Using Counterfactual Regret Minimization to Find Nash Equilibrium

Timothy Li Eddie Tu

December 2019

1 Motivation

1.1 General Motivation

Our project focuses on the theory behind counterfactual regret minimization, and the development of regret minimization algorithms that will weakly solve zero-sum games where players have imperfect information. In the real world, there are many such games: some examples include Rock, Paper, Scissors, Avalon, Resistance, and Texas Hold'Em limit poker. We decided on this project in part because we play many of these games, and thus are interested in the optimal playing strategy.

We hope to show that our regret minimization algorithm will develop an "optimal" strategy by training against itself over many iterations, revising its model in each round according to a positive regret measurements.

1.2 Relation to the Course

We define optimal play for some player i as a strategy, given all of the other players' strategies, that maximizes the expected utility for that player i. In game theory terminology the state of every player playing the best strategy in response to the strategies of the other players as a Nash Equilibrium. This is to say that no players can improve their expected utility by employing any other strategy, and we will prove this statement using concepts discussed in the course. We will then find, for some games discussed in this course (or games similar to those in the course), exactly this equilibrium point between players as an application of counterfactual regret minimization in a game theory-based context.

2 Background

2.1 Rock, Paper, Scissors, and Simultaneous Games

Simultaneous move games can be defined as a game between a finite number players, where each player has a finite set of actions available to them. In a single instance of the game, each player selects from their set of available actions a single action without knowledge of the actions selected by the other players. Every agent does, however, know all of the available actions to every other agent as well as the utility functions of the other players.

First, we introduce the popular one-shot, or simultaneous move game: Rock, Paper, Scissors. In every game, each player has the option to play either rock, paper, or scissors. Then, at the same time, the two players reveal their chosen action to each other. Both player's utility function can be modeled as follows: If the same action is chosen by both players, the round ends in a tie and neither player gains or loses any utility. Otherwise, victory is granted in the following way: rock beats scissors, paper beats rock, and scissors beats paper. We imagine the player that chooses the winning action to gain some small constant amount of utility for a single game, and the player that loses to lose utility of the same magnitude, making it a zero-sum game between both players.

Rock, Paper, Scissors, is a simultaneous, zero-sum game. It is simultaneous because the actions chosen by the two players are revealed at the same time, and both players clearly understand the utility function of the opposing player, as well as their finite set of available actions. It is a zero-sum game because after every round, the game ends in either a tie or one player wins and the other loses.

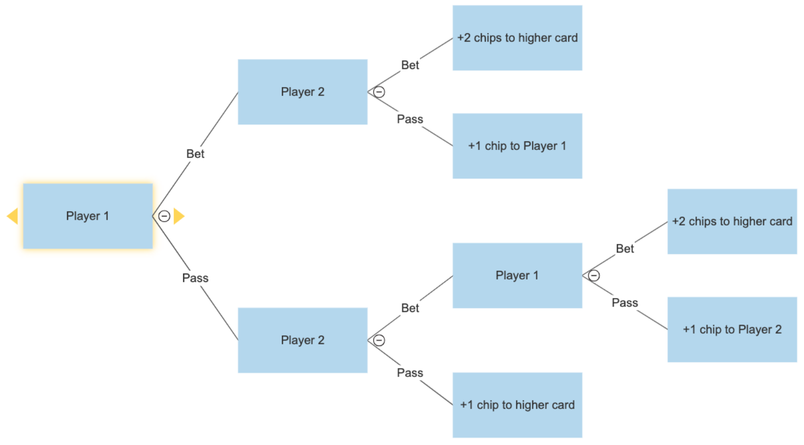
We can see that the Nash equilibrium of Rock, Paper, Scissors is to play rock, paper, and scissors with equal probability by using symmetry. Also, we know that if we play against an opponent who is playing a mixed strategy which is not the Nash equilibrium, we can exploit this player by playing a pure strategy. To prove this, assume that the opponent's mixed strategy is represented by the following distribution: $[\alpha, \beta, \phi]$, where $\alpha + \beta + \phi = 1$. Furthermore, we assume that we are not playing the Nash equilibrium $[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$. Thus, we know that one of the variables must be the largest of the three (or there can exist a tie between two variables). Without loss of generality, we assume that α is the largest variable in the distribution, which implies that our opponent plays rock the most frequently. Then we see that the strategy that maximizes our utility is to always play paper, which is a pure strategy Nash equilibrium. If there is a tie for the most frequent probability, we can again assume without loss of generality that $\alpha = \beta > \phi$. In this case, we would still always play paper, as we would win and tie with equal probability. Thus, for any mixed-strategy there exists a pure-strategy Nash equilibrium which is a best response.

2.2 Kuhn's Poker and Sequential Games

Unlike simultaneous move games, sequential games consist of a sequence of actions by players in turn-based play. At each turn, the decision made by the opponent in the previous turn will influence the action of the player in the current turn.

Kuhn's poker represents a sequential move game with two players defined as follows: at the start of the game, both players are dealt a single card from a set of three cards with values 1, 2, and 3. The value of a player's card is held as private information to the player before the end of the game. Initially, both players are required to bet "1" chip, or point of value into the prize pool (for the winner to collect upon conclusion of the game). From this starting state, player actions alternate between rounds, starting from player 1. During a turn, a player may choose to either 'pass', which will end the turn, or 'bet', where the player will add an additional chip into the prize pool. The game concludes if any of the three following conditions are met: if a player makes a 'pass' immediately after a 'bet', the opponent takes the chips in the pot. If two 'passes' are made successively, both players reveal their cards and the player with the higher value card wins the pot. Similarly if two 'bets' are made successively, the player with the higher value card wins the pot upon reveal. Kuhn Poker is a game of imperfect information as neither player knows the value of the opponent's card with certainty. Thus the action of a player during a given turn is influenced by the value of their own card as well as the decision made by their opponent in a previous turn.

We can represent the utility function of a player by analyzing each possible series of actions that leads to the conclusion of a game as a tree.



2.3 Counterfactual Regret Minimization

The regret minimization algorithm tracks regrets from past plays of a series of games and devises and optimal strategy by prioritizing towards future rounds actions that are proportional to the positive regret generated in the games' history. After a particular play of the game, a player's regret for not having played another action can be calculated as the difference in utility of having played their current action over that particular action.

This regret heuristic can be explained in a simple example of Rock, Paper, Scissors: if our player hypothetically chooses rock and the opponent chooses paper, the utility we experience from losing with rock is some negative constant, -1 in our simulations. Our regret from not having played paper can then be calculated as the utility we would have gained from matching our opponent's paper subtracted by our utility from playing rock which is $0 - (-1) = 1$ for a positive regret of 1. Our regret from not having played scissors is similarly calculated as the difference between the potential utility of winning with scissors (a positive constant 1) and the utility of our chosen action, $1 - (-1) = 2$.

The regret minimization algorithm will update its strategy based on these results to prioritize actions with higher regret. Assuming a completely neutral starting strategy, our next action will be chosen proportionally to the positive regrets we found in the first round. Thus in the next round the player will choose randomly between rock, paper, and scissors with a $\frac{0}{3}$, $\frac{1}{3}$, and $\frac{2}{3}$ probability.

With this process, the counterfactual regret minimization algorithm will minimize the expected regret of any strategy through self-training over some number of iterations. As we have shown above for the Rock, Paper, Scissors game, the average strategy profile devised through such a process converges to a Nash Equilibrium for zero-sum games.

3 Method

3.1 Rock, Paper, Scissors

Listing 1: Counterfactual Regret Algorithm Initialization

```
class RockPaperScissors():
    def __init__(self):
        self.NUM_ACTIONS = 3
        self.regretSum = [0] * self.NUM_ACTIONS
        self.strategy = [0] * self.NUM_ACTIONS
        self.strategySum = [0] * self.NUM_ACTIONS
```

In order to run our counterfactual regret minimization algorithm on the game Rock, Paper, Scissors, we defined a class which contained a few different vectors. The first vector, regretSum, keeps track of the total amount of regret accumulated throughout the iterations of the game. The strategy vector contains the current strategy that the algorithm is testing. Finally, strategySum keeps track of the sum of the strategies we have tried throughout the iterations.

Listing 2: Getting the Next Strategy

```
def getStrategy(self):
    normalizingSum = 0
    for a in range(self.NUM_ACTIONS):
        if self.regretSum[a] > 0:
            self.strategy[a] = self.regretSum[a]
        else:
            self.strategy[a] = 0
        normalizingSum += self.strategy[a]
    for a in range(self.NUM_ACTIONS):
        if normalizingSum > 0:
            self.strategy[a] /= normalizingSum
        else:
            self.strategy[a] = 1/self.NUM_ACTIONS
        self.strategySum[a] += self.strategy[a]
    return self.strategy
```

In this method, our function creates a variable called normalizingSum. Then, we select all the positive values in our regretSum vector and include them in our strategy. The other values in strategy (i.e. values where the regretSum is 0 or negative) receive a value of 0 in our strategy vector. The sum of the positive values in regretSum is the value of normalizingSum. Finally, we normalize all the values in strategy by normalizingSum, resulting in a vector where all the elements sum to be one. If normalizingSum is less than zero, which implies that there are no positive values in regretSum, then we set our strategy to be uniform across our three outcomes ($[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$). Finally, we return this strategy.

Listing 3: Counterfactual Regret Algorithm Initialization

```
def getAction(self, strategy):
    a = 0; cumulativeProb = 0
    r = random.random()
    while a < self.NUM_ACTIONS - 1:
        cumulativeProb += strategy[a]
        if r < cumulativeProb:
            break
        a += 1
    return a
```

This function simply takes as input a strategy and uses this strategy as a distribution from which to choose an action. Thus, if a strategy was $[\frac{1}{4}, \frac{1}{2}, \frac{1}{4}]$, getAction would return the output 0 with probability $\frac{1}{4}$, the output 1 with probability $\frac{1}{2}$, and the output 2 with probability $\frac{1}{4}$.

Listing 4: Training Function

```
def train(self, myAction, otherAction):
    actionUtility = [0] * self.NUM_ACTIONS
    actionUtility[otherAction] = 0
    if otherAction == 0:
        actionUtility[1] = 1
        actionUtility[2] = -1
    elif otherAction == 1:
        actionUtility[0] = -1
        actionUtility[2] = 1
    elif otherAction == 2:
        actionUtility[0] = 1
```

```
            actionUtility[1] = -1
        for a in range(self.NUM_ACTIONS):
            self.regretSum[a] += actionUtility[a] - actionUtility[myAction]
```

For our training function, we first take as input the action we played and the action the other opponent has played. Then, we create an actionUtility vector of zeros, which we will use to keep track of the regret we had playing our action compared to other actions. If we had played the same action as our opponent (i.e. if she had played rock and we also had played rock), then we tie and thus experience no regret. The remaining three cases look at what our regrets would have been playing other actions. For example, in the first if statement, we see that the other player has played rock. Thus, we regret the most not playing paper, as we would have gained utility of one. We regret the least not playing scissors, as then we would have lost one utility. We update the other actionUtilties similarly. Finally, after populating our actionUtilities vector with the correct values, we add this actionUtility to the regretSum vector. Thus, we have trained one additional step of our algorithm.

Listing 5: Returning the Average Strategy

```
    def getAverageStrategy(self):
        avgStrategy = [0] * self.NUM_ACTIONS
        normalizingSum = 0
        for a in range(self.NUM_ACTIONS):
            normalizingSum += self.strategySum[a]
        for a in range(self.NUM_ACTIONS):
            if normalizingSum > 0:
                avgStrategy[a] = self.strategySum[a]/normalizingSum
            else:
                avgStrategy[a] = 1/self.NUM_ACTIONS
        return avgStrategy
```

This function is what we will eventually use to return the Nash equilibrium that we calculate. As per the algorithm for counterfactual regret minimization, the strategy that eventually converges to a Nash equilibrium is calculated by normalizing the strategySum vector. Finally, the function returns this avgStrategy.

Listing 6: Training the CFR Algorithm Against a Mixed Strategy

```
a = RockPaperScissors()
for i in range(10000):
    aAction = a.getAction(a.getStrategy())
    bAction = a.getAction([0.4, 0.3, 0.3])
    a.train(aAction, bAction)
```

We used this to train the instance ”a” of our RockPaperScissors class to learn how to play against an opponent who played the mixed equilibrium [0.4, 0.3, 0.3]. We ran 10000 iterations, and in each iteration we got a’s current strategy and used that to generate a’s action. Note that a.getAction([0.4, 0.3, 0.3]) has no effect on any variables in a, and that we only use a’s method to get the action of our opponent who plays a mixed strategy. Finally, we train a on the two actions we have generated.

3.2 Kuhn’s Poker

Listing 7: Counterfactual Regret Applied to Kuhn’s Poker - Initialization

```
class Node():
    def __init__(self):
        self.infoSet = ''
        self.NUM_ACTIONS = 2
        self.regretSum = [0] * self.NUM_ACTIONS
        self.strategy = [0] * self.NUM_ACTIONS
        self.strategySum = [0] * self.NUM_ACTIONS
```

We initialize the same vectors as the previous CFR algorithm for Rock, Paper, Scissors: regretSum, strategySum, and strategy. In addition to these vectors, we also initialize an empty information set called infoset. An information set contains all the information available to the player, including the player's own card value and the actions taken by the opponent in previous turns. Note that the total number of actions that a player can make at any decision point is two: 'pass' or 'bet'.

Listing 8: Kuhn's Poker - Getting the Next Strategy

```
    def getStrategy(self, rweight):
        normalizingSum = 0
        for a in range(self.NUM_ACTIONS):
            if self.regretSum[a] > 0:
                self.strategy[a] = self.regretSum[a]
            else:
                self.strategy[a] = 0
            normalizingSum += self.strategy[a]
        for a in range(self.NUM_ACTIONS):
            if normalizingSum > 0:
                self.strategy[a] /= normalizingSum
            else:
                self.strategy[a] = 1/self.NUM_ACTIONS
            self.strategySum[a] += rweight * self.strategy[a]
        return self.strategy
```

To get the strategy for the new round, we initialize the same variable as our RPS algorithm, normalizingSum and use the sum total of regret for across each strategy as the proportional probability of play divided by the normalizing sum. The difference for the Kuhn's Poker implementation is the consideration of a realization weight (rweight) which is used to weight the choice of strategy based on its reach probability. This represents the likelihood that our simulated game should reach this game state at any point.

Listing 9: Kuhn's Poker - Returning the Average Strategy

```
    def getAverageStrategy(self):
        averageStrat = [0] * self.NUM_ACTIONS
        normalizingSum = 0
        for a in range(self.NUM_ACTIONS):
            normalizingSum += self.strategySum[a]
        for a in range(self.NUM_ACTIONS):
            if normalizingSum > 0:
                averageStrat[a] = self.strategySum[a] / normalizingSum
            else:
                averageStrat[a] = 1/self.NUM_ACTIONS
        return averageStrat
```

This method is also the same as its RPS equivalent. We aggregate our realization weighted strategy across every training iteration in our strategySum vector - across all strategies this becomes our normalizingSum which will act as sample space for our average strategy.

Listing 10: Kuhn's Poker - Initialization & Training

```
class KuhnTrainer():
    def __init__(self):
        self.nodeMap = dict()
        self.NUM_ACTIONS = 2

    def train(self, iters):
        cards = [1,2,3]
        utility = 0
        for i in range(iters):
            for j in reversed(range(1, len(cards))):
                rand = random.randrange(0, j+1)
                cards[j], cards[rand] = cards[rand], cards[j]
            utility += self.cfr(cards, "", 1, 1)
        print("Average game value: " + str(utility / iters))
        for nodeval in self.nodeMap.values():
            print(nodeval)
```

Every iteration of training will generate a new, random ordering of the cards - index 0 of the cards array represents the card value dealt to player 1 and index 1 represents the card value dealt to player 2. We will then call the `cfr` method which will update the strategy profile at each of the nodes in nodeMap (represented as a dictionary) and aggregate the utility generated by that particular instance of training to a total so we may report average game value upon conclusion of our training method.

Listing 11: Kuhn's Poker - CFR Application and Updating Strategy

```
    def cfr(self, cards, history, p0, p1):
        plays = len(history)
        player = plays % 2
        opponent = 1 - player
        if plays > 1:
            terminalP = history[plays-1] == 'p'
            doubleBet = history[plays-2:plays] == 'bb'
            isHigher = cards[player] > cards[opponent]
            if terminalP:
                if history == 'pp':
                    if isHigher:
                        return 1
                    else:
                        return -1
                else:
                    return 1
            elif doubleBet:
                if isHigher:
                    return 2
                else:
                    return -2
        infoSet = str(cards[player]) + history
        if infoSet not in self.nodeMap:
            node = Node()
            node.infoSet = infoSet
            self.nodeMap[infoSet] = node
        else:
            node = self.nodeMap[infoSet]
        if player == 0:
            strategy = node.getStrategy(p0)
        else:
            strategy = node.getStrategy(p1)
        utility = [0] * self.NUM_ACTIONS
        nodeUtil = 0
        for a in range(self.NUM_ACTIONS):
            if a == 0:
                nextHist = history + 'p'
            else:
                nextHist = history + 'b'
            if player == 0:
                utility[a] = -self.cfr(cards, nextHist, p0 * strategy[a], p1)
            else:
                utility[a] = -self.cfr(cards, nextHist, p0, p1 * strategy[a])
            nodeUtil += strategy[a] * utility[a]
        for a in range(self.NUM_ACTIONS):
            regret = utility[a] - nodeUtil
            if player == 0:
                node.regretSum[a] += p1 * regret
            else:
                node.regretSum[a] += p0 * regret
        return nodeUtil
```

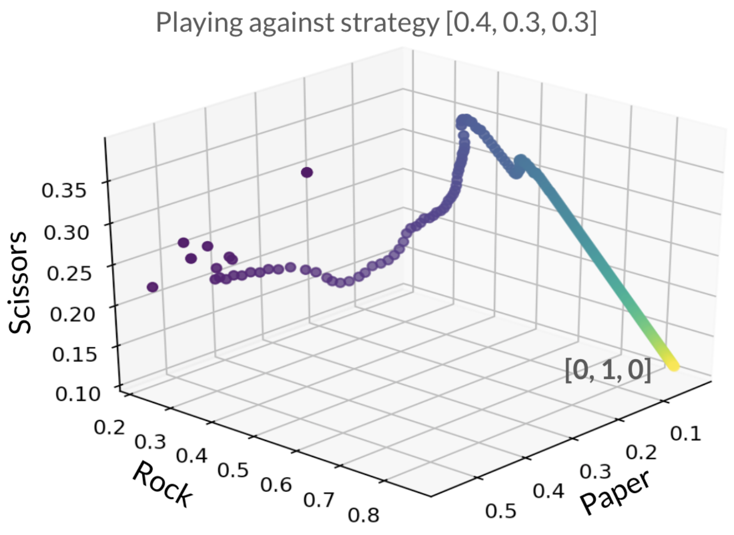
The base case for our recursive `cfr` method checks if the current call is on a history at a terminal state. The execution of the algorithm continues if it is not, otherwise returning according to the utility function we described in the Section 3.2 above. Each node strategy is determined using the regret matching/minimization method and is computed with the following process.

At every unique information set, create an initial node to represent a neutral strategy for our player's action, otherwise pulling from our nodeMap the trained strategy up to this point. We compute the node strategy using the realization weights, p_0, p_1 depending on which player the current turn belongs to. To determine the utility of the two actions in the current turn, we initialize a blank vector called utility and make recursive calls to the `cfr` method with updated realization weights. Each recursive call will return the expected utility of a particular action at this node, either 'pass' or 'bet'. We are able to then update our strategy profiles for each node with this utility vector by calculating for our player's regret for a given action. Again, it is important to note the significance of p_0, p_1, representing the likelihood of the opponent to play according to the current information set, which we use to update our total regretSum - also known as the counterfactual regret.

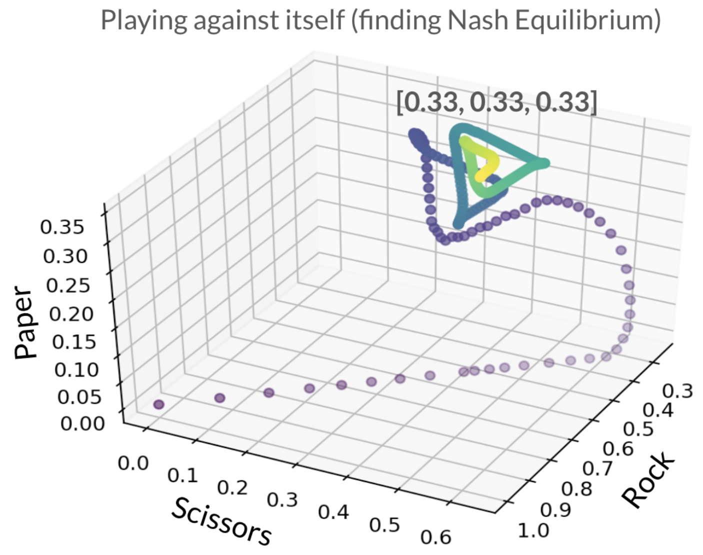
4 Results

4.1 Rock, Paper, Scissors

We first tested the counterfactual regret minimization algorithm against a mixed strategy [0.4, 0.3, 0.3], which can be interpreted in the context of Rock, Paper, Scissors, as an opponent who plays rock with probability 0.4 and plays paper and scissors each with probability 0.3. As shown in the introduction, there exists a pure Nash equilibrium to any mixed strategy in Rock, Paper, Scissors, and our algorithm correctly finds this pure strategy. The following figure depicts the average strategy as the number of iterations counts up, where the darker purple part of the gradient indicates the beginning of the iterations, and the lighter yellow part indicates the later iterations of training.

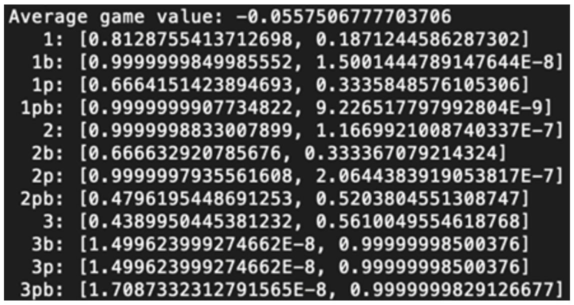


As we can see, at the beginning our algorithm behaves more erratically, yet as the iterations increase we see that we eventually converge to the point (0, 1, 0). After running 10000 iterations, we got the following values as our average strategy: [0.006730266723543586, 0.9922297332764565, 0.0010399999999999997]. Thus, we can say with sufficient certainty that our algorithm eventually converges to the pure strategy [0, 1, 0]. This is consistent with our expectations, as we expect that to play optimally against an opponent who prefers to play rock, we should always play paper as that guarantees that over time we will win more than we lose.



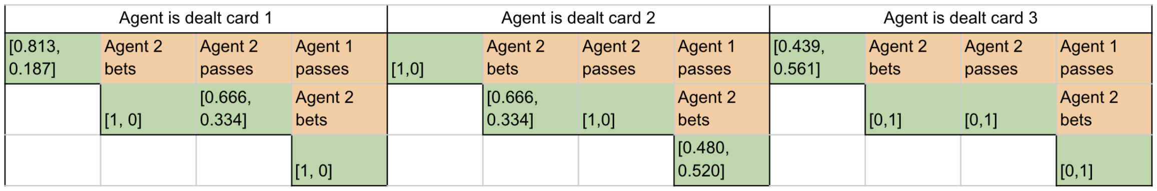
4.2 Kuhn's Poker

For Kuhn's Poker, we ran the counterfactual regret minimization algorithm against itself to find a Nash equilibrium. After running it several times, we realized that some values were always the same, while other values had some variation. First, we give as an example the Nash equilibrium of one of the times we ran our algorithm. The raw output is as follows:



```
Average game value: -0.0557506777703706
   1: [0.8128755413712698, 0.1871244586287302]
  1b: [0.9999999849985552, 1.5001444789147644E-8]
  1p: [0.6664151423894693, 0.3335848576105306]
 1pb: [0.9999999907734822, 9.226517797992804E-9]
   2: [0.9999998833007899, 1.1669921008740337E-7]
  2b: [0.666632920785676, 0.333367079214324]
  2p: [0.9999997935561608, 2.0644383919053817E-7]
 2pb: [0.4796195448691253, 0.5203804551308747]
   3: [0.4389950445381232, 0.5610049554618768]
  3b: [1.499623999274662E-8, 0.99999998500376]
  3p: [1.499623999274662E-8, 0.99999998500376]
 3pb: [1.7087332312791565E-8, 0.9999999829126677]
```

We round the results to three significant digits and use a format which is easier to understand:



Agent is dealt card 1				Agent is dealt card 2				Agent is dealt card 3			
[0.813, 0.187]	Agent 2 bets	Agent 2 passes	Agent 1 passes	[1,0]	Agent 2 bets	Agent 2 passes	Agent 1 passes	[0.439, 0.561]	Agent 2 bets	Agent 2 passes	Agent 1 passes
	[1, 0]	[0.666, 0.334]	Agent 2 bets		[0.666, 0.334]	[1,0]	Agent 2 bets		[0,1]	[0,1]	Agent 2 bets
			[1, 0]				[0.480, 0.520]				[0,1]

To explain the graph, we represent the history of the game with the orange color, and the green colored box represents the current round of the game. For example, the last column of the section "Agent is dealt card 1" represents a state of the game where in the first round, player 1 passed. Then in the second round, player 2 bets. Thus, this was the history of the game, and now player one must choose whether to bet or whether to pass. In this case, the strategy that was generated from running our counterfactual regret minimization algorithm told us to pass with a probability of 1. (In reality, the strategy that our algorithm returned for this history was a number very close to 1: 0.9997873388252149, and thus we approximate it to be 1).

Kuhn demonstrated that there are an infinite number of mixed-strategy Nash equilibrium which can be formulated as follows: for all $\alpha \in [0, \frac{1}{3}]$, a mixed-strategy can be formed by betting with probability α in round one when given card 1 (this corresponds to the action represented by the first column). When the player has card 3 in round one, she should bet with a probability of 3α. If the player has card 2 and the following history has occurred: player 1 passes and player 2 bets, then the player should bet with $\alpha + \frac{1}{3}$. The strategies for other states of the game are fixed in the Nash equilibrium, and are represented by the other columns in our table (i.e. all columns excluding column 1, 8, and 9 are the same for all mixed-strategy Nash equilibrium, and are unaffected by the choice of α).

From the results of our training, we see that the strategy outputted by our algorithm does follow the requirements to be a mixed-strategy Nash equilibrium. In the particular example shown by the graph, we see that the value of α chosen was 0.187. Doing a quick calculation, we see that $3\alpha = .561$ and $\alpha + \frac{1}{3} = .520$, which does correspond with the values we see from our mixed strategy. Thus, we show that our algorithm for calculating a mixed-strategy Nash equilibrium for Kuhn's poker does function as intended.

Finally, we explain the value of average game value. The decimal number -0.05575 we calculated is very close to $-\frac{1}{18}$. This also follows with Kuhn's proof that on average, the value of playing a game is $-\frac{1}{18}$ to the first player, and because Kuhn's poker is a zero-sum game, the average value of playing a game to the player that goes second is $\frac{1}{18}$.

5 Discussion

In many ways, this project is a proof of concept, or an application of a theory that we found particularly interesting in a game theory context. The most interesting finding that we confirmed was the guaranteed success of our algorithm in finding an optimal Nash Equilibrium solution for zero-sum games regardless of the agent's starting strategy, or the strategy of the opponent that we faced. It's important to note that our self-training method is motivated a weakness in CFR to opponents that understand the regret matching approach, and could devise an exact strategy to counter our approach. To avoid such a constant exploitation of our action probabilities, we train our model through self-play to develop an optimal strategy before encountering other opponents.

We were surprised that such an algorithm could be applied to games with an extreme number of game states, such as Avalon. The complexity of games that we applied this process to already seemed like non-trivial problems for even human players to solve, despite having significantly smaller sample spaces.

6 Future Directions

In our project, we chose to focus on the application of counterfactual regret minimization to zero-sum games. We hope to extend the application of our algorithm to games with larger sample spaces, and the next logical step would be to apply this method to a popular game like a restricted, zero-sum version of poker: Heads-up Limit Texas Hold 'Em.

We would also like to analyze what strategy counterfactual regret would recommend for games that are not zero-sum or constant-sum to determine if they are Nash Equilibrium, or some more general form of correlated equilibrium. Future applications of this algorithm also extend to a wider range of games where there are more players. We have seen counterfactual regret minimization applied to the popular sequential game Avalon, which has more than two players participating. However, additional steps have to be taken in order to limit the sample space, as otherwise it would not be feasible to compute, which is why we are also interested in looking at ways to reduce the sample space while still obtaining the correct strategy.

References

[1] Noam Brown et al., "Deep Counterfactual Regret Minimization," ArXiv:1811.00164 [Cs], May 22, 2019, http://arxiv.org/abs/1811.00164.

[2] Kuhn, Harold W. Lectures on the Theory of Games (AM-37). STU - Student edition ed., Princeton University Press, 2003. JSTOR, www.jstor.org/stable/j.ctt7t5mk.

[3] Jack Serrino et al., "Finding Friend and Foe in Multi-Agent Games," ArXiv:1906.02330 [Cs, Stat], June 5, 2019, http://arxiv.org/abs/1906.02330.

[4] Oskari Tammelin et al., "Solving Heads-up Limit Texas Hold'em," n.d., 8.