Pokerbots Course Notes

Massachusetts Institute of Technology
6.9630: Pokerbots Competition[*]

IAP 2023

Contents

[*]Contact: pokerbots@mit.edu

1 Lecture 6: Advanced Topics I

This lecture is taught by Matt Mcmanus.[1] All code from lecture can be found at the public github.com repository mitpokerbots/reference-lecture-6-2023. The slides from this lecture are available at the public github.com repository mitpokerbots/class-resources-2023

At this lecture (and all others), we raffle off a pair of Sony Headphones. Synchronously attend lectures if you want a chance to win them!

We'd like to thank our twelve Pokerbots 2023 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.

Announcing our Mini-Tournament 2 results, which had over 95 teams participating, CallingStation was again our winner with an ELO of 2136.

Announcement: The Final Tournament logistics have been finalized: the event will take place on Friday, February 3, 2023, from 4:00–7:00pm 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 2023 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.

1.1 Machine Learning

We will begin the lecture with a quick overview on machine learning. In general, machine learning is a topic that is commonly heard of all the time in computer science research and products. It has become a hot topic in the computer science field over the last several years, and it has earned such recognition due to its success in solving some really important problems. Over the last year alone, there were a few big advancements in the field of machine learning, two of them being DeepMind's AlphaFold for predicting protein structures and OpenAI's GPT-3 language model. Despite the confusing terms that are often associated with news about advances in machine learning, machine learning boils down to a relatively simple concept. Machine learning is a catch-all term that describes a class of algorithms that gets better with experience. They complete a process or task, like the algorithms we have introduced previously in this class, but they get better at completing that task with experience. In fact, at a certain point, they can become good enough to handle the task with near perfect accuracy. There are many applications of these algorithms. One application is prediction, such as prediction the answer to whether or not it will rain tomorrow. Another application is decision making, which is the area in which your Pokerbot algorithm falls under, as you are making decisions in betting rounds and card allocation rounds. Lastly, we also have the area of general automation, which combines aspects of both prediction and decision making, including speech and visual recognition.

Generally all algorithms that perform the tasks described above get better through experience. Such experience mainly comes from either looking at data or trying a new task over and over again. This process is known as **training**. Training is the term used to describe this period of time. A typical training plan begins by starting with a simple strategy. The next step is to take in experience by looking at a data point and trying something new. With this new experience, the algorithm is able to learn something new about the strategy it is attempting. From this new piece of knowledge, the algorithm updates its strategy and repeats the process. This framework is the general workflow of a machine learning algorithm in its process to optimize its strategy.

Next, we are going to introduce the typical algorithms used in machine learning that build upon this training process. These algorithms will generally fall into one of three categories. The first category is **supervised learning** algorithms, in which we show our algorithm many input and output examples in attempt to teach the algorithm to learn those input-output mappings itself. By learning and building on these mappings, the algorithm improves its ability to predict future inputs that it has not seen before. Next, we have **unsupervised learning** algorithms, in which we show our algorithm many input examples but do not tell the algorithm what the correct output answer is. In this case, the algorithm has to determine some pattern within the data without outside help. Finally, we have **reinforcement learning** (RL) algorithms,

[1] Email:mattmcm@mit.edu.

in which our algorithm learns from an environment. Typically, our algorithm is some sort of player or agent, and that agent learns to make decisions/navigate its environment in order to get some reward.

In today's lecture, we will touch on examples of each of the three algorithms mentioned above. We will begin with a quick intro to supervised and unsupervised algorithms and spend a bulk of the remaining lecture focusing on reinforcement learning algorithms.

An example of a typical supervised learning framework would be the following. We want to build an algorithm that is able to take a photo and tell us if it is a cat or a bird. We also have pre-existing example photos that come with labels that describe if that photo is one of a cat or a bird. To build our algorithm, we begin with the training process. We show the algorithm a bunch of different photos of cats, which act as our data, and with each photo we add an appropriate label. In this case, the label would be a string representing the word, cat. Similarly, we do the same process for bird photos, and we occasionally show our algorithm pictures of a bird with its corresponding bird label. Over time, the algorithm will get a good idea of what looks like a cat photo and what looks like a bird photo. After a while of training, we will be able to show our algorithm a completely new photo of either a cat or a bird, and if trained well enough, it will be able to correctly identify the animal in the picture. Generally, neural networks have had great success working with image data, whether that be images of animals or even handwritten digits. In contrast, there have also been numerous other algorithms that support numerical data such as decision trees, regression, and support vector machines (SVM).

Alternatively, we also could have used an unsupervised learning approach to solve a similar problem. Given a group of images of both birds and cats, instead of creating an individual label for each image, we can use an unsupervised learning algorithm that groups the images into two groups (commonly referred to as clusters) based on how similar they are. We may want to do this approach in the case when we have too many images and adding individual labels is too time-consuming. Because the algorithm is unsupervised, it performs the grouping by recognizing certain patterns between the different images without putting a real label on what exactly those similarities are. Many unsupervised learning algorithms utilize clustering from data analysis techniques such as k-means clustering, k-nearest neighbours, and the Expectation-Maximization Algorithm (EM).

1.2 Reinforcement learning

1.2.1 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.[2]

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.

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.

1.2.2 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.[3] 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,

[2]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.

[3]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.

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 39 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.

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

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. 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.

1.3 CFR

CFR (Counterfactual Regret Minimization) 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.[4] 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

[4]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."

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 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.[5] 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.[6] 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

[5]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.

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

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've done 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).[7]

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.[8] 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.

1.4 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.

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 could also map your starting hand to a distribution of "swapped hand strength." Finally, you can map from information sets and game states to buckets as defined in CFR, 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. 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 against a fixed strategy, it will lead you to beat that one strategy but not play poker well.Finally 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.

[7]We highly recommend reading more about CFR here: http://modelai.gettysburg.edu/2013/cfr/index.html.

[8]Note that CFR+ is even more complicated than CFR, so proceed at your own risk.

Thank you for attending (or reading notes from) the Pokerbots 2023 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, February 3$^{\text{rd}}$, and wish you the best of luck for the rest of Pokerbots 2023!