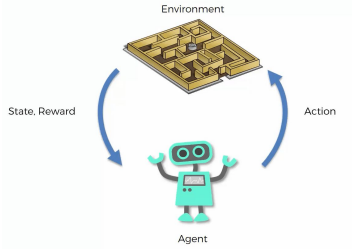


Agent
Theory of Mind with Reinforcement Learning

Techniques and models in RL for Agents Modeling Other Agents (AMOA)

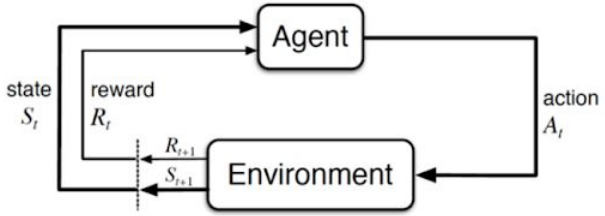
Image sources: https://www.ejable.com/tech-corner/ai-machine-learning-and-deep-learning/theory-of-mind-ai-in-artificial-intelligence/
https://k21academy.com/datascience-blog/machine-learning/reinforcement-learning/. Image sources not explicitly cited can be found in the write-up.

Our Core question from game theory: how do agents model other agents?

Reinforcement learning provides a body of methods for this problem

Basics of Reinforcement Learning

Markov Decision Processes (MDPs)

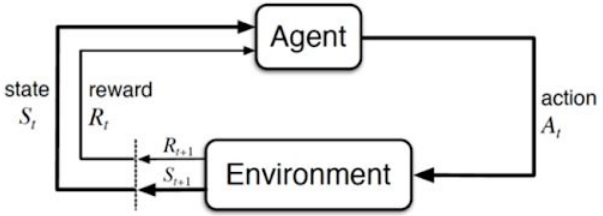


- MDPs are the core concept in Reinforcement Learning (RL): they formalize the dynamic between agents and their environments
- An MDP consists of:
  - A set of states $\mathcal{S}$
  - A set of actions $\mathcal{A}$
  - A set of rewards $\mathcal{R}$
  - A transition function $\mathcal{T} : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$
- This generates a sequence that looks like:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

Image source for this slide and following: https://vinaylanka.medium.com/multi-agent-reinforcement-learning-marl-1d55dfff6439

Where does the learning come in?



- An RL agent learns a policy π that tells it what action to take at any state in the MDP
- The typical approach is to learn a representation of the value for each state, and then act greedily with respect to that value
- The value is meant to represent the combination of immediate rewards of taking that action and the discounted rewards from following states

Value Functions

- Our value function can be represented with the following Bellman equation:

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

- The optimal policy solves this equation
- When we know the underlying transition function, this works great. We can compute the value function through iteration then pick actions greedily
- What about when we don’t know the underlying function?

Q-Functions and TD-Learning

- We use a Q-function that takes both state and action as argument:

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

- We can estimate this by running through the MDP. A technique called TD(0) uses the reward and estimate of a following state to update the value for a state action pair:

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

- There are many ways of doing this updating, including deep learning

Game Theory in MARL

Stochastic Games

- Stochastic games generalize the idea of an MDP
- Thus, stochastic games have multiple RL agents, each with their own reward function, and the transition function takes the actions of ALL agents as inputs
- This unites RL with game theory - we can also think of this as a generalization of a game that includes an underlying state
- A common assumption is that agents can only see some subset of the features of the state and the actions of other players - this is called a partially observable stochastic game

Some helpful translations from RL to game theory

Environment = Game

Agent = Player

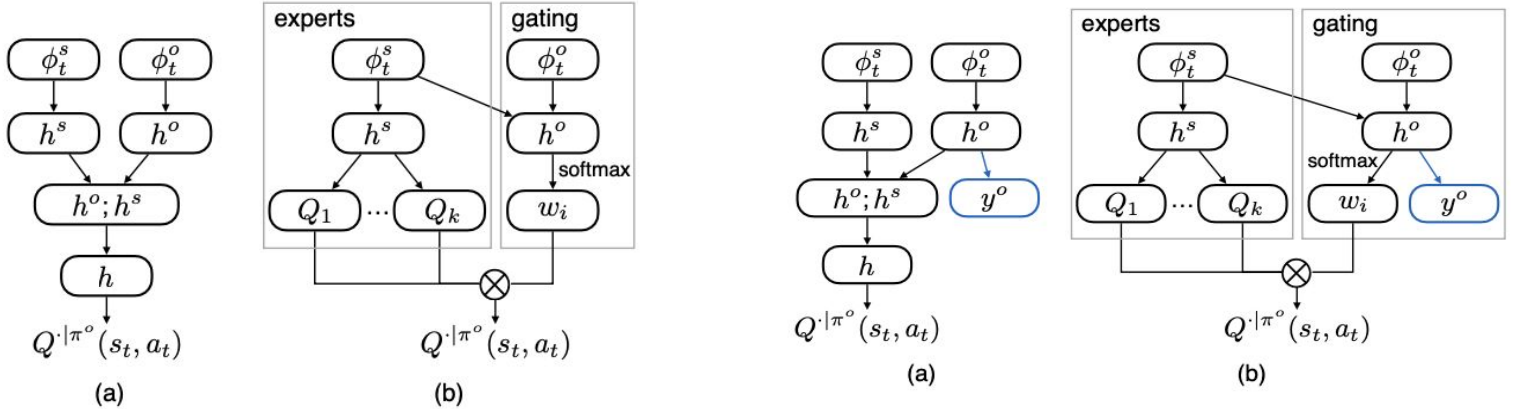
Reward = Payoff or Utility

Policy = Strategy

AGENTS MODELING AGENTS IN MARL

Deep Learning Opponent Models

- A natural idea: split off information about opponent agents from environment in a DL MARL agent's neural network



Experimental Results for Deep Learning Opponent models

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	-	-

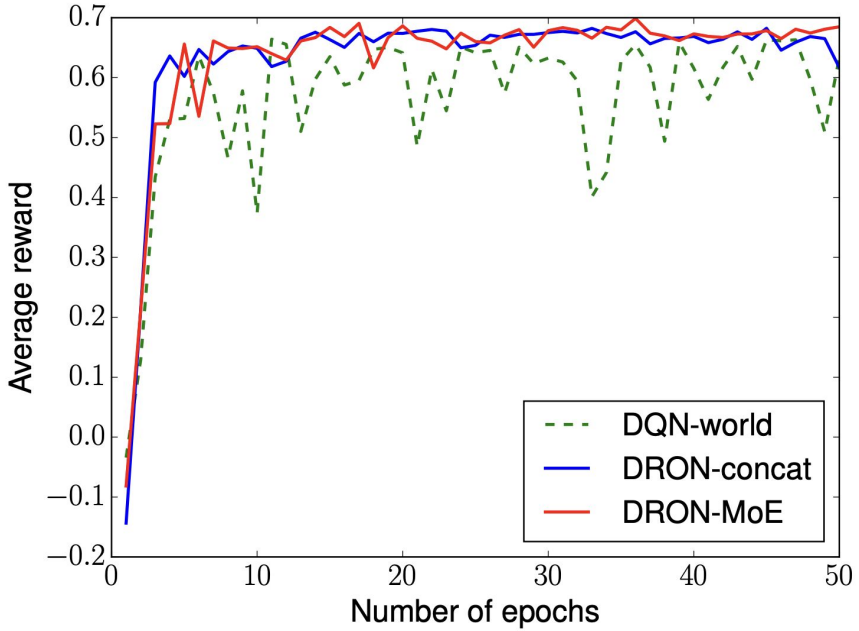
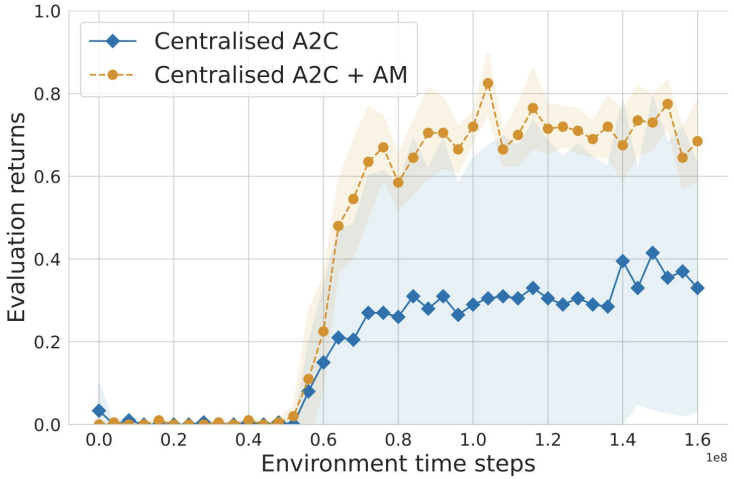


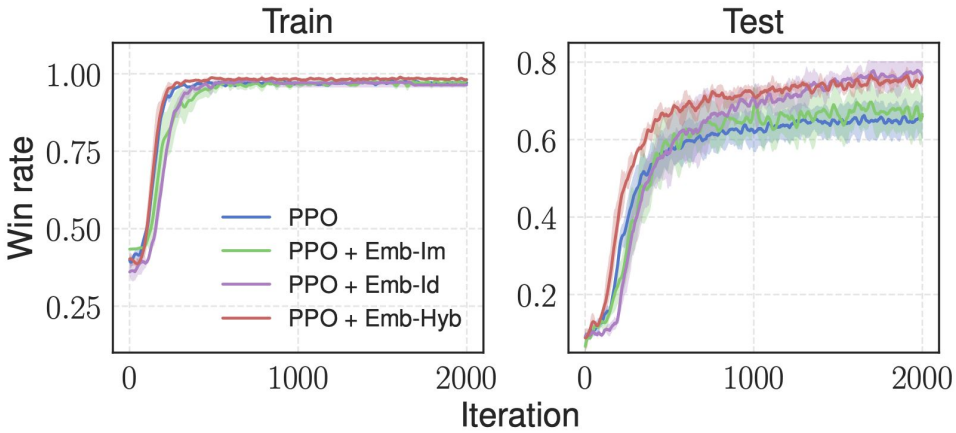
Image source: https://tenor.com/view/its-beautiful-orson-krennic-star-wars-gif-27444911

Learning Representation of Policies with Autoencoders

- This is a similar idea as the supervised prediction in the previous slides: an encoder takes information about opponents, then passes an embedded representation of that information to a decoder. The decoder then predicts the probability of opponents' actions. The whole architecture is called an autoencoder
- This is flexible in two ways:
  - The resulting probabilities can be used to condition the actions for any MARL algorithm
  - Autoencoders can be engineered to fine-tune opponent prediction

Experimental Results for Autoencoders





Self Other Modeling (SOM)

- The main equation for SOM is:

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

- Before using the above equation though, we first need to find $\bar{z}^i_{other}$ and this is done using the following neural network:

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

- Then, after calculating $\bar{z}^i_{other}$ using the above neural network, we plug it into the first equation to find both the value estimate and the probability distribution over actions

Experimental Results for SOM

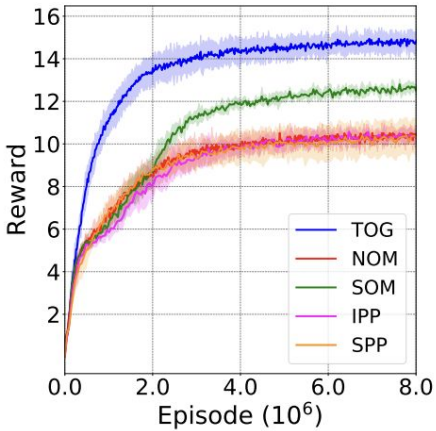


Fig 1: Rewards for each baseline on cooperative coin game in an 8x8 grid

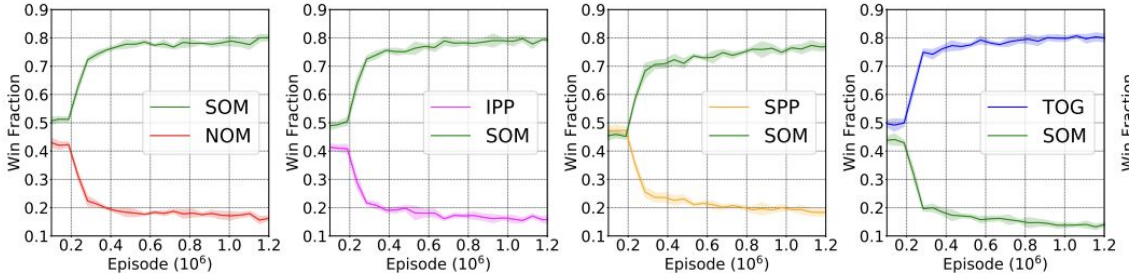


Fig 2: Rewards for each baseline on adversarial recipe game in a 4x6 grid

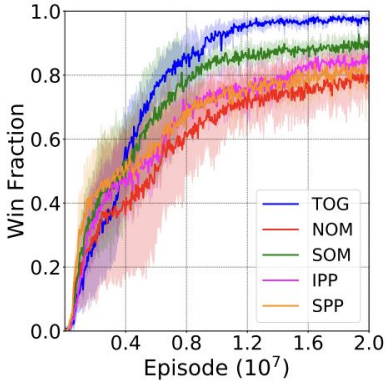


Fig 3: Rewards for each baseline on cooperative door game with asymmetric roles in a 5x9 grid

Bayesian Modeling

- Each agent has a belief state (prior distribution over set of models and strategies) updated every iteration as follows:

  $b = (P_M, P_S, s, h)$

- This belief state is modified using an experience vector at every iteration resulting in a new belief state:

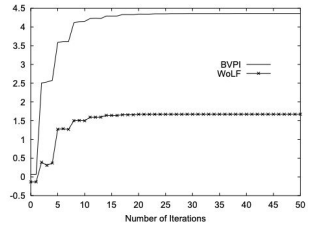
  $b' = b(s, a, \mathbf{r}, t) = (P'_M, P'_S, t, h')$

Here, bayes rule is used to compute both P'_M and P'_S respectively, and the bayes rule equation for both is shown below.

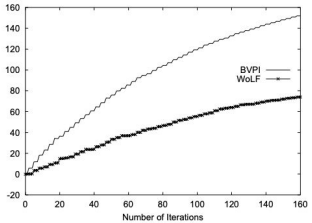
$$P'_M(m) = zPr(t, r|a, m)P_M(m)$$

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

Experimental Results for Bayesian Modeling

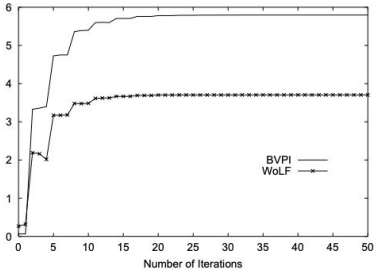


(a) $\gamma = 0.75$

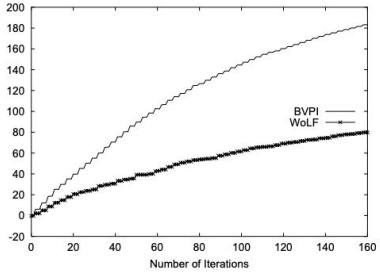


(b) $\gamma = 0.99$

Fig 1: Accumulated rewards for 50 iterations of the opt in or out game run with medium noise using BVPI and WoLF algorithms

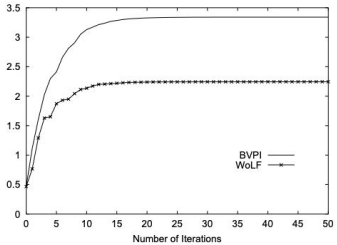


(b) $\gamma = 0.75$

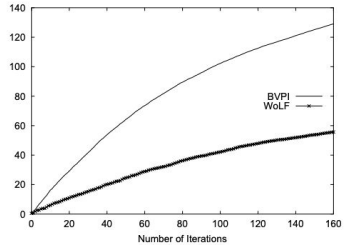


(c) $\gamma = 0.99$

Fig 2: Accumulated rewards for 50 iterations of the opt in or out game run with low noise using BVPI and WoLF algorithms



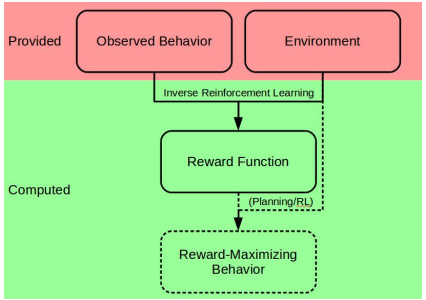
(b) $\gamma = 0.75$



(c) $\gamma = 0.99$

Fig 3: Accumulated rewards for 50 iterations of the chain world game run using BVPI and WoLF algorithms

Inverse Reinforcement Learning (IRL)



A different but important form of Reinforcement learning

Image source for this slide: https://dkasenberg.github.io/inverse-reinforcement-learning-rescue/

Max Margin

- Max Margin algorithm finds an arbitrary reward function R satisfying the following equation which ensures that for R, a_1 is indeed the most optimal action:

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

- This inequality captures essence of max margin, but a penalty term could be added to enforce stricter bounds on the reward function

Experimental Results using Max Margin

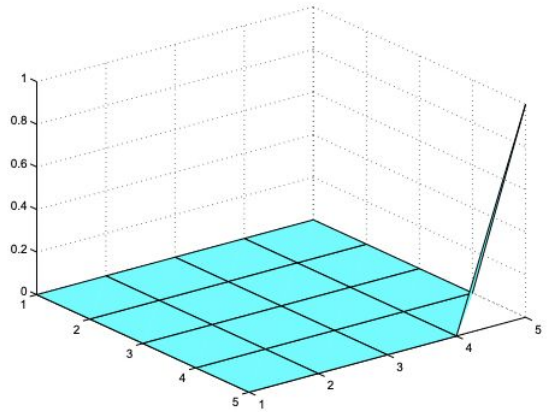


Fig 1: True reward function for 5x5 grid world

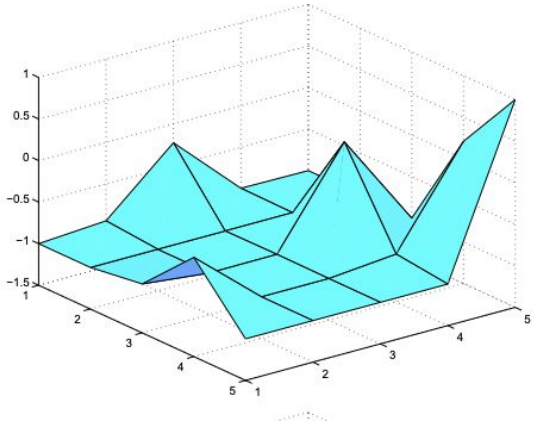


Fig 2: Reward function for 5x5 grid world after using max margin

Projection

- Projection also finds a reward function R, but here R is not arbitrary (unlike the Max margin case), but is a linear combination of known features. So, reward function R is defined as:

$$R(s) = \alpha_1\phi_1(s) + \alpha_2\phi_2(s) + \ldots + \alpha_d\phi_d(s)$$

- Now the value function for the optimal policy π is defined as:

$$V^{\pi} = \alpha_1 V_1^{\pi} + \ldots + \alpha_d V_d^{\pi}$$

- Now the final inequality for the projection algorithm proves the optimality of action by ensuring that performs at least as well as all other actions in the set of actions A. The inequality is:

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

Experimental Results using projection

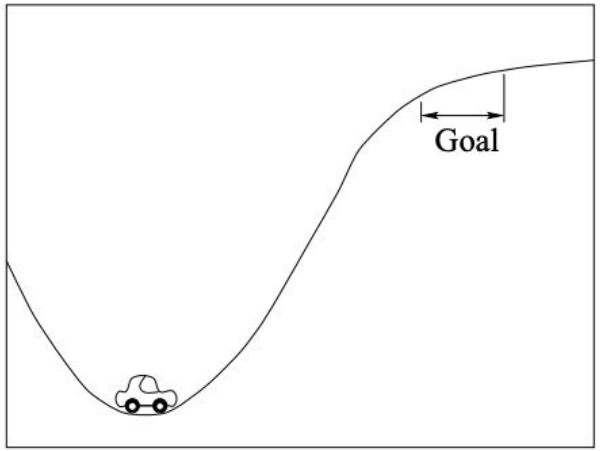


Fig 1: True reward function for mountain car

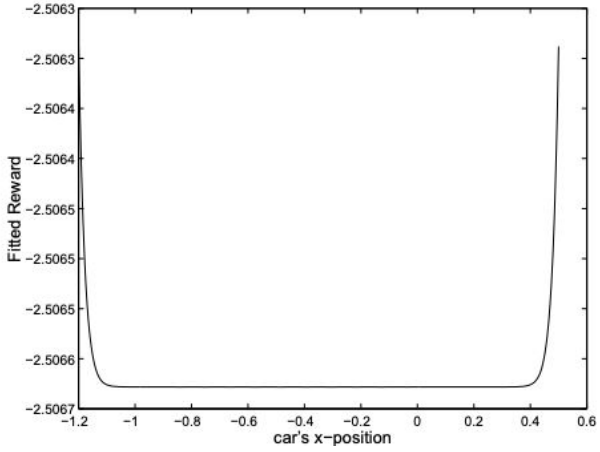


Fig 2: Reward function after running projection on the mountain car

Apprenticeship Learning: An Application of IRL

- Observes expert to learn optimal reward function; tries to find policy π such that discovered reward function within epsilon of true reward function or:

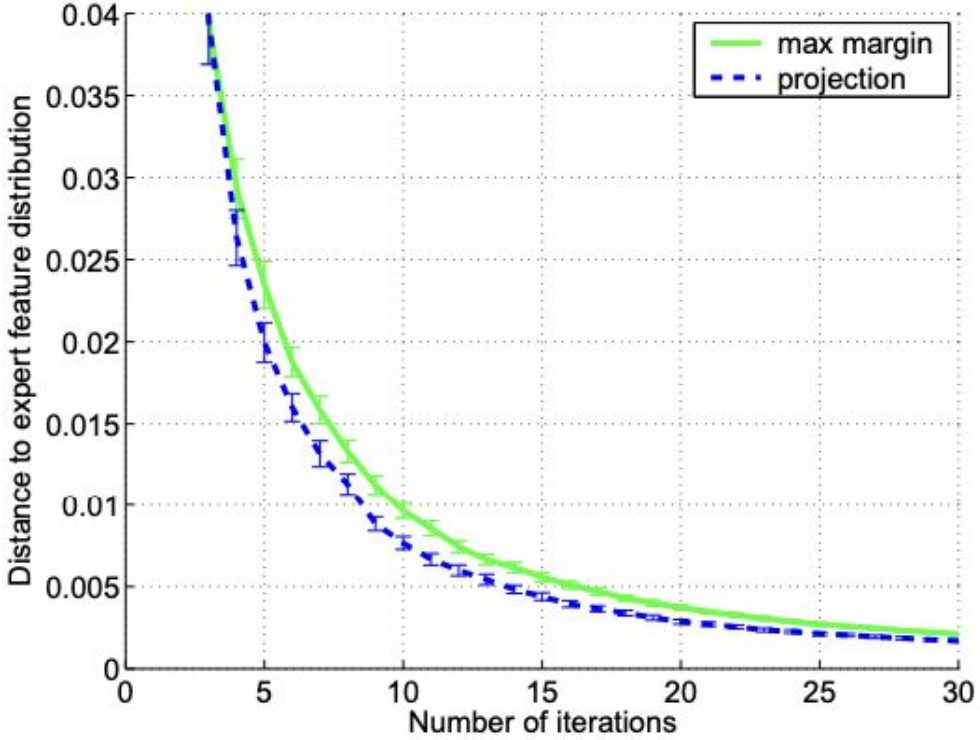
$$|\mu\pi - \mu E| = \epsilon$$

- Steps of algorithm:
  - Pick arbitrary policy π
  - Apply IRL algorithm to discover true reward function of expert using π picked
  - If difference between true reward function and expert reward <= epsilon, end, otherwise continue
- Iterations to find optimal policy π:

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

(quasi linear in terms of number of policies k)

Experimental Results for Apprenticeship Learning



Max margin and projection algorithms compared to expert's feature distribution with respect to an increasing number of iterations in a 128x128 grid world

Conclusions

- MARL experiments consistently show improvements from modifying RL models to explicitly model other agents
- But… there's no general framework for evaluating these techniques against one another, or understanding where one technique might be more appropriate than another
- IRL can provide such a framework, since its focus on learning underlying reward functions can be applied in any MARL environment
- This is a promising avenue for MARL moving forward

Thank you!