No Game (Theory) No Life: The Machine Learning Approach to Playing Games

Stephen Huan

April 27, 2020

Figure 1: No Game (Theory) No Life [1]

1 Introduction

Contrary to its name, game theory has deep implications beyond playing games very well - life itself can be thought of as a difficult game.

A connection between game theory and cutting edge machine learning exists under a mathematical intrepretation of Generative Adversarial Networks (GANs) - many papers describe GANs as playing a minimax game and converging to a Nash equilibrium. A machine learning club lecture was written on this topic [2]. Another connection exists when considering Reinforcement Learning (RL) as commonly applied to game-playing (exemplified by Google DeepMind's recent Go, Shogi, and Chess RL system called AlphaZero [3]). RL, however, is a more general technique that has often been applied to robotics (see OpenAI's robot hand that solves a Rubik's cube [4]). Notably, GANs, RL, and energy based models (EBMs) are all mathematically equivalent (under certain conditions) [5] which gives a nontrivial reason to study game theory - it advances the cutting edge of machine learning as advances in playing games directly leads to advancements in GANs, RL, and EBMs.

2 Algorithms

2.1 Minimax

Minimax is a perfect information game (i.e., a game without hidden states) player. It views a game as a tree of states, with actions being state transitions (edges in the tree) and attempts to pick the best subtree at each node. It does this via a depth-first search: one move away from an end game state, the best move is obvious - the move that maximizes the immediate utility. Two moves away, it has the score of all its children, and just pick the best moves from those. Move up the tree and it can reconstruct the optimal move for each state.

However it's infeasible to actually compute the entire game tree for a nontrivial game. Therefore, we need to use a heuristic which caps the search depth to something that is reasonable and is usually human designed (see the Othello project in AI).

One such approach to creating a good Othello player is documented in the Logisthello bot [6]. He essentially assumed the heuristic was a linear function of features he hand-defined, and solved linear equations for an optimal heuristic iteratively with backpropagation (neural networks weren't as popular in the 90's). The ideal heuristic is obviously the actual utility function - whether one wins or loses from a given position given optimal play, and having to select features by hand is time consuming, requires a in-depth knowledge of the given domain, and may not be optimal. One might consider training a neural network to estimate the true utility function but there is a paradox - where does one get the training data for optimal play when one doesn't know how to play optimally? Logisthello used human Othello games but obviously that may not be optimal. This paradox is usually resolved through self-play, where the machine gets iteratively better as more games are played.

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an algorithm similar to Minimax in the sense that both are tree searches. However, MCTS is more in line with the "self-play" philosophy - it essentially uses the result of a large number of random games as heuristic for how good a particular position is. MCTS was used by the AlphaZero system along with RL.

3 Counterfactual Regret Minimization

Both algorithms, however, require perfect information. In real life, however, we rarely have perfect information - one is often uncertain excactly what effect an action can have and other variables. One such problem is Reconnaissance Blind Chess (RBC), proposed by John Hopkin's Applied Physics Lab [7]. RBC is played like regular chess except one does not know what move their opponent made. Instead, every turn each player can "sense" a 3x3 region on the board and learn the positions of every piece in that region. Minimax, MCTS, and RL do not apply well to such a game. Thus, the creation of another algorithm is necessary to account for uncertainty.

3.1 Definitions

For a perfect information game, playing a game "optimally" is well defined because each state has exactly one "optimal" move. However, for a game with uncertainty, what does it even mean to play optimally?

For simplicity, let us simplify games into ***normal-form games*** [8]. A normal-form game is game with N, the set of n players, S_i, the ***actions*** for a player i, A, the set containing all possible combinations of actions for each player, and u, the vector giving ***utilities*** (the rewards) for each player (a particular player's utility is u_i). These games are commonly represented by tables, where the row player is player 0 and the column player is player 1. An entry in the table is pair of numbers in the form (u_0, u_1), representing the utility for player 0 and player 1.

	R	P	S
R	0, 0	-1, 1	1, -1
P	1, -1	0, 0	-1, 1
S	-1, 1	1, -1	0, 0

Table 1: The payoff table for rock paper scissors

A game is ***zero-sum*** if each entry adds up to 0. Finally, a ***strategy*** for a game is a function representing the probablity that an action is chosen. We notate strategies as $\sigma_i(s)$, representing the probability for player i to use action $s \in S_i$. $-i$ refers to the opponent, so in the case of two players $S_{-i} = S_{1-i}$.

Since games under uncertainty are played in a probabilistic manner, playing the same strategies against each other multiple times will have different results. Rather than track simple win/losses for a single game, we instead define the ***expected utility***, the amount of utility that the player expects to get per game over the long term. The definition of expected value is sum of each value times the probability of that value occuring. To use this definition in our game, we sum over each possible value (each entry in the table) and the probability that such a value occurs takes into account both player's strategies.

$$u_i(\sigma_i, \sigma_{-i}) = \sum_{s \in S_i} \sum_{s' \in S_{-i}} \sigma_i(s)\sigma_{-i}(s')u_i(s, s')$$

Under this framework, we finally arrive at the defintion of optimal play for uncertain games. A ***best response*** strategy is one that maximizes a player's expected utility, given the strategies of the opponent(s). If both players play best responses, the combination of strategies is the ***Nash equilibrium***. As we shall see, one intrepretation of the Nash equilibrium is that players cannot change strategies to gain an advantage - any change in strategy will yield the same expected utility (this is consistent with the definition of "maximizing" because, intuitively, any movement yields the same function value, so our current position must be a local extrema). A player is ***indifferent*** if all actions have the same utility. If both players are indifferent, than that is a Nash equilibrium.

3.2 Examples

3.2.1 Rock Paper Scissors (RPS)

It seems obvious that the optimal strategy for RPS is to play rock with $\frac{1}{3}$ probability, and the same for paper and scissors. Suppose that one deviated from the optimal strategy and played $\frac{1}{2}$ rock, $\frac{1}{4}$ scissors, and $\frac{1}{4}$ paper. What would the optimal response be? Basic analysis shows that it would be to punish the over-emphasis on rock as much as possible, by always playing paper (a strategy always playing a single action is known as a ***pure strategy***, and a non-pure strategy is a ***mixed strategy***). The strategy of always playing paper yields an expected value of $\frac{1}{2}(1)+\frac{1}{4}(-1)+\frac{1}{2}(0) = \frac{1}{4}$. Since playing any action with greater than $\frac{1}{3}$ leads to a pure strategy punishing that deviation, the optimal strategy must be the one stated above.

We can also reason why the the solution fulfills our definition of the Nash equilibrium. Computing the expected values for playing rock, paper, and scissors:

$$u_R = \frac{1}{3}(0) + \frac{1}{3}(-1) + \frac{1}{3}(1) = 0$$
$$u_P = \frac{1}{3}(1) + \frac{1}{3}(0) + \frac{1}{3}(-1) = 0$$
$$u_S = \frac{1}{3}(-1) + \frac{1}{3}(1) + \frac{1}{3}(0) = 0$$

Since the expected value for playing any action is the same, we are indifferent, and so it must be the Nash equilibrium.

3.2.2 Pure Strategies

We saw in RPS that a deviation from the Nash equilibrium led to a pure strategy being optimal to take advantage of that deviation. Is that generally true? The answer is yes, and to prove it requires a basic understanding on linear programming. In normal-form games, our expected utility must be a linear function of the probabilities. The constraint on the probabilities is that they sum to one and that each probability is greater or equal to 0. Given a linear objective function we want to maximize and a linear constraint, linear programming states that the optimal value occurs at a vertex of the simplex, or the region of feasible values for each parameter. Since the constraint $x_1+x_2+\cdots+x_n = 1$ along with $x_i \geq 0$ has a feasible region which is the hyperplane's intersection with each axis, the vertex of the simplex must be when one probability is 1 and the rest are 0, fulfilling the definition of a pure strategy.

3.2.3 A Variant of RPS

To see one way to compute the Nash equilibrium for normal-form games, one needs to have a less obviously solved game. Consider RPS, but one player is handicapped in the sense that they are at a disadvantage if they play scissors. All interactions are identical to standard RPS except when player 1 plays scissors: player one scissors loses against player two scissors, ties against player two paper, and loses against player two rock.

Figure 2: Utility table for the game (from *No Game No Life* episode 2, circa 5 minutes)

Let us calculate the best response strategy for Stephanie, the girl on the right. Let player 0 be Sora, the boy on the left, and player 1 be Stephanie. Let $p_1 = \sigma_1(R)$, $p_2 = \sigma_1(S)$, and $p_3 = \sigma_1(P)$ (encapsulating Stephanie's strategy). We have:

$$u_R = p_1(-1) + p_2(0) + p_3(-1) = -p_1 - p_3 \tag{1}$$
$$u_S = p_1(-1) + p_2(-1) + p_3(0) = -p_1 - p_2 \tag{2}$$
$$u_P = p_1(1) + p_2(-1) + p_3(0) = p_1 - p_2 \tag{3}$$
$$p_1 + p_2 + p_3 = 1 \tag{4}$$

Let $u_R = u_S = u_P$, giving us three equations and set (2) equal to (3):

$$-p_1 - p_2 = p_1 - p_2$$
$$p_1 = 0$$

Set (1) equal to (2):

$$-p_1 - p_3 = -p_1 - p_2$$
$$p_2 = p_3$$

Because of (4), $p_2 = \frac{1}{2}$ and $p_3 = \frac{1}{2}$. Thus, the optimal strategy for Stephanie is to never play rock and flip a coin between scissors and paper. Doing similar logic for Sora, his optimal strategy is to play $\left[\frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{4}\right]$. Using the expected value formula from earlier, Stephanie's expected value is $\frac{1}{2}$ and since this is a zero-sum game, Sora's is $-\frac{1}{2}$.

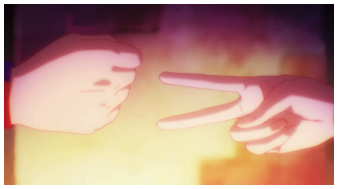

Figure 3: The end result of the game, which is consistent with our calculations.

3.3 Regret

Normal-form games can be solved as a linear system of equations as shown above. However, for games that are not normal-form, a more general algorithm is necessary to calculate the Nash equilibrium. One such algorithm is based off the concept of regret.

Suppose we are playing (standard) RPS and we played rock while our opponent played paper. We regret not playing paper and drawing, but we regret not playing scissors, and winning, even more. ***Regret*** of an action is formally defined as the difference between the utility of that action and the action we originally chose. Given this definition, we regretted not playing paper 1 $(0 - (-1))$ and not playing scissors 2 $(1 - (-1))$. In order to use regret to find the Nash equilibrium, one obvious goal is to minimize regret. However, simply picking the action we regret most at a point in time is ineffective as our opponent can track our regrets and predict our actions. Instead, we use ***regret matching***, or creating a strategy where the probability of playing an action is proportional to the amount we regret not playing it. In order to turn our regrets into probabilities, take the sum of all the positive regrets (if the regrets are negative, we will never play those actions). Then, the probability of playing a given action is the amount we regret it divided by the total regret. However, even regret matching doesn't guarentee convergence to the Nash equilibrium. Instead, the ***average*** of the strategies generated by regret matching converges to the Nash equilibrium as the number of trials approaches infinity. The algorithm is as follows:

- Initialize regrets to 0
- For some number of iterations:
  - Compute a strategy with regret-matching
  - Add the strategy to the cumulative strategy sum
  - Select actions based off the strategy
  - Compute regrets
  - Add the regrets to the cumulative regrets
- Return the average strategy, the strategy sum divided by the number of iterations

It is not obvious how this algorithm generalizes to non normal-games, but it basically becomes a tree search similar to minimax. The resulting algorithm is called counterfactual regret minimization (CFR) and is extremely general and can be applied to non normal-games, like Poker and RBC.

However, CFR has the same problems as minimax, namely the tree size growing exponentially, making a full traversal impractical. One way to resolve this is to use Monte Carlo methods, in a similar way to MCTS [9], creating Monte Carlo CFR (MCCFR). Instead of searching through every possible action, one can randomly sample a subset of the possible actions. Another way to resolve this is to use a heuristic, but in this case the "heuristic" is known as an ***abstraction***. An abstraction is a simplification of the

original game, losing information as a trade off for time. Abstractions are commonly used in Poker, one example would be treating a bet of 90 chips as the same as a bet of 100 chips.

The same problems with heuristics apply to abstractions. CFR applied to Poker often relies on hand-crafted, domain-specific, and ad-hoc abstractions. Machine learning provides a solution however, by using deep neural networks (NNs) on CFR.

3.4 Deep CFR

The idea of Deep CFR is to use a NN to approximate the results CFR would have produced without abstractions [10]. In particular, they use MCCFR to sample states and train one NN (called the "value" network) on the regrets produced by MCCFR. They also train another network, called the "policy" network, which approximates the average strategy, the thing that converges to the Nash equilibrium.

3.5 Single Deep CFR

An optimization on Deep CFR is Single Deep CFR, or a variant which only uses a single value network, discarding the policy network [11]. They accomplish this by storing all iteration strategies, increasing the memory usage but reducing prediction error and removing the need for the policy network to predict the average strategy.

3.6 Neural Fictitious Self-Play (NFSP)

A competitor to Deep CFR is NFSP, which relies on the concept of fictitious self-play instead of regret minimization, and uses RL as deep learning [12]. Fictitious self-play is less well documented than CFR, however.

4 Conclusion

Game theory is becoming increasingly relevant as machine learning becomes more and more popular. Traditional game-playing algorithms like minimax, MCTS, and RL struggle on imperfect information games, thus algorithms like CFR and NFSP are the next step in game theory. Machine learning techniques can be applied to CFR in order to remove the human-designed elements common to most game-playing algorithms.

References

[1] NekoSpectrus. *File:No-Game-No-Life-anime-logo.svg*. 2014. URL: https://commons.wikimedia.org/wiki/File:No-Game-No-Life-anime-logo.svg.

[2] Neil Thistlethwaite. *Advanced GANs*. Feb. 2018. URL: https://tjmachinelearning.com/lectures/1718/advgan/.

[3] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.

[4] OpenAI et al. "Solving Rubik's Cube with a Robot Hand". In: *arXiv e-prints*, arXiv:1910.07113 (Oct. 2019), arXiv:1910.07113. arXiv: 1910.07113 [cs.LG].

[5] Chelsea Finn et al. "A Connection between Generative Adversarial Networks, Inverse Reinforcement Learning, and Energy-Based Models". In: *CoRR* abs/1611.03852 (2016). arXiv: 1611.03852. URL: http://arxiv.org/abs/1611.03852.

[6] Michael Buro. "Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello". In: *Games in AI Research*. 1997, pp. 77–96. URL: https://skatgame.net/mburo/ps/improve.pdf.

[7] Jared Markowitz, Ryan W. Gardner, and Ashley J. Llorens. "On the Complexity of Reconnaissance Blind Chess". In: *CoRR* abs/1811.03119 (2018). arXiv: 1811.03119. URL: http://arxiv.org/abs/1811.03119.

[8] Todd W. Neller and Marc Lanctot. "An Introduction to Counterfactual Regret Minimization". In: July 2013. URL: http://modelai.gettysburg.edu/2013/cfr/cfr.pdf.

[9] Richard G. Gibson et al. "Efficient Monte Carlo Counterfactual Regret Minimization in Games with Many Player Actions". In: *NIPS*. 2012, pp. 1889–1897. URL: http://papers.nips.cc/paper/4569-efficient-monte-carlo-counterfactual-regret-minimization-in-games-with-many-player-actions.

[10] Noam Brown et al. "Deep Counterfactual Regret Minimization". In: *CoRR* abs/1811.00164 (2018). arXiv: 1811.00164. URL: http://arxiv.org/abs/1811.00164.

[11] Eric Steinberger. "Single Deep Counterfactual Regret Minimization". In: *CoRR* abs/1901.07621 (2019). arXiv: 1901.07621. URL: http://arxiv.org/abs/1901.07621.

[12] Johannes Heinrich and David Silver. "Deep Reinforcement Learning from Self-Play in Imperfect-Information Games". In: *CoRR* abs/1603.01121 (2016). arXiv: 1603.01121. URL: http://arxiv.org/abs/1603.01121.