thinking wires

⊕

Inverse Reinforcement Learning pt. I

Published 2018-02-13 by Johannes Heidecke

Overview

In this blog post series we will take a closer look at *inverse reinforcement learning* (IRL) which is the field of learning an agent's objectives, values, or rewards by observing its behavior. For example, we might observe the behavior of a human in some specific task and learn which states of the environment the human is trying to achieve and what the concrete goals might be.

This is the first part of this series in which we will get an overview of IRL and look at three basic algorithms to solve the IRL problem. In later parts we will explore more advanced techniques and state of the art methods[1].

To follow this tutorial, basic knowledge in reinforcement learning (RL) is required. If you are not familiar with RL or want to refresh your knowledge, I recommend the following resources:

- UCL course on reinforcement learning by David Silver in 10 video lectures (~1 hour 30 minutes each)
- Blog series in seven parts: Dissecting Reinforcement Learning, especially suitable if you like programming and are familiar with python
- If you prefer learning from a book, the standard reference is this book by Sutton and Barto. The newest draft is available online for download as PDF

IRL versus RL:

First of all, let's look at what distinguishes inverse reinforcement learning from reinforcement learning. In RL, our agent is provided with a reward function which, whenever it executes an action in some state, provides feedback about the agent's performance. This reward function is used to obtain an optimal policy, one where the expected future reward (discounted by how far away it will occur) is maximal.

In IRL, the setting is (as the name suggests) inverse. We are now given some agent's policy or a history of behavior and we try to find a reward function that explains the given behavior. Under the assumption that our agent acted optimally, i.e. always picks the best possible action for its reward function, we try to estimate a reward function that could have led to this behavior.

In [1]⊕ this is informally characterized as follows:

Given:

1. measurements of an agent's behavior over time, in a variety of circumstances
2. if needed, measurements of the sensory inputs to that agent (environment states)
3. if available, a model of the environment (transition probabilities)

Determine: the reward function being optimized.

Since points 2 and 3 in the list above also appear in any reinforcement learning problem, the key characteristic of IRL is being given observed behavior of some other agent. We can boil this down to the following table, only focussing on the differences of IRL and RL:

	IRL	**RL**
given	policy π or history sampled from that policy	(partially observed) reward function $\mathcal{R}$
searching	reward function $\mathcal{R}$ for which given behavior is optimal	optimal policy π for given reward

IRL and inverse optimal control (IOC) are often used as synonyms. If you see some paper talking about IOC, the most important difference to be aware of is that optimal control talks about costs while RL talks about rewards. We can simply turn rewards into costs and vice versa by adding a negative sign.

Motivation

Why might we be interested in searching for the reward function of a given problem? In most reinforcement learning tasks there is no natural source for the reward signal. Instead, it has to be hand-crafted and carefully designed to accurately represent the task. Often, engineers manually tweak the rewards of the RL agent until desired behavior is observed. A better way of finding a well fitting reward function for some objective might be to observe a (human) expert performing the task in order to then automatically extract the respective rewards from these observations.

As Ng and Russel state in their foundational paper on IRL from 2000 [1]:

> After all, the entire field of reinforcement learning is founded on the presupposition that the reward function, rather than the policy, is the most succinct, robust, and transferable definition of the task.

The reward function is a *succinct* description of a task since it describes in a compact way which states of the environment are desirable. On the other hand, if we use a policy to describe a task, it will be less succinct since for each state we have to give a description of what the behavior should look like. This only tells us implicitly, what the task is actually about. In a way, the reward function captures the *salient* parts of the task, while the policy will contain many (potentially irrelevant) intermediate steps of reaching the task's objective.

The reward function is in some way more *robust* than the policy. When given the policy or trajectories sampled from it, we can use it for *behavior cloning*, i.e. exactly mimicing what the observed agent did. However, this will not be robust to any changes in the environment's transition mechanics. If some part of the environment changes, behavior cloning will lead to suboptimal behavior, while on the other hand the reward function contains enough information to robustly adapt the agent's behavior to still achieve the task.

The reward function is also a *transferable* description of the task. Even when the observed agent is very different from the target agent (e.g. has an entirely different set of actions, as is common with human and robots), the reward function will contain all relevant information in a way that is usable for the other agent. Clearly, this is important when trying to transfer human tasks to robots.

Maybe the biggest motivation for IRL is that it is often immensely difficult to manually specify a reward function for a task $\oplus$. If we look at successful RL applications right now, such as Alpha Go Zero for the game Go, most of them are games which naturally provide a reward signal (i.e. winning or losing, or the score achieved). So far, RL has been successfully applied in domains where the reward function is very clear. But in the real world, it is often not clear at all what the reward should be and there are rarely intrinsic reward signals such as a game score. Manually designing and tweaking a reward function for a task is very challenging and comes with many pitfalls and potential for errors. In many cases it is easier to instead observe expert behavior and let them demonstrate how to achieve the

desired goal. Instead of simply copying this expert behavior, we can then try to *learn* the underlying reward function which the expert is trying to optimize.

Let's say we want to design an artificial intelligence for a self driving car. Think about your own 'reward structure' when driving a car. Can you specify exactly, how you would trade off between safety, speed, comfort, gas usage, legal risks, and the many other aspects that are taken into account while driving? When exactly are you rewarded for driving? When reaching your destination? When driving safely? How do you trade-off between different aspects? This should make clear, how hard it would be to manually specify a reward function for the task *driving*. IRL tries to solve this problem and offer automatically extracted reward functions based on observed behavior.

Another motivation for IRL is that it might provide us with a theoretical *model* that describes the objectives of some agent that we observe. This can be relevant in many different scenarios, e.g. biologists trying to understand animal behavior or social scientists or psychologists aiming for more understanding of human values. In the case that one day some artificial intelligence reaches super-human capabilities, IRL might be one approach to understand what humans want and to hopefully work towards these goals.

Notation

For this blog series we will look at many different IRL publications which proposed a variety of different algorithms and solutions. Even though they mostly talk about the same things (MDPs, states, actions, ...) many of them use different notations and symbols. I will use a unified notation here (which might in turn deviate from specific papers). All symbols and notation used will be introduced in this section:

A Markov decision process (MDP) is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, T, \gamma \rangle$ with:

- $\mathcal{S}$ is the set of n environment states
- $\mathcal{A}$ is the set of k actions
- $\oplus \mathcal{R}$ is the reward function which maps each state to a real valued reward $\mathcal{S} \to \mathbb{R}$
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ contains the transition probabilities, e.g. $T(s'|s, a)$ is the probability of landing in state s' after performing action a in state s
- γ is the discount factor for future rewards

A policy π maps states to actions. If the policy is stationary deterministic, it outputs exactly one deterministic action for each state: $\pi : \mathcal{S} \to \mathcal{A}$. If the policy is stochastic,

it will contain probabilities of choosing each action for each state:
$\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$.

In some cases, the observed behavior will not be provided as a policy, but instead as a history of trajectories ζ through the MDP. A trajectory is an ordered list of states and actions which were sampled from some policy: $\zeta = \langle (s_0, a_0), (s_1, a_1), \ldots \rangle$. The indices in this list correspond to the progression of time, i.e. s_0 is the first observed state, s_1 the second observed state, and so on.

We will refer to the real reward function (only known to the demonstrator / expert) as $\mathcal{R}$ and to the estimated reward function as $\hat{\mathcal{R}}$.

The value of a state s under policy π is denoted as $V^{\pi}(s)$. By the Bellman equation, this value is defined as:

$$V^{\pi}(s) = \mathcal{R}(s) + \gamma \sum_{s' \in \mathcal{S}} T(s'|s, \pi(s)) V^{\pi}(s')$$

The value of state-action pairs under some policy π is denoted as $Q^{\pi}(s, a)$ and is defined as:

$$Q^{\pi}(s, a) = \mathcal{R}(s) + \gamma \sum_{s' \in \mathcal{S}} T(s'|s, a) V^{\pi}(s')$$

IRL algorithms

Over the course of this blog post series we will look at a variety of different IRL algorithms which were proposed in the last ~20 years. In this post, we will look at the three algorithms proposed by Ng and Russel in 2000 in their paper "Algorithms for Inverse Reinforcement Learning". They introduced high-level descriptions for three algorithms, one for each of the following scenarios:

- Optimal policy π is known. Small state space
- Optimal policy π is known. Large or infinite state space
- Optimal policy π is unknown. Behavior trajectories ζ can be sampled from π

In later parts of this series, we will look at more modern approaches, for example:

- Apprenticeship Learning via IRL, Abbeel & Ng, 2004
- Bayesian IRL, Ramachandran & Amir, 2007

- Maximum Entropy IRL, Ziebart et al., 2008
- Maximum Causal Entropy IRL, Ziebart et al., 2010
- Maximum Entropy Deep IRL, Wulfmeier et al., 2016

Let's start now with the three basic approaches to IRL from paper [1].

Linear Programming for small state space

The first IRL algorithm we will look at is meant for finite, small state spaces, where we know the expert's policy π. Let's start by defining the *solution set* for the IRL problem: the set of all reward functions for which the given policy is optimal.

The policy π observed from the agent is optimal and stationary deterministic (for a given state it always chooses the same action, $\pi : \mathcal{S} \rightarrow \mathcal{A}$). $\mathbf{T}^{\pi}$ is the policy transition matrix of size $n \times n$ containing state transition probabilities for always choosing $\pi(s)$ in each respective state. (If something is not entirely clear, it might help to scroll down and look at the example)

For each state, there are $k-1$ actions a that are not picked by the policy ($a \neq \pi(s)$). We can order these other actions arbitrarily for each state and construct $k-1$ non-policy transition matrices $\mathbf{T}^{\neg\pi} = \{\mathbf{T}^1, \dots \mathbf{T}^{k-1}\}$. $\mathbf{T}^i$ is a $n \times n$ transition matrix containing state transition probabilities for always choosing the i-th non-policy action in each respective state. $\oplus$

Let $\succeq$ denote vectorial inequality, i.e. $\mathbf{x} \succeq \mathbf{y}$ iff $\forall i : \mathbf{x}_i \geq \mathbf{y}_i$. E.g. $(1,2,4)^T \succeq (1,1,3)^T$.

The solution set can then be defined as follows:

$$\forall \mathbf{T}^i \in \mathbf{T}^{\neg\pi} : (\mathbf{T}^{\pi} - \mathbf{T}^i)(\mathbf{I} - \gamma \mathbf{T}^{\pi})^{-1}\mathbf{R} \succeq 0$$

This inequality above will probably not look obvious to you at first sight, so let's have a look at how it was derived:

The solution set is the set of all reward functions for which the given policy π is optimal. This means that the policy-chosen action $\pi(s)$ at any state s corresponds to an expected value which is larger or equal than the expected value of all other actions in this state:

$$\forall s \in \mathcal{S} : \pi(s) = \text{argmax}_{a \in \mathcal{A}} \sum_{s'} T(s'|s,a)V^{\pi}(s')$$

which is equivalent to:

$$\Leftrightarrow \forall s \in \mathcal{S}, \forall a \in \mathcal{A} : \sum_{s'} T(s'|s, \pi(s)) V^{\pi}(s') \geq \sum_{s'} T(s'|s, a) V^{\pi}(s')$$

In the inequality above, we force the expected value of the policy actions $\pi(s)$ to be larger or equal than the expected value of any other action for all states. We can re-formulate this using matrices and vectors $\oplus$ for transitions and values, where $\mathbf{V}^{\pi}$ is a n-dimensional vector containing the state values under the policy π:

$$\Leftrightarrow \forall \mathbf{T}^{i} \in \mathbf{T}^{\neg \pi} : \mathbf{T}^{\pi} \mathbf{V}^{\pi} \succeq \mathbf{T}^{i} \mathbf{V}^{\pi}$$

Let $\mathbf{R}$ be the n-dimensional vector containing the rewards for all n states in $\mathcal{S}$. Since our policy is stationary deterministic and always chooses a single action $\pi(s)$ in some state s, we can rewrite the policiy's value as:

$$\begin{aligned} \mathbf{V}^{\pi} &= \mathbf{R} + \gamma \mathbf{T}^{\pi} \mathbf{V}^{\pi} \\ \Leftrightarrow \mathbf{V}^{\pi} - \gamma \mathbf{T}^{\pi} \mathbf{V}^{\pi} &= \mathbf{R} \\ \Leftrightarrow (\mathbf{I} - \gamma \mathbf{T}^{\pi}) \mathbf{V}^{\pi} &= \mathbf{R} \\ \Leftrightarrow \mathbf{V}^{\pi} &= (\mathbf{I} - \gamma \mathbf{T}^{\pi})^{-1} \mathbf{R} \end{aligned}$$

Using this, we can go back to our definition of the solution set and re-write it:

$$\begin{aligned} &\forall \mathbf{T}^{i} \in \mathbf{T}^{\neg \pi} : \mathbf{T}^{\pi} \mathbf{V}^{\pi} \succeq \mathbf{T}^{i} \mathbf{V}^{\pi} \\ \Leftrightarrow &\forall \mathbf{T}^{i} \in \mathbf{T}^{\neg \pi} : \mathbf{T}^{\pi} (\mathbf{I} - \gamma \mathbf{T}^{\pi})^{-1} \mathbf{R} \succeq \mathbf{T}^{i} (\mathbf{I} - \gamma \mathbf{T}^{\pi})^{-1} \mathbf{R} \end{aligned}$$

which is equivalent to our original definition of the solution set$\oplus$:

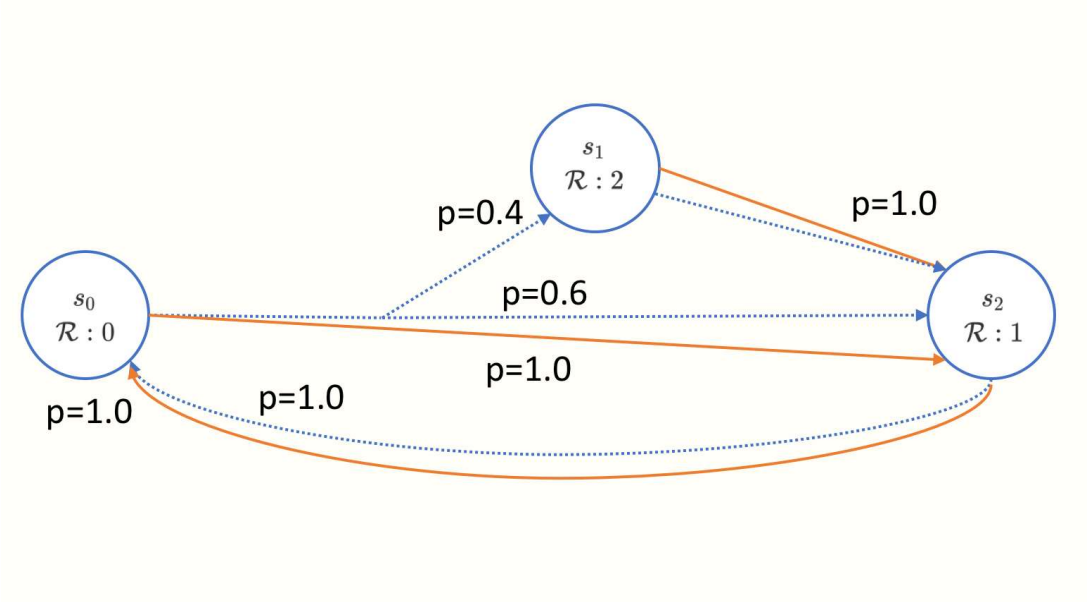
$$\begin{aligned} \Leftrightarrow \forall \mathbf{T}^{i} \in \mathbf{T}^{\neg \pi} : \mathbf{T}^{\pi} (\mathbf{I} - \gamma \mathbf{T}^{\pi})^{-1} \mathbf{R} - \mathbf{T}^{i} (\mathbf{I} - \gamma \mathbf{T}^{\pi})^{-1} \mathbf{R} \succeq 0 \\ \Leftrightarrow \forall \mathbf{T}^{i} \in \mathbf{T}^{\neg \pi} : (\mathbf{T}^{\pi} - \mathbf{T}^{i}) (\mathbf{I} - \gamma \mathbf{T}^{\pi})^{-1} \mathbf{R} \succeq 0 \end{aligned}$$

The inequality above gives us a set of n linear inequalities for each of the $k-1$ non-policy actions. We can use a basic linear programming algorithm to solve them and find solutions.

Example:

Let's look at a very simple example which will help to understand how this would be applied in practice. Let's assume we have three states and two actions with the reward structure and transition probabilities as depicted in the following figure:

⊕



First of all, let's assume we observed the following optimal policy π:

$\mathbf{s_i}$	$\pi(\mathbf{s_i})$
s_0	blue
s_1	orange
s_2	orange

This would correspond to the following policy transition matrix $\mathbf{T}^{\pi}$ (the transition probabilities for each state when following the policy's actions)⊕ :

$$\mathbf{T}^{\pi} = \begin{bmatrix} 0 & 0.4 & 0.6 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Since there are only two actions ($k = 2$), there will only be one non-policy transition matrix $\mathbf{T}^{1}$. It will contain the transition probabilities for always performing the *other*, non-policy action:

$$\mathbf{T}^{1} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

We now construct the inequalities corresponding to $\mathbf{T}^{\pi}$ being better than $\mathbf{T}^{1}$. These inequalities contain the three unkown rewards for each state ($\hat{\mathcal{R}}(s_0)$, $\hat{\mathcal{R}}(s_1)$, $\hat{\mathcal{R}}(s_2)$) which we want to estimate:

$$
\left(\begin{bmatrix}0 & 0.4 & 0.6\\ 0 & 0 & 1\\ 1 & 0 & 0\end{bmatrix} - \begin{bmatrix}0 & 0 & 1\\ 0 & 0 & 1\\ 1 & 0 & 0\end{bmatrix}\right)\left(\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} - \gamma\begin{bmatrix}0 & 0.4 & 0.6\\ 0 & 0 & 1\\ 1 & 0 & 0\end{bmatrix}\right)^{-1}\begin{bmatrix}\hat{\mathcal{R}}(s_0)\\ \hat{\mathcal{R}}(s_1)\\ \hat{\mathcal{R}}(s_2)\end{bmatrix} \succeq
$$

$$
\Leftrightarrow \begin{bmatrix}0 & 0.4 & -0.4\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}\begin{bmatrix}1 & -0.36 & -0.54\\ 0 & 1 & -0.9\\ -0.9 & 0 & 1\end{bmatrix}^{-1}\begin{bmatrix}\hat{\mathcal{R}}(s_0)\\ \hat{\mathcal{R}}(s_1)\\ \hat{\mathcal{R}}(s_2)\end{bmatrix} \succeq 0
$$

$$
\Leftrightarrow \begin{bmatrix}-0.1619 & 0.3417 & -0.1799\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}\begin{bmatrix}\hat{\mathcal{R}}(s_0)\\ \hat{\mathcal{R}}(s_1)\\ \hat{\mathcal{R}}(s_2)\end{bmatrix} \succeq 0
$$

This simplifies to the three following equations, of which clearly only the first one is useful:

$$
\begin{gathered}
-0.1619\hat{\mathcal{R}}(s_0) + 0.3417\hat{\mathcal{R}}(s_1) - 0.1799\hat{\mathcal{R}}(s_2) \geq 0\\
0 \geq 0\\
0 \geq 0
\end{gathered}
$$

Let's see if our vectorized original reward function $\mathcal{R} = (0, 2, 1)^T$ fulfills the above inequality

$$
-0.1619 \cdot 0 + 0.3417 \cdot 2 - 0.1799 \cdot 1 \approx 0.5036 \geq 0
$$

And indeed, as we can see the original reward function is part of our solution set. However, it is quite obvious that the solution set will contain many more reward functions, many of which are *degenerate*. For example, the function assigning 0 reward to all states will always be a solution. This is very problematic; how can we know which of the many reward functions in our solution set is the correct one?$\oplus$ In the next section we will look at some heuristics proposed in the same paper to remove these degenerate solutions.

Degeneracy Heuristics

Ng and Russel came up with some heuristics to remove degenerate solutions which can easily be incoprorated into their linear programming formulation.

The first one, one could call it *costly single-step deviaton*, tries to maximize the distance between the policy's Q-values and other action's Q-values:

$$
\text{maximize}: \sum_{s \in \mathcal{S}}(Q^{\pi}(s, \pi(s)) - \max_{a \in \mathcal{A} \setminus \pi(s)} Q^{\pi}(s, a))
$$

This heuristic will pick the reward function from our solution set which makes the difference between the policy action and the second best action in all states as large as possible. This will succesfully remove many degenerate reward functions (e.g. the 0-reward function), but it also makes quite strong assumptions on how to interpret the expert's behavior. In some cases, this heuristic might lead to reward functions that are very different (most likely more "extreme") than the real reward function.

The second heuristic proposed by Ng and Russel in the same paper assumes that (all other things being equal) reward functions with many small rewards are more natural and should be prefered. They introduce a penalty term to the maximization which leads to a minimization of the reward vector's norm $\oplus$, for example the L1-norm:

$$-\lambda||\hat{\mathbf{R}}||_1$$

The hyper-parameter λ can be used to trade-off between the single-step deviation heuristic and this heuristic. In their paper they introduce a method to automatically search a suitable λ.

In addition, a upper bound for the values of the reward function is enforced:

$$\forall s \in \mathcal{S} : |\hat{\mathcal{R}}(s)| \leq R_{\max}$$

These heuristics are combined with the constraints defining the solution set and passed to a linear programming solver. However, this approach would not work in infinite (or very large) state spaces. In the next section we will look at their proposed solution for this scenario.

Linear programming for infinite state spaces

The second algorithm proposed in [1] is an adaption of the previous algorithm for very large or even infinite state spaces. For the sake of concreteness we will limit ourselves to the case of $\mathcal{S} = \mathbb{R}^n$. To find the reward function $\hat{\mathcal{R}}$, we now need to work with the function space $\mathbb{R}^n \to \mathbb{R}$. While this is possible in theory, it is very hard to process algorithmically.

Instead, Ng and Russel use a simple trick from the toolbox of function approximation and limit their algorithm to find a linear combination of d fixed, known, and bounded basis functions ϕ_i:

$$\hat{\mathcal{R}}(s) = \alpha_1\phi_1(s) + \alpha_2\phi_2(s) + \ldots + \alpha_d\phi_d(s)$$

By doing this, we turned our difficult function search into a much easier linear combination of basis functions (with parameters α_i) which enables us to still use linear programming. However, this simplification comes with a huge drawback: we can no longer be sure that the true reward function will be in our solution set. Our

model becomes *biased* and can only express a smaller subset of all possible reward functions.

Before we turn to this issue, let's have a quick look at how V^π can be expressed in terms of the basis functions. We can calculate the value function for each basis function, V_i^π, assuming that $\hat{\mathcal{R}}(s) = \phi_i$. By the linearity of expectations, the value with the "full" estimated reward function will then be:

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

Now, how can we define the solution set using this value function? Our state space is now infinite (or very large), so we cannot express transition probabilities as matrices anymore. Instead, we will estimate the expected value based on a sampled subset of spaces $\mathcal{S}_0 \subset \mathcal{S}$ and enforce that policy actions lead to higher expected value than non-policy actions:

$$\forall s \in \mathcal{S}_0, \forall a \in \mathcal{A} \setminus \pi(s) : \mathbb{E}_{s' \sim T(s'|s,\pi(s))}[V^\pi(s')] \geq \mathbb{E}_{s' \sim T(s'|s,a)}[V^\pi(s')]$$

Since we know that V^π is linear in the parameters α_i, the inequality above will provide us with a set of linear equations based on which we can estimate fitting parameter values.

However, as we noticed earlier, the real reward function might not be representable as a linear combination of our selected basis function. This could lead to our solution set being empty as soon as two or more constraints are not fulfillable with the choice of any values for the parameters α_i. To resolve this issue, we *relax* our linear constraints and simply penalize whenever they are violated. For this we introduce some penalization function p given by $p(x) = x$ if $x \geq 0$, and $p(x) = c \cdot x$ otherwise. The hyperparameter c defines how much constraint violations should be penalized. In their paper, Ng and Russel chose $c = 2$ and stated that their results were not very sensitive to the choice of $c\oplus$.

The full linear programming formulation then becomes:

$$\text{maximize} \sum_{s \in \mathcal{S}_0} \text{min}_{a \in \mathcal{A} \setminus \pi(s)} p(\mathbb{E}_{s' \sim T(s'|s,\pi(s))}[V^\pi(s')] - \mathbb{E}_{s' \sim T(s'|s,a)}[V^\pi(s')])$$

$$\text{such that } |\alpha_i| \leq 1, i = 1, \ldots d$$

We can see that the formulation above tries to maximize the smallest difference it could find, i.e. it selects the best non-policy action and tries to maximize the difference to the policy action in each state. This is essentially the *costly single-step deviation* heuristic that we saw in the last section, just adapted for the case of large and infinite state spaces.

In their experiments for the paper, Ng and Russel used Gaussian basis functions which were spaced evenly over the state space.

Linear programming with sampled trajectories

The last of the three algorithms proposed by Ng and Russel in 2000 deals with the scenario where we don't know the exact policy but are able to observe trajectories ζ of states and actions generated by some optimal policy based on the unkown reward function. This is in general a much more realistic scenario; many of the more recent IRL methods are developed exactly for this case.

As in the previous algorithm, we will use a linear combination of known, fixed, and bounded basis functions ϕ_i to approximate the true reward function. However, now we want to calculate the empirical value of a trajectory, not the expected value of a policy, i.e. how much reward did some trajectory *actually* yield. To do this, we use our current estimate $\hat{\mathcal{R}}$ and calculate the corresponding value estimate $\hat{V}_i(\zeta)$ for each basis function:

$$\hat{V}_i(\zeta) = \sum_{s_j \in \zeta} \gamma^j \phi_i(s_j)$$

Since we set $\hat{\mathcal{R}}$ to be a linear combination of the basis function, the overall empirical value of a trajectory is

$$\hat{V}(\zeta) = \alpha_1 \hat{V}_1(\zeta) + \ldots + \alpha_d \hat{V}_d(\zeta)$$

Let's call the trajectory generated by the expert ζ_{π^*}. In some cases there might be several trajectories generated by the same expert (e.g. from different initial states), in this case we can construct the (weighted) mean of their respective empirical values. For the sake of simplicity in notation, let's assume that there is only one expert trajectory.

Now our goal is to find parameters α_i such that our optimal expert trajectory ζ_{π^*} yields a higher empirical reward than other trajectories which were generated by different policies. We could start with some arbitrary (e.g. random) policy π^1 and use it to generate a trajectory ζ_{π^1}. Clearly we want the expert trajectory to obtain more value than the other one:

$$\hat{V}(\zeta_{\pi^*}) \geq \hat{V}(\zeta_{\pi^1})$$

We now use linear programming to find the best fitting parameters α_i, just as in the algorithm before. However, just comparing the expert trajectory to one arbitrary random trajectory will most likely not suffice. Optimally we want to compare to many more trajectories and especially trajectories that are more "competitive" than random policies. This is where the inductive step of the algorithm comes in. After each new estimation of the reward function, we run a RL algorithm to find a policy which is optimal for the current reward estimate. This policy will most likely be better than the expert trajectories for the current reward estimate and if we use it for additional constraints it will hopefully drive the estimate closer to the actual reward function.

For each new policy π^i generated this way, we will generate a corresponding trajectory ζ_{π^i}. In the first iteration our set of competing trajectories only consisted of ζ_{π^1}. After m iterations of the algorithm the linear programming constraints will grow to

$$\forall \zeta_{\pi^{\neg *}} \in \{\zeta_{\pi^1}, \ldots, \zeta_{\pi^m}\} : \hat{V}(\zeta_{\pi^*}) \geq \hat{V}(\zeta_{\pi^{\neg *}})$$

This algorithm then runs for some large number of iterations until a satisfactory reward function estimate is found. For this algorithm to work we have to make two important assumptions:

- We can generate trajectories for any given policy
- Given any reward function, we can generate a policy which is optimal for this reward function

As in the approach before, we might run into cases where the true reward function cannot be expressed as a linear combination of the fixed basis functions. To still find a solution which is hopefully at least close to the real one, we again use the penalization function p which penalizes violated constraints more heavily than satisfied constraints are "rewarded". Also, just as in the algorithm before, we limit the maximum value of the parameters α_i. The final linear programming formulation then becomes:

$$\begin{aligned} &\text{maximize} \sum_{i=1}^{m} p(\hat{V}(\zeta_{\pi^*}) - \hat{V}(\zeta_{\pi^i})) \\ &\text{such that } |\alpha_i| \leq 1, i = 1, \ldots, d \end{aligned}$$

Summary

Congratulations! If you've made it this far you have succesfully learned the basics of inverse reinforcement learning and the three algorithms described in the constitutive paper of Ng and Russel from the year 2000. Of course, in the last years more approaches have been proposed and there is quite a lot more to learn. Before we explore these more recent papers in the upcoming posts of this series, let's do a quick summary of what we've seen so far.

One of the hardest challenges in many reinforcement learning tasks is that it is often difficult to find a good reward function which is both learnable (i.e. rewards happen early and often enough) and correct (i.e. leads to the desired outcomes). Inverse reinforcement learning aims to deal with this problem by *learning* a reward function based on observations of expert behavior.

The problem of IRL is to find a reward function under which observed behavior is optimal. This comes with two main problems:

- IRL is an *underspecified* problem: for most observations of behavior there are many if not infinite fitting reward functions. The set of solutions often contains many degenerate solutions, i.e. assigning zero reward to all states. In this post we've seen heuristics to tackle this problem. Later in this series, we will also investigate principled ways to arrive at one single solution.
- The IRL algorithms so far assume that the observed behavior was optimal. This is a strong assumption, arguably too strong when we talk about human demonstrations. Again, in later posts we will see approaches which make weaker assumptions about this.

We have seen three algorithms to solve the IRL problem, each of them for a different scenario. In the first two we are given the exact expert policy, first in a small state space and then in a large or infinite state space. In the third algorithm we do not have access to the expert's policy but we are provided with example histories of states and expert actions. All three algorithms use linear programming constraints to find a solution set and maximize some heuristics to ignore degenerate solutions and find the best possible one.

I encourage you to have a look at the results section of Ng's and Russel's paper and maybe even implement some experiments yourself. If you've done a cool project with IRL let me know and I will link it here.

What's next?

I am currently working on the next part of this series which will focus on the paper "Apprenticeship Learning via Inverse Reinforcement Learning" by Abbeel and Ng from the year 2004. Following posts will advance chronologically through important publications of the field.

Once I finished the next part, you will be able to find it here. If you want you can also add your email address to the newsletter below to receive an update when I finish another post.

English is not my first language, so if you find any mistakes or formulations that are hard to understand, please let me know. I'm happy about any kind of feedback; you can leave a comment or contact me privately.

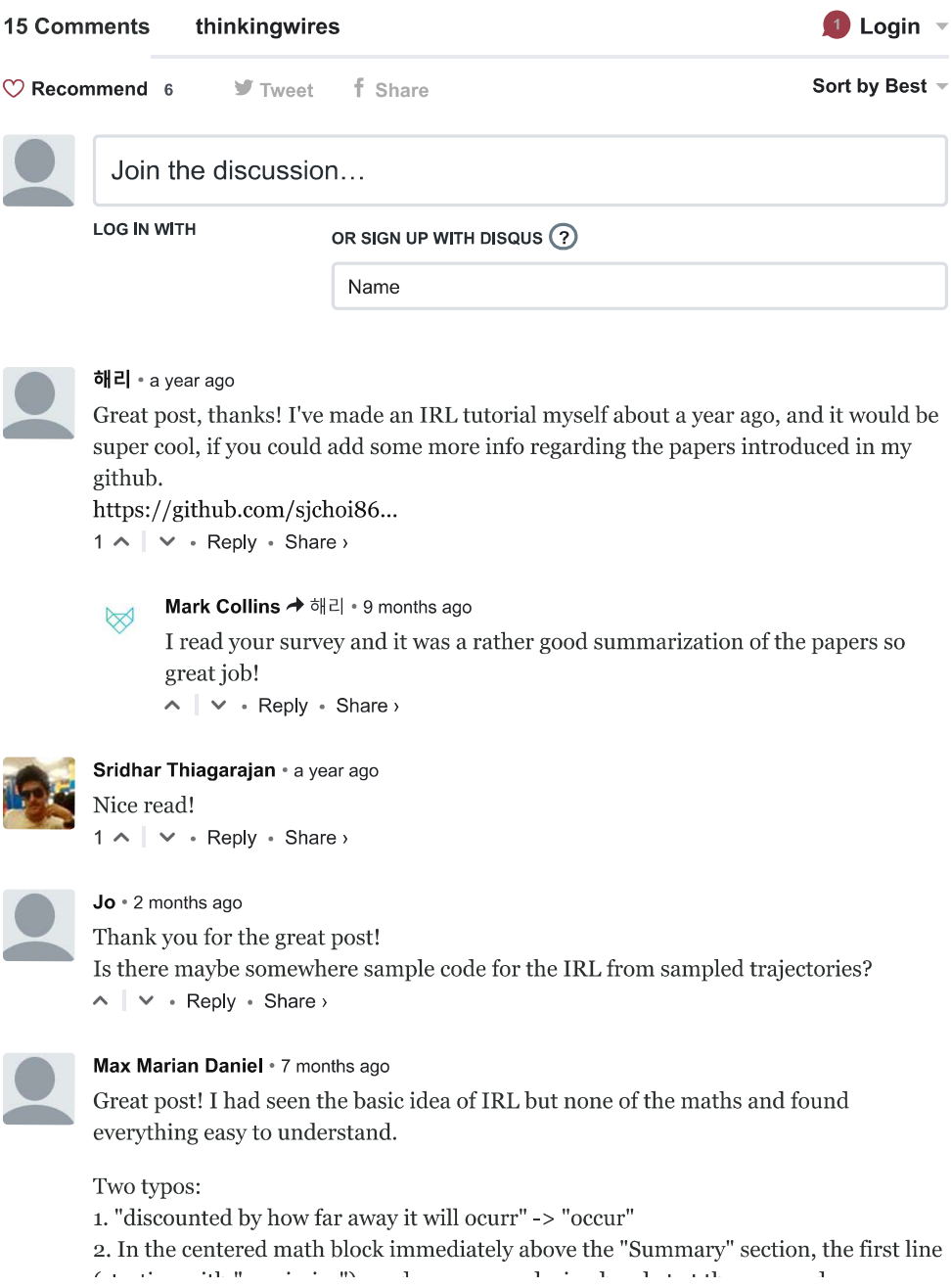
Subscribe to Newsletter

Subscribe here to automatically receive new posts on thinking wires via email:

email address

Subscribe

Back to main page



(starting with "maximize") needs one more closing bracket at the very end.

Reply • Share ›

thinkingwires Mod → Max Marian Daniel • 7 months ago

Thanks for your feedback! The typos should be fixed now :)

Reply • Share ›

HYUNKI SUNG • 7 months ago

So nice article! Thank you :)

Reply • Share ›

cgo • 8 months ago

I am confused about your example. T^\pi is the stationary policy, so I think the matrix should read T^\pi = [0 0 1\\ 0 0 1\\ 1 0 0]. The matrix you placed there with 0.4 and 0.6 are simplify transition probabilities (not policies?). Also, T^1 is supposed to be the nonpolicy action, meaning, T^1 = [0 1 0\\ 0 0 1\\ 1 0 0].

Reply • Share ›

Kranthi Kiran • 9 months ago

Great post! When is the part 2 coming?
Eagerly waiting for it. :)

Reply • Share ›

thinkingwires Mod → Kranthi Kiran • 7 months ago

I'm working on the next part now, but can't promise to finish it too soon ;)

Reply • Share ›

Shamane Siriwardhana • a year ago

where is the other post ?

Reply • Share ›

MB • a year ago

Typo: In the first inequality after "This simplifies to the three following equations, of which clearly only the first one is useful:", R^\hat(s_1) appears twice.

Reply • Share ›

MB → MB • a year ago

Also, after "The solution set can then be defined as follows:", I think you need an R in that inequality.

Reply • Share ›

thinkingwires Mod → MB • 10 months ago

Thanks for noting, fixed both!

Reply • Share ›