Game Theory Research Report

Jessica Opsahl-Ong

July 2022

Fixed Strategies

The first thing I did to introduce game theory concepts to this research was encode playable common iterative strategies to the prisoners dilemma. The strategies are as follows:

1. **Cooperator**: Cooperates on every move
2. **Defector**: Defects on every move
3. **Random**: Makes a random move with 50/50 chance
4. **Tit for Tat**: Copies the opponent's last move
5. **Grim Trigger**: Cooperates, until the opponent defects, and thereafter always defects.
6. **Tit for Two Tat**: Cooperates on the first two moves, and defects only when the opponent defects two times.
7. **Two Tit for Tat**: Same as Tit for Tat except that it defects twice when the opponent defects.
8. **Gradual**: Cooperates on the first move, and cooperates as long as the opponent cooperates. After the first defection of the other player, it defects one time and cooperates two times. After the nth defection it reacts with n consecutive defections and then cooperates twice.
9. **Soft Majority**: Cooperates on the first move, and cooperates as long as the number of times the opponent has cooperated is greater than or equal to the number of times it has defected, else it defects.
10. **Hard Majority**: Defects on the first move, and defects if the number of defections of the opponent is greater than or equal to the number of times it has cooperated, else cooperates.
11. **Remorseful Prober**: Like Tit for Tat, but it tries to break the series of mutual defections after defecting 5 times.
12. **Soft Grudger**: Like GRIM except that the opponent is punished with D,D,D,D,C,C.
13. **Prober**: Starts with D,C,C and then defects if the opponent has cooperated in the second and third move; otherwise, it plays TFT.

Furthermore, each strategy can be "noisy" by utilizing the "unexpected prob" parameter, which means that the agent plays unexpectedly "unexpected prob" percent of the time.
Code: https://github.com/jessica-ops/game-theory/blob/main/strategies/strategy.py

Strategy Likelihood

With these strategies, I implemented a way to determine which strategy is being used in a game where one of the players is using one of the 13 strategies above. This is built on Bayesian decision processes and updates the likelihood of each strategy after each round of the game. The process begins with a flat prior and the posterior is calculated each round by the following algorithm:

Algorithm 1: Calculating Strategy Likelihood

Input : list of all moves the agent has played thus far, *agentActs*

Input : list of all moves the opponent has played thus far, *oppActs*

Output: Probability distribution over all strategies indicating likelihood of each strategy given observed data

1 **for** each s in strategies **do**

2 compute P(*agentActs*, *oppActs* | strategies = s)

3 P(*agentActs*, *oppActs*) = $\sum_{\text{each } s \text{ in strategies}} P(agentActs, oppActs|\text{strategies} = s) * P(\text{strategies})$

4 P(strategies | *agentActs*, *oppActs*) = [P(*agentActs*, *oppActs* | strategies) * P(strategies)]/P(*agentActs*, *oppActs*)

5 **return** *P(strategies | agentActs, oppActs)*

code: https://github.com/jessica-ops/game-theory/blob/main/probabilistic_playing/strategy_inference.py

Computing likelihood for individual strategies

Because each strategy has distinct behavior, there is a different function for each strategy for the computation in line 2. Depending on the class of strategy, there are generally different ways of computing the likelihood. However, there are variations within the groups as well.

Fixed Strategies - Cooperator, Defector

These always have the same expected action. So for a new round of the game, if the agent has played its respective expected action, the likelihood = previous likelihood * (1 - unexpected probability)

Responsive Strategies - Tit for Tat, Tit for Two Tat, Two Tit For Tat, Remorseful Prober, Soft Grudger

All of these strategies rely on a fixed memory of the previous game to play. The most standard is Tit for Tat, whose expected move is always the opponent's previous move. Meaning given a new round r of the game, if $agentAct_r = oppAct_{r-1}$, then likelihood = previous likelihood * (1 - unexpected probability). This operates similarly for Tit for Two Tats and Two Tits for Tat, except that there is a memory of two instead of one previous actions required. Similarly, Remorseful Prober requires a memory of five.
Soft Grudger technically is a responsive strategy; however, it can be difficult to discern where in the punishing sequence the agent is by just looking back. So in actual practice, I use a similar strategy to Gradual to compute the likelihood for Soft Grudger.

Memory Strategies - Grim Trigger, Gradual, Soft Majority, Hard Majority, Prober

These are strategies which must retain some memory of the whole game thus far in order to play their ensuring action. For some of the strategies, the information which they "remember" is relatively straightforward, making their likelihood straight forward to compute.
For Grim Trigger, if the opponent has ever played a 1, Grim Trigger acts as a Defector. Otherwise, it acts as a Cooperator. This means that the likelihood can be computed in a similar way to the Fixed

Strategies.
Prober also adopts a new strategy depending on its history. Depending on the outcome of the first three rounds of the game, Prober will either become a Defector or play Tit for Tat. This means that the likelihood can be computed as either one of those strategies. For Soft Majority and Hard Majority, their action depends on the ratio of opponent defects to total plays. This can be calculated using the action history and easily determine if the expected move is 1 or 0.
Gradual is more complicated in that the number of times that the opponent defects impacts the following sequence that Graudal will play. Becuase of this, I found that it works best to re-calculate the likelihood from the beginning for each round, instead of replying on the previously calculated likelihood. By doing this, I can keep track of the number of defects of the opponent at each round in the game and subsequently keep track of where Gradual should be in its sequence. For each expected and unexpected move, the likelihood is updated accordingly.

Random

Because Random will play a random move each round, the likelihood of any move is always 0.5. Meaning the overall likelihood = 0.5 * previous likelihood.

Determining the Unexpected Probability

Despite being used in the computation of the likelihood, the unexpected probability is also an unknown and must be estimated while trying to determine which strategy the agent is playing. I currently have a rudimentary way of updating the unexpected probability. The functions which calculate likelihood for each strategy also take in as an input and return as an output the number of unexpected moves the agent has played thus far. The unexpected probability = number of unexpected moves * epsilon. The epsilon value is a parameter of the Likelihood class. Currently, I typically set the value to 0.01. This means that when the number of rounds = 100, unexpected probability = number of unexpected moves / total moves. And before then, it isn't too hasty with increasing the value of the unexpected probability.

code: https://github.com/jessica-ops/game-theory/blob/main/strategies/likelihood.py

Reinforcement Learning Agents

In order to create an agent which is responsive to different strategies it encounters, I have been using RL to train policies which play the prisoner's dilemma. I have been using the Poximal Policy Optimization (PPO) algorithm from Stable Baselines 3 and made a custom environment in line with the OpenAI Gym framework.
The environment is set up such that one episode is a full game consisting of a set number of rounds (typically I have trained it on 100 rounds) and one step is a single round of the game. The agent is rewarded based off the payout matrix. The observations which are given to the policy to select actions include a history of the previous 10 rounds of actions (both the agent's actions and opponent's). I have experimented with including other observations - including average number of opponent and agent defections as well as the round number.
When starting out, in order to get a feel for the responsiveness of an RL trained agent, I trained against all the strategies. A random opponent strategy was chosen for each episode and the agent would play against it. I also trained agents against the policies which I had saved.
To examine the responsiveness and general strategies of these agents, I played against them and created a "tournament" in which each agent would play games against every other agent and strategy.
Currently, I am training policies against a single strategy. Because these policies only see a single oppo-

nent when training, it's belief will be that any opponent it faces will be using that strategy. This will allow us to test IRC on a game theory set-up.

Drawbacks

However, RL poses an issue for the future steps of IRC because the policy is not necessarily learning which strategy its playing against and incorporating that knowledge into its belief about the world. In some cases, the policies I trained against a strategy did not learn the optimal way to play against its respective strategy. For instance, the policy trained against Grim Trigger learned to defect against opponent, not cooperate. This shows a lack of understanding of the strategy it is playing against, which would make it difficult to understand that policies belief state.

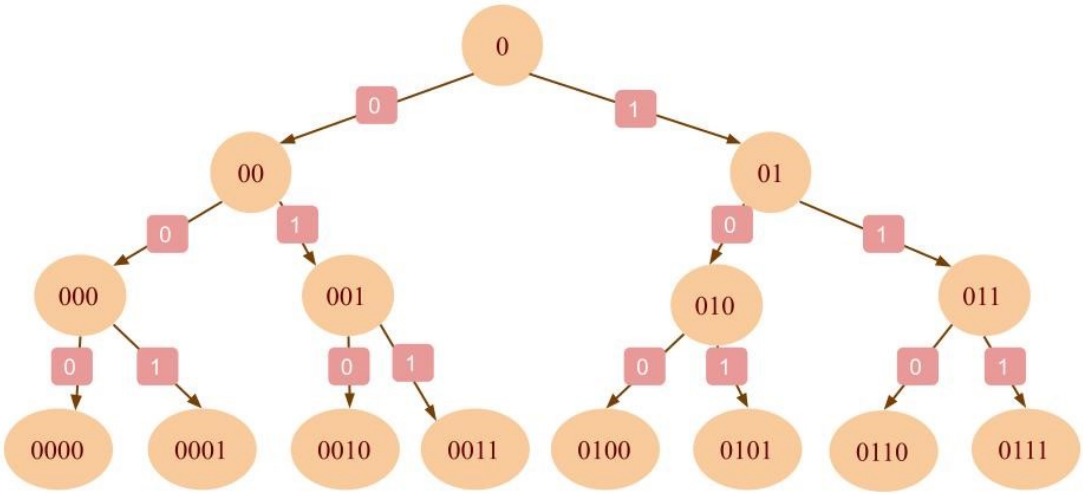
Value Iteration Learning

Formulation of Problem

Because of the drawbacks posed by RL, we reformulated the problem as a tree representing potential states for each round of the game and the transitions between states. The world can be represented as:

- states as the current history of the opponent's actions
- transitions between states as a single action of the agent

An example of this representation for playing against a Tit for Tat strategy:



Unexpected actions can be accounted for by adding transition probabilities such that each expected state transition (as depicted on the diagram) occurs with probability $1 - \epsilon$ and each unexpected state transition occurs with probability ϵ.
Something to note is that at a given state, it is assumed that the strategies move is known. However, since the actual game occurs with both players acting simultaneously, the state would not be fully known. In instances where the strategy plays no unexpected moves, this does not pose an issue. However, there may be challenges when we account for strategies playing unexpectedly.

Code for building transition tree: https://github.com/jessica-ops/game-theory/blob/main/value_iteration/learn.py (at def build_transition_probs)

Learning

To learn the best strategy against the strategies (formulated as above) I am using the below value iteration formula:

Algorithm 2: Value Iteration Algorithm

Input : valid states as a tuple of strategy history and agent history, *states*. An exxample of a valid state for a tit for tat strategy is $(0100, 100)$, where the first element is the tit for tat strategy and the second element is the agents actions thus far.

Input : valid terminal states as a tuple of strategy history and agent history, *terminalStates*

Input : transition probabilities from state s to s' given action a, $P_a(s'|s)$

Input : payoff matrix for game, R. $r_1, r_2 = R[a_1][a_2]$ indicates the reward for player one paying a_1 and player two playing a_2, where their rewards are r_1 and r_2 respectively.

Input : threshold at which to stop learning, θ

Input : discount factor, γ

Output: q table indicating value of action at each state, Q

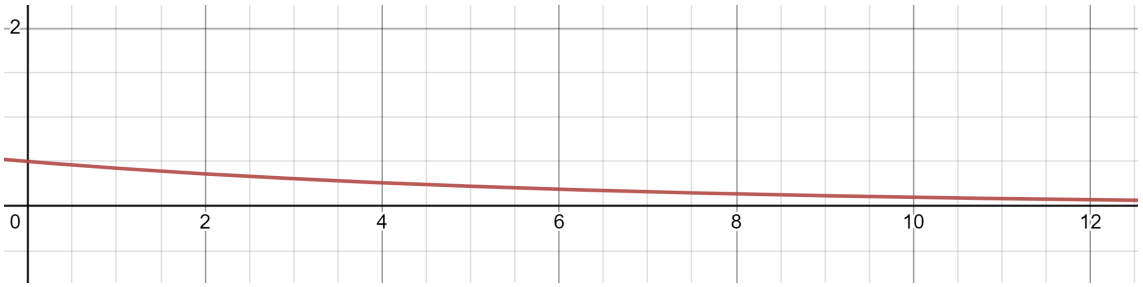
$V_s \leftarrow 0$ for all s in *states*
$V_s \leftarrow \max(R_{0,s[-1],0}, R_{1,s[-1],0})$ for all s in *terminalStates*
$Q_{s,a} \leftarrow R_{a,s[-1],0}$ for all s in *terminalStates* and a in $\{0, 1\}$
$\Delta \leftarrow -1$
while $\Delta \neq -1$ or $\Delta > \theta$ **do**
    for s in *states* **do**
        for a *in* $\{0, 1\}$ **do**
            $Q_{s,a} \leftarrow 0$ **for** s' in $\{0, 1\}$ **do**
                $stratState \leftarrow s_0 + s'$
                $actions \leftarrow s_1 + a$
                $Q_{s,a}$ += $R_{a,s_0[-1],0} + P_a((stratState, action) \mid s) \times V_{(stratState,action)}$
        $\Delta \leftarrow \max(delta, |\max_{\text{for } a' \text{ in } \{0,1\}}(Q_{s,a}) - V_s|)$
        $V_s \leftarrow \max_{\text{for } a' \text{ in } \{0,1\}}(Q_{s,a})$
return Q

code with algorithm: https://github.com/jessica-ops/game-theory/blob/main/value_iteration/learn.py (at def learn)

Non-determinism

To create more overlap between the policies, I implemented non-determinism for both the strategies and the policies. For the strategies, I expanded the world diagram to include the potential for unexpected moves with a certain ϵ value. By including this within the transition probabilities and states, the policies can optimize their play against strategies with any value of ϵ.
code: https://github.com/jessica-ops/game-theory/blob/main/value_iteration/learn.py (at def include unexpected moves)

To include non-determinism within the policies themselves, I still wanted the policies to defer to the Q values they have learned. Therefore, the policies are more likely to choose a sub-optimal move if the q value difference between the optimal and sub-optimal move is smaller. Specifically, the choice is dictated by $y = 0.5 * (0.85)^x$, where x is the difference between the optimal and sub-optimal move and y is the probability that the agent will choose the sub-optimal move.



Algorithm 3: Non-deterministic optimized policy action

Input : q values for each state of (opponent actions, agent actions), Q

Input : function for computing action distribution of opponent's next move, **stratNextMove**. Inputs: strategy history, opponent history, unexpected move parameter

Input : unexpected move parameter for opponent strategy, ϵ

Input : strategy history, $strat$

Input : agent history, $agent$

Output: selected move for the round

$V \leftarrow [[Q_{(strat+0,agent),0}, Q_{(strat+1,agent),0}], [Q_{(strat+0,agent),1}, Q_{(strat+1,agent),1}]]$

$stratNext \leftarrow$ **stratNextMove**$(strat, agent, \epsilon)$

multiply each element in column 0 of V by $stratNext_0$

multiply each element in column 1 of V by $stratNext_1$

$actionValue \leftarrow [\sum_{c=\{0,1\}} V_{0,c}, \sum_{c=\{0,1\}} V_{1,c}]$

$action \leftarrow \text{argmax}(actionValue)$

$p \leftarrow$ random from $[0, 1)$

$diff \leftarrow |actionValue_0 - actionValue_1|$

$threshold \leftarrow 0.50 * (0.85)^{diff}$

if $p \leq threshold$ **then**

  $action \leftarrow (action + 1) \bmod 2$

return $action$

code: https://github.com/jessica-ops/game-theory/blob/main/value_iteration/learn.py (at def play)

Observations

In order for an observer to make a decision about what strategy the policy was optimized against, it must have some known information about the strategy behavior's or the policy's behaviors. I decided that its realm of knowledge should be based on the policy's behavior towards the strategy it was trained against. To implement this, I run a certain number of games of the non-deterministic policy (actions described above) against the strategy it was optimized against and save the probability of the agent choosing a certain action given the history of the game.

There are two ways to describe the history of the game. The first way I tried was by using the full history of (strategy action history, policy action history). Because this is how the states of the Q table are encoded, this is accurate to how the policy makes it decisions. However, this makes the states very specific and less likely to experience certain combinations of strategy actions and policy actions. Therefore, when applying the policy to strategies it has not been optimized against, many states will not be included in prior observations.

Because of this limitation, I created a compressed version of the observed action probabilities where the state is only the strategies actions. This provides more information when the policy plays outside of its world; however, it is less accurate to the way a choice is made by the policy.

Using the observed behaviors, the "observer" will know the probability that a policy optimized against

a certain strategy chooses a certain action given the history of the game. This will allow the observer to determine which strategy the policy was trained against by finding the policy which has the highest probability of playing the actions observed.
code: https://github.com/jessica-ops/game-theory/blob/main/value_iteration/observe_policies.py