Iterated Prisoners Dilemma with Reinforcement Learning

Keven (Kedao) Wang
Department of Computer Science
Stanford University
kvw@stanford.edu

Abstract

This project uses recurrent neural network based reinforcement learning to play Iterated Prisoner's Dilemma. Multiple experiments are carried out with varying strategy compositions: RL-agent vs. stationary strategies, RL-agent vs. RL-agent, more than two RL-agents, and a mix of strategies. Cooperative behavior emerged in some RL-agent vs. RL-agent scenario. Non-uniform strategies evolved in some tournaments where number of RL-agents greater than two. Q-table and markov matrix are used to analyze agents' learned strategies. A high variance is observed across trials when using Q-learning without experience replay.

1 Intro

Reinforcement learning effectively maximizes an agents expected cumulative discounted reward. In a conventional reinforcement setting, the environment is fixed. What if there is a game among multiple agents, and the agents' actions change the environment? In particular, in a prisoner's dilemma game, an agent's reward depends on all agents actions. I hope to explore this problem via iterated games of prisoners dilemma. i.e. prisoner's dilemma repeated multiple times. For a single game of prisoners dilemma, the nash equilibrium is where both agents defect, which is not a globally-optimal result. The questions this project tries to answer are: Can an RL agent learn to play:

- vs. a deterministic strategy?
- vs. another RL agent?
- among multiple RL agents?
- among mixed strategies?

Further, are changes in hyper-parameters needed in order for different strategies to occur? How important is the environment in terms of players strategies? How do strategies evolve with respect to the distribution of strategies in the environment? Most of these questions have been answered by tournament with hand-engineered strategies. The goal of this project is to use reinforcement learning to train neural network agents.

2 Related Work

Iterated Prisoner's Dilemma is a widely studied topic across disciplines. This project gets inspiration from the 1996 paper by Sandholm[1], where an Elman recurrent neural network was used learn the Q values. Essentially a single previous step is taken as input to the RNN. In comparison, this project uses the newly developed LSTM cells for RNN, with longer time steps and up to two layers. This project also explores the tournament settings, where more than two players are involved.

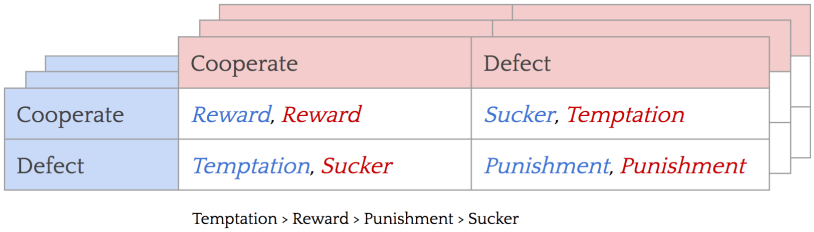

Figure 1: Iterated Prisoner's Dilemma consist of multiple single games of prisoner's dilemma, repeated across time.

	Cooperate	Defect
Cooperate	3, 3	0, 5
Defect	5, 0	1, 1

Figure 2: A typical prisoner's dilemma payoff matrix

3 Approach

3.1 Game Set up

A single prisoner's dilemma game consist of two players. Each player choose one of two actions: cooperate (C) or defect (D). A payoff matrix specifies the reward given the action pairs: T (Temptation), R (Reward), P (Punishment), S (Sucker). For a valid prisoner's dilemma game: $T > R > P > S$. The most frequently used reward values are $T = 5, R = 3, P = 1, S = 0$. In this paper, unless otherwise stated, we will use these default reward values.

There are two types of games played:

- Finite-episode game: the game ends after n number of episodes. The cumulative discounted reward is defined as $\sum_{t=1}^{n} r_t \gamma^t$, where r_t is reward of single prisoner's dilemma game at episode t, and γ is discount rate of reward. It can be proven through backward induction that the nash equilibrium for a finite-episode iterated prisoner's dilemma, the nash equilibrium is to always defect. A quick proof is as follows: at the last episode, both agent should defect. Given this information, both agent should defect in the previous episode. So on and so forth. I want to use the finite-episode as a sanity check, to prove that our agent is learning the value properly.
- Infinite-episode game: the game never ends. The cumulative discounted reward is defined as $\sum_{t=1}^{\infty} r_t \gamma^t$. In order to provide teaching signal to the RL agent, at each episode I compute the loss for the last n episodes, and use that as the teaching signal. The loss is defined below. The cumulative discounted reward from the last n episodes is computed, and used as a metric for an experimenter to judge an agent's performance against others. This score is evaluated as $\sum_{t=(i-n)}^{i} r_t \gamma^t$, where i is the current episode being played.

3.2 Markov Matrix

In prisoner's dilemma, a markov matrix specifies the probability of an action given a state. Here the possible actions are to defect or cooperate. The state is the pair of previous actions of self and oponent: `(prev_own_action, prev_opponent_action)`. Therefore we can define one-step markov matrix as four probabilities: $P(D'|CC), P(D'|CD), P(D'|DC), P(D'|DD)$. For example, $P(D'|CD)$ represent the probability of an agent defecting, given that in the previous

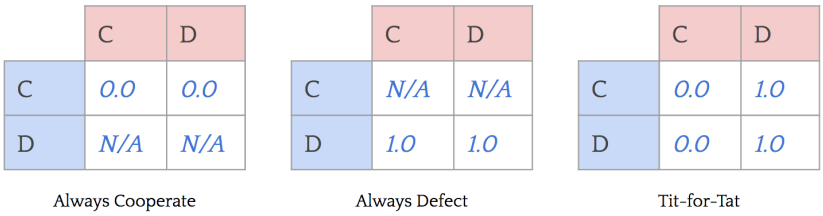

Figure 3: Markov matrix of different strategies: always cooperate, always defect, tit-for-tat

episode the agent himself cooperated while the opponent defected. Since there are only two actions, knowing the probability of defecting gives us the probability of cooperating: $P(C) = 1 - P(D)$. In Figure3 we provide some common markov matrices of stationary prisoner's dilemma strategies.

There has been research showing that only one single previous state is needed, in order to define any prisoner's dilemma strategy[2]. This suggests that the one-step markov matrix offers sufficient insight into an agent's behavior. In the experiments below, I use the markov matrix as a tool to analyze an agent's behavior.

3.3 Reinforcement Learning Agents

Our RL agents are not envious, meaning an agent only cares about maximizing his own score, not achieving a higher score than his opponent. This is achieved by optimizing an agent's expected cumulative discounted reward. This is a simplifying assumption.

I also allow our RL agents to have memory, meaning an agent can use previous actions of opponent and self in order to inform his next action. This is achieved by using a recurrent neural network that takes previous n episodes of history as input. n is a hyper-parameter that can be tuned.

The agent learns value function via Q-learning, a reinforcement learning technique that iteratively updates expected cumulative discounted reward Q given a state s, and a future action a.

$$Q_{new}(s_t, a_t) = (1 - \alpha)Q_{old}(s_t, a_t) + \alpha(r_{t+1} + \gamma \cdot \max_a Q_{old}(s_{t+1}, a)) \tag{1}$$

Where α is the learning rate, γ is the discount, r_{t+1} is the reward received after taking action a_t in the current episode.

3.4 Network Architecture

In our game, the state for a single episode is the pair of previous actions of self and oponent: `(prev_own_action, prev_opponent_action)`. At inference time, the RNN takes a list of action pairs up to n previous episodes as inputs in order to infer the next action. At training time, after every n episodes, the RNN back-propagate the loss across last n episodes through the network to update weights. The loss is defined as:

$$loss = \frac{1}{n}\sum_{i=1}^{n} ||Q - Q_{pred}||^2 \tag{2}$$

Where Q_{pred} is the Q-table iteratively learned by the network, Q the label is the Q-table from n episides before. In other words, we have a moving target of label Q, which is learned over time. Counter-intuitively, this technique works well to make Q-table converge after extended period of training. To approach the Q-learning task, I used one to two LSTM layers.

4 Experiments & Results

I performed a number of experiments varying the hyper-parameters. Notably, there is significant variance across trials while using the exact same hyper-parameters. I was suggested that by using experience-replay, one can introduce independency between the episodes played, and the variance

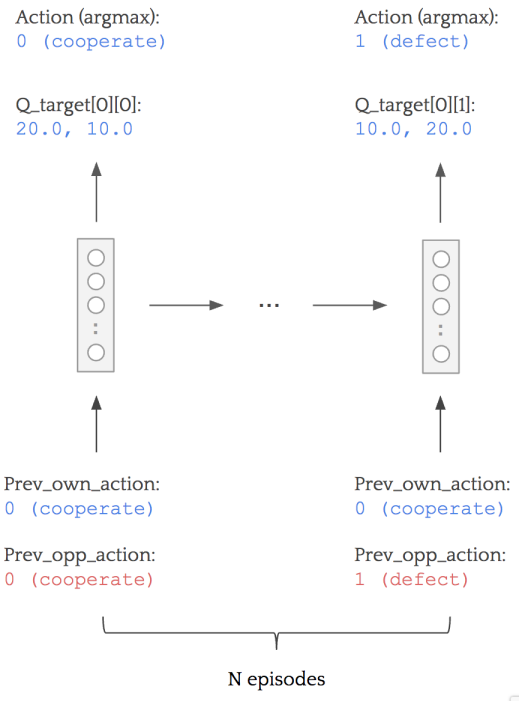

Figure 4: Q-learning agent's network architecture

could be reduced. I ran out of time to explore this option. In the following experiments, where run time is reasonable, I run the game multiple times and report the output distribution.

4.1 Tournament Set up

A game consist of multiple single prisoner's dilemma games. We call each individual game an episode. A game can have a finite or infinite number of episodes. In the case of infinite-episode game, I break up the episodes into rounds, for human evaluation purposes. Each round consist of n number of consequtive episodes. At the end of each round I evaluate cumulative discounted score for the round, and use that to evaluate the performance across agents. In the case of two agents, a round consist of a single sub-game: agent 1 vs. agent 2. In the case of three agents, a round consist of three sub-game: agent 1 vs. agent 2, agent 2 vs. agent 3, agent 1 vs. agent 3, aka the pairwise combination among all agents. The Q-table for each agent is persisted across rounds.

4.2 Hyper-parameters

A number of hyper-parameters affect the performance of the network. In particular, the following parameters are used, unless otherwise stated:

learning rate: single layer: `0.01`, two layer: `0.001`
Reward discount γ: `0.95`
Hidden state size: `10`
Reward structure: `T = 5, R = 3, P = 1, S = 0`
Epsilon ϵ: `0.2`
Epsilon decay rate: `0.9999`
Number of episodes per round n: `10`
Strategy types: `Q-learning single layer, Q-learning two layer, Tit-for-Tat, Always Cooperate, Always Defect`

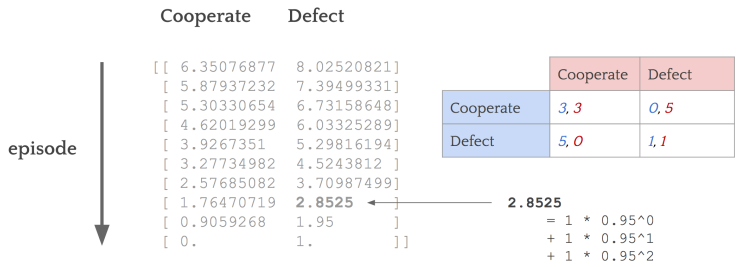

Figure 5: In a finite-episode game, Q-learning agent successfully learns cumulative discounted reward

4.3 Finite Episodes

As a sanity check, I set up a finite-episode game of 10 episodes. Here each game has exactly one round. The future reward after n episode is zero. The game is repeated multiple times. I ran a one layer Q-learning agent against an always defecting agent. The RL agent learns to defect each episode. Table 5 shows action value vs. episode index. The agent is able to learn a mathematically accurate representation of the action value for each episode.

Notably, the earlier episode's Q values takes longer time to converge as compared to later episodes. This is because the earlier episodes requires adequate information from later episodes, which takes time to propagate. This problem worsens with increased number of episodes per round n.

4.4 Infinite Episodes

After verifying that our RL agent is learning, I apply it to the more interesting case of infinite-episode games. Roughly 400 experiments are run, in order to tune the hyper-parameters and collect interesting results.

The agent is played against Tit-for-Tat. Tit-for-Tat mirrors opponent's action from last episode, and is shown to perform well in an evolutionary setting[3]. The agent converged to multiple distinct strategies across trials. The following four strategies emerged for a given round, where 0 means cooperate, 1 means defect. For a given trial, the agent would repeatedly play one of the following action sequences toward the end of the game:

[0 0 0 0 0 0 0 0 0 0]: RL agent learns to always cooperate

[0 1 0 1 0 1 0 1 0 1]: RL agent learns to alternative between cooperate and defect

[1 1 0 1 1 0 1 1 0 1]: RL agent learns to cycle the "cooperate, cooperate, defect" action sequence

[1 1 1 1 1 1 1 1 1 1]: RL agent learns to always defect

Notably, in all four cases, the agent does not receive worse scores consistently than its opponent. This is a surprising result, given that the agent's RL objective does not take into account the opponent's score. For each case, the markov matrix and Q-table learned by the agent is vastly different. Figure 6 lists three converged strategies. The markov matrix can be understood as choosing the best actions given state with probability, which maximizes cumulative discount reward. For example, for cyclic 110 strategy, the Q value of state DCD' `(prev_own_action: defect, prev_opponent_action: cooperate, current_own_action: defect)` = 38.65 is higher than Q value of Q(DCC') = 13.38. This corresponds to the markov matrix entry P(DC) = 1, indicating the agent will always defect, if in the previous episode agent defected and opponent cooperated.

I ran 10 trials for Q-learning agent vs. Tit-for-Tat, varying the number of LSTM layers (1 and 2). Experiments results are in Table 1.

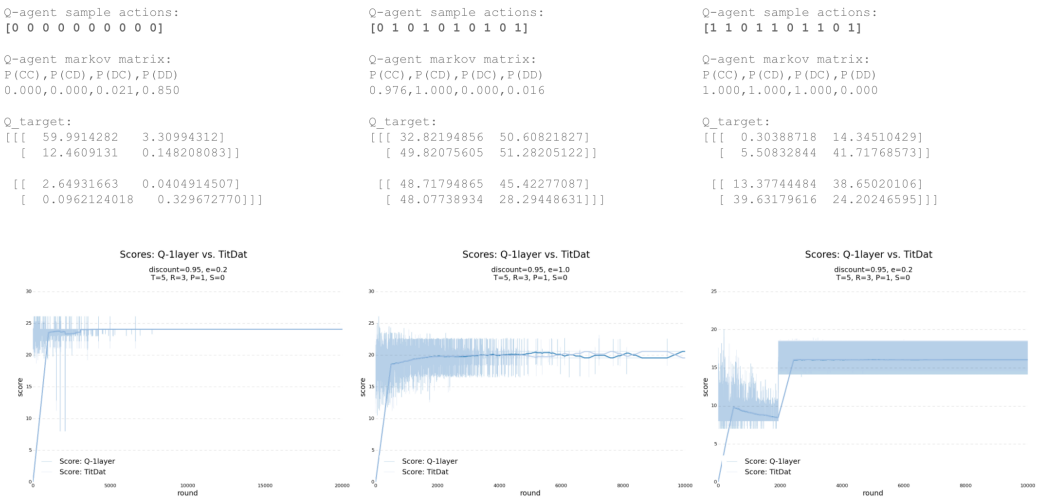

Figure 6: Q-learning against tit-for-tat in a game and three of the converged strategies. Left: always defect (111). Center: alternating (1010). Right: cyclic (110).

Table 1: Converged Strategies: RL agent against Tit-for-Tat

Strategy	Count (1 layer)	Count (2 layer)
000	5	7
110	5	0
111	0	3

4.4.1 Q-learning Agent vs. Tit-for-Tat

4.4.2 Q-learning Agent vs. Q-learning Agent

I also let two Q-learning agents play against each other across 10 trials, and varying the number of LSTM layers (1 and 2). The two layer network fails to discover the globally optimal strategy of always cooperating. While the one layer network discovers it 2 out of 10 trials. In both cases, the two networks' learned Q-tables are similar in value. Experiments results are in Table 2. Typical action sequences and Q-values are shown in Figure 7.

Interestingly, a degenerative case occured 1 out of 10 trials, for both sets of trials. In the degenerative case, one network dominates the other network, by defecting all the time. While the other agent alternates between cooperate and defect. It is not clear to me why this degenerative case occured.

4.4.3 Multiple Q-learning Agents

When greater than two Q-learning agents are playing against each other, the converged strategy is most often always defect. However, there are some interesting cases, where distinct strategies evolved. In most cases, one or few high-flyer strategy evolved to take advantage of all other agents,

Table 2: Converged Strategies: RL agent against RL agent

Strategy	Count (1 layer)	Count (2 layer)
000	2	0
111	7	9
1111 - 1010 (degenerative)	1	1

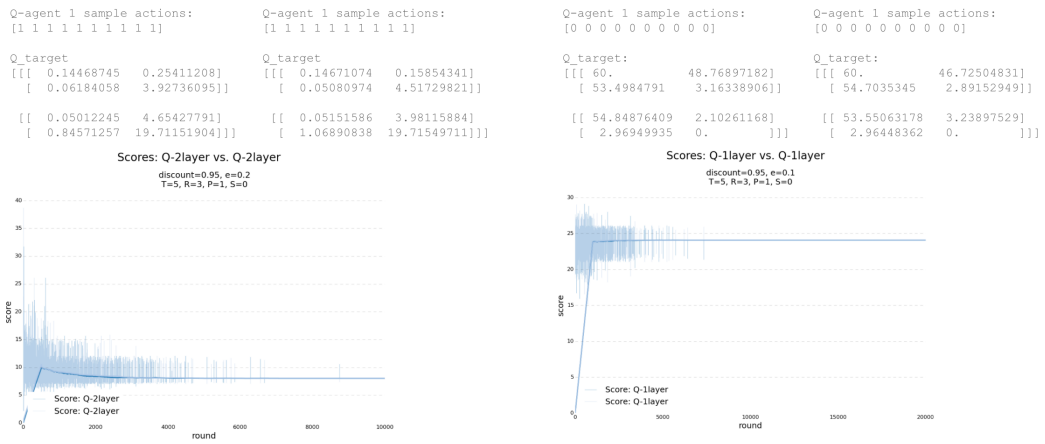

Figure 7: Two Q-learning agents in a game and multiple converged strategies. Left: always defect outcome. Right: always cooperate outcome.

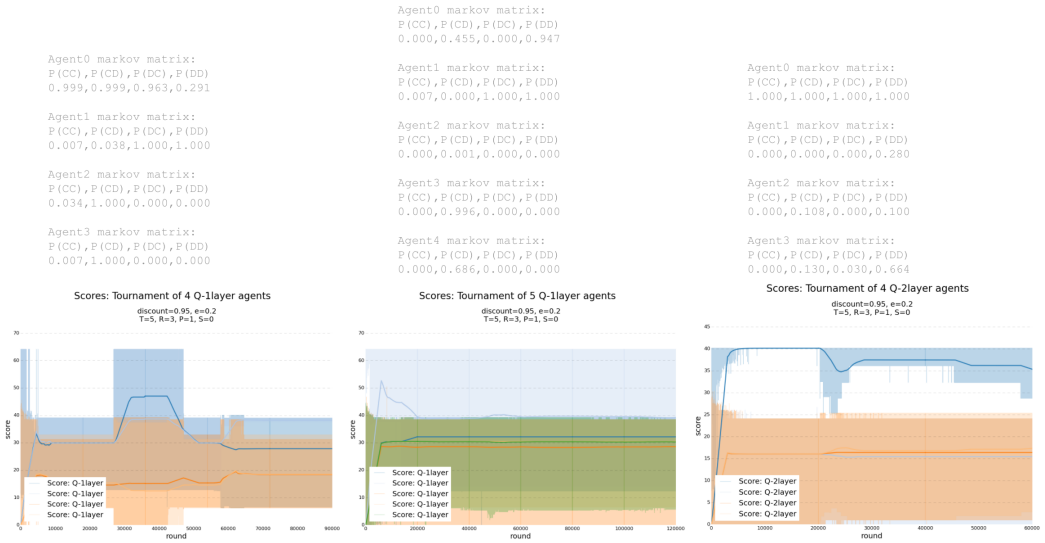

Figure 8: Greater than two Q-learning agents in a tournament. Non-uniform strategies converged.

by defecting all the time. While the other agents converge in a local optimal by always cooperating among themselves. Due to the long run time required, I did not perform multiple trials to demonstrate any statistical significance. Some trials are shown in Figure 8.

4.4.4 Q-learning Agents and Stationary Strategies

I then seeded the player pool with stationary strategies. When the player pool consist of two Q-learning agents and one always defect agent, all three agents learn to defect always. The hope was that an agent can learn to identify different opponents (always defect vs. not), and learn to play differently depending on the opponent type. However, this did not happen. There are two hypothetical approaches to develop this capability: use a larger number of episodes per round, so that the network can identify action sequences that represent signatures of a strategy. Another approach would be to append a player identifier (e.g. an `int`) to the state input, that simply identify the opponent without revealing the particular strategy.

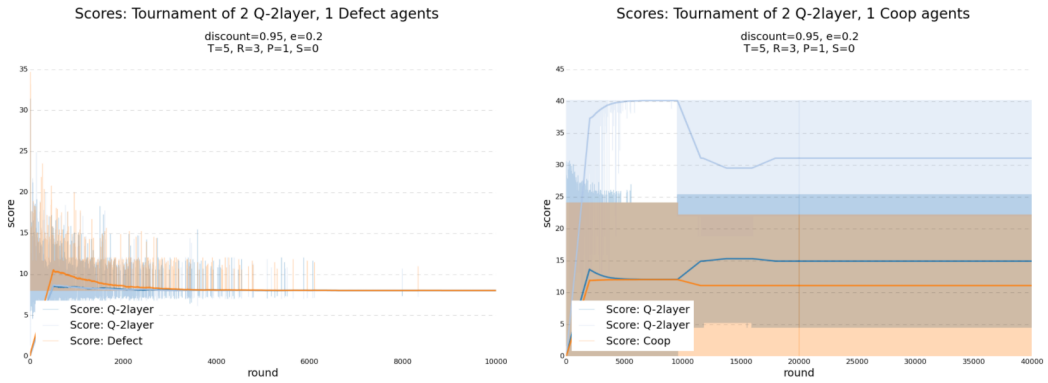

Figure 9: Left: 2 Q-learning agent + 1 always defect agent. Right: 2 Q-learning agent + 1 always cooperate agent.

When the player pool consist of two Q-learning agents and one always cooperate agent, interesting behavior occurs in one trial. One of the Q-learning agent learns to defect always, and therefore receiving the highest score. The other Q-learning agent learns to almost always cooperate. This is likely because the aggressive Q-learning agent learned early on that defecting is optimal, and therefore starts defecting always. While the passive Q-learning agent receives conflicting teaching signals. From the always cooperating agent, the passive player learns that cooperating results in reward of 3. While from the agressive agent, the passive player learns that defecting results in reward of 1. This distorted payoff matrix might have slowed down the exploration process for the agent. Some results are shown in Figure 9.

5 Discussion

In this project, I showed that a Q-learning agent can learn to play against a Tit-for-Tat agent without losing, that two Q-learning agents can learn to cooperate with each other with some low probability. In the case of more than two players, the results are harder to interpret.

There was a significant variance across trials. One direction would be to introduce experience replay that has been shown to decrease variance. Another direction is to implement policy gradient, and compare the variance in results.

In order to balance the explore vs. exploit ratio, I adopted an annealing epsilon. Thompson sampling has been shown to explore the environment faster[4]. In particular, dropout has been shown to approximate Thompson Sampling when applied to neural networks in a reinforcement learning setting[5]. It will be an interesting experiment to compare the converence speed between the two approaches.

In the case of mixed strategy players, an RL agent has not been able to identify the opposing player. Ideally the RL agent would learn this via the action sequences of opponent. However, to bound the computation, one could provide a hint by adding the opponent player ID in the input state.

The team concept has been introduced to multi-player tournament. At the 20th-anniversary iterated prisoner's dilemma competition, a team from Southampton University in England introduced a new strategy that relied on collusion among agent teammates to win the competition[6][**?**]. The strategy relied on the fact that, the performance of a team depends only on the score of the highest player. Therefore, the agent will play a sequence of actions as signatures for opponent to identify itself, afterwhich one agent will sacrifice itself by always cooperating, while the other agent always defecting to receive the highest possible score and hence win the comptition. It will be interesting to introduce the team concept with neural network based RL agents, and explore if similar colluding behavior occurs.

Acknowledgments

Thanks to Steven Hansen and James McClelland for the project advice for this Psych 209 course project at Stanford.

References

[1] Tuomas W Sandholm and Robert H Crites. Multiagent reinforcement learning in the iterated prisoner's dilemma. *Biosystems*, 37(1-2):147–166, 1996.

[2] William H Press and Freeman J Dyson. Iterated prisoners dilemma contains strategies that dominate any evolutionary opponent. *Proceedings of the National Academy of Sciences*, 109(26):10409–10413, 2012.

[3] The axelrod tournaments. `https://lawrules.wordpress.com/2011/09/05/the-axelrod-tournaments/`. Accessed: 2017-03-21.

[4] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*, pages 2249–2257, 2011.

[5] What my deep model doesn't know... — yarin gal - blog — cambridge machine learning group. `http://mlg.eng.cam.ac.uk/yarin/blog_3d801aa532c1ce.html`. Accessed: 2017-03-21.

[6] Alex Rogers, Rajdeep K Dash, Sarvapali D Ramchurn, Perukrishnen Vytelingum, and Nicholas R Jennings. Coordinating team players within a noisy iterated prisoners dilemma tournament. *Theoretical Computer Science*, 377(1-3):243–259, 2007.

[7] Wikipedia. Prisoner's dilemma — wikipedia, the free encyclopedia, 2017. [Online; accessed 21-March-2017].

[8] Prisoner's dilemma strategies.

[9] David W Stephens, Colleen M McLinn, and Jeffery R Stevens. Discounting and reciprocity in an iterated prisoner's dilemma. *Science*, 298(5601):2216–2218, 2002.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[11] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[12] Daniel G Horvitz and Donovan J Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American statistical Association*, 47(260):663–685, 1952.