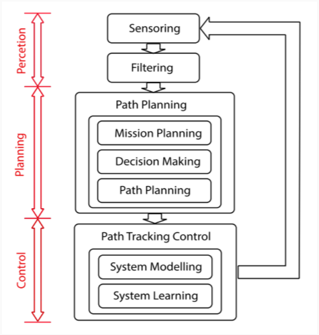
Advanced planning for autonomous vehicles using reinforcement learning and deep inverse reinforcement learning

You, Lu, Filev, Tsiotras

Laura Graves

Overview

Implements planning for autonomous vehicles using RL and inverse RL systems.

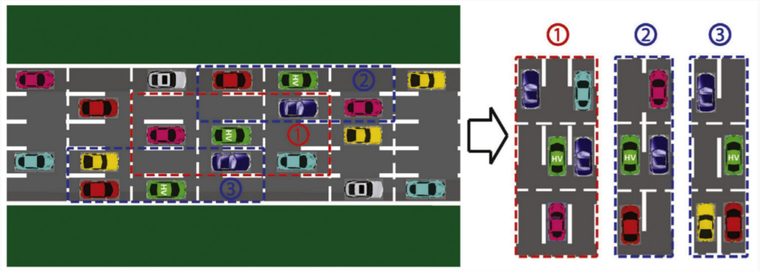


This is done in two ways:

- Develop a reward function matching desired behavior and use an MDP to develop a policy
- Extract a reward function from recordings of an expert driver and use that reward function to develop a policy

Environment

The head vehicle (HV) is in an environment that positions the surrounding environment vehicles (EVs) and head vehicle in a multi-lane street.

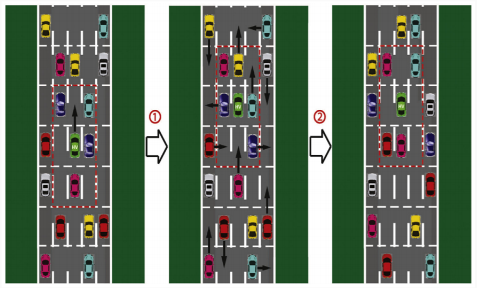


If the vehicle is in the center lane, the number of states is $2^8 = 256$, and if it is in a side lane the number of states is $2^5 = 32$, for a total of 320 states. Further, the researchers count left-turn and right-turn road segments for a total of 960 states.

The following actions are defined: $A = \{$maintain, accelerate, brake, left-turn, right-turn$\}$.

State Transitions

State transactions are done in a turn-order manner, where the HV selects and executes an action and the EVs select and execute an action based on the HV action. An environmental policy ensures that the EVs are not permitted to take actions that result in collision.



The above graphic shows one transition cycle, where the HV selects the "accelerate" action and the surrounding vehicle subsequently select their actions. The HV selects their action based only on the surrounding vehicles, denoted by the red grid.

Manually Engineered Reward Function

The first approach the authors took was to develop their own reward functions. These take the form of a linear combination of features $R(s, a) = \omega^T \phi(s, a)$ where ω is the weight vector and $\phi(s, a)$ is the feature vector. In this work they define the following features:

- Action features
- HV position
- Overtaking strategy
- Tailgating
- Collision incident

With values for these rewards chosen to suit the desired behavior (for example, a vehicle that prefers overtaking has positive rewards for acceleration and negative rewards for braking), the authors used a standard Q-learning setup using ϵ-greedy action selection to determine the optimal policy.

Extracting the Reward Function from an Expert Driver

If a reward function is unknown, we can use inverse reinforcement learning to learn an optimal policy from observations taken from an expert.

The authors do this and ensure correctness in two ways:

First, using the maximum entropy principle and a function approximator, we can estimate the reward function from our set of observations. The maximum entropy principle ensures that our estimated reward function is based on the estimation that introduces the least amount of assumption from our observations.

Second, we can use the estimated reward function to extract an optimal policy using RL techniques. This can be done in a similar way to the previous method, using Q-learning or another MDP.

Maximum Entropy Principle

If we're given a set of observations in a distribution and a set of constraints, the maximum entropy principle lets us find the distribution that satisfies the constraints with the *largest remaining uncertainty* so we don't accidentally introduce any assumptions into our distribution.

In inverse reinforcement learning, we can utilize the maximum entropy principle to find a reward function depending on either a state $R(s)$, a state/action pair $R(s, a)$, or a state/action/next state tuple $R(s, a, s')$. In order to do so, we introduce a DNN that serves as a parameterized reward function, either in a form that takes s, a as features and produces $R(s, a,)$, or in a form that takes s as a feature and produces a reward vector where each value represents the reward given by each action.

Q-Learning with Model Learning

In the case without a model (where the transition table $\mathbb{P}(s'|s, a)$ is unknown), we are able to utilize Q-learning to extract a model. During the learning process, we can count the s, a, s' occurrences to estimate the transition probabilities using

$$\mathbb{P}(s'|s, a) = \frac{v(s,a,s')}{\sum_{s' \in S} v(s,a,s')}$$

As the number of observations of each s, a, s' tuple approaches infinity, the estimated probabilities $\mathbb{P}(s'|s, a)$ approach their actual values.

The authors implement model-learning as part of their Q-learning algorithm, so the Q-learning algorithm returns not only the Q values and policy, but also the model (see **Algorithm 2**).

The MaxEnt Deep IRL Algorithm

Putting all of this together brings us to the **MaxEnt Deep IRL Algorithm**, which I summarize as follows:

- Set the initial neural network parameters θ and visitation counts $v(s, a, s')$.
- while θ has not converged:
    - Update reward function $R = NN(\theta)$
    - Update policy, visitation counts, and model using Q-learning
    - Using policy and model, calculate expected s and s, a counts
    - Determine maximum-entropy gradients
    - Update θ using gradient (and any regularization terms)
- $\theta^* = \theta$
- $R* = NN(\theta^*)$
- Calculate π^* using Q-learning
- Return $\theta^*, R^*, \pi^*, \mathbb{P}$

MaxEnt IRL Refinement

Two further versions of this algorithm are proposed that attempt to minimize the error resulting from having a low number of visitations on some s, a pairs. Further, the number of demonstrations may not be enough to fully represent the random behavior of the environment, especially over long-term simulations.

The first algorithm views only a single time step at a time, and thus avoids calculating the expected transition probabilities altogether, instead calculating the gradient with regard to the policy and the expected empirical policy from the demonstrations.

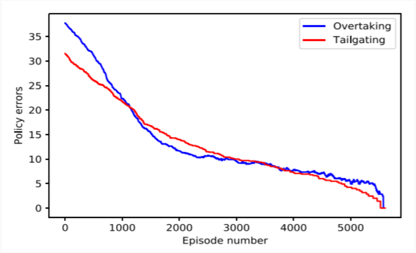
The second algorithm takes a small number of time steps, splitting the demonstrations into a number of small segments to avoid making long-term errors.

These refinements can be found in section 5.3.

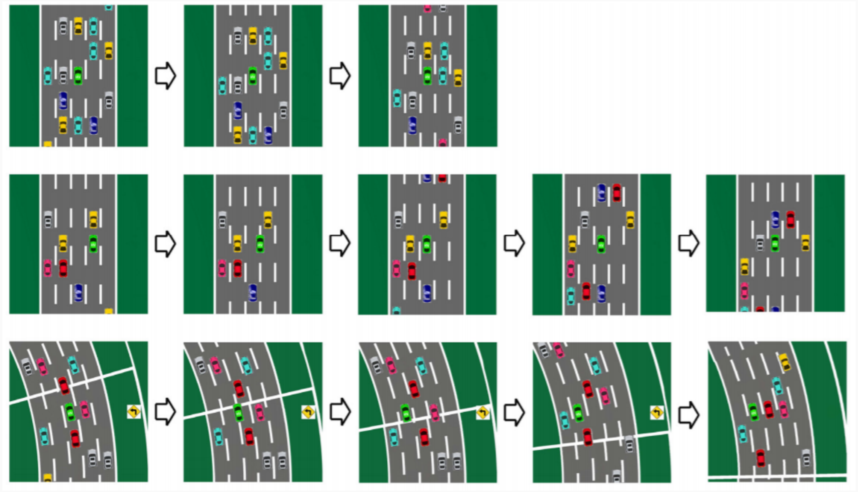
Results - Driving from RL

Simulations were done using pygame, where each road has a sequence of straight and curved segments, and each segment has five lanes. The HV is free to move about freely, and two initial reward functions have been chosen. There is an *overtaking* behavior, which rewards vehicles for moving lanes and accelerating to overtake vehicles in front of them. There is also an *tailgating* behavior, where the vehicle attempts to stay directly behind some vehicle through acceleration and lane changes.

The algorithm was ran until convergence for each behavior, and the convergent policy was used to simulate vehicle behaviors.



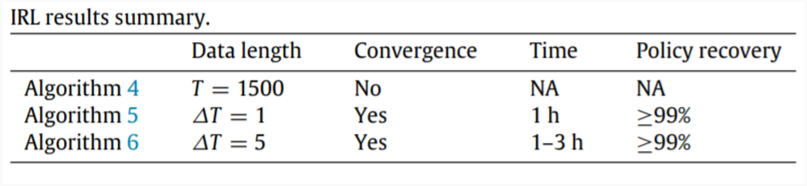
Simulated Behavior from RL



Results - Driving from IRL

To test the inverse RL algorithms, the authors used simulated data from the previous driving simulations to attempt to extract the reward function and policy that the previous vehicles had. The behavior of each vehicle was captured in a set of 500 simulations, with each simulation consisting of 1500 time steps.

The three MaxEnt algorithms were tested, and the authors found that the standard algorithm did not converge, while the modified algorithms did converge, recovering the policies with greater than 99% accuracy.



IRL results summary.

	Data length	Convergence	Time	Policy recovery
Algorithm 4	$T = 1500$	No	NA	NA
Algorithm 5	$\Delta T = 1$	Yes	1 h	$\geq$99%
Algorithm 6	$\Delta T = 5$	Yes	1–3 h	$\geq$99%

Discussion

The simulation is very simplistic, with some very big caveats. Vehicles move only in single-time steps, environment vehicles can't cause a collision, and there's no way for vehicles to be between cells or between lanes. A system that isn't as strictly parameterized would be much better.

The hand-tailored reward functions are insufficient to convince me that they correctly model that behavior.

I wish we had seen some other situations - for example, a policy that is able to safely move a vehicle in the far left lane into the far right lane (so they can exit the highway).