four means four levels of back and forth actions between us and our opponent.
[38] This augmented CFR was created by the authors of the DeepStack paper.

time, we simply plug in our turn neural network, and get high quality data at the flop stage as well. At this stage, we have two neural networks—one for the turn, and one for the flop—and these neural networks let us solve for continuations at three–card scenarios. This means we can apply it to pre–flop situations, which is what we want! For every possible flop that we are dealt, we can plug into our flop, and this will give us the approximate value of our pre–flop stage; we have finally turned this into something which is tractable at the very beginning.

There is one last aspect of DeepStack we have not talked about; in spite of all these optimizations, we have not fully solved for poker. If we considered every possible bet size, four back-and-forth actions would be about 200^4 game states, which is too many to deal with in a handful of seconds. To deal with this, DeepStack buckets actions and only deals with ten or so for an action. This action bucketing is much better than CFR information set bucketing, however; when we bucket on actions, we only group together possible continuations, but in CFR bucketing information sets, we lose information regarding the game state we are currently playing. We also reduce the depth of the game tree, through neural networks, which makes the game tree a tractable size. The resulting algorithm will play ordinary heads–up no–limit Texas Hold'em and runs in under 5 seconds on gaming laptop–tier hardware, which is a great improvement over previous algorithms which required university resources. Even though it can run on gaming laptop–level hardware, it still beat 33 world poker experts with statistical significance, and is considered the state–of–the–art computer poker playing algorithm.

6.2 Advanced inference

6.2.1 Graphical models

We first need to introduce a bit of new language for Bayesian inference, which we covered in lecture 3 when we introduced the particle filter algorithm. This time the algorithms will be a bit different, but still talking about the same problem of solving for properties of probability distributions.

Probabilistic graphical models, sometimes called Bayesian networks, express a *statistical dependency structure* between random variables. In the real world, these are ideally *causal relationships*. Slide 23 has an example: the physical world relationship between sprinklers, rain, and wet grass. We look at a bigger example on slide 24, which is a graphical model for fire alarms. We have student evacuated from their dorm, and want to figure out what the reason might be. This causal graphical model has a lot of possible examples for what might have caused the fire alarm being set off. Of course, burning food and chemicals could start a fire which could start a fire alarm, but it could also have been faulty or set off as a prank by a student. If the alarm goes off students will have to evacuate, but they might also have to evacuate in a flood or some other disaster situation.

Let's get back to thinking about poker. The engine samples a permutation, and every showdown you see depends only on this permutation. This is a very simple graphical model, and there's not much going on here. We can expand this to get a more expressive graphical model, which describes more of what's going on in this scenario. To do so, we're going to introduce more random variables to take a look under the hood. For example, TV(2) is the true value of 2, and TV(3) is the true value of 3. These true values are what drive the showdowns, as the true value of every card that you see in a showdown is what causes the outcome of a specific showdown. More complete graphical models often impose more structure on the hidden variables, in this case TV(2), TV(3), etc; however, in the case of Permutation Hold'em, these two models are equivalent because the true values are in one–to–one correspondence with the permutation.

The more complex graphical model might be more convenient if our goal is to perform inference on one true value individually. For example, this might be the case if we are currently holding a pair of 2s, and our goal is to figure out how strong that pair of 2s might be. In any case, it is up to you to decide whether the added complexity of the graphical model is worth that convenience. Now that we have a graphical model in mind for Permutation Hold'em, we get to the fun part of choosing an algorithm which we wish to use to find information about these random variables.

We will need to do approximate inference in Permutation Hold'em, since doing exact inference would require too much computational time. Exact inference would require us to at least consider each of the 13! permutations, which is infeasible given our 30 second game clock requirement. To do approximate inference,

we recall Bayes' theorem from our inference lecture:

$$P(M|S) = \frac{P(S|M) \cdot P(M)}{P(S)}.$$

We remember that S is the observation (showdown outcome), and M is a candidate permutation. The reason we care so much about the posterior distribution is that the posterior distribution exactly describes which permutations are still possible, and with what probabilities. If we had a magic oracle which gave us samples from the posterior distribution, then we would have reduced Permutation Hold'em to ordinary Texas Hold'em. That's why $P(M|S)$ is our target for approximate inference. The idea of approximate inference is we wish to approximate the posterior distribution $P(M|S)$ using $Q(M)$, and the key idea of approximate inference is Q belongs to a family of easy–to–work–with distributions:

$$P(M|S) \approx Q(M).$$

6.2.2 Variational Bayesian methods

One such family of distributions is variational Bayesian methods. They assume that $Q(M)$ *factorizes* over the hidden random variables

$$Q(M) = Q_2(M_2) \cdot Q_3(M_3) \cdots Q_A(M_A),$$

where $Q_2(M_2)$ is the probability that the true value of 2 is M_2, and so on.[39] This is called the "mean field approximation."

Unfortunately, this factorization assumes that all 13 of these true values are independent of each other. If this were close to being true, we would get some pretty strong results; however, this is problematic with Permutation Hold'em, as there would be degenerate cases with repeated assignments. The mean field approximation would allow for such cases as every card being a 2, or every card being an A—clearly both impossible. We are also ignoring that one showdown imposes a constraint on multiple true values, which prevents us from fully incorporating information from straights or one pair being higher than another pair (as this would require encoding information about two types of cards). It turns out that any Bayesian approach which assumes the true values are independent is going to run into this issue. For example, the belief propagation algorithm and many other Bayesian inference algorithms are also invalidated by this assumption. The reason this mean field approximation is so popular is it makes a lot of other math very easy, but it unfortunately fails us in this case. We instead look to making a different simplifying assumption, which leads us to a different algorithm with a lot more promise than variational Bayesian methods.

6.2.3 Markov Chain Monte Carlo (MCMC)

We're not going to go over the proofs in this section, but the procedure we provide will give you a good approximation of techniques very helpful in this year's Pokerbots variant.[40]

We're going to take a random walk through the space of permutations, making slight tweaks to our permutation with each random step, with the hope that we can bias our walk towards correct permutations over time.

What does such a random walk look like? Let's start with an idea of random swaps. We're going to randomly pick two indices of true values in our original permutation of no change, and randomly swap them to create a new permutation. These random walks have some technical constraints, which you can search more about online using these names. The first such constraint is called "detailed balance." Detailed balance means that if we have two permutations M_1 and M_2, the probability

$$P(\text{go from } M_1 \text{ to } M_2) = P(\text{go from } M_2 \text{ to } M_1).$$

Note that this doesn't mean you have to be able to go to every permutation, but for a single random step the probability should be equal in either direction. Random pairwise swaps do clearly satisfy detailed balance. The other constraints on our random walk are that it is "positive recurrent" and "aperiodic." Positive

[39] Another way of saying this is that $Q(M)$ is a product of "marginals."

[40] If you *are* interested in the proofs, there are many good materials available online from MIT's inference classes.

recurrent means that you visit all permutations with positive probability, and aperiodic means your random walk does not revisit permutations at regular intervals. If your random walk visited certain permutations every even random step, that would be a periodic random walk. Random pairwise swaps can indeed reach every permutation, and it is also aperiodic if you no–swap with positive probability. You don't need to know what these constraints mean mathematically, but you should be able to check for them.

If we verify all these constraints, we can present the Metropolis algorithm (sometimes called the Metropolis–Hastings algorithm), in this case applied to Permutation Hold'em.[41] We start out with an arbitrary starting permutation M_0; since our random walk can go to any possible permutation, it doesn't really matter where we start. We are then going to do T iterations of generating a candidate permutation before deciding whether or not to random walk to that candidate permutation. We're going to call M' our candidate permutation, generated by our random walk. We will define the *acceptance ratio*, the chance that we decide to take the random walk to M', as

$$\alpha = \frac{P(M')}{P\left(M_{i-1}\right)} \cdot \frac{P(S|M')}{P\left(S|M_{i-1}\right)}.$$

We then randomly draw $u \sim \text{Uniform}[0, 1]$, and if u is less than the acceptance ratio we perform the random walk to get to this new candidate permutation.[42] Finally, we return our ending candidate permutation after T iterations, M_T, which is an approximate sample from the posterior. In case you forgot from lecture 3, the reason we want to sample from the posterior distribution is this exactly describes what permutations could be the correct permutation for this game.

There are, unfortunately, some issues with the Metropolis–Hastings algorithm in Permutation Hold'em. If $P(S|M')$ is 0, then we're always going to reject that candidate permutation, which means we only accept when it's 1. This results in many rejections, or in the worst case a stranded random walk: this could sound very unlikely, but it is quite possible. Imagine if the only remaining valid permutations start with [234 . . .] and [423 . . .], which means we can't go between them with only one pairwise swap. If this happened, MCMC would not work because positive recurrence would be violated. One way to get around this is to randomly sample a new permutation at each step of our random walk, but this would just mean randomly selecting a new permutation each step which is just a less effective particle filter.

One of our best alternatives might be to use Metropolis to sample from an analogue of our posterior, rather than from the true posterior distribution. We could imagine sampling from $P(M|S')$, which is all showdown information ignoring straights. This seems to work great, but has a caveat as always since we are throwing out information about straights—this means our opponent could have more information than us, if they are keeping track of straights.

Thank you for attending (or reading notes from) the Pokerbots 2020 lectures! We hope you could all learn something new from our six classes, and find these topics useful in building your pokerbot. We look forward to seeing you again at our final tournament event on Friday, January 31$^{\text{st}}$, and wish you the best of luck for the rest of Pokerbots 2020!

[41] Here is a visualization of the Metropolis-Hastings algorithm:
https://chi-feng.github.io/mcmc-demo/app.html?algorithm=RandomWalkMHtarget=banana.

[42] This acceptance ratio might be dividing by 0, but if this is the case then we imagine that the acceptance ratio is positive infinite. In this case, we are always going to accept our proposed permutation.