Agent Theory of Mind with Reinforcement Learning

Matthew LeBar and Arnav Agrawal

May 28, 2025

1 Introduction

One of the fundamental topics in Game Theory is how agents predict the actions of other agents. Classical solution concepts like Nash Equilibria rely on agents knowing each other's value over each outcome and assuming that everyone will act perfectly rationally. Of course, in many circumstances, neither of these holds. Thus, a natural question arises: How can we formally model agents' beliefs about other agents? The field of artificial intelligence has a long history of agents modeling other agents (AMOA)[3]. In this review, we focus specifically on agent modeling using Reinforcement Learning (RL). There are relevant techniques in two areas of RL: Multi-Agent Reinforcement Learning (MARL) and Inverse Reinforcement Learning (IRL). MARL provides the bridge between game theory and RL by (as the name suggests) modeling interactions between multiple reinforcement learning agents. However, within MARL, many models assume that agents cannot see each other's actions or can only implicitly model other agents. Thus, we review a few models of agents' representations of other agents in MARL, many of which make use of deep learning. In IRL, on the other hand, there is a less direct connection to game theory, but the core problem in IRL is learning the underlying values maximized by an artificial agent. This means that any IRL technique can be understood as an agent modeling another agent; thus, we summarize two fundamental IRL algorithms and discuss the specific application of IRL in apprenticeship learning.

2 Basics of RL

Before we discuss the theory of mind in Reinforcement Learning, it will be helpful to have some basic definitions and techniques in place. Definitions are drawn from Sutton and Barto or Albrecht et al.[11, 2]
.

2.1 Markov Decision Processes

Firstly, what is reinforcement learning? "**Reinforcement learning (RL)** algorithms learn solutions for sequential decision processes via repeated interaction with an environment [2]." To specify this formally, we understand sequential decision processes as **Markov Decision Processes (MDP)**. An MDP is built around a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a set of rewards $\mathcal{R} \subset \mathbb{R}$, and a sequence of time steps $t = 0, 1, 2, 3, ...$ We can think of $\mathcal{A}$ as a function with domain $\mathcal{S}$, $\mathcal{A}(S)$ gives the set of actions available in S. A **trajectory** of the MDP can be given by a sequence of the form

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$$

A trajectory from an initial state to a terminal state is called an **episode**. At each time step t, the current state is $S_t \in \mathcal{S}$, the agent selects $A_t \in \mathcal{A}(S_t)$, receives rewards $R_{t+1} \in \mathcal{R}$, and transitions to state $S_{t+1} \in \mathcal{S}$. Thus, to complete the characterization of our decision process, we need a transition function $\mathcal{T} : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ where $\mathcal{T}(s', r|s, a) = \Pr\{St = s', R_t = r|s = S_{t-1}, a = A_{t-1} = a\}$. What makes a decision process a *Markov* decision process is the assumption that s', r are determined only by s, a - in the words of Sutton and Barto, "the probabilities given by p completely characterize the environment's dynamics". Specifically, this means we don't need any information about earlier states, so the previous state s is said to have the **Markov Property** [11]. We now know what a sequential decision process via repeated interaction with an environment looks like; let's examine several solution concepts.

2.2 Value Functions

Typically, solutions in RL find a way to represent the **value** of a particular state so that the agent's behavior is optimized by behaving greedily at that particular state. This means that the value at a particular state must be a function of both the immediate rewards of actions and the expected rewards in future states based on actions at the current state. This can be thought of as a Dynamic Programming problem. We apply a **discount rate** γ to future rewards; thus, with a **policy** π that maps each state to a probability distribution over the agent's possible actions at that state, we have

$$V^\pi(s_t) = \sum_{a \in \mathcal{A}(s_t)} \pi(a|s_t) \sum_{s_{t+1} \in \mathcal{S}, r_{t+1} \in \mathcal{R}} \mathcal{T}(s_{t+1}, r_{t+1}|s_t, a)[r_{t+1} + \gamma V^\pi(s_{t+1})]$$

This recursive formula is called the Bellman Equation.[2] The optimal policy solves this equation. Since optimal policies under this model pick just one action, we generally assume that each policy picks a unique action, so think of it as mapping to that action rather than a probability distribution. We also drop π where it is clear from context what the policy is.

There are two basic methods for calculating the value function and deriving an optimal policy from it: **Value Iteration** and **Policy Iteration**. This presentation follows Sutton and Barto chapter 4.[11] In Value Iteration, we initialize arbitrary estimates of $V(S)$ for all $s \in \mathcal{S}$, then loop through states, updating each state's estimate:

$$V(s) \leftarrow \max_a \sum_{s',r} \mathcal{T}(s', r|s, a)[r + \gamma V(s')]$$

This is repeated until the estimates converge, at which point for all $s \in \mathcal{S}$ we set

$$\pi(s) \leftarrow \arg\max_a \sum_{s',r} \mathcal{T}(s', r|s, a)[r + \gamma V(s')]$$

Thus we first calculate the value function which allows for greedy action selection, then greedily select actions.

In Policy Iteration, we bounce between updating our value function based on the current policy (*policy evaluation*), and updating our policy based on the current value function (*policy improvement*). We start by initializing V and π arbitrarily for all $s \in \mathcal{S}$. Then, until our policy has stabilized, policy iteration will first evaluate the policy by looping through each s and updating:

$$V(s) \leftarrow \sum_{s',r} \mathcal{T}(s', r|s, \pi(s))[r + \gamma V(s')]$$

Once this has converged, it improves the policy by updating π greedily:

$$\pi(s) \leftarrow \arg\max_a \sum_{s',r} \mathcal{T}(s', r|s, a)[r + \gamma V(s')]$$

Then, if this round of updates has changed the policy at all, the algorithm returns to the policy evaluation step. Thus this is quite similar to value iteration, only instead of saving the policy update until the end of value estimation, it's a part of our iteration.

Note that both of these methods required models of our function $\mathcal{T}$. We next turn to TD Learning, which implicitly learns a bootstrapped estimate for the transition function through the agent-environment interaction process. With these in hand we will be ready to turn to the game theory concepts and terminology needed for MARL.

2.3 TD Learning

In **TD** (Temporal Difference) **Learning**, to estimate the value of state s_t at time t under policy π, the agent observes their reward at consecutive states, and uses that to estimate the discounted reward for s_t. The simplest form of TD is TD(0), where only one consecutive state is visited for the estimate. With an update step size of α and discount rate γ, our update rule is then

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

We can think of $r_{t+1} + \gamma V(s_{t+1})$ as a target value for $V(s_t)$ so that $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is akin to the error of our estimate. Thus to do a full TD(0) estimate, we initialize our value estimates arbitrarily, then iterate through a full episode, at each state s_t where $t \neq 1$ estimating the value of s_t with r_{t+1} and $V(s_{t+1})$. Note that we do not need any estimates of transition probabilities; the value is learned through playing through the MDP, and thus is bootstrapped.[11] Note TD(0) as described only gives us a policy evaluation approach, but not an approach to policy improvement. To perform policy improvement with TD Learning, we use the **Q-function**, a generalization of the value function. The Bellman Equation for the Q-function (treating $\pi(s)$ as an action rather than a probability distribution) is

$$Q^{\pi}(s_t, a) = \sum_{s_{t+1} \in S} \mathcal{T}(s_{t+1}, r | s_t, a)[r + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1}))]$$

There are then two main methods for estimating a Q-function and using it to find an optimal policy: SARSA and Q-learning.

2.3.1 SARSA

SARSA is a technique for learning the optimal policy while acting by that policy. We include some small probability of picking from the actions uniformly, rather than picking the action given by the policy. Let α be a step size parameter and ϵ a the parameter that gives the likelihood of choosing a random action instead of the policy's action. In the algorithm, when we select an action, we directly pick $\arg\max_a Q(s, a)$ with probability $1-\epsilon$, and pick uniformly among all actions (including the argmax one) probability ϵ. This is called **epsilon-greedy** action choice. As we can see, the basic idea is to use TD(0) to estimate the

Algorithm 1 SARSA[11]

```
Initialize $Q(s,a)$ arbitrarily for all $s \in S$, $a \in \mathcal{A}(s)$, except if $s$ is terminal
$Q(s,a) = 0$ for any $a$
for Each episode do
    Initialize $s$
    Select $a$ as specified above
    while $s$ is not terminal do
        Take action $a$, observe $r, s'$
        Choose $a'$ given $s'$ as above
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
        $s \leftarrow s', a \leftarrow a'$
    end while
end for
```

values of each state action pair, and use epsilon-greedy action choice to balance **exploring** (taking new actions) and **exploiting** (taking actions believed to be optimal).

2.3.2 Q-Learning

Q-learning is very similar to SARSA in that it uses TD estimates to update the Q function. The only difference is it does not use the actual action taken at s_{t+1} to estimate the value at s_t - rather, it always uses the *optimal* action at s_{t+1}. Thus Q-learning is considered off-policy, since the policy it uses to estimate the Q function is not the same as the policy it uses to choose actions.

Algorithm 2 Q-Learning[11]

```
Initialize $Q(s,a)$ arbitrarily for all $s \in S$, $a \in \mathcal{A}(s)$, except if $s$ is terminal $Q(s,a) = 0$ for any $a$
for Each episode do
    Initialize $s$
    while $s$ is not terminal do
        Choose $a$ given $s$ as above
        Take action, observe $r, s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$
    end while
end for
```

2.3.3 Deep Q-Learning

One technique for Q-Learning is to use deep learning to use the Q function. This is particularly helpful when one wants the Q function to respond to features of the underlying state. The formulation in the previous section evaluates each state completely separately, but in many applications of Q-learning (e.g., learning to play video games), there are a very large number of states bearing structural similarities to one another, and a neural net can learn how those those structural similarities affect the value of a state. Thus, rather than directly updating the Q value as above, the difference $[r + \gamma \max\limits_{a'} Q(s',a')] - Q(s,a)$ (or its square) gives the loss for a neural network called a **Q-Network**, and it uses gradient descent to learn the appropriate Q function. However, training the Q function while also using it to estimate a target can cause wild fluctuations in the Q function that make it difficult to converge to an optimal estimate. A typical solution is to make a frozen copy of the neural net every so often, and use the parameters from that to provide the target $r + \gamma \max\limits_{a'} Q(s',a')$ value.[2] In general, in this literature review we will treat the learning of neural networks as a black box. Thus, while we might discuss the architecture of a neural network or the intuition behind it, we do not always discuss details of the loss function, the computational specifics of the layers of the net, or information on backpropagation and stochastic gradient descent, except insofar as it is helpful for understanding AMOA.

2.4 Policy Gradient Descent: A2C

A common way to test techniques for AMOA in MARL is by extending a **Policy Gradient** algorithm named **Advantage Actor Critic (A2C)**. Our presentation of A2C follows Albrecht et al., chapter 8 section 2.[2] In policy gradient algorithms, rather than learning a value function and then making a policy by choosing actions greedily, the policy is learned directly. Thus the output of the neural network will be a probability distribution over the possible actions. A2C does this with what is called an **actor-critic** algorithm that trains a network (the actor) to learn the policy alongside a network to learn the value (the critic). A2C uses a function called the **advantage function**:

$$Adv^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Intuitively, this is the difference between taking action a at state s and then following policy π, and simply following policy π at s and onwards. To compute the advantage, Q is estimated as in SARSA, and the critic is trained to output the value function using MSE with SARSA-style bootstrapped estimates as targets. Then the actor model trains to maximize the advantage. The full algorithm can be seen in Figure 1.

3 Game Theory in MARL

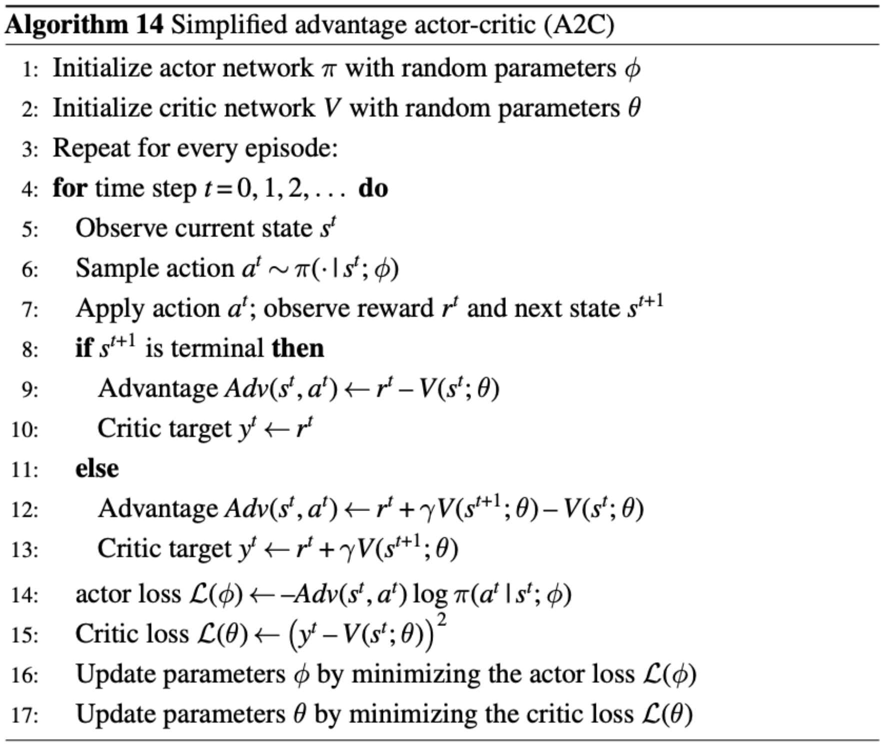
Before finally turning to MARL, we need a formal framework that allows us to unite reinforcement learning and game theory. **Stochastic games** are the key idea. Stochastic games combine the game theoretic notion of rewards as a function of a set of actions from multiple agents with the notion of an underlying environmental state from RL. Thus, Albrecht, Christiano, and Schäfer define stochastic games as consisting of:

- A set of agents or players $I = 1, ...n$
- A finite set of states S with a subset of terminal states $\overline{S} \subset S$
- For all $i \in I$:
  - A finite set of actions A_i
  - A reward function $\mathcal{R}_i : S \times A \times S \mapsto \mathbb{R}$, where $A = \prod_{i=1}^{n} A_i$
- A transition probability function $\mathcal{T} : S \times A \times S \mapsto [0,1]$ such that

$$\forall s \in S, a \in A, \sum_{s' \in S} \mathcal{T}(s,a,s') = 1$$

- An initial state distribution $\mu : S \mapsto [0,1]$ such that

$$\sum_{s \in S} \mu(s) = 1 \text{ and } \forall s \in \overline{S} : \mu(s) = 0$$



Algorithm 14 Simplified advantage actor-critic (A2C)

```
1: Initialize actor network $\pi$ with random parameters $\phi$
2: Initialize critic network $V$ with random parameters $\theta$
3: Repeat for every episode:
4: for time step $t = 0, 1, 2, \ldots$ do
5:     Observe current state $s^t$
6:     Sample action $a^t \sim \pi(\cdot \mid s^t; \phi)$
7:     Apply action $a^t$; observe reward $r^t$ and next state $s^{t+1}$
8:     if $s^{t+1}$ is terminal then
9:         Advantage $Adv(s^t, a^t) \leftarrow r^t - V(s^t; \theta)$
10:        Critic target $y^t \leftarrow r^t$
11:    else
12:        Advantage $Adv(s^t, a^t) \leftarrow r^t + \gamma V(s^{t+1}; \theta) - V(s^t; \theta)$
13:        Critic target $y^t \leftarrow r^t + \gamma V(s^{t+1}; \theta)$
14:    actor loss $\mathcal{L}(\phi) \leftarrow -Adv(s^t, a^t) \log \pi(a^t \mid s^t; \phi)$
15:    Critic loss $\mathcal{L}(\theta) \leftarrow \left(y^t - V(s^t; \theta)\right)^2$
16:    Update parameters $\phi$ by minimizing the actor loss $\mathcal{L}(\phi)$
17:    Update parameters $\theta$ by minimizing the critic loss $\mathcal{L}(\theta)$
```

Figure 1: A2C Algorithm [2]

Thus, a stochastic game is like an RL MDP, but modified to include multiple agents. Specifically, 1) the transition and reward functions depend on the full set of agent actions, not just one agent and 2) each agent has its own reward function.
Often in MARL, it is assumed that each player has only a partial view of their environment and of other agents. This model is called a **Partially observable stochastic game**. A partially observable stochastic game is a stochastic game along with, for all agents $i \in I$ [2]:

- A finite set of observations O_i
- An observation function $\mathcal{O}_i : A \times S \times O_i \mapsto [0, 1]$ such that

$$\forall a \in A, s \in S : \sum_{o_i \in O_i} \mathcal{O}_i(a, s, o_i) = 1$$

Observations could be defined in many different ways, but typically they will comprise some subset of the state and action at the latest time. We designate agent i's observation at time t as o_i^t. This completes our unified framework for MARL; Christiano, Albrecht and Schäfer provide a table (copied here in figure 2) to translate between RL and game-theoretic (GT) terminology.

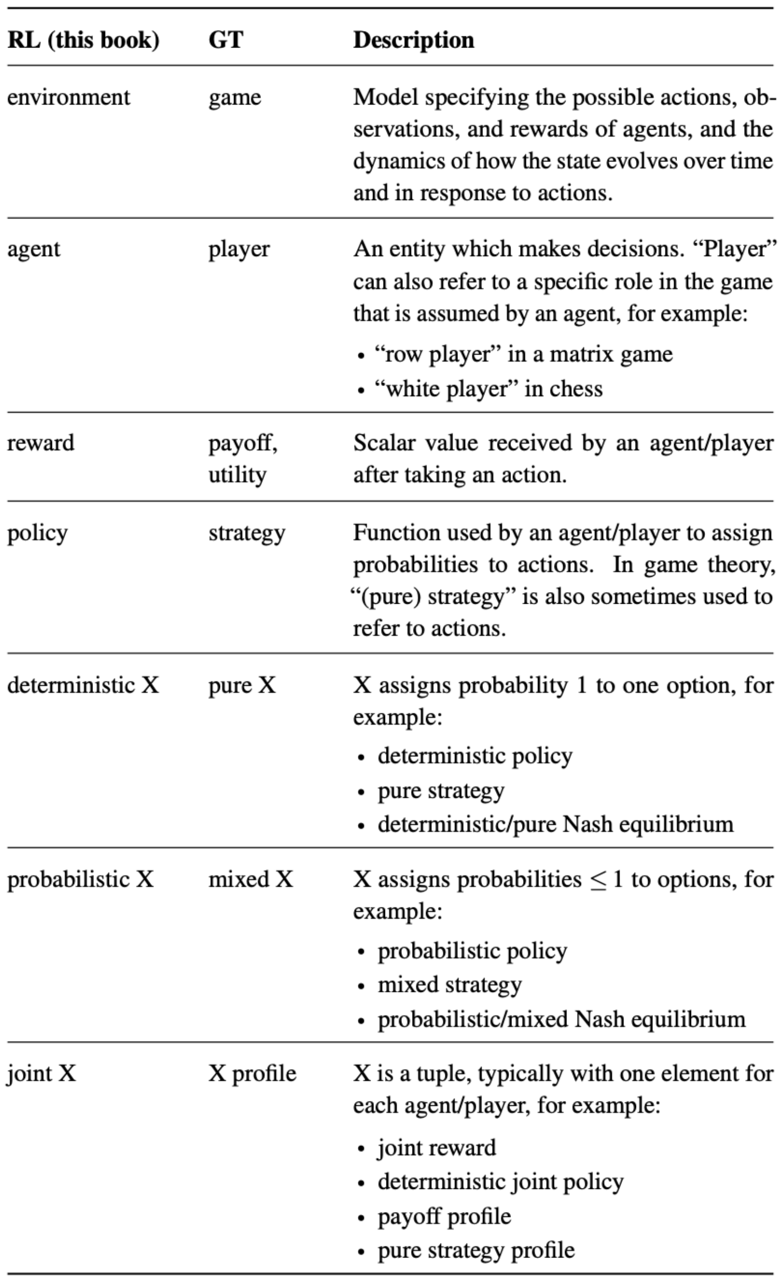
4 Agents Modeling Agents in MARL

4.1 Fictitious Play and JAL

Now we can finally turn to agents modeling other agents AMOA in MARL. We start with the simplest way this is done: each agent models the other agents with a probability distribution $\hat{\pi}_j$ over their actions, based on the play history (either previous actions within the episode, or actions within previous episodes). We discuss this approach following Albrecht et al..[2] Thus, if player j has played action a_j $C(a_j)$ times, every other agent will model the probability of agent j taking action a_j as

$$\hat{\pi}_j(a_j) = \frac{C(a_j)}{\sum_{a'_j \in A_j} C(a'_j)}$$

This is a very rudimentary form of AMOA, as agents do not attempt to represent anything about other agents' reward functions.
There are two basic MARL algorithms in which this is used; in one, **Fictitious Play**, each agent simply plays their best response given that they expect all other agents to act according to the $\hat{\pi}_j$ model. This is used in non-repeated normal form games, where probabilities are based on previous episodes.
More general is **Joint-Action Learning (JAL)**, which is combines Q-Learning with fictitious play, and can be used for stochastic games. To extend fictitious play to stochastic games, our probability distribution will include information



RL (this book)	GT	Description
environment	game	Model specifying the possible actions, observations, and rewards of agents, and the dynamics of how the state evolves over time and in response to actions.
agent	player	An entity which makes decisions. “Player” can also refer to a specific role in the game that is assumed by an agent, for example: • “row player” in a matrix game • “white player” in chess
reward	payoff, utility	Scalar value received by an agent/player after taking an action.
policy	strategy	Function used by an agent/player to assign probabilities to actions. In game theory, “(pure) strategy” is also sometimes used to refer to actions.
deterministic X	pure X	X assigns probability 1 to one option, for example: • deterministic policy • pure strategy • deterministic/pure Nash equilibrium
probabilistic X	mixed X	X assigns probabilities ≤ 1 to options, for example: • probabilistic policy • mixed strategy • probabilistic/mixed Nash equilibrium
joint X	X profile	X is a tuple, typically with one element for each agent/player, for example: • joint reward • deterministic joint policy • payoff profile • pure strategy profile

Figure 2: Translation between RL and GT Vocabulary [2]

about state. Thus we now have (if $C(s, a_j)$ is the number of times agent j takes action a_j in state s)

$$\hat{\pi}_j(a_j|s) = \frac{C(s, a_j)}{\sum\limits_{a'_j \in A_j} C(s, a'_j)}$$

Again each agent $i \neq j$ models j the same way. Each agent i also has a q function $Q_i : S, A \mapsto \mathbb{R}$. Note this takes as an argument the *profile* of actions, not just the action of agent i. It is helpful to define a function AV_i that gives the expected return for agent i at state s taking action a_i:

$$AV_i(s, a_i) = \sum_{a_{-i} \in A_{-i}} Q_i(s, \langle a_i, a_{-i} \rangle) \prod_{a_j \in a_{-i}} \hat{\pi}_j(a_j|s)$$

Now we are in position to see our algorithm for JAL with Agent Modeling (JAL-AM). This algorithm specifically controls agent i, and chooses its action epsilon greedily. Thus, with probability ϵ it will choose among the actions with uniform probability, and otherwise at time t will choose $a^t = \arg\max_{a_i} AV_I(s_t, a_i)$ (we use superscript for time instead of subscripts since we use subscripts to identify agents). This provides a basic method integrating AMOA with gameplay;

Algorithm 3 JAL-AM[2]

```
Initialize $Q(s,a)$=0 for all $s \in S$, $a \in A$
Initialize $\hat{\pi}_j$ uniformly for all $j \neq i$
**for** Each episode **do**
    **for** $t = 0, 1, 2, ...$ **do**
        Observe current state $s_t$
        Choose action $a_i^t$ as specified above
        Observe joint action $a^t = (a_1^t, a_2^t, ...a_n^t)$, reward $r_i^t$, next state $s^{t+1}$
        **for** $j \neq i$ **do**
            Update $\hat{\pi}_j$ with $(s^t, a_j^t)$
        **end for**
        $Q_i(s^t, a^t) \leftarrow Q(s^t, a^t) + \alpha[r_i^t + \gamma \max_{a'_i} AV(s^{t+1}, a'_i) - Q_i(s^t, a^t)]$
        $s \leftarrow s'$
    **end for**
**end for**
```

it is a highly simple form of AMOA, but unlike many MARL algorithms, it involves agents *explicitly* representing other agents. Any effective MARL algorithm would have to at least implicitly represent other agents, but it might not be in a form that allows for extracting representations of agents from the environment as a whole, or for extracting the representation of an individual from the representation of a group of agents. Thus, fictitious play and JAL largely serve as illustrative examples for us to see what explicit AMOA looks like in MARL

4.2 Deep Learning Opponent Models

We have seen already in section 2.3.3 how deep learning can be used to learn a Q-function; He et al. extend this to MARL by decomposing the neural net learning the Q-function into two modules, N_Q and N_o.[6] They dub the result Deep Reinforcement Opponent Net (DRON). N_Q is a Q-network that learns Q-values with respect only to state action pairs *without* information about opponent strategies; N_o only models the expected behavior of opponents. These are then combined to produce Q-values appropriately sensitive to opponent strategies. N_Q takes as input features of the state ϕ^s and learns an embedding of them a hidden (i.e., not observable from "outside" the neural network) representation space h^s, while N_o takes as input features of the opponent (note their model has a singular opponent) ϕ^o and learns to embed them in a hidden representation space h^o.

He et al. have two variants of DRON based on how the two modules are combined: DRON-CONCAT and DRON-MOE. As the name suggests, DRON-CONCAT combines the modules through concatenation: elements of h_s and h_o are concatenated and sent as input to a final neural net layer that outputs the Q values. In DRON-MOE, a Mixture-of-Experts model is used to model Q-values without opponent information. The Mixture-of-Experts model is an ensemble method, meaning it combines weighted predictions of the Q values from several networks. This allows DRON-MOE to explicitly marginalize out the effects of opponents strategies on Q-values; in this architecture, ϕ^o is combined with ϕ^s to produce h^o, which is passed into a further layer that produces the weights. This allows the weights to serve as information about the opponent's strategy, so that each expert Q_i can be understood as giving Q values conditioned on information about opponent strategies:

> Unlike DRON-CONCAT, which ignores the interaction between the world and opponent behavior, DRON-MOE knows that Q-values have different distributions depending on ϕ^o; each expert network captures one type of opponent strategy. [6]

See Figure 3, copied from their paper for a side-by-side of the two architectures. He et al. also include modifications both architectures to include an output from h^o so that they can train their model on the accuracy of its predictions about opponent behavior (this architecture is quite similar to the ones discussed in the next section).

He et al. performed artificial experiments with these architectures two settings: one a model of soccer and one a model of Quiz bowl, a quiz game. They use a Deep Q-Learning model DQN that does not separate opponent and model representations as a baseline. In soccer, both DRON models had much stabler learning curves (see figure 4) than DQN and outperform it (see figure 5). See figures Interestingly, varying the number of experts between 2 and 4 didn't seem to affect the outcome much, and directly supervising the representation of opponent strategies improved DRON-CONCAT but not DRON-MOE.

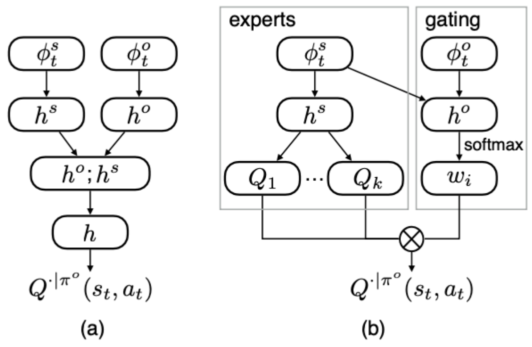


Figure 1. Diagram of the DRON architecture. (a) DRON-concat: opponent representation is concatenated with the state representation. (b) DRON-MoE: Q-values predicted by K experts are combined linearly by weights from the gating network.

Figure 3: DRON Architectures [6]

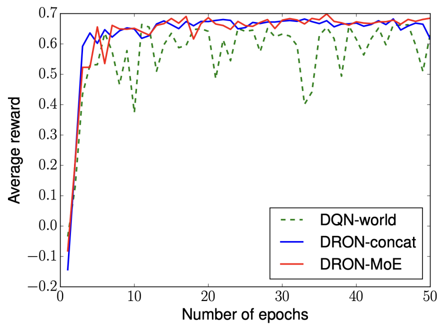
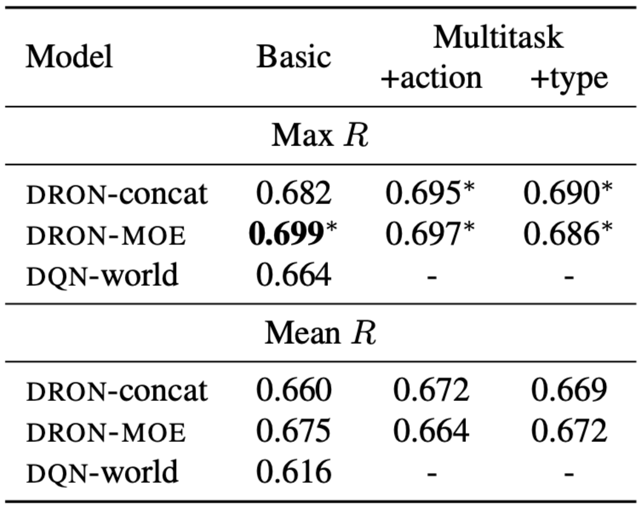


Figure 4: Learning Curve For DRON vs DQN[6]



Model	Basic	Multitask +action	Multitask +type
Max R			
DRON-concat	0.682	0.695*	0.690*
DRON-MOE	**0.699***	0.697*	0.686*
DQN-world	0.664	-	-
Mean R			
DRON-concat	0.660	0.672	0.669
DRON-MOE	0.675	0.664	0.672
DQN-world	0.616	-	-

Figure 5: Performance in DRON vs DQN. The multitasking columns evaluate the two architectures with additional supervision on the opponent models, either predicting their action or their type (offense or defensive)[6]

In Quiz Bowl, players want to be the first to buzz to answer a question. This is where opponent modeling comes into play:

> To test depth of knowledge, questions start with obscure information and reveals [sic] more and more obvious clues towards the end... Therefore, the buzzing model faces a speed-accuracy tradeoff: while buzzing later increases one's chance of answering correctly, it also increases the risk of losing the chance to answer. A safe strategy is to always buzz as soon as the content model is confident enough. A smarter strategy, however, is to adapt to different opponents: if the opponent often buzzes late, wait for more clues; otherwise, buzz more aggressively.[6]

In this experiment, they used both a DQN-world model, which does not disentangle opponent features from world features and a DQN-self model that does not include opponent features at all. Both DRON models outperform both baselines, with DRON-MOE outperforming DRON-CONCAT. Again, DRON-MOE did not improve performance with direct prediction supervision. Their work provides a convincing demonstration that engineering a Q-network to explicitly model opponent strategies can have significant performance payoffs. Albrecht et al. offer a similar model, extending JAL to use learned Q-functions and separately learned representations of opponent policies.[2] These also outperformed versions of Q-learning that did not explicitly separate opponent strategies from the environment in artificial experiments.

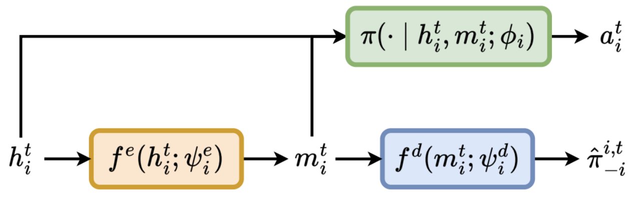


Figure 6: Representation of Autoencoder Architecture[2]

4.3 Learning Representations of Policies with Autoencoders

We can also approach learning representations of opponents in a way that is agnostic as to how those representations are used; the previous representation models were tailored for specific algorithms. This can be accomplished with another technique from machine learning: a type of artificial neural network called an **autoencoder**. Our basic exposition follows Albrecht et al. chapter 9, section 6.3.[2] In general, an autoencoder consists of encoder neural network f^e and a decoder neural network f^d - see Figure 6. In the particular application of learning to model opponent strategies, the autoencoder is trained by taking as input an observation history at a time h_i^t for some agent i at time t. The encoder network will output a representation m_i^t from this input that is then passed to the decoder network, which then provides a model predicting all other agent actions: $\hat{\pi}_{-i}^{i,t} = f_d(m_i^t)$. This can be trained based on the accuracy of the predictions using cross-entropy loss, a common loss function for predictions in machine learning. The resulting model $\hat{\pi}$ can then be used to condition an agent's choice of action in any MARL learning algorithm - it could be Deep Q-Learning as in the previous section, JAL as in the first section, or other MARL algorithms not discussed in this review. Albrecht et. al extend A2C with this structure. They test the resulting model in an artificial foraging environment that requires agent cooperation, and they mix in agents that learn with the standard A2C algorithm without explicitly learning representations of opponents. The agents using the model with the autoencoder significantly outperformed the agents without the autoencoder - see figure 7. This experiment is more sophisticated than the others discussed in this section, in that the agents being modeled are actively learning their policies; in all other experiments in this section, autoencoder networks are trained on agents whose policies have already been determined.

Grover et al. use a similar model, but their model only has two agents per episode (though many more agents across all episodes).[5] This allows them to augment the loss function so that it distinguishes between players - they do this by including a term in the loss that encourages representations of a player's strategy in some episode to be closer to a representation of that same

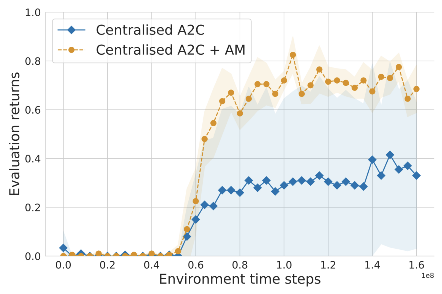


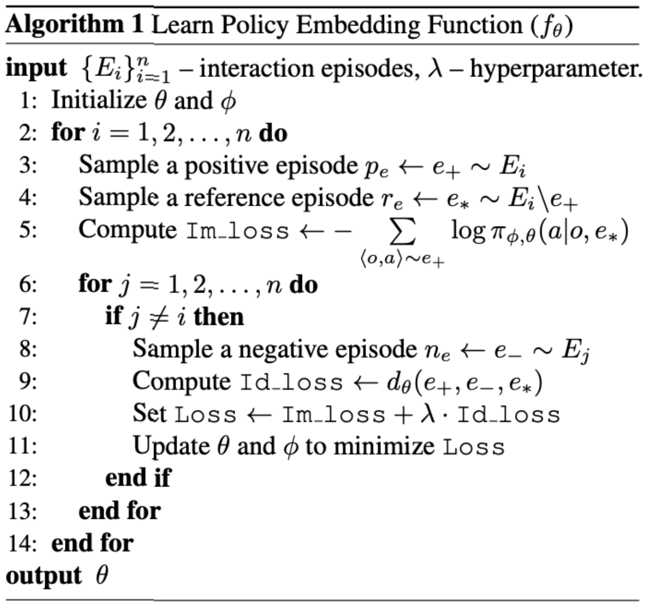
Figure 7: Performance of A2C with and without Agent Modeling [2]

player's strategy in a different episode than the representation of a different player's strategy. See figure 8 for the formal algorithm integrating this with cross-entropy loss. Here d_θ gives the loss that pushes representations of a single agent's actions closer together: using the notation from the figure, if p_e is the representation of e_+, r_e the representation of e_*, and n_e the representation of e_-,

$$d_\theta(e_+, e_-, e_*) = (1 + \exp\{||r_e - n_e||_2 - ||r_e - p_e||_2\})^{-2}$$

They call the embedding using only cross-entropy loss Emb-Im ("Im" is short for "imitation learning", which is similar to apprenticeship learning, discussed in section 5.2), the embedding using only the above d_θ Emb-Id, and the embedding using both Emb-Hyb. Grover et al. tested a common policy gradient model (PPO) using these embeddings to condition action selection; their experiments use a competitive artificial wrestling environment called RoboSumo, and a cooperative environment called ParticleWorld, where one agent called a speaker tries to send messages to another agent called a listener to navigate to a landmark in an artificial world; rewards are shared between the two. In RoboSumo, regardless of which embedding is used, the networks with opponent embeddings outperform an optimization network without opponent embeddings. In particular, while all tested networks perform well against agents they are *trained* against, networks with opponent embeddings generalize to agents outside of the training set significantly better. See figure 9. Unsurprisingly, Emb-Hyb performs the best, with Emb-Id close behind. In ParticleWorld, speakers were specifically trained to model listeners, and again they find that models of speakers including embeddings of opponent strategies outperform those that do not, with Emb-Hyb again being the most successful.

Papoudakis et al. show how autoencoder networks can represent other agents using limited information.[8] Their model for this task is called LIAM (Local Information Agent Modelling). In LIAM, one controlled agent is trained with the autoencoder architecture to model another agent. To do this, the encoder is passed a representation z_t of the controlled agent's observation and action history up to time t. Notice this is the *controlled* agent's history; this is what



Algorithm 1 Learn Policy Embedding Function (f_θ)

```
input $\{E_i\}_{i=1}^n$ – interaction episodes, $\lambda$ – hyperparameter.
 1: Initialize $\theta$ and $\phi$
 2: for $i = 1, 2, \ldots, n$ do
 3:     Sample a positive episode $p_e \leftarrow e_+ \sim E_i$
 4:     Sample a reference episode $r_e \leftarrow e_* \sim E_i \setminus e_+$
 5:     Compute Im_loss $\leftarrow - \sum_{\langle o,a \rangle \sim e_+} \log \pi_{\phi,\theta}(a|o, e_*)$
 6:     for $j = 1, 2, \ldots, n$ do
 7:         if $j \neq i$ then
 8:             Sample a negative episode $n_e \leftarrow e_- \sim E_j$
 9:             Compute Id_loss $\leftarrow d_\theta(e_+, e_-, e_*)$
10:             Set Loss $\leftarrow$ Im_loss $+ \lambda \cdot$ Id_loss
11:             Update $\theta$ and $\phi$ to minimize Loss
12:         end if
13:     end for
14: end for
output $\theta$
```

Figure 8: Policy Representation Algorithm [5]

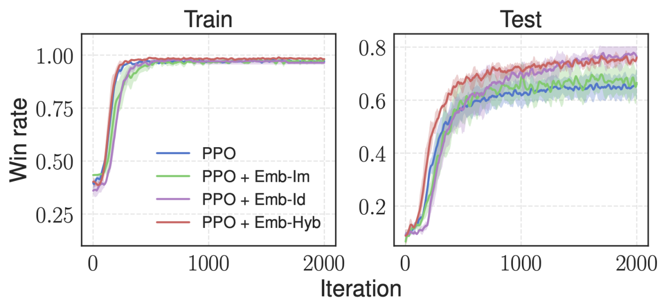


Figure 9: Performance of standard PPO vs PPO with the three varieties of learned embedding [5]

allows the encoder to model another agent based only on the controlled agent's observations. The decoder $f_{\mathbf{u}}$ produces from this representation an observation and a policy for the modeled agent, $f^o_{\mathbf{u}}(z_t)$ and $f^\pi_{\mathbf{u}}(\cdot|z_t)$ respectively. If o_t and a_t are the true observation and action history of the modeled agent, respectively, then the loss for the autoencoder is

$$\mathcal{L} = \frac{1}{H}\sum_{t=1}^{H}[(f^o_{\mathbf{u}}(z_t) - o_t)^2 - \log f^\pi_{\mathbf{u}}(a_t|z_t)]$$

This is trained on full information from both controlled and modeled agents, but at test time only receives information from the controlled agent.
Overall, LIAM works well when tested in several artificial MARL environments (cooperative speaker-listener experiments, cooperative foraging experiments, and competitive predator-prey experiments). They augment A2C with LIAM, and compare with several baseline models to incorporate information about opponents, one of which is FIAM, which unlike LIAM is trained on complete information about the modelled agent; this provides an upper bound on the performance of LIAM. LIAM consistently performed better than the other baselines and approached its upper bound:

> Our results support the idea that effective agent modeling can be achieved using only local information during execution: the same RL algorithm generally achieved higher average returns when combined with representations generated by LIAM than without, and in some cases the average returns are comparable to those achieved by an ideal baseline which has access to the modeled agent's trajectory during execution.[8]

Another use of autoencoders for AMOA in MARL comes from Zintgraf et al., who extend a purely observational AMOA model from Rabinovitz et al. to be used by an agent in a MARL environment, and find it outperforms an agent that does not use an autoencoder to explicitly represent opponents.[12, 9] The model uses two encoders: a character net meant to capture stable traits of the agent across time, and a mental state net meant to represent their current mental state. These are passed to a decoder that combines their representations to predict the agents' actions. This model is quite successful in the observational context, developing sufficiently complex representations to correctly predict agents acting on false beliefs, so it is no surprise that it improved model performance. In both cases, the encoder structure is trained directly on its predictive accuracy. Interestingly, however, Zintgraf et. al did not observe a significant difference between using this double-encoder architecture and architectures that use only a single encoder.
Autoencoders offer a highly flexible and general technique for AMOA. They can be combined with any policy algorithm in MARL with minimal modification, can be trained to differentiate agents, can operate with restricted observations, and can represent complex models of individual agents. The success found in

tailoring the autoencoder network for different predictive tasks and their consistency in improving performance relative to models with only implicit representations of other agents suggest they could have a highly useful role in MARL moving forward.

4.4 Self-Other Modeling

The paper "Modeling Others Using Oneself in Multi-Agent Reinforcement Learning" discusses Self-Other Modeling (SOM), a crucial MARL algorithm. The SOM algorithm described in the paper restricts games to 2 players. The algorithm uses a neutral network f that takes the current agent's goal as input z_{self}, an estimate of other player's goals $\bar{z}_{other}$, and observation state from the current agent's perspective, s_{self}. Then, using these inputs, the algorithm outputs a probability distribution over actions π and value estimate V. The total SOM estimate for agent i then becomes:

$$(\pi^i, V^i) = f^i_{self}(s^i_{self}, z^i_{self}, \bar{z}^i_{other}; \theta^i_{self}) \quad [10]$$

Here, θ^i are agent i's neural network parameters for the neural network f. The actions are sampled from some policy π; the observation state s^i_{self} contains the location of the acting agent (the one whose action is decided by the neural network f^i), as well as the location of the other agent.

Since each agent computes both its actions and values, each agent will have the same neural network used in two different ways. The first way of using the neural network is part of the acting mode, and it is as follows:

$$f^i_{self}(s^i_{self}, z^i_{self}, \bar{z}^i_{other}; \theta^i_{self}) \quad [10]$$

The other way of using it is part of the inferring mode and the equation for the neural network is as follows:

$$f^i_{other}(s^i_{other}, \bar{z}^i_{other}, z^i_{self}; \theta^i_{self}) \quad [10]$$

f_{self} is used by an agent to compute its values and actions, while f_{other} is used to infer the other agent's goals. The parameters for the f_{self} and the f_{other} are the same, but the difference comes in the location of $\bar{z}_{other}$ and z_{self} or the positioning of the current agent's goals and positioning of an estimate of other player's goals respectively. At each step, the agent needs to infer $\bar{z}_{other}$ (and this is inferred using f_{other} or the inference mode) to plug into f_{self} and choose the corresponding action (since f_{self}, or the active mode, is used to produce a probability distribution over actions π and the value estimate V where V is the scalar estimate of expected return for the current agent).

The paper ran the SOM algorithm for 3 special cases: the coin game which was a fully cooperative game with symmetric roles for the agents, the recipe game which was an adversarial game with symmetric roles for the agents, and finally, the door game which was a cooperative game but with asymmetric roles for the agents. All of these games were played in a grid-world environment. These games were run with other baseline models such as True-Other-Goal (TOG), No-other-model (NOM), integrated-policy-predictor (IPP), and separate-policy-predictor (SPP). TOG is similar to SOM and uses the same architecture, but instead of having access to estimates of other agent's goals for $\bar{z}_{other}$, this algorithm has access to the other agent's true goal as its $\bar{z}_{other}$. NOM only takes in the observation state s_{self} and its own goal z_{self}; it has no explicit model of the other agent or estimate of the goal, and uses the same architecture as SOM. IPP has the same setup as NOM, but it also has an additional final layer that outputs the probability distribution over the next action of the other agent. SPP (an extension of the model from He. et al in section 4.2) has a policy network for deciding the current agent's actions and an opponent network for predicting the other agent's actions. The opponent network takes the state S and its own goal z_{self} and outputs a probability distribution for the action taken by the other agent at the next step, and once again it uses the same architecture as SOM. One crucial difference between SPP and SOM is that SPP does not infer the agent's goal explicitly like SOM does, but it builds an implicit model (by predicting the agent's actions at each step).

In the coin game, since it is cooperative, both agents gain more rewards by obtaining both agents' goals instead of just picking coins from their own goal. This game was played in an 8x8 grid world containing 12 coins of 3 different colors. Agents are randomly assigned to any of the three colors at the start and can go either up, down, left, right, or not move at each step. A single run of the game ends after 20 steps and the reward is a function of both the current agent's color and the other agent's color, and there was a penalty term that penalizes the agents for not finding coins from either agent's goals (this can happen because there are 3 colors and only 2 agents), so that agents are forced to only pick from their own or the other agent's color set (picking a coin makes it disappear from the grid). The results of running all the above baseline models with this setup are shown in Figure 10.

As seen in Figure 10, TOG, which knows the true goal of the other agent, performs the best, and this is expected because it knows the true goal, but SOM beats all other baseline models for this cooperative version. This is because with SOM, every agent could analyze and pick both the self and other coins to maximize the reward for the agent, but most other baselines ended up picking only coins to satisfy their own goals. Since the reward was maximized when picking both the self and the other agent's goals, these other baselines did not perform as well. So, SOM was the best model for a cooperative game, as shown in the coin game experiment above.

In the second experiment, the SOM and other baselines were run on a recipe game. In this game, agents had to create compositional recipes containing multiple items from the environment. Agents are given the names of their goal recipes as input without the components necessary to make the recipe. Since

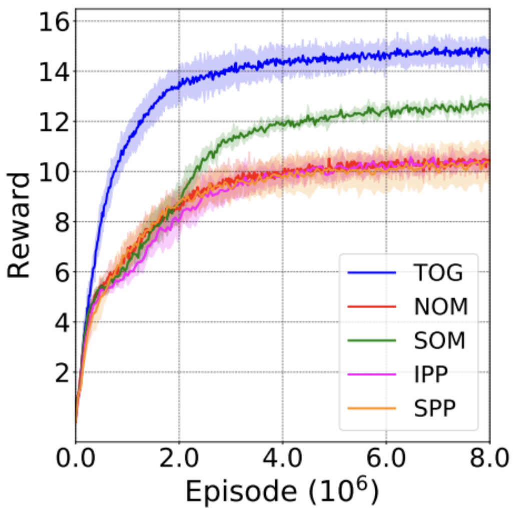


Figure 10: Reward for each baseline when run on the cooperative coin game [10]

this is an adversarial game, only one agent can end up getting the recipe because resources were scarce. There were 4 types of items (8 total, 2 of each type), 4 recipes, and the game was played in a 4x6 grid. The first agent is given an arbitrary recipe from the 4 possible recipes, but the other agent is given a competing recipe so that only one agent is successful for a single run. Each agent receives a reward of +1 for fully crafting the recipe and a penalty of -0.1 for not picking up items needed for the recipe at the end of a single run (only 2 baseline models run against each other in a single run). All agents were placed away from the items so that no agent started with an advantage (agents started at the leftmost and rightmost corners arbitrarily). Agents had 6 actions to choose from: up, down, left, right, choosing not to move, or picking the item (picking an item makes it disappear from the grid), and the game ended after 50 steps (50 steps constituted a single run). The result of running all baseline models with this adversarial setup is shown in Figure 11.

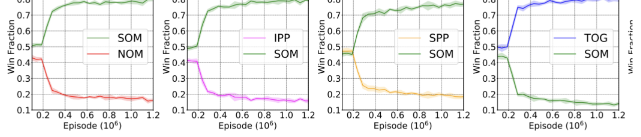


Figure 11: Reward for each baseline run against SOM [10]

As seen in Figure 11, TOG once again is the only baseline model that defeats SOM, and this is expected because TOG knows the true goal and is a prophet of sorts. All other baseline models perform much worse than SOM. So, just like in a cooperative game, even in an adversarial game, SOM outperforms the baseline models (except for TOG), thus proving that SOM is an effective MARL algorithm that infers other agents' goals and also maximizes the expected return of the current agent.

For the final experiment, SOM and other baselines ran on a cooperative door game, but agents this time had asymmetric roles. This door game was played in a 5x9 grid with two agents and had 5 goals behind 5 walls on the left edge of the grid, and 5 switches on the right wall of the grid. Players start on random squares of the grid (not on squares where doors, goals, or switches exist) and can either go up, down, left, right, or not move at each step. A single run of the game ends after 22 steps or when one of the agents reaches their goals. This game also has invalid actions, so if an agent goes outside the border or to a square with a closed door then the action is invalid. Agents got a +3 reward for getting to their goals, and a -0.1 penalty for every step taken. Doors for goals are open only as long as one of the agents is on the switch for that door (otherwise the door is closed and the action becomes invalid). Agents have to cooperate in that one agent must sit on the switch and ignore its own goal for the other agent to get their goal (provided that the sacrificing agent has inferred the other agent's goal correctly). Result of running all the baseline models with this cooperative but asymmetric setup is shown in Figure 12.

As seen in Figure 12, SOM again outperforms the other baseline models except for TOG, which knows the true goals. In this cooperative game, unlike the coin game where the differences between the agents were very large, here all models have almost the same win fraction with small variability. This happens because each agent plays both roles of sacrificing and achieving their goals in some runs. So, while SOM still beats the other baseline models, asymmetric roles ensure that agents are more cooperative and sacrifice their own goals for the other agents, thereby reducing variability among other baseline models.

Based on these 3 experiments, SOM clearly beats all the other baseline models, not just for cooperative and adversarial environments with symmetric roles but also for a cooperative environment with asymmetric roles. SOM's dominance shows that it is an effective MARL model that can infer other agent's goals and also maximize the current agent's expected return.

4.5 Bayesian Modeling

The paper "Coordination in Multiagent Reinforcement Learning: A Bayesian Approach" presents a Bayesian MARL algorithm called BVPI. The paper then compares BVPI to another Bayesian model to test their performance in different environments.

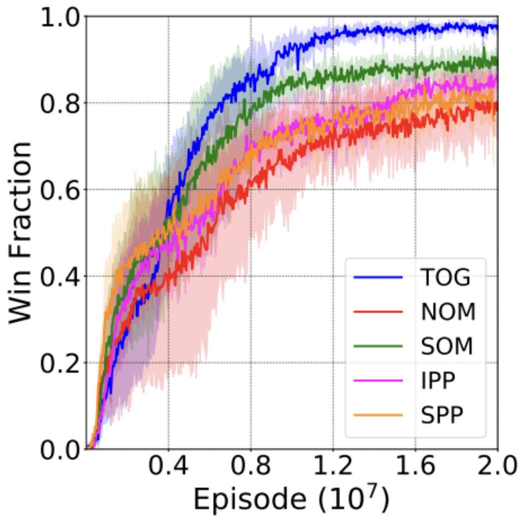


Figure 12: Reward for each baseline when run on the cooperative door game with asymmetric roles [10]

In the Bayesian model described in this paper, the agent knows the game structure but does not know the reward or transition models. So at each iteration, an agent learns the experience vector $(s, a, \mathbf{r}, t)$ where s is the previous state where the action was taken, a is the action taken, $\mathbf{r}$ is the vector of rewards received, and t is the resulting state. Every Bayesian MARL agent has a prior distribution over the set of possible models and strategies being used by other agents and these distributions are updated after the agent observes the results of its actions and those of other agents. Therefore, every Bayesian agent has a belief state b that is updated every iteration. The belief state is of the form

$$b = (P_M, P_S, s, h) \quad [4]$$

where P_M is the density over possible game structures, P_S is the joint density over possible strategies played by other agents, s is current state of the system and h accounts for game history. Game history is important here because in Bayesian modeling, the models of some agents are history dependent, so each Bayesian agent must account for the game history to accurately predict the models of all agents. At each iteration, the agent learns the experience vector $(s, a, \mathbf{r}, t)$ as described above and using this experience vector, the belief state is updated to

$$b' = b(s, a, \mathbf{r}, t) = (P'_M, P'_S, t, h') \quad [4]$$

Here, the updates to the density of game structures and joint actions is done via Bayes rule. This means that we have the following update rules:

$$P'_M(m) = zPr(t, r|a, m)P_M(m) \quad [4]$$

and

$$P'_S(\sigma_{-i}) = zPr(a_{-i}|s, h, \sigma_{-i})P_S(\sigma_{-i}) \quad [4]$$

Here, σ_{-i} is the reduced strategy profile for all other agents except for the current agent, h' is the updated history, and z is the normalization factor. After updating the belief state, the agent selects optimal actions, and process is repeated.

The model described above (updating belief states and repeating for a certain time interval) helps determine the optimal policy as a function of the belief state. Unfortunately, this model may not always converge to an optimal policy because priors that fail to reflect the true model can mislead an agent, which results in an optimal policy not being achieved. The paper also calculates an additional expected value of information (EVOI) which is the impact of action a on a particular state s. Calculating this value allows the agent to explore more intelligently and helps improve decision making for subsequent steps; the value allows the agent to pick not just the most optimal action but also the most optimal action that has an impact on future decisions as well. This is just an optimization to ensure more efficient calculation of the optimal policy.

The paper ran three experiments using two main Bayesian methods: the one-step lookahead (BOL) and the sampling method as described above (BVPI). In all experiments, the Bayesian agents used a fictitious play model to represent uncertainty over other agent's actions. In each game, the different models were sampled to compute the expected reward gain for each agent's actions.

For the first game, or the penalty game, the strategy priors for each Bayesian agent are set to be 0.1 or 1 for each opponent action. The model priors are also uninformative, and each state-action pair gives the same prior distribution over reward and transition distributions. The game is altered so that joint actions provide a stochastic reward. The Bayesian approach (BVPI) described above is compared to the KK algorithm that focuses on exploring so that optimal equilibria can be achieved. The Optimistic Boltzmann (OB) and the Combined OB (CB) are also compared with the Bayesian algorithm described above. KK performs exploration without observing but both the OB and CB algorithms observe and predict the other agent's actions in addition to just exploring like the KK does. The last algorithm being tested for this game was the WoLF-PHC algorithm which works on stochastic games and has no special heuristics for equilibrium selection. For each run of the game, two agents of the same type are chosen to play the game (so BOL and BVPI play together, while KK, OB and CB go together as well).

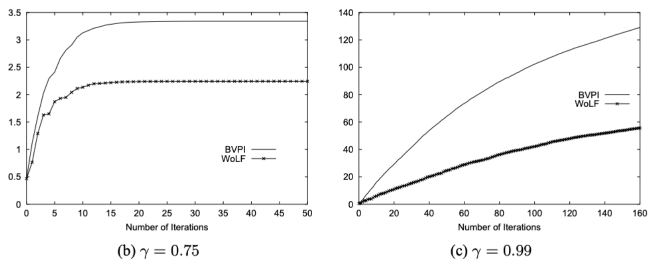


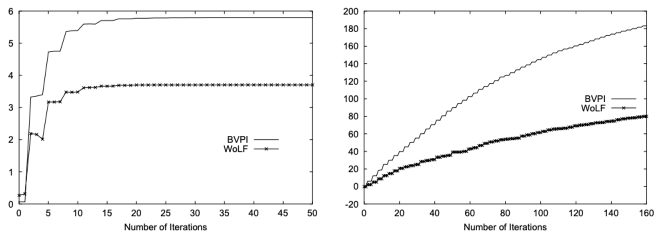
Figure 13: Accumulated reward functions for 50 iterations for BVPI and WoLF in the Chain World game [4]

For the first two iterations of this game, the paper used two values of the discount factor, 0.95 and 0.75. The results showed the total discounted reward accumulated by each of the agents averaged over 30 trials. Bayesian methods (both BVPI and BOL) performed better than all other algorithms for this experiment, and although all 3 of KK, OB, and CB converged to a policy, they perform worse than the Bayesian methods, and even worse than the WoLF-PHC algorithm (where WoLF-PHC was ranked third when discount factor was 0.95, but when discount factor for 0.75, WoLF-PHC beat BVPI and was ranked second while BVPI was ranked third). For the third and final run of this game, the penalty was increased to -100 but the discount parameter was unchanged at 0.95. KK and OB never converged to an optimal policy, while WoLF-PHC and CB converged to the optimal equilibrium each time. Since the penalty was increased, all four methods had their performance worsen with respect to the discounted reward, but the Bayesian methods performed much better.

So, in this first experiment of the penalty game, except for the case where discount factor was 0.95 (in which case WoLF-PHC outperforms BVPI but not BOL), both the Bayesian look ahead and the Bayesian algorithm described above beat all the other algorithms thereby proving their effectiveness in determining an optimal policy.

The second experiment was playing a Chain World game which was an identical-interest, multi-state, stochastic game. Transitions are said to be noisy with a 10% chance of the agent's action having the opposite effect to that intended. Only two algorithms were compared here. BVPI and WoLF (algorithm described above with no special heuristic for equilibrium selection) were both run using discount factors of 0.75 and 0.99 respectively, again averaged over 30 runs. The results are shown in Figure 13.

As seen in Figure 13, BVPI performs considerably better than WoLF for both the discount factors of 0.75 and 0.99. BVPI performed much better here because it managed to find the optimal policy in 7 of the 30 runs for the dis-



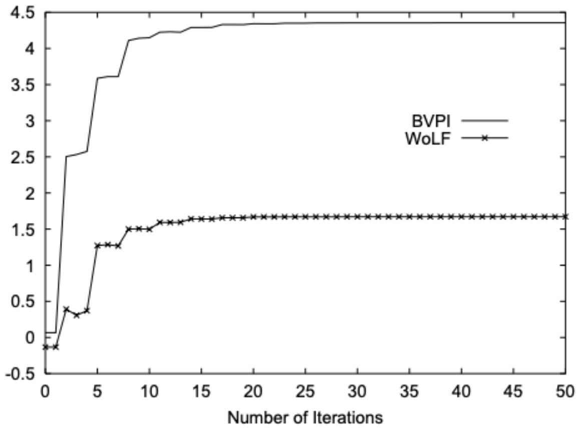
(b) $\gamma = 0.75$ (c) $\gamma = 0.99$

Figure 14: Accumulated reward functions for 50 iterations for both BVPI and WoLF in the Opt In or Out game with low noise [4]

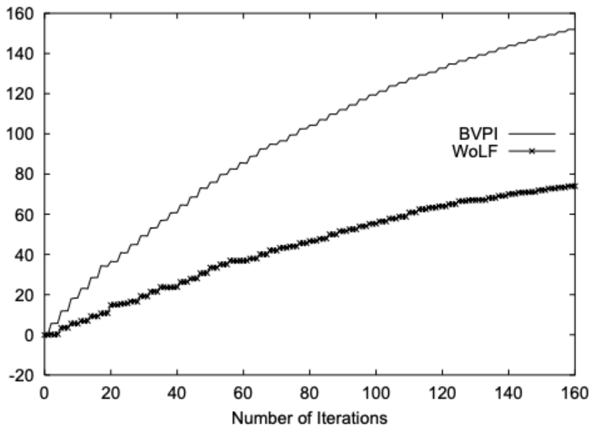
count factor = 0.99 and in 3 of the 30 runs for the discount factor = 0.75 while WoLF was unable to reach s_5 for the discount factor of 0.99 but managed to reach 2 the optimal policy in 2 of the 30 runs for discount factor = 0.75. So, the Bayesian algorithm described in this paper is more intelligent in terms of exploration and this is why it outperforms WoLF here which has no special heuristic for equilibrium selection. The algorithm described in this paper uses EVOI (an added optimization) that allows the paper to optimize future decision making by selecting each action not just on its utility but also on its impact on future decisions, and this is something that the WoLF algorithm lacks (it has no special heuristic to select an equilibrium) which is why it performs more poorly than the Bayesian algorithm described in this paper.

The final experiment was the "Opt In or Out" game which also had stochastic transitions and the same conditions of some actions having unintended consequences with some probability p. In the first version, the probability of the unintended consequences was 0.05 (low 'noise'), and in the other version, this probability was 0.11 (medium 'noise'). The discount factor was again set to 0.75 and 0.99 as in the previous game, and once again the Bayesian algorithm described in this paper is compared to the WoLF-PHC algorithm. The results for the first version are shown in Figure 14(low noise).

As seen in Figure 14 (low noise); again the Bayesian algorithm described in this paper beats the WoLF-PHC algorithm. This happens because in the low noise case, the Bayesian algorithm found the optimal policy 18 out of 30 times with discount factor being 0.99 and 15 out of 30 times when discount factor was 0.75, while WoLF-PHC achieved the optimal policy only once out of 30 iterations for discount factor being 0.99, while for discount factor of 0.75, WoLF-PHC found the optimal policy 17 times, which was better than the Bayesian case but not significantly better. Since WoLF-PHC was slightly better for the discount factor being 0.75 for the low noise case but significantly worse for the discount factor being 0.99, the Bayesian algorithm described in this paper still outperforms the WoLF-PHC algorithm. The results for the medium noise version are



(a) $\gamma = 0.75$



(b) $\gamma = 0.99$

Figure 15: Accumulated reward functions for 50 iterations for both BVPI and WoLF when the discount factor is varied between 0.75 (top) and 0.99 (bottom) for the Opt In or Out game with medium noise [4]

shown in Figure 15.

Figure 15 (medium noise version), just like in Figure 14 (low-noise version), also shows that the Bayesian algorithm outperformed the WoLF-PHC algorithm. Now here, even though eventually the WoLF-PHC algorithm always converged to some policy for both discount factor of 0.75 and 0.99 while the Bayesian algorithm only found the optimal policy 10 times for discount factor of 0.99 and 13 times for discount factor of 0.75, the Bayesian algorithm still performs better. This happens because the algorithm described in the paper is not always concerned with finding an optimal policy but instead it is concerned with maximizing the reward during its run; on the other hand, the WoLF-PHC algorithm is only concerned with converging to some policy, so in a short time interval, due to the maximizing nature of the algorithm described in this paper, BVPI will still end up performing better even if it does not find the optimal policy at each run.

So, the three games show that the Bayesian algorithm described in this paper is a powerful algorithm for MARL and typically beats all its competitors in the stochastic game environment.

5 Inverse Reinforcement Learning

Inverse Reinforcement Learning (IRL) is a machine learning technique where an agent learns an implicit underlying reward function by observing an expert. IRL differs from reinforcement learning in that, unlike reinforcement learning, the reward function is not given for IRL. The expert does not have a formally defined reward function. Still, they are assumed to be following some set of optimal steps to maximize their reward 'function'. The agent's job is to observe the expert and find this function. Once the agent has discovered the underlying reward function, the agent then decides whether to apply the same reward function (model the expert) or some different reward function based on their environment (since their environment could differ from that of the expert) to maximize the reward that they get. We looked at two main IRL papers; the first paper talked about two simple yet fundamental IRL algorithms that captured the essence of IRL and its use in the real world, while the second paper talked about an Apprenticeship Learning algorithm, a more specific application of IRL algorithms.

5.1 Two Fundamental IRL algorithms

The first paper we examined was about two fundamental IRL algorithms, namely **max margin** and **projection**. The paper titled "Algorithms for Inverse Reinforcement Learning" by Andrew Y. Ng and Stuart Russell goes into detail, talking about what these two fundamental IRL algorithms actually are, and also describes the performance of these two algorithms in a set of grid world experiments.

The max margin algorithm described in the paper tries to find a reward function R such that the observed expert behavior a_1 is optimal, i.e. better than all actions in the set of other actions $a_{-1} = a_2, a_3, \ldots, a_k$ under this particular reward function. In other words, the goal is to make the expert's behavior look like a reward-maximizing behavior. This algorithm searches over arbitrary reward functions instead of using a single fixed reward function R. The assumption here is that a_1 is the optimal fixed action taken by the expert in all observed states. The equation for the max margin algorithm, as seen in the paper, is shown below. The main aim of the max margin algorithm is to find a reward function R such that given a finite set of states S, a set of actions A, along with transition probability matrices Pa, where these probability matrices are the state transition probabilities for each action (The transition probability matrices can be thought of as having the probability of ending up at some state s' given that we started at state s and took action a), R can satisfy the inequality given below:

$$(P_{a_1} - P_{a_{-1}})(I - \gamma P_{a_1})^{-1} R \geq 0 \quad [7]$$

The inequality is ensuring that we find a reward function such that for that particular reward function, the expert's action a_1 is indeed the optimal action (weakly dominates) over all other actions in action set A.

While the above inequality captures the essence of the algorithm, the paper tries to enforce stricter bounds on the reward function by adding a penalty term and also focuses on maximizing the worst-case margin between the optimal action a_1 and the other actions from the action set A. The modifications help constrain the reward function further, but the main idea is still that the reward function for this particular algorithm must at least satisfy the inequality given above.

The projection algorithm described in the paper also tries to find a reward function R such that the observed expert behavior a_1 is as optimal as possible for all other actions a_{-1} under this particular reward function, or in other words, the goal is to make the expert's behavior look like a rewards-maximizing behavior. The key difference here is that this algorithm is not searching over an arbitrary R, but instead the R is fixed to be a linear combination of known features. The algorithm works with an infinite set of states S, and the reward function is fixed and maps the set of states to a set of reals. The reward function R can be represented as follows:

$$R(s) = \alpha_1 \phi_1(s) + \alpha_2 \phi_2(s) + \ldots + \alpha_d \phi_d(s) \quad [7]$$

Where $\phi_1, \ldots, \phi_d$ are some features of a particular state s (and this is why they are functions of s as well), while the $\alpha_1, \ldots, \alpha_d$ are the unknown parameters that have to be fit to ensure that a_1 is still optimal. So, the projection algorithm is trying to find the unknown parameters $\alpha_1, ..., \alpha_d$ that make a_1 optimal.

The value function for the optimal policy π is defined as:

$$V^\pi = \alpha_1 V_1^\pi + \ldots + \alpha_d V_d^\pi \quad [7]$$

Where $V_1^\pi, \ldots, V_d^\pi$ are the expected sum of features $\alpha_1, \ldots, \alpha_d$ for the current state and this is why the total value function is a linear combination of the feature expectations for the current state.

Now, using these value functions and the definition of R from above, for some policy π to ensure that a_1 is the optimal action taken by the expert compared to all other actions, the final inequality for the projection method becomes:

$$E_{s'} \sim P_{sa_1}[V^\pi(s')] \geq E_{s'} \sim P_{sa_{-1}}[V^\pi(s')] \quad [7]$$

This inequality says that the expected value of taking action a_1 for some state s' is at least as large as expected value when taking any other action a_{-1} for that same state. This means that action a_1 always performs at least as well as all other actions in the set of actions A, thus proving the (weak) optimality of a_1.

So, both the max margin and the projection algorithms tried to find a policy π such that for that policy, they could find a reward function R that would maximize the optimal action a_1 taken by the expert. The main difference between max margin and projection is that the former works on finite sets and works with arbitrary reward functions, while the latter works on infinite sets and the reward function is fixed and is a linear combination of the features of a particular state s. Furthermore, while max margin tries to directly optimize the reward function, the projection algorithm tries to maximize the expectation of the value functions of each state, and these value functions indirectly help estimate the reward functions.

Now, the paper ran two experiments, one with the grid world where they tested out the max margin algorithm and one with a mountain car where they tested out the projection algorithm. For the grid world case, the paper used a 5x5 grid world where the agent starts from the lower left square of the grid and makes its way to the upper right grid square, where it receives a reward of 1. Just like in the last grid world case, actions are in the four main directions and there is a 30% chance of resulting in a move in a random direction. The true reward function was supposed to be as seen in Figure 16.

The max margin algorithm without a penalty gave the reward function as shown in Figure 17.

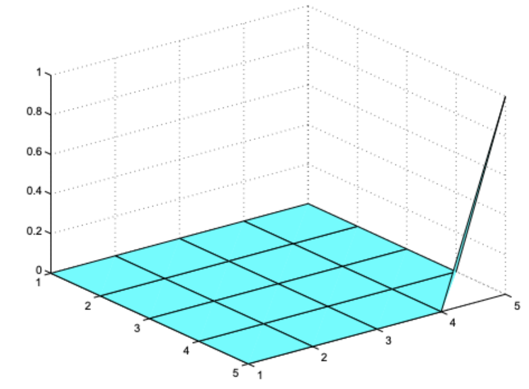


Figure 16: True Reward Function for 5x5 Grid World [7]

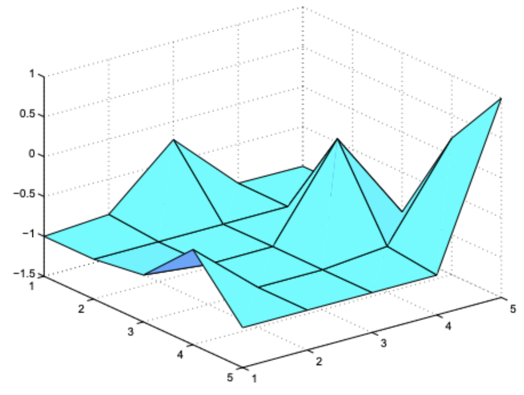


Figure 17: Reward Function after running Max Margin on 5x5 Grid World [7]

Now as seen in Figure 17, the reward function matches with the true reward function but has a lot more “bumps”. These bumps are due to the lack of a penalty term, and this is why the authors of the paper added a penalty term. The addition of this penalty term reduced the bumps in the reward function significantly and made the function look a lot smoother.

So, the max margin algorithm was able to recover most of the expert’s true reward function in the grid world, and while there were some bumps, these bumps could be eliminated using a penalty term, which would help the max margin reward function match more closely with the true reward function.

To test the projection algorithm, the paper used a continuous state space of a mountain car. The true reward per step was set to -1 until the top of the hill was reached. The authors only chose to use the x position of the car as the state space modeling functions (or in other words, the feature functions for any state were just based on the x position of the car, instead of other things like the velocity of the car for example), so their reward function was a function of the different x positions of the car (they used 26 different positions, so a linear combination of all the 26 positions). The goal of the mountain car was to reach the top of the hill as shown in Figure 18.

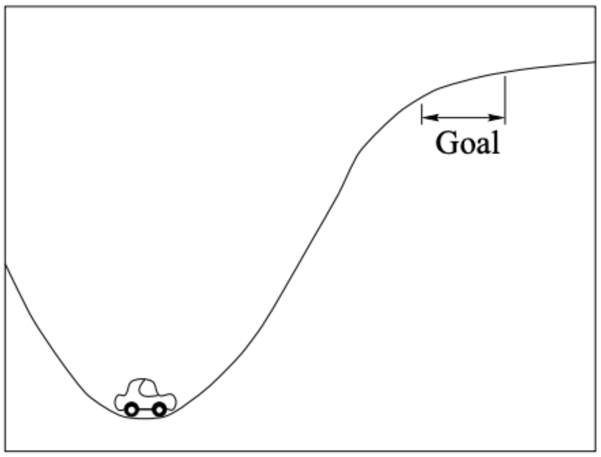


Figure 18: True Reward Function for Mountain Car [7]

Now after running the projection algorithm, the reward function obtained was as seen in Figure 19.

In Figure 19, we can see that the projection algorithm actually performed slightly better than the max margin algorithm in that it was able to accurately

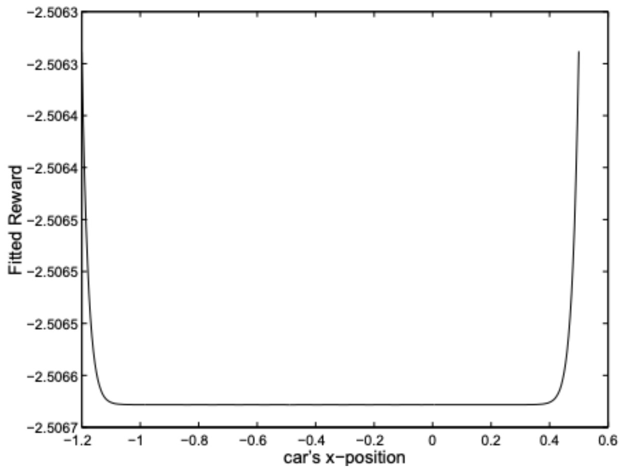


Figure 19: Reward function obtained via Projection algorithm for mountain car [7]

capture the structure of the true reward function without the need for a penalty term, and there were no 'bumps'.

All in all, the results of the experiments show that both the max margin and projection algorithms were able to capture the true reward function almost completely. The projection algorithm performed slightly better than the max margin algorithm, but the addition of the penalty term to the max margin algorithm will allow it to capture the true reward function more accurately. The experiments show that both the max margin and projection algorithms are apt algorithms to capture the true reward function of the expert. The next paper talks about how Apprenticeship Learning uses either of these two algorithms to accurately learn the true reward function from the expert via observation.

5.2 Apprenticeship Learning: An Application of IRL algorithms

The second paper we looked at was about **Apprenticeship Learning**, a more specific application of IRL. The paper titled "Apprenticeship Learning via Inverse Reinforcement Learning" talks about such an apprenticeship learning algorithm by Pieter Abbeel and Andrew Y. Ng where they try to learn the optimal reward function by observing an expert, and the paper also goes into details about how the algorithm fares in the real world.

The main algorithm used in the paper relies on finding a policy π that would ensure that the algorithm's discovered reward function is very close to the expert's true reward function, such that the difference between the expectation of the discovered reward function $\mu\pi$ and the expectation of the expert's reward function μE is just ϵ, or:

$$|\mu\pi - \mu E| = \epsilon \quad [1]$$

The algorithm starts off by first picking an arbitrary policy π either via Monte Carlo methods or just through random selection. The next step is to apply an IRL algorithm (this can be either max margin or projection IRL) using this policy to discover the 'true' reward function of the expert. The algorithm tries to find a reward function for the current policy such that the expert's reward function only does better by some fraction t, and the algorithm terminates if this t $\leq \epsilon$ because if t $\leq \epsilon$ then a 'close enough' reward function has already been found. If t $> \epsilon$, then the algorithm continues to the next iteration and repeats this process until it finds some policy π with a discovered reward function close enough to the expert's reward function. The paper also mentions that the algorithm will find a policy relatively quickly; the paper mentions that it takes approximately

$$O([k/(1-\gamma)^2(\epsilon)^2]log[k/(1-\gamma)\epsilon]) \quad [1]$$

iterations where γ is the discount factor used in the algorithm to ensure that the reward function is not 0 and k is the number of policies.

The paper tested this algorithm described above on a grid world first, and then also on a car driving simulation. For the grid world, a 128x128 grid world was used where the algorithm sampled trajectories from the expert's (optimal) policy. The agent has 4 actions, one for each direction, but with a 30% chance that an action fails and some random move is made. The 128x128 grid is divided into 16x16 macro cells, and only a few of them (a total of 64) have positive rewards. The paper shows that the max-margin algorithm (IRL algorithm used in step 2 above) and the projection algorithm (an alternate, IRL algorithm that could be used in step 2 above to discover a reward function such that the difference between expert's reward function and the current reward function is $\leq \epsilon$) end up being relatively close to the expert (feature) distribution especially as the number of iterations increase. The plot of the results for the grid world is shown in Figure 20.

As seen in Figure 20, the Apprenticeship learning algorithm described above using the max-margin algorithm (in step 2) differs by only 0.04 from the expert's distribution with a small number of iterations (say iteration count = 5) and as the number of iterations increased to 30, the distance fell to < 0.005 which is really close to the expert's distribution function showing that the algorithm is very effective in finding the optimal reward function in at least the grid world scenario (the projection IRL seems to outperform the max margin IRL but the difference is almost negligible).

For the car driving simulation experiment, the paper applied the apprenticeship learning algorithm above to try to learn five different driving styles. The

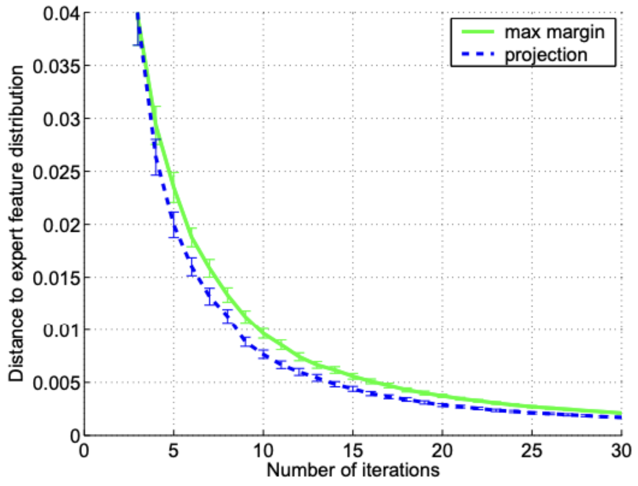


Figure 20: Max margin and projection IRL algorithms compared to the expert's reward function distribution [1]

expert's features were estimated from a single trajectory of 1200 samples (over a 2-minute interval). The features indicated the car lane and distance to the closest car in the current lane (distance of 0 means collision). The algorithm ran for 30 iterations and policy for each iteration was selected manually rather than by inspection for each of the 30 iterations. The algorithm considered these 5 driving styles of the expert: Nice= avoid collisions and prefer right over middle over left lanes; Nasty = Hit as many cars as possible; Right Lane Nice= Drive in right lane but go off lane to avoid collisions; Right Lane Nasty = Drive off road on right and come back in to hit the cars in right lane, and finally Middle Lane= drive in middle lane and crash into cars if they come in middle lane (ignore other lanes). The algorithm made sure to have negative rewards for collisions and for driving off-road, but larger positive rewards for driving in the right lane (compared to other lanes).

The paper concludes that the algorithm was qualitatively able to mimic demonstrated driving style. There were no true reward functions specified by the expert (just a qualitative description as above) so then the paper can only qualitatively show that the apprenticeship learning algorithm employing one of the two IRL algorithms as described above, works as intended even in a real world scenario like a car driving simulation (even though it is a very simplified real world scenario).

So, from the grid-world experiment and the real-world (simplified) driving scenario, the paper concludes that the apprenticeship learning algorithm they used (described in paragraph 2) is a reasonably good algorithm that applies IRL (in step 2 through either max margin IRL algorithm or projection IRL) and can find a rewards function that is very close to the expert's 'true' reward function (the difference between the expectation of the best policy and the true reward function's expectation is only ϵ). So, the algorithm described above is a useful algorithm incorporating IRL that can accurately predict and find the expert's true reward function, and this algorithm shows how IRL can be used effectively to find the 'true' reward functions in the real world.

6 Conclusion

The field of Reinforcement Learning provides a rich set of tools for AMOA. In this paper, we draw on two major sources for these tools: MARL and IRL. MARL provides a natural bridge between game theory and RL, so it is no surprise that we should find material relevant to game-theoretic inquiry. That being said, many MARL models do not explicitly incorporate AMOA, for two major reasons. One is that, in many environments, agents cannot observe other agents' actions, in which case there is nothing with which to model them; another (more significant) reason is that other agents are often not separated from the environment. Thus, agents model other agents only implicitly and in a way that does not enable isolation of those models.

However, the results in the MARL part of this paper consistently show that explicitly separating models of other agents from the environment significantly improves performance in MARL. This suggests that as MARL grows and develops as a field, we are likely to see AMOA become a more significant aspect of model optimization. We also see that more carefully crafted AMOA techniques often outperform simpler ones (Zintgraf et al. provides a counterexample, however) - for example, in He et al., DRON-MOE outperformed the simpler DRON-CONCAT, and in Grover et al., EMB-HYB and EMB-ID both outperform EMB-IM. Due to the relative youth of the field, however, there is nothing approaching a synoptic overview of the various techniques in AMOA (this paper excluded), and due to this relatively little comparison *between* the models that we have examined. It is unclear then when an autoencoder might be a better choice than a SOM for AMOA, and given that the experiments in each paper were fairly tailored for the technique being analyzed, it is not clear how well these techniques will generalize to arbitrary MARL environments. The problem is exacerbated by the paper using different underlying algorithms to augment with an AMOA. This is where IRL can help MARL.

If there is one safe assumption that could be made about agents across the plethora of MARL environments, it is that the agents will be Reinforcement Learners. Since we know this, we know in **any** MARL environment, we can make the assumptions key to IRL: specifically, that the agent being modeled has a reward function, and is acting in order to maximize that reward function. Therefore, the modeling techniques developed in IRL should apply to MARL environments in general. This would allow IRL to provide a general-purpose modeling baseline against which other techniques for AMOA can be evaluated. Other techniques could be compared either by comparing predictive accuracy, or (if the technique allows) extracting a representation of the reward function from the technique and comparing it to those derived with IRL. IRL allows us to see that a successful example of AMOA should involve at least *implicitly* a model of the other agent's reward function, but SOM is the only MARL technique we have seen that does this explicitly. Notably, SOM outperformed baseline MARL models that include AMOA but not an explicit reward representation, supporting this paper's contention that IRL provides a helpful perspective for augmenting MARL AMOA techniques.

This gives us a clear path for developing AMOA in RL in the future. MARL's game theoretic framework and a well-developed library of policy optimization algorithms provide an ideal testing ground for AMOA techniques, and IRL provides a natural framework for evaluating those techniques. Thus, we hope to see a systematic comparison of the many MARL AMOA techniques discussed in this paper, using IRL as a baseline (and as inspiration for future developments) across a wide variety of MARL environments, all while controlling for the underlying MARL algorithm. This should allow us to observe the situations (MARL environments and policy algorithms) that are optimal for particular AMOA models, or perhaps careful investigation will reveal that one technique stands above the others.

References

[1] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-first International Conference on Machine Learning (ICML)*, pages 1–8, 2004.

[2] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches.* MIT Press, 2024.

[3] Stefano V. Albrecht and Peter Stone. Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence*, 28:66–96, 2018.

[4] Georgios Chalkiadakis and Craig Boutilier. Coordination in multiagent reinforcement learning: A bayesian approach. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 709–716, Melbourne, Australia, 2003. ACM.

[5] Aditya Grover, Maruan Al-Shedivat, Jayesh Gupta, Yuri Burda, and Harrison Edwards. Learning policy representations in multiagent systems. In *International conference on machine learning*, pages 1802–1811. PMLR, 2018.

[6] He He, Jordan Boyd-Graber, Kevin Kwok, and Hal Daumé III. Opponent modeling in deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning(ICML)*, 2016.

[7] Andrew Y. Ng and Stuart J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML)*, pages 663–670. Morgan Kaufmann, 2000.

[8] Georgios Papoudakis, Filippos Christianos, and Stefano Albrecht. Agent modelling under partial observability for deep reinforcement learning. *Advances in Neural Information Processing Systems*, 34:19210–19222, 2021.

[9] Neil Rabinowitz, Frank Perbet, Francis Song, Chiyuan Zhang, SM Ali Eslami, and Matthew Botvinick. Machine theory of mind. In *International conference on machine learning*, pages 4218–4227. PMLR, 2018.

[10] Roberta Raileanu, Emily Denton, Arthur Szlam, and Rob Fergus. Modeling others using oneself in multi-agent reinforcement learning. *arXiv preprint arXiv:1802.09640*, 2018.

[11] Richard S. Sutton and Andrew G. Barto. *Introduction to reinforcement learning*. MIT Press, 2 edition, 2012. (draft 2nd ed.).

[12] Luisa Zintgraf, Sam Devlin, Kamil Ciosek, Shimon Whiteson, and Katja Hofmann. Deep interactive bayesian reinforcement learning via meta-learning. *arXiv preprint arXiv:2101.03864*, 2021.