"Multi-task Inference and Planning in Board Games using Multiple Imperfect Oracles"
Research Project Report – Graphs in ML & RL

Lilian Besson*
Department of Mathematics
École Normale Supérieure de Cachan
Cachan, France
lilian.besson@ens-cachan.fr

Basile Clement
Department of Computer Science
École Normale Supérieure
Paris, France
basile.clement@ens-cachan.fr

Abstract

For our research project, we studied the framework of inverse reinforcement learning (IRL) applied to learn a board game's optimal strategy. In this small project report, we first present quickly the required hypotheses on the game and the usual notations, along with a sum-up of the usual approach of IRL on what we call "linearly-represented" games. Then we consider multi-task, where we have a database of games from multiple experts. Our main contribution is to extend the work of [PD15b] and [DR12] to imperfect oracles, where we do not accord the same trust to each expert, but introduce a prior or compute a posterior on their strength.

Project Advisor: Christos Dimitrakakis (Inria Lille & Chalmers University)
For: the *"Graphs in Machine Learning"* course, by Michal Valko
and the *"Reinforcement Learning"* course, by Alessandro Lazaric
For the Mathématiques, Vision, Apprentissage (MVA) Master 2 at ENS de Cachan
Grade: We got 18/20 for our project.

---

1 Presentation

In this short project report, we first present the IRL framework quickly, and then recapitulate notations, mainly inspired by those used in [DR12, TD13, PD15b]. In section 2, we present the main idea for an IRL algorithm trying to generalize [TD13]'s approach to a multi-task setting, when we give it a prior on the relative strength of the experts (relative scoring). Then we present an alternative algorithm, adapting Pengkun's algorithm [PD15b], and discuss two approaches to compute a posteriori the experts distribution (*e.g.* based on their performances on the available demonstrations).
The last section presents our implementation, and sums up our experiments on 3-by-3 and 4-by-4 Tic-Tac-Toe, on which we re-implemented both methods from [TD13] and [PD15b] and our extended algorithm.

*If needed, see on-line at http://lbo.k.vu/gml2016 for an e-version of this report, as well as additional resources (code, complete bibliography etc), open-sourced under the MIT License.

2 The IRL framework for board games

Here are detailed the hypotheses and notations we make on the board games we are trying to solve. We only consider full-knowledge, turn-based board games with two players. We first present the RL setting and the IRL problem quickly, and then recapitulate notations from [DR12, TD13, PD15b].

2.1 States and actions

As usually done in Reinforcement Learning (RL), we start with $\mathcal{S}$ a set of states (finite), and $\mathcal{A}$ a state of actions (also finite).
For the n-by-n Tic-Tac-Toe[1] games studied in this project, the set of actions $\mathcal{A} \overset{\text{def}}{=} \{1, \dots, n\} \times \{1, \dots, n\}$ corresponds to the different spaces on the grid where players can put a mark. The state set $\mathcal{S}$ is not harder to write formally: $\mathcal{S} \overset{\text{def}}{=} \{X, O, \cdot\}^{n^n}$. When a player applies an action a to a state s, the result $s + a = s'$ is the state where the a-th space has been replaced with the player's mark. For instance, applying the action $(1, 3)$ for player X to the initial state $s = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$, we obtain the state $s' = \begin{bmatrix} \cdot & \cdot & X \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$. Below is showed an example[2] of a game, being a succession of choice of action for the player X and his opponent X:

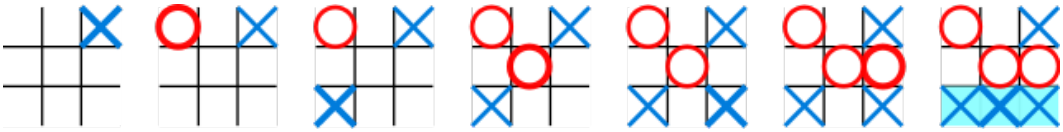

Figure 1: Example of a game for 3-by-3 Tic-Tac-Toe, X winning against O.

The RL problem is to learn a way to play the game (a policy), using the idea of learning by playing, where we get a reward $\rho(a, s)$ after a decision $\pi(a, s)$, and the reward is usually drawn from a MDP. We can learn either if we have a full knowledge on the underlying MDP of the game or a prior on the MDP.
We call trajectory (or demonstration) a game history, seen as a finite sequence of states s_t and actions a_t chosen by the first player (e.g. X). We do not keep track of the state s'_t neither of the action a'_t chosen by its opponent (e.g. O), because they are implicitly contained in the next state s_{t+1}. In effect, from the point of view of player X, actions from player O corresponds to the environment's transitions.
A demonstration d of length T is therefore written $d = [(s_1, a_1), \dots, (s_T, a_T)]$. A collection (or samples) of i demonstrations will be written: $\mathcal{D} \overset{\text{def}}{=} \{d^{(1)}, \dots, d^{(i)}\}$. On the other hand, the inverse RL problem is about learning a policy by using existing demonstrations (and not playing with the MDP anymore).

2.2 Naive approach: minimax tree search

To keep this report short, we do not present here the well-known usual approach in (board) game inference: the minimax tree search. In a nutshell, it is a complete tree search to select the move which maximizes the end-game score. For more details, please read the Wikipedia page for example.

It was quick and easy for the 3-by-3 Tic-Tac-Toe: so we implemented it and used it for our experiments. But it only works for (very) small games!
Note that minimax is *optimal* for 3-by-3 Tic-Tac-Toe: it never looses (either win or draw). This is why when comparing a policy against the minimax one in the experiments reported below, we focussed on the draw rate and not the win rate. Below is illustrated[3] a small minimax tree search:

[1] For the sake of conciseness we do not present the game here, please see online for more details.
[2] Illustration from Wikimedia, `https://en.wikipedia.org/wiki/Tic-tac-Toe`.
[3] Illustration from `beej.us/blog/data/minimax/`.

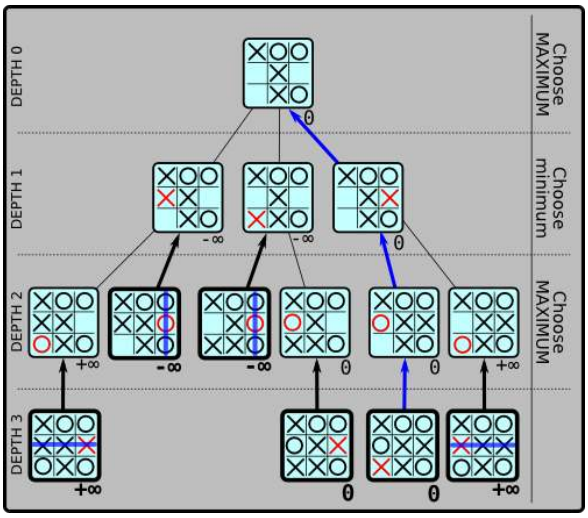

Figure 2: Illustration of "minimax" tree search for 3-by-3 Tic-Tac-Toe.

But when there is too many policies, we are facing a combinatorial explosion! Minimax obviously fails if the game tree is too big (too deep or too wide, or both). No need for huge games (Go, chess), 4-by-4 Tic-Tac-Toe is already too big! Note that optimized or randomized tree search has been studied a lot, e.g. [HK14] or the Monte-Carlo tree search algorithm ("MCTS"). For our project, we chose to explore another approach: learning from demonstrations.
So we will introduce below one more hypothesis on the game: we restricted our work to **"linearly representable"** games.

2.3 From one expert to M experts

Now for the multi-expert setting, we introduce a few additional notations. Instead of having only one database of demonstrations, we consider $M \geq 1$ different experts, indexed with $m = 1, \dots, M$. For each expert m, we have $\mathcal{D}_m = \{d_m^{(1)}, \dots, d_m^{(i_m)}\}$ a certain number of demonstrations $d_m^{(i)}$.
As before, we can write thus the entire database of demonstrations as $\mathcal{D} = \cup_{m=1}^{M} \mathcal{D}_m$.

2.4 A prior distribution on the M players

We now introduce a way to represent a (prior) knowledge on the relative strengths of the expert. For instance, we might know that Alice ($m = 1$) is a very clever player, while Bob ($m = 2$) has just been introduce to the game, and so the purpose of this score on players is to represent this difference of skills, which in the IRL setting will imply to trust more the player Alice and trust less the player Bob.
We thought of at least three points of view for this "relative strength":

- an *absolute* score, we would write it $\text{score} : \{1, \dots, M\} \to \mathbb{R}, m \mapsto \text{score}(m)$ (e.g. the ELO score for chess, going to $[0, 4000]$).
- a *relative* ranking on M players, it would just be a permutation on $\{1, \dots, M\}$.
- But the more interesting point of view, in order to learn a stochastic policy (i.e. a probability distribution on actions), is to simply talk about this ranking or score as a discrete probability distribution on the players $\{1, \dots, M\}$.

In our example, we might want to model the fact that Alice should be trusted way more than Bob, so we take $e(1) = 0.8$ and $e(2) = 0.2$. This is the hypothesis we make from now, of having a distribution $e : \{1, \dots, M\} \to [0, 1]$.
This is also called imperfect oracles (or experts) in the literature, as this e represents how

trustworthy are the experts, and a small $e(m)$ can be interpreted as m being an imperfect oracle.
We also explain below how to compute such a distribution $e(m)$ a posteriori, based on different approaches to evaluate the strength of the experts.

2.5 Linearly-representable games

Another hypothesis we make on the game is what we called "linearly-representable" above. More formally, this means that a couple state, action (s, a) is not described as just an index $(s, a) \in \mathcal{S} \times \mathcal{A}$ in the product space, but through real-valued vectors[4] $g_\rho(s, a)$ and $g_Q(s, a)$, respectively used for computing ρ (the reward function) and the Q-value function.

The main hypothesis done here is to only use linear combinations of the components of the features vector $g(s, a)$. Therefore, we have $\rho_m(s, a) = g_\rho(s, a)^T w_\rho^{(m)}$ for the reward function, and $Q_m(s, a) = g_Q(s, a)^T w_Q^{(m)}$ for the Q-value function (see [SB98] for more details about Q functions). For board games, these functions are called *board features function*: $g : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^F$ if we chose to have F features. Both[5] g_ρ and g_Q will depend on both the current state s and action a, as $g : (s, a) \mapsto g(s, a) \in \mathbb{R}^F$, and use the same number of board features.

These two g_ρ, g_Q have to be implemented and changed for every game, and we will not cover the intricacies of designing and perfecting a good feature function. Note that we never use g_ρ afterward, but it could be chosen equals to g_Q. As this aspect represents our knowledge of the game, we assume g_ρ, g_Q to be the same for each expert.

Concretely for Tic-Tac-Toe, we use[6] an extension of the features presented in [TD13, Part 5.3] and [KBB09] to handle the nxn grid. We use the $4n - 2$ following features, as well as their second-order products (multi-variate binomial terms), for a total of $(4n-2)(4n-1)/2$ features:

- number of i-lets for each player, i.e. lines/columns/diagonals with exactly i marks of the corresponding player and all other spaces blank,
- i-diversity, the number of directions for i-lets for each players,
- the number of marks on the diagonals for each player.

An important motivation for choosing these features is their invariances by the grid symmetries. In the end, we use $O(n^2)$ features.

Note that in our first experiment (section 4.1), we obtained better results with a quadratic number F'_n of features than with a linear F_n: and this is logical, because having more (well chosen) features allow to fit the expert's policy more efficiently.

However, what changes for each expert m is the weights he/she decides to put on each features. For multi-expert, we will have a weight vector for each expert, and they will be written, for m, as $w_\rho^{(m)}$ and $w_Q^{(m)}$. Therefore, we have $Q_m(s, a) = g_Q(s, a)^T w_Q^{(m)}$ for the Q-value function, and this expression will be used for our algorithm below.

2.6 From Q-value to a policy

We recall that a policy is a way of deciding – deterministically or probabilistically – which action to select when we are in a state. The first way to create an efficient policy from a

[4] This change of point of view extends the idea of what is usually done, in a simpler way, for multi-class classification: instead of working with an index $1 \leq k \leq K$, we work with a "one-hot" encoding $\overrightarrow{u} \in \{0, 1\}^K$.

[5] [TD13] suggests that maybe g_ρ can be only indexed on the state, but it seemed more logical to have the same dimension for both.

[6] For details about this part of the implementation, refer to the `features` methods in the `TicTacToe` class.

Q-value function is to take the $\arg\max$, like:

$$\pi_Q(s) = \arg\max_{a \in \mathcal{A}} Q(s,a).$$

However, this is a strategy that will always pick the same action for each state - if we are wrong that this is the "right" action, the penality will be harsh! A better choice is to use a *soft-max* function, which smoothly approximates the maximum.

$$\pi_Q(a|s) = \mathrm{softmax}_\beta(Q)(s,a). \tag{1}$$

For $\beta \in \mathbb{R}_+^\star$, a parameter called *temperature*, $\mathrm{softmax}_\beta(Q)$ is defined as: $\mathbb{R}^{\mathcal{S}\times\mathcal{A}} \to \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$:

$$\mathrm{softmax}_\beta(Q) : (s,a) \mapsto \pi_Q(a|s) = \frac{\exp\left(\beta Q(s,a)\right)}{\sum\limits_{a' \in \mathcal{A}} \exp\left(\beta Q(s,a')\right)} \tag{2}$$

The sum in the denominator is simply a renormalization factor, in order to have a stochastic matrix $\mathrm{softmax}_\beta(Q)$: $\sum_{a\in\mathcal{A}} \mathrm{softmax}_\beta(Q)(s,a) = 1$, and so a vector $\pi_Q(\cdot, s)$ which is really a stochastic distribution.
Intuitively, the $\mathrm{softmax}_\beta$ function behave in the following way: in the denominator, the larger term is the one for $Q(s,a^\star) = \max_{a\in\mathcal{A}} Q(s,a)$, and so if it's the temperature is high enough ($\beta \gg 1$) then for other actions $a' \neq a^\star$, $\exp(\beta Q(s,a)) \ll \exp(\beta Q(s,a^\star))$. So $\pi_Q(s|a) \ll \pi_Q(s|a^\star)$, and stochastically $a^\star$ has a very high probability of being chosen which increases with β.

This β can either be a fixed constant (*e.g.* $\beta = 1$), or a parameter that will be maximized along with w_ρ or w_Q. We will see that β can be set to 1 as the optimization problems for w_ρ and w_Q are unconstrained, and only use βw (they are homogeneous in βw so the value of β is implicitely included in the objective w).
But β can also be considered as a random variable, drawn from a unknown parametric distribution that we would also try to learn. This is done for instance in [DR12], where a form of parametric distribution for β is first chosen (a Gamma one, with two parameter g_1, g_2), and then introduce a prior distribution on these two parameters (it is thus referred to as a hyper-prior). Another article proposes an Exponential prior for β [TD13, Part 4.1]. We chose to first work with a fixed $\beta = 1$ for our implementation, for the sake of simplicity.

3 Aggregation learning with imperfect experts

We now present the main contribution of our project, a new learning algorithm which generalizes [TD13]'s approach to a multi-task setting, by including a prior on relative strength of the experts in order to learn an aggregated policy, and then extend it with different approaches that compute this distribution a posteriori.

3.1 Probabilistic justifications

To come back to a more formal presentation of the problem, to learn the behavior of the m-th expert, what we try is to find a $\pi_m^\star$ which maximizes this likelihood:

$$\max_{w_m} \mathbb{P}(\mathcal{D}_m | \pi_m^\star, Q_m^\star)$$

The goal is to learn a $\pi_m^\star$ (for the expert m) which best fits the observed demonstrations $\mathcal{D}_m$, but is also able to generalize. This exactly means that we want to avoid over-fitting, as usual in machine learning. It will be solved separately for each expert.

Now for the aggregation part, we try to find a $\pi^\star$ which maximize this second likelihood:

$$\max_{w} \mathbb{P}(\mathcal{D} | \pi^\star, Q^\star, \rho^\star) = \max \prod_m \mathbb{P}(\mathcal{D}_m | \pi^\star, Q^\star, \rho^\star).$$

Which is equivalent to maximize a log-likelihood, except the second form highlights the separability between experts: $\max_w \sum_m \mathcal{L}(\mathcal{D}_m|\pi^\star, Q^\star, \rho^\star)$.

Because we are learning weights w, and assume the Q-function to be linear in w, it make sense to propose this scheme for combining the $Q^\star_m$:

$$Q^\star = \mathbb{E}_m(Q^\star_m) = \sum_{m=1}^{M} e(m) Q^\star_m.$$

Note that is is equivalent to a convex combination on the weights w (or an expectation), but doing the same for policies cannot be justified probabilistically.

3.2 A first multi-task learning algorithm

The first algorithm we present is adapted from [TD13], and its goal is basically to learn a policy $\pi^\star_m$ for each of the M experts. It requires a prior distribution on experts, used to linearly combine the weight features, in the hope of getting a better policy $\pi^\star$. The second algorithm explained later tries to aggregate the experts in a more subtle way, by learning the opponents MDPs.

Data: g_Q: board features function (for Q-value functions),
Data: M, and a database $\mathcal{D}_m$ of demonstrations for each expert m.
Data: A prior $(e(m))_{m\in\{1,\dots,M\}}$ on the experts strength.
Data: A temperature β for the softmax ($\beta = 1$ works).
Result: $\pi^\star$ the aggregated optimal policy we learn.
`/* (For each expert, separately or in parallel) */`
for $m = 1$ *to* M **do**
  `/* Learn` $\pi^\star_m$ `from` $\mathcal{D}_m$ `the LSTD-Q algorithm */`
  Compute the function $l_m : w \mapsto l_m(w)$;
  Compute its gradient $\nabla l_m : w \mapsto \nabla l_m(w)$;
  Chose a starting point $w^{(0)}_m = [0, \dots, 0]$;
  `/* Optimize` l_m `with a 1-st order optimization method: */`
  $w^\star_m \leftarrow \mathrm{LBFGS}(l_m, \nabla l_m, w^{(0)}_m, \dots)$;
end
$w^\star = \mathbb{E}_m[w^\star_m]$, $Q^\star = g \cdot w^\star$ (expectation based on the distribution $e(m)$);
Result: $\pi^\star = \mathrm{softmax}_\beta(Q^\star)$ the aggregated optimal policy we learn.

Algorithm 1: Naive Multi-Task Learning Algorithm for Imperfect Oracles, with a prior on their strength.

It is important to notice that getting this $\pi^\star$ is the initial goal of all this framework, concretely what we really want is to be able to *play* the game.

But having this prior knowledge $e(m)$ seems in fact not realistic at all. We will present below three approaches to compute $e(m)$, first as the performance of expert m on its demonstrations, and then by using the Gaussian learning algorithm from [PD15b].

3.3 The LSTD-Q algorithm

For expert m, the goal of this algorithm is to find the weights w_m that fits as well as possible the demonstrations in $\mathcal{D}_m$. It boils down to maximize a log-likelihood, which can be written as a maximization problem $w^\star_m = \arg\max_w l_m(w)$, with objective l_m function:

$$l_m(w) \overset{\text{def}}{=} \frac{1}{|\mathcal{D}_m|} \sum_{d \in \mathcal{D}_m} \sum_{t=1}^{T_k} \Big\{ \beta g_Q(a_{d,m}^{(t)}, s_{d,m}^{(t)})^T w - \ln\big(\sum_{a' \in \mathcal{A}} \exp(\beta g_Q(a_{d,m}^{(t)}, s_{d,m}^{(t)})^T w) \big) \Big\} \quad (3)$$

This function is from [TD13, Eq. (4.18) Part. 4.2]. We chose to not details that part as proceed exactly the same way. Refer to their article [TD13] for more details. We only changed it by dividing it with the number i_m of demonstrations for expert m, $i_m \overset{\text{def}}{=} |\mathcal{D}_m|$, in order to have the same scale for the gradients even if one player has much more demonstrations. Indeed we want to stay at the order of values for weights w_m, and as we use an iterative method similar to an ascent of gradient. In practice we observed more consistent results after this division.

This function of w has the advantage of being *concave* (as a sum of affine terms, and a negation of $\log \operatorname{sum} \exp$ which is convex), and therefore the LSTD-Q algorithm boils down to calling an efficient concave maximization solver.

- *How big is this maximization problem?*
  The variable is w (w_Q in practice), and it lives in the space $\mathbb{R}^F$. In practice for n-by-n Tic-Tac-Toe, $F'_n = O(n^2$ so it grows linearly in the size of the board. The other parameters do not influence the dimension of the maximization problem, but the computational cost of each evaluation of l_m (and ∇l_m) is about $O(|\mathcal{D}_m| F_n \max_k T_k)$.
- *Is l_m it smooth?*
  Yes, and it is (at least) twice differentiable, so we can maximize it with standard first- or second-order methods, like a gradient ascend or Newton's method. Let us now compute the gradient of l_m:

$$\frac{\partial l_m}{\partial w_i}(w) = \frac{1}{|\mathcal{D}_m|} \sum_{d \in \mathcal{D}_m} \sum_{t=1}^{T_k} \Big\{ \beta g_Q(a_{d,m}^{(t)}, s_{d,m}^{(t)})_i - \big(\sum_{a' \in \mathcal{A}} \beta \nu_{a'} g_Q(a_{d,m}^{(t)}, s_{d,m}^{(t)})_i \big) \Big\} \quad (4)$$

$$\text{With } \nu_{a'} = \frac{\exp(\beta g_Q(a_{d,m}^{(t)}, s_{d,m}^{(t)})^T w)}{\left(\sum_{a' \in \mathcal{A}} \exp(\beta g_Q(a_{d,m}^{(t)}, s_{d,m}^{(t)})^T w) \right)}$$

  As expected, we recognize a well-known form for this weight: it is exactly a $\operatorname{softmax}$! This is not surprising because l_m comes the log-likelihood we try to maximize in π_m (as function of the weights $w_Q^{(m)}$). Note that the form of l_m and its gradient allow us to notice here a link with the *exponential families*, where the sufficient statistics are the vector $g_Q(s, a)$, and the parameters are w_Q.

In practice, we have implemented both the function and the gradientof l_m in a generic manner (accepting as many demonstrations as we want, for n-by-n Tic-Tac-Toe). For some more efficient methods, we need to differentiate one more time to get the Hessian, but that's a pain. Note that the formula for the Hessian starts to be long (and is not included here for sake space). In particular, it is hopeless to think of having a closed form or exact solution for this maximization problem.

We can use any concave maximization method (or conversely, convex minimization), like a gradient ascent (1-st order) or Newton's methods (2-nd order). In practice, we have used the GSL library and their convex minimizer toolbox. We have used the efficient and popular L-FBGS algorithm [LN89], which is a "clever" 1st-order method that make use of both l_m and ∇l_m, and we feed it a arbitrary starting point ($w_0 = [0, \dots, 0]$).
Now that we have made clear both l_m and ∇l_m, and the convex optimization method, our algorithm 1 is completely explicit.

3.4 Simulating random games against a fixed opponent

Instead of relying on a prior $e(m)$, once we have the weights $w_m^\star$, a way to evaluate the strength of our M linearly-represented experts is to make them play K games again a certain opponent π_0. It can be a random opponent, or an Bayesian opponent from [PD15b]. This way, we can count how many games each $w_m^\star$ won or lost (or got a draw), normalize these winning rates in order to get relative success rate $e(m)$, and finally use these rates to aggregate the weights and produce a final policy $\pi^\star$.

3.5 Trying to use temperatures to aggregate the experts

We also quickly present another approach trying to aggregate the experts. Following the interpretation of β as a temperature, we observe that we might constraint the weights vector w_m of each expert to have a constant norm, let say $\|w_m\|_2 = 1$ for the l^2 norm. We can still rely on the same maximization solver, but instead of just producing a w_m, we also compute the β_m parameter as $\beta_m = \|w_m\|_2$, and return both β_m and $w'_m = w_m/\beta_m$ the normalized weight vector.
This way, every expert has a weight vector with the same norm (l^2 or any norm). And we have a *different* temperature β_m for expert m, not a constant one $\beta = 1$. Following the intuition that a huge β_m is for a "hot" expert that exploit a lot (i.e. us confident about their weights), and small β_m are for a "cold" expert which has to randomly explore a lot because he is a bad player. Then we can normalize the $\beta_{1..M}$ vector, and get a probability distribution on players. However it is very hard to justify why this would work, and in practice after implementing it, we observed nothing significant: on several runs, we generated demonstrations, and learned two $w_1^\star, w_2^\star$.
The norms of weight vector representing a purely random vector was usually smaller than the norm of the minimax weight vector (for 3-by-3 Tic-Tac-Toe), but this observation varied a lot for each run (high variance, higher for the random player), and their average was $\hat{\beta}_{\text{random}} = 53.7$ vs $\hat{\beta}_{\text{minimax}} = 30.2$ (on 100 runs), and 85% of the runs give a higher temperature for the random vector. Unfortunately, these results tends to be against the intuition exposed above, where a cold β would imply a small $e(m)$, so we have not even tried to aggregate multi-expert weights or policies based on their temperatures.

3.6 Extending Pengkun's algorithm for multi-expert

One of the ideas we wanted to incorporate to our work was the Coherent Inference algorithm from [PD15b, PD15a]. In [PD15b], a probabilistic search descent algorithm is proposed that tries to approximate the optimal value of the game (i.e., the best result – win or loss – that can be obtained starting at each node if both players play optimally). This search algorithm splits the optimal value into two parts, an on-policy g score that represents a node's value under random roll-outs and a Δ value that represent approximates a node's value based on its position in the game tree.

This algorithm approximates Expectation-Propagation by operating a succession of descent in the game tree. At each descent, expectation propagation is performed on a small subset of visited nodes around the game tree following an exploration policy; when an unvisited node is reached, its value is approximated by a random roll-out before being added to the set of visited nodes. [PD15b] uses uniform probability amongst all successor states when performing this roll-out to model the fact that the opponent's actions are unpredictable.

One idea is to use instead a prior on the opponent's distribution in this roll-out (π_m^O); which would bias the score in such a way that the actions that our prior says are more likely for our opponent are more heavily explored. Following this idea, we can first learn an approximate feature weight for each expert's opponents, as in 1 but with a symmetric point of view, then uses the inference algorithm from [PD15b] from the current state to give us an estimation of the best move we can make against this biased opponent distribution. Using the game-theoretical principles of min-max, we will then choose amongst those estimate

the one with (stochastically) worse outcome for us, and let the inference algorithm tell us our (stochastically) best move against this worst opponent.

Note that this correspond to estimating transition function for the MDP associated with the game: indeed, the opponents' actions are seen by us as a transition from one state to another. Thus, the symmetry in the game for both players is what allows us to perform this "reversal" and learn an expert's policy not for ourselves, but for our opponent, in order to better learn how to play against them.

This yields the algorithm outlined in algorithm 2, which uses both the LSTD-Q step from [TD13] outlined in subsection 3.3, and the inference algorithm from [PD15b] as building blocks.

Note that we only use the G-inference algorithm since [PD15b] found that using the v-value inference (incorporating the Δ value) was not a significant improvement.

Data: g_Q: board features function
Data: M, and a database $\mathcal{D}_m$ of demonstration for each opposing expert m
`/* 1. Off-line learning step */`
for $m = 1$ *to* M **do**
  Learn $\pi^{\star,O}_m$ for the opposing player from $\mathcal{D}_m$ using LSTD-Q.
end
`/* 2. Play step (on line while playing) */`
Data: b: the current board state,
for $m = 1$ *to* M **do**
  `/* Use the coherent inference algorithm from [PD15b] */`
  Learn the G values starting from b using $\pi^{\star,O}_m$ for the opponent's distribution;
  Sample r_m from the distribution of G at b;
  for $a \in A$ **do**
    Sample c_a from the distribution of G at $b + a$ (state after playing move a).
  end
  Let a_m be $\arg\max_a c_a$ be the best answer to $\pi^{\star,O}_m$;
end
Let $m^\star$ be $\arg\min_m r_m$ be the strongest opponent;
Let $a^\star$ be $a_{m^\star}$ the best answer to the strongest opponent (minimax idea).

Algorithm 2: Multi-Task Algorithm for Imperfect Opposing Experts

4 Implementation and experimental results

This section presents our implementations; mainly done on 3-by-3 and 4-by-4 Tic-Tac-Toe. For our implementation, we chose to work in `C++`, in order to be coherent with the previous works by Liu [PD15a] and Dimitrakakis [Dim15] (both available on GitHub). Our code and our project has also been published with an open-source license, on Bitbucket.

About our experiments, we first wrote the n-by-n Tic-Tac-Toe rules, the "board state to board features" function g, and then both a fully-random player and an optimal minimax tree-search player for 3-by-3 (to have a not-so-bad adversarial policy). We can also combine the two with a certain probability, to have a "drunk" player, this helps us to have not only two policies π_{random} and π_{minimax}, but a family of (random) policies $\pi_\varepsilon = (1-\varepsilon)\pi_{\text{minimax}} + \varepsilon\pi_{\text{random}}$.

Then we implemented algorithm 1, relying on a "blackbox" optimizer for the maximization step using L-BFGS from the GSL library. And finally, we have been able to implement our second algorithm 2. A short sum-up of the experimental results is given below.

4.1 First experiment, reproducing and extending [TD13]

First, we chose to re-implement the first algorithm from [TD13] (to learn $\pi^\star_m$ from $\mathcal{D}_m$), and we wanted to reproduce the main results they obtained.

For 3-by-3 Tic-Tac-Toe, we implemented three kind of players: an optimal minimax tree-search, a fully random player, and a mixture of the two, called a "drunk" player as above. We have been able to use any combination of two players, e.g. minimax vs minimax or drunk vs random, to generate a database of demonstrations, and we produced 100 demonstrations for each. Note that this first part does not uses g_Q nor weights.

Then we use the algorithm 1 to learn the feature weight vectors $w^\star_m$ (in our code, we have a function `FitExpert` for this purpose, accepting one or more demonstrations $\mathcal{D}_m$) for each of these 9 combinations (expert m, opponent o_i, for $m = 1..3$ and $i = 1..3$). After that, we get the policy $\pi^\star_{m,i}$, and we have to evaluate how well the LSTD-Q step has been able to learn to represent linearly the three players (the optimal minimax, random and drunk player).

How to evaluate this? We have both the exact player and the linearly-represented one, and the initial opponent, so we chose to simulate make K games, with $\pi^\star_{m,i}$ against its hard-coded opponent o_i (the same one). In practice, we were able to try $K = 10, 100, 1000, 10000$, as all this was extremely quick when playing against the purely random opponent, but only up to $K = 100$ against the minimax opponent because it is slower (as it is computational non-trivial to make this minimax tree search).
Below is display a sum-up of these experiments:

Player and opponent	Win rate	Lost rate	Draw rate
Optimal vs random (always winning)	100%	0%	0%
Learning from 100 "optimal vs **optimal**" demonstrations			
Learned vs *random*	77%	7%	16%
Learned vs *minimax*	0%	0%	100%
Learning from 100 "random vs **optimal**" demonstrations			
Learned vs *random*	58%	28%	14%
Learned vs *minimax*	0%	92%	8%
Learning from 100 "random vs **random**" demonstrations			
Learned vs *random*	93%	3%	4%
Learned vs *minimax*	0%	0%	100%
Learning from 100 "optimal vs **random**" demonstrations			
Learned vs *random*	53%	28%	19%
Learned vs *minimax*	0%	77%	23%

Figure 3: Win, loss and draw rates for $K = 100$ games for several linearly-represented players, for 3-by-3 Tic-Tac-Toe, learned from 100 demonstrations.

This table confirms the use of LSTD-Q [TD13], as we see that we are able to learn linear representations of our 3-by-3 minimax optimal player or random player which behave similarly.

It is maybe simpler to understand these results on the next table 4. On the left, we made play the learned expert again a full-random policy (uniformly choosing its move), on the right, the linearly-represented expert (with $w^\star_m$) is tested against the optimal minimax policy. Therefore, the left table uses the win+draw rate, while the right one only accounts for the draw rate (remember that for 3-by-3 Tic-Tac-Toe, the minimax policy is optimal and ubeatable – one can only hope to obtain a draw against the "Opt." policy).

Both tables show four "win" rates, and each rate is evaluating the performance of the linearly-represented policy learned from 100 demonstrations, generated with optimal-vs-optimal, optimal-vs-random, random-vs-optimal, and random-vs-random (respectively):

Let us quickly explain what these results mean and why they are satisfying:

Player \ Opp.	Opt.	Rand.
Optimal	77%	93%
Random	58%	53%

(a) **Learned** *vs* Random.

Player \ Opp.	Opt.	Rand.
Optimal	100%	23%
Random	8%	0%

(b) **Learned** *vs* Optimal.

Figure 4: Combined **"Win"** % rate (learned from 100 demonstrations, tested on 100 games).

- Learning from optimal-vs-optimal produces a very efficient expert against the optimal policy (100%),
- And in general, learning from optimal-vs-x produces an efficient expert, both against the optimal and random policies (77%),
- Learning from optimal-vs-random produces a very efficient expert against the random policy (93%) but less efficient against the optimal policy,
- Learning from random-vs-optimal produces a "random" expert (but linearly-represented!), performing as well as a random policy against itself ($58\% \simeq 50\%$), but almost always loosing against the optimal policy (8%),
- And similarly, learning from random-vs-random produces a "random" expert, performing as well as a random policy against itself ($53\% \simeq 50\%$), but always loosing against the optimal policy (0%).

4.2 Second step, reimplementing [PD15a]

Then, we chose to pursue the work initiated in [PD15a], and we wanted to reproduce the main results announced in his thesis [PD15b]. Because of a lack of documentation, it was not obvious and not instantaneous, but we managed to obtain the same kind of performance [PD15b, Fig 4.3 p-28], for 3-by-3 but also 5-by-5 and 8-by-8 Tic-Tac-Toe. Apparently, our implementation was more optimized and more efficient, as we obtained the same results quicker when comparing with the code from [PD15a]. Due to space constraint, we prefered to not include any of these results here.

4.3 Last step, implementing our multi-expert algorithm 2

Finally, we implemented our multi-expert algorithm algorithm 2 using the previous algorithms as building blocks. As last experiments, it works for both 3-by-3 and higher-dimensional Tic-Tac-Toe games.

Let us present here one experiment designed to assess the quality of our algorithm 2. On 3-by-3 Tic-Tac-Toe, we learned using subsection 3.3 two experts for the second players, each over 100 runs, from:

- An optimal player *vs* a "drunk" player with $\varepsilon_1 = 0.3$ (second player),
- A "drunk" player with $\varepsilon_2 = 0.4$ *vs* a "drunk" player with $\varepsilon_3 = 0.2$.

Then, we built two experts using algorithm 2 for each of the opposing experts alone, as well as the two experts combined. Our witness expert is an expert using the regular inference algorithm from [PD15b].
Sample results from 1000 tests runs against an optimal player are reproduced on Figure 5, with performance expressed as a "draw percentage" (the optimal expert can never lose, so the best a player can achieve is to not lose). The results vary depending on the run and learned vectors for the opposing players.

As we can see, combining several experts (in blue) is usually improving performance over using a single expert – the presence of a good opposing expert is reducing the penalty from having a bad opposing expert (e.g. in run 2), and having several good opposing experts will usually improve over the using a single one of them (e.g. in run 3).
However, unfortunately, this method does usually not improve over the performance (although it does not loses much from it either) of simply using a simple implementation of

coherent inference from [PD15b] and is dependent on the performance of the opposing vector learned.

Player and opponent	Run 1	Run 2	Run 3	Run 4
Opposing Expert 1	25%	34%	37%	44%
Opposing Expert 2	34%	74%	27%	13%
Aggregated 1 and 2 Opposing Experts	29%	64%	41%	41%
Coherent Inference [PD15b] (average)	40%			

Figure 5: Draw % rate using different opposing experts.

---

5 Ideas for future perspectives

In this last section we list a few ideas about possible future development of the work presented here: other applications of our Bayesian algorithm, or additional theoretical developments.

5.1 Another game?

One interesting possible future work could be to apply (and maybe adapt) the algorithms and methods explained in this article to other 2-player (zero-sum) board game. We only focused on the n-by-n Tic-Tac-Toe, but there are many other games we could have chosen to study: chess, Go, Shogi, checkers, 4-in-a-row, and even "meta" Tic-Tac-Toe! [HK14] studied Shogi, while for instance Sebag's team at LRI (Orsay, France) worked on Go for years.

5.2 Other extensions?

Another direction could have been to extend the learning algorithms from 2-player to n-player games. Examples of such games include generalized n-chess, 4^+-player mahjong, or Chinese checkers (2 to 6 players). But we could even think of applying these algorithms to more general games, there is no reason to only consider *board* games. However, it seems really harder to apply an algorithm the ones presented here to a game which is not turn-based (real time, or more complicated), not full-knowledge (e.g. poker) or hard to represent with linear features[7].

5.3 Use real-world data, instead of (randomly) generated demonstrations

Of course, an interesting direction of work we have not been able to explorewas to use real-world data instead of generated trajectories [PD15b]. The most known example of "interesting" game for which real-world trajectories databases are available is chess. Countless websites offer a database of games, including two serious references: `www.chessgames.com`, and `database.chessbase.com`. With more time we could have implemented the chess rules, as we did for Tic-Tac-Toe, and a g function for the chess board (it is harder but exists) download a huge database of trajectories (e.g. 6 million from `database.chessbase.com`), and then learn from the database, either in a parallel way, or an on-line way (note that our algorithm would need an adaptation to become on-line). This idea is similar what was done by the famous IBM Deep Blue in 1996, with one of world's #1 super-computer to learn very fast before the game, and then to explore a pruned search tree very very quickly during the game. Ideally, a final output of our project

[7] Like a complex board games or a video games... Although the now famous Google DeepMind project on Atari games [MKS+13] tends to indicate they found a way to do that. Basically, they "watch" the TV screen as a low-dimension colored image, and then feed it to a huge image analysis deep-net – so, in a way, they do represent the game state as a finite-dimensional vector on which they apply linear (and non-linear) transformations.

could have been a usable chess computer player, with a GUI to play with etc (or for another game). But obviously, we did not have time to do any of that.

5.4 Use a non-linear kernel?

One of the restricting hypothesis done in this article is that we only try to learn board value functions that are linear combination of the board features, as done in [HK14, PD15b]. As it is suggested in the conclusion of [TD13], we could also try here to use a non-linear kernel to combine the board features, and hope for better performance. As we will model a more general function of the features, we can indeed hope for better results, but learning a non-linear kernel is in general harder than learning a linear function (the first issue is storing, as it not sufficient to just store the linear coefficients). *Note:* As we are both interested in understanding more about this idea and about kernel methods, we plan to follow Jean-Philippe Vert's course (*"Machine Learning with Kernel Methods"*) during the second semester.

---

6 Conclusion

The Reinforcement Learning framework as used here allows to numerically compute approximated optimal policies for (off-line) decision making. Basic RL algorithms allows to compute a good policy if we know how the system evolves (i.e. if the MDP is fully known if we chose to model by a MDP), like PI or VI. The Inverse RL framework does not assume to know the MDP anymore, and tries to *learn from demonstrations*, when the only thing we know about the dynamics of the system is only a sample of trajectories played by an expert. Introducing a prior knowledge on the experts relative strength seemed a clever way to make a real use of multi-expert trajectories, but finding a good prior independent from the demonstrations seems impossible, so we explained an approach to compute distribution on experts a posteriori. One goal of such aggregation is to at least perform as well as each expert, but it is only useful if it outperforms every experts. In practice, we have not been able to obtain satisfactory results on this last goal.

6.1 Our contribution and its limits

- We exposed a simple concave-maximization algorithm to compute $w_m^\star$ separately for each expert from the sample of demonstrations $\mathcal{D}_m$, in a totally parallelizable way, based on [TD13, DR12], and its real computation efficiency only depends on the numerical optimization library.
- We explained how to infer the most probable policy for each expert ($\pi_m^\star$), and then we justified how to combine them with a prior score on experts $e(m), m = 1..M$. This aggregation of M policies $\pi_m^\star$ is cheap in term of both memory and computation time, but having a good prior distribution on experts is not realistic.
- So we proposed an intuitive way to test the performance of the aggregated policy $\pi^\star$, based on evaluation against a fixed opponent, and another way by computing their temperature (but it fails).
- Finally, when trying to validate experimentally our second algorithm 2, the aggregated policy $\pi^\star$ proved to be about as efficient as that the best $\pi_m^\star$, but unfortunately we have not been able to significantly improve its performance.

Acknowledgments

We would mainly like to thank a lot Christos Dimitrakakis our project advisor, he replied quickly to our queries and provided useful insights. Thanks also to both professors Michal Valko and Alessandro Lazaric for their two interesting courses at the MVA Master program (Cachan, France). Thanks also to Émilie Kaufmann who agreed to evaluate us for the oral presentation of our project (see here for the slides, `http://lbo.k.vu/rl2016`).

Appendix

References

[Dim15] Christos Dimitrakakis (December 2015). *BeliefBox, a Bayesian framework for Reinforcement Learning (GitHub repository)*. URL `https://github.com/olethrosdc/beliefbox`, online, accessed 20.12.2015.

[DR12] Christos Dimitrakakis and Constantin A. Rothkopf (2012). *Bayesian Multitask Inverse Reinforcement Learning*. In *Recent Advances in Reinforcement Learning*, pages 273–284. Springer. URL `http://arxiv.org/abs/1106.3655v2`.

[DVWRT10] Finale Doshi-Velez, David Wingate, Nicholas Roy, and Joshua Tenenbaum (2010). *Non-parametric Bayesian policy priors for Reinforcement Learning*. *Neural Information Processing Systems Foundation (NIPS'10)*. URL `http://hdl.handle.net/1721.1/66107`.

[HK14] Kunihito Hoki and Tomoyuki Kaneko (2014). *Large-scale Optimization for Evaluation Functions with MiniMax Search*. *Journal of Artificial Intelligence Research*, 49:527–568. URL `http://www.jair.org/papers/paper4217.html`.

[KBB09] Wolfgang Konen and Thomas Bartz-Beielstein (2009). *Reinforcement Learning for Games: Failures and Successes*. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2641–2648. ACM. URL `http://doi.acm.org/10.1145/1570256.1570375`.

[LN89] Dong C Liu and Jorge Nocedal (1989). *On the Limited Memory BFGS Method for Large Scale Optimization*. *Mathematical programming*, 45(1-3):503–528.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller (2013). *Playing Atari with Deep Reinforcement Learning*. *arXiv preprint arXiv:1312.5602*. URL `http://arxiv.org/abs/1312.5602`.

[PD15a] Liu Pengkun and Christos Dimitrakakis (June 2015). *Code for Implementation and Experimentation of Coherent Inference in Game Tree (GitHub repository)*. URL `https://github.com/Charles-Lau-/thesis`, online, accessed 20.12.2015.

[PD15b] Liu Pengkun and Christos Dimitrakakis (June 2015). *Implementation and Experimentation of Coherent Inference in Game Tree*. Master's thesis, Chalmers University of Technology.

[SB98] Richard S. Sutton and Andrew G. Barto (1998). *Reinforcement Learning: An Introduction*, volume 1. MIT Press, Cambridge, MA. URL `http://webdocs.cs.ualberta.ca/~sutton/book/the-book.html`.

[TD13] Aristide C. Y. Toussou and Christos Dimitrakakis (2013). *Probabilistic Inverse Reinforcement Learning in Unknown Environments*. *arXiv preprint arXiv:1307.3785*. URL `http://arxiv.org/abs/1307.3785v1`.

License and copyright?

Note that this article has **not** been published on any conference, journal or pre-print platform. It was just the result of a small research Master project.
