

# Lecture notes for Multi-Agent Systems

Danny Kuminov

13-Dec-2006

## 1 Introduction

This is a summary of the lecture by Moshe Tennenholtz at the MAS course. The normal text (such as this) represents a nearly verbatim (up to mistakes and permutations of sentences) recording of the lecture.

(DEK) *Any addition/clarifications/open questions added by the transcriber (Danny Kuminov, dannykv@tx.technion.ac.il) will be written like this and prefaced by (DEK).*

## 2 Optimal Teaching Policy

This section is based on a paper "[On Partially Controlled Multi-Agent Systems](#)" by Ronen Brafman and Moshe Tennenholtz.

First, let us recall the teacher-learner setting. Let  $G$  be some repeated two-player game in strategic form. We assume that one player, the student, uses an *a priori* known strategy (possibly mixed) and the utility of the other player (the teacher) is a function of the student's actions only. This setting models a situation in which one player (the learner) uses some known learning algorithm (for example, HCR derivative) in order to learn to play some game, and the teacher's objective is to teach the learner to use a specific action (for example, to cooperate in the "Prisoner's dilemma").

(DEK) *A repeated two-player game is the same as  $n$ - $k$ - $g$  game described in the previous lecture, with  $n = k = 2$  and the same underlying one-shot game.*

**Definition 1** (Teaching Policy). A *teaching policy*  $\pi$  is a function from the set of finite histories of the game to an action of the teacher. Given a learning algorithm of the student,  $\pi$  induces a probability distribution  $Pr_{\pi,k}$  over the student's actions at time  $k$ . Therefore, the value of  $\pi$  for the teacher is

$$Val(\pi) = \sum_{k=0}^{\infty} \gamma^k E_k(u)$$

where

$$E_k(u) = \sum_{a \in A} Pr_{\pi,k}(a)u(a)$$

and  $u$  is the teacher's utility function.

**(DEK)** *Looking at the setting from the non-cooperative game theory point of view and using the repeated game formalism, the problem of finding the optimal teacher policy is equivalent to the following:*

*Given*

- *a repeated game such that the utility of player 2 (the teacher) in the underlying one-shot game depends only on the actions of player 1 (the learning algorithm)*
- *a specific strategy for player 1 for the repeated game*

*find the best-response strategy for player 2.*

*However, it can be easily seen that without putting any limitations on the strategy of player 1, this problem cannot be solved by a Turing machine (to be more accurate, the best-response strategy might be impossible to implement by a Turing machine). To see why it is so, consider the following strategy for player 1: interpret the action of player 2 in turn  $k$  as an answer to the question "Does Turing machine number  $k$  under fixed enumeration of TMs halt?" and play 1 as long as player 2 answers correctly, otherwise play 2 forever. To "teach" player 1 to play 1, player 2 must solve the infamous Halting Problem, which is undecidable. However, the above strategy for player 1 is by itself impossible to implement by a Turing machine; therefore, it makes sense to put computational limitations on the possible strategies of player 1, which is equivalent to limiting the space of possible learning algorithms. Specifically, we will assume from now on that the algorithm of player 1 can be represented by a finite-state automaton (which is the same as a Turing machine that uses fixed amount of memory). Note that this is a very strong assumption that does not hold for many known learning algorithms, because it does not allow, for example, to count elapsed stages or to compute the cumulative utility (one needs  $O(\log t)$ , where  $t$  is the running time, to do that).*

We would like to characterize optimal teaching policies. Under the previous assumptions, the teacher's problem can be modeled as finding the optimal policy for a *Markov decision process (MDP)*. An MDP models a repeated interaction between an agent with finite amount of actions and an environment with finite amount of states, with the next state depending stochastically on the previous state.

**Definition 2** (MDP). A *Markov decision process (MPD)* is a tuple  $(S, A, P, r)$ , where

- $S$  is a finite set of world states.
- $A$  is a finite set of agent's actions.
- $P$  is a probabilistic state transition function:  $P : S \times S \times A \rightarrow [0, 1]$ , where  $P(s_1, s_2, a)$  is the probability of transition from state  $s_1$  to state  $s_2$  given agent's action  $a$ .
- $r$  is the reward function:  $r : S \rightarrow \mathfrak{R}$ , where  $r(s)$  is the agent's reward from being in state  $s$ .

Given an initial state  $s \in S$  and some policy  $\pi : S \rightarrow A$ , the distribution function  $Pr_{s,\pi,k} : S \rightarrow [0, 1]$  is well defined:  $Pr_{s,\pi,k}(s')$  is the probability of getting to state  $s'$  after  $k$  steps, starting from the initial state  $s$  and using policy  $\pi$ . A strategy is called  $\gamma$ -optimal for a specific MDP if it maximizes the  $\gamma$ -discounted expected agent's reward:

$$\sum_{k=0}^{\infty} \gamma^k \left( \sum_{s' \in S} P_{s,\pi,k}(s') r(s') \right)$$

There is a classic theorem that shows existence and efficient computation of optimal strategy (using dynamic programming). Note that this strategy is optimal for *any* starting state.

We assume that the learner's algorithm has finite number of states  $\Sigma$ . Therefore, its state transition function can be represented as  $\tau : \Sigma \times A_S \times A_T \rightarrow \Sigma$  (where  $A_S$  and  $A_T$  are the sets of student's and teacher's actions accordingly) and its probabilistic decision function – as  $\rho : S \times A \rightarrow [0, 1]$  (which means that  $\rho(s, a)$  is the probability of acting  $a$  in state  $s$ ).

Recall that we assume that the teacher knows everything, in particular he knows the learner's algorithm and its starting state. Therefore, the teacher can simulate it – assuming that he knows the current state of the algorithm and sees the actions selected, he can predict the next state of the algorithm. Therefore his problem is effectively reduced to selecting his actions given the student's state.

**Definition 3** (TMDP). We will define a Teacher's MDP (TMDP)  $(\Sigma, A_T, P, v)$  as follows:

- The set of the states is exactly  $\Sigma$ , the set of the learning algorithm states.
- The set of actions is exactly  $A_T$ , the set of teacher's actions.
- The transition function is

$$P(s, s', a_T) = \sum_{a_S \in A_S} \rho(s, a_S) \delta_{s', \tau}(s, a_S, a_T)$$

where  $\delta_{s', \tau}(s, a_S, a_T) = 1$  if  $\tau(s, a_S, a_T) = s'$  and zero otherwise.

- The reward function is

$$u(s) = \sum_{a_S \in A_S} \rho(s, a_S) u(a_S)$$

**Claim 1** (Reduction correctness). *The optimal policy for the TMDP represents the Optimal Teaching Policy as defined above.*

*Proof.* The optimal policy for the TMDP maximizes the expression

$$\sum_{k=0}^{\infty} \gamma^k \left[ \sum_{s' \in \Sigma} P_{s, \pi, k}(s') u(s') \right]$$

By substituting the expression for  $u(s')$  we get:

$$\sum_{k=0}^{\infty} \gamma^k \left[ \sum_{s' \in \Sigma} P_{s, \pi, k}(s') \left( \sum_{a_S \in A_S} \rho(s, a_S) u(a_S) \right) \right]$$

And changing the order of summation gives us:

$$\sum_{k=0}^{\infty} \gamma^k \left[ \sum_{a_S \in A_S} \left( \sum_{s' \in \Sigma} \rho(s, a_S) P_{s, \pi, k}(s') \right) u(a_S) \right]$$

Note that

$$\sum_{s' \in \Sigma} \rho(s, a_S) P_{s, \pi, k}(s')$$

is the probability that the action  $a_S$  is chosen by the student at time  $k$  given starting state  $s$ , and therefore the whole expression represents  $Val(\pi)$ , the expected utility of the teacher using policy  $\pi$ .  $\square$

Therefore, we can find the optimal teaching policy for a specific learning algorithm by reducing to an MDP, finding the optimal policy for it and defining the teaching algorithm as

- Simulate the learning algorithm, so that its state is known at each point in time
- Do the action that the MDP optimal policy dictates for this state

### 3 Examples of the learning algorithms for the student

#### 3.1 Q-Learning

The Q-Learning algorithm was first proposed by [Watkins, C.J.C.H. \(1989\). Learning from Delayed Rewards. PhD thesis, Cambridge University, Cambridge, England.](#)

This [site](#) contains a nice explanation of Q-learning by presenting specific examples (including working demonstrations).

*Q-learning* is the name for a family of learning algorithms, parameterized by  $\gamma$  and  $\alpha$ . The special case  $\gamma = 0$  is called *Classical Reinforcement Learning*.

All algorithms in this family have a finite set of states and the transitions between them are determined by the input the algorithm receives from the environment in each stage. Furthermore, the states and the transition function are known *a priori* and fixed (for example, consider an algorithm that is limited to remembering the results of the last three games – its states represent all possible vectors of outcomes of length three and the transition function for state  $(x_1, x_2, x_3)$  and current outcome  $x$  transfers to a state  $(x, x_1, x_2)$ ). Therefore, the only parameter that can be tuned by the learning process is the action function (which determines the algorithm’s action in each state). The algorithm will assign to each state-action pair a “quality” measure  $q(s, a)$  and the action function will select an action according to the values of  $q(s, a)$  for the current state. Let  $0 < \alpha < 1$  be the learning coefficient and  $0 \leq \gamma < 1$  the external parameter, then the value of  $q(s, a)$  will be updated by the learning algorithm according to the following formula:

$$q(s, a) := (1 - \alpha)q(s, a) + \alpha(R + \gamma v(s'))$$

where  $v(s') = \max_{a \in A_s} q(s, a)$  and  $R$  denotes the reward received in the current stage.

The formula has two “levels” - first of all, the new value of  $q(s, a)$  is a weighted average (with coefficient  $\alpha$ ) of the old value and the result of the current action. Thus,  $\alpha$  determines the relative weight of the new experience to the memory of previous experiences in determining the agent’s action. Secondly, the result of the current action is considered to be a weighted sum of the immediate reward and the value that the agent is expected to gain in the new state.  $\gamma$  is the relative weight that the algorithm assigns to this “foresight”.

**(DEK)** Note that  $\gamma < 1$  ensures that  $q(s, a)$  stays bounded (by  $(1 - \gamma)^{-1}R$ ) as the number of games played goes to infinity.

The description above does not state how the action is selected given the matrix  $q(s, a)$ . Here, a tradeoff exists between “exploitation” - selecting actions that were tried before and collected high utility (or led to a “good” state), and “exploration” - selecting previously untried or low-utility actions in order to explore the possibility that they might lead to previously undiscovered “good” states. One of the methods to resolve this dilemma is: let  $P_s(a)$  denote the probability of playing action  $a$  in state  $s$ , then

$$P_s(a) = \frac{e^{\left(\frac{q(s,a)}{T}\right)}}{\sum_{a' \in A} e^{\left(\frac{q(s,a')}{T}\right)}}$$

This expression is known in physics as Boltzmann distribution with “temperature” parameter  $T$ . The idea here is to start with a high  $T$  (which implies

a near-uniform distribution on the actions, which allows effective exploration) and gradually reduce it as more games are played (this is similar to a known AI technique called “Stimulated Annealing”). As  $T \rightarrow 0$ , the distribution converges to selecting the highest-q-value action with probability 1.

**(DEK)** *A trick question: suppose you have a Q-learning algorithm based on states that represent the last three games played, with the appropriate transition function. How much memory does this algorithm use and how far back does its memory go?*

*Answer:*

1. *Since the algorithm maintains q-values that are based on accumulating payoffs from all games played by the algorithm, there is no limit on how back its memory goes. Of course, the payoffs are not kept explicitly and cannot be restored from q-values, but still: one cannot say, for example, that after enough games were played the memory of the algorithm is flushed and its state is essentially reset, as we did in the proof for HCR.*
2. *The answer to this question depends on the representation the algorithm uses for real values (of q and the probability function). Accurate representation requires linear memory, while rounding to fixed number of bits may affect algorithm’s correctness - I, for one, don’t know whether the algorithm can be implemented with fixed memory. In any case, the fact that the number of states is fixed does not promise anything about the size of the memory used.*

*The point of the question is to point out that I don’t think that Q-learning can be represented by a finite-state machine.*

Under mild assumptions on the action selection function and the model, Q-learning has the following property:

**Claim 2** (Q-learning). *If the Q-learning algorithm is executed in an environment that determines the rewards according to an unknown MDP, the algorithm will converge to the optimal policy for that MDP. However, the convergence time is not guaranteed to be polynomial (and indeed might not be).*

Therefore, the problem (which we will solve in the next section) is to construct a learning algorithm with polynomial convergence time that converges to an optimal policy against unknown MDP environment.

## 3.2 Reinforcement Learning

This section is based on a paper [”R-MAX - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning”](#) by Ronen Brafman and Moshe Tennenholtz.

The subject of Reinforcement Learning is to learn to behave in an unknown environment on the basis of observed reactions. We have seen what can be done when the environment is an MDP. However, adversarial behavior of other rational agents cannot be modeled by MDP. For a setting that includes multiple strategic agents, the proper model is a more complex model (that extends both MDP and the repeated game setting), called a “Stochastic Game”.

**Definition 4** (Stochastic Game). A fixed-sum, two-player stochastic game [SG] on states  $S = \{1, \dots, N\}$  and actions  $A = \{a_1, \dots, a_k\}$  consists of:

- **Stage Games:** each state  $s \in S$  is associated with a two-player, zero-sum strategic form game where the action set of each player is  $A$ . We use  $R_i$  to denote the reward matrix associated with stage-game  $i$ .
- **Probabilistic transition function:**  $P_M(s, t, a, a')$  is the probability of a transition from state  $s$  to state  $t$  given that the first player (the *agent*) plays  $a$  and the second player (the *adversary*) plays  $a'$ .

In this game, the players play an infinite sequence of games, selected from a given set of games according to the current state. After playing each game, the players receive their payoffs according to the game matrix and the next game to be played is determined (stochastically) based on the players’ joint action and the current state. Note that we assume w.l.o.g. that the action set is the same for both players in all games.

This model includes both MDP (where the adversary has only one possible action in each game) and repeated game setting (where there is only one state). We assume that the player does not know beforehand what the game matrices and the transition function are, but he can observe the joint action in each state and the state he is currently in (that is, he can recognize a state he already visited). We also assume that he knows the number of states  $N$  and the maximal possible payoff in all the games  $R_{max}$ . Our goal is to build an algorithm that can achieve a near-optimal guaranteed expected average reward quickly, in the following sense:

- **expected average** - our performance measure in the infinite game is the  $\liminf$  of the expected average T-step reward as  $T \rightarrow \infty$ , where the expectation is taken over the transition probabilities and the random decisions that the players are making.
- **guaranteed** - we would like to guarantee the maximal possible reward that can be achieved regardless of the opponent strategy (i.e. we are looking for a safety-level strategy).

**Formally:** Let  $U_M(s, \pi, p, T)$  denote the expected average reward over the first T steps, when

- the starting state is  $s$
- the opponent’s strategy (possibly mixed) is  $p$

- the player’s strategy (defined by our algorithm) is  $\pi$
- the stochastic game is  $M$

Let  $U_M(s, \pi, T) = \min_p U_M(s, \pi, p, T)$  be the probabilistic safety-level value, and let  $U_M(s, \pi) = \liminf_{T \rightarrow \infty} U_M(s, \pi, T)$ . Then the value of the player’s strategy is  $U_M(\pi) = \min_s U_M(s, \pi)$ .

- **near optimal, quickly** - if we knew the exact game *a priori* and used the optimal (w.r.t. the above criteria) strategy from the first step, we could have guaranteed payoff value which is  $\epsilon$ -close to the safety-level after  $k$  steps. We want our algorithm to guarantee at least the same value after a number of steps that is polynomial in  $k$ .

**(DEK)** *Note that both the optimal strategy and our algorithm are probabilistic, and therefore they can only make guarantees of the kind: “with probability at least  $1 - \delta$ , the payoff is at least  $V - \epsilon$  after  $k = \text{poly}(\frac{1}{\epsilon}, \frac{1}{\delta})$  steps”. This is because there always exists a sequence of random decisions that will cause them to fail altogether (with probability at most  $\delta$ ) and they can never guarantee the exact expected value, only get  $\epsilon$ -close to it.*

*The term  $\epsilon$ -return mixing time is used to denote the number of games  $k$  that an algorithm needs to get  $\epsilon$ -close to its expected value.*

The algorithm is called  $R_{max}$ , after the parameter that represents the maximal possible reward in the game. It maintains an optimistic fictitious model of the environment as follows:

- Model initialization – it creates  $n$  states and 1 fictitious state. In each state, it initializes all entries of the game matrix with  $R_{max}$  and assumes all joint actions transfer to the fictitious state with probability 1 (in particular, that means that once the player gets to that fictitious state, he stays there forever and always gets  $R_{max}$ ). In addition, all entries in all game matrices are initially marked as “unknown”.
- Model update - After a joint action is played in a specific state, the algorithm does the following:
  - Records the observed payoff in the appropriate entry in the matrix
  - Records the observed transition for the joint action in the specific stage
  - Once enough transitions are observed for the specific entry in a game matrix, marks the entry as “known”, updates the transition function with the observed frequencies and recomputes the optimal strategy for the model

Given the fictitious model, the algorithm always plays the optimal strategy relative to it.

**Claim 3** (Main Theorem). *Let  $M$  be an SG with  $N$  states and  $k$  actions. Let  $\epsilon > 0$  and  $0 < \delta < 1$  be the desired error bounds. Denote the set of policies for  $M$  whose  $\epsilon$ -return mixing time is  $T$  by  $\Pi_M(\epsilon, T)$  and the optimal expected return achievable by such policies by  $OPT_M(\Pi(\epsilon, T))$ , then with probability  $\geq 1 - \delta$  the algorithm will attain actual average return of no less than  $OPT_M(\Pi(\epsilon, T)) - 2\epsilon$  within  $\text{poly}(N, T, k, \frac{1}{\delta}, \frac{1}{\epsilon})$  steps.*

*Proof.* The proof is based on the following definition and two lemmas:

**Definition 5** ( $\alpha$ -approximation). An SG  $M'$  is an approximation of  $M$  iff

- They have the same state and action sets
- $|p_M(s, t, a, a) - p_{M'}(s, t, a, a)| < \alpha$  for all states and actions
- for every state  $s$ , the stage game is the same in  $M$  and  $M'$

**Claim 4** (The simulation lemma). *Let  $M, M'$  be SG's over  $N$  states, where  $M'$  is an  $\frac{\epsilon}{N \cdot T \cdot R_{max}}$ -approximation of  $M$ . Then, for every state  $s$ , agent policy  $\pi$  and adversary policy  $p$ :*

$$U_M(s, \pi, p, T) - U_{M'}(s, \pi, p, T) < \epsilon$$

**Claim 5** (The implicit explore or exploit lemma). *Let  $M$  be an SG,  $M_L$  the model maintained by  $R_{max}$ , where  $L$  is the set of unknown states. Assume that  $M_L$  is identical to  $M$  on known states. Let  $p$  be an arbitrary adversary policy,  $0 < \alpha < 1$  an arbitrary constant,  $V_{R-max}$  the expected  $T$ -step average reward on  $M$  by the optimal policy for  $M_L$ .*

*Then either  $|OPT_M(\Pi(\epsilon, T)) - V_{R-max}| < \alpha$  or an “unknown” entry will be played within  $T$  steps with probability at least  $\frac{\alpha}{R_{max}}$ .*

The last lemma ensures that we either “do well” or learn something new within fixed time. Since there are only polynomially many entries to discover, at worst after polynomial number of steps the algorithm will achieve an  $\epsilon$ -optimal return.  $\square$

**(DEK)** *Note that what the theorem actually says is the following: if the algorithm is given  $T$  as its input, then it is guaranteed to do as well as the best policy that has  $\epsilon$ -return mixing time  $T$ . If we knew the value  $T$  for the optimal policy a priori (or at least an upper bound on it) and had given it as an input to the algorithm, then we could say that the algorithm is guaranteed to perform as well as the best policy. Unfortunately, there is no way to know the  $\epsilon$ -return mixing time of the optimal policy in advance, and therefore a scheme for searching for the minimal sufficient  $T$  has to be incorporated into the learning algorithm. The paper by Brafman and Tennenholtz that is referenced at the start of the section provides a solution for it; I will not describe it here, since it's outside the scope of the lecture.*