RL Project Report
Bayesian Inverse Reinforcement Learning

Niklas Kaspareit
Max Serra

August 2022

1 Introduction

Reinforcement learning (RL) boils down to learning how to act in an environment. To choose among the set of possible actions, we use the concept of reward and in each state we take the action that maximises the expected cumulative reward. The reward is a (scalar) signal that measures "goal achievement", and it is commonly inherent to the environment and the state. During the semester we have learned about multiple paradigms and algorithms to find the such optimal policy (the one that maximises the expected cumulative reward) and this is all fine, as long as the reward is known.

But, what happens when we do not have access to the functional expression of the reward? How do we find the optimal policy in such situations? Can we define the reward function arbitrarily and then train a Reinforcement Learning agent?

Assuming we are asked to program a self driving car. How do we define the rewards? How much more important is it to drive safely than to get to our destination under reasonable time? If the rewards we assign are not balanced correctly we might get a very safe but slow driving car or a very fast and dangerous one. Hopefully the idea is clear; defining a reward function in a multidimensional and time changing environment is not an easy task and it might yield unintended behaviour. So how can we find the optimal policy?

When we have access to demonstrations by an "expert" (an agent that knows and behaves according to the optimal policy) we can attempt two tasks:

1. Learn the policy: given the behaviour of the expert we can act as similar to it as possible by replicating what the expert did in each state.

2. Learn the reward: from the demonstrations we can find a reward function that would yield the same behaviour.

The statement described in 1. is known as Imitation Learning, whereas in 2. the idea of Inverse Reinforcement learning (IRL) is defined. The difference might

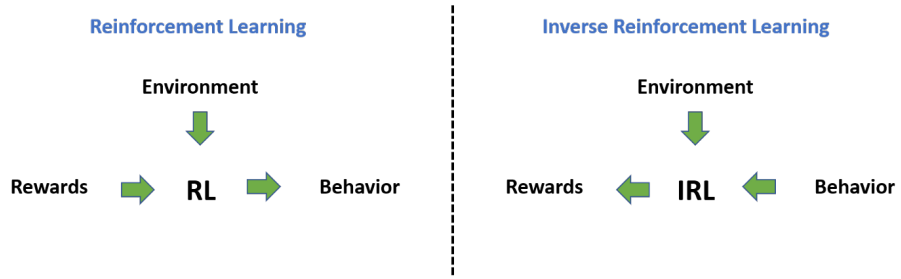

Figure 1: RL vs IRL (Source)

seem subtle but it is actually quite significant. The advantage of learning the reward over the policy is that the former is the most informative representation of the goal at hand. While the policy answers the "How to reach the goal?", the reward answers the "What is the goal?", and this is inherently a much more robust and transferable information. Knowing the reward opens the door to finding alternative optimal policies, or to adapting to minor alterations of the environment.

> Note from the authors: We believe it is worth at this point to raise a note on the matter of data requirements. It is clearly much more interesting to obtain the reward, but at what cost? It seems that Imitation Learning might be able to get by with little data from the expert while IRL needs much more data and is more computationally intensive.

Hopefully by now we have convinced you that the "higher" aim is to find the reward function using demonstrations by an optimal agent when the reward function is unknown. This is what Inverse Reinforcement Learning is about: reverse engineering the RL process (as shown in figure 1).

In the scope of this project we implement the "vanilla" IRL algorithm proposed in [Ng and Russell, 2000] to estimate the reward function from sampled trajectories (demonstrations by an expert agent). And we go one step further and looked into Bayesian IRL, as introduced by [Ramachandran et al., 2007], and also implement the algorithm they suggest to generate samples from the posterior distribution. In the following we will present our understanding of both papers.

2 Paper 1: Algorithms for IRL

In this paper three different scenarios are considered. One algorithm is presented for each of these. The notation in this report follows Section 2 of the paper, so you can refer to it in case of doubt.

As in most RL tasks, it is assumed that the underlying environment can be described according to a Markov decision processes (MDP). For a refresh on the topic, you can refer to Section 2 of the paper.

2.1 IRL in Finite State Spaces

In the scenario where the transition probabilities P are known, it can be derived from the Bellman equations and the Bellman optimality theorem that the optimal policy $\pi(s) = a_1$ is indeed optimal if for all the other actions in the action space it holds that:

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

This result has two weak points:

1. $R = 0$ satisfies the equation for all states
2. and there might be many choices of R that satisfy this.

To tackle these issues, the authors propose the heuristic of searching for the reward that makes the optimal action as distinct from the rest of the actions as possible. This can me formulated as:

$$maximise \sum_{s \in S} (Q^{\pi}(s, a_1) - max_{a \in A \backslash a_1} Q^{\pi}(s, a_1)) \tag{2}$$

This choice is "arbitrary" and other heuristics could be applied here, but this one seems reasonable enough. In combination with equation 1 this heuristic leads to the formulation:

$$\begin{aligned} maximise \sum_{i=1}^{N} min_{a \in A \backslash a_1} \{(P_{a_1}(i) - P_a(i))(I - \gamma P_{a_1})^{-1} R\} - \lambda \|R\|_1 \\ such\ that\ (P_{a_1} - P_a)(I - \gamma P_{a_1})^{-1} R \geq 0 \\ \forall a \in A \backslash a_1 \\ |R_i| \leq R_{max}, i = 1, ..., N \end{aligned} \tag{3}$$

Notice that we have added a λ term. This functions as a regularization term, pushing for the norm of R to be small.

This formulation seems quite complicated, but as indicated in the paper it can be formulated as a linear program. Linear programs (or programming) (LP) are a very simple and well studied system of equations, we refer the curious reader to the Wikipedia page on the topic.

2.2 Linear Function Approximation in Large State Spaces

When the state space is infinite we can still work with a MDP formulation but the transition probabilities P become unwieldy. To tackle this, the authors suggest to:

1. Generalize the equation 1 to the condition:
$$E_{s' \sim P_{sa_1}}[V^\pi(s')] \geq E_{s' \sim P_{sa}}[V^\pi(s')] \tag{4}$$
Which brings us back to the initial idea of optimality: maximise cumulative expected reward. Recall that the value function is a measure of precisely that.

2. Approximate the reward function in terms of a linear approximation:
$$R(s) = \alpha_1 \phi_1(s) + \alpha_2 \phi_2(s) + \cdots + \alpha_d \phi_d(s) \tag{5}$$
Where the ϕ_i are known and fixed real-valued function on the state space and the α_i their "weights".

The generalization in 1. is a "safe" move. Hence, we will not provide further details. However, the approximation in 2. might require some commenting for clarification. The choice of the basis function ϕ_i has an impact on the form and values that R can take. So choosing the ϕ_is should be done carefully. If the choice is inappropriate, we might not be able to successfully approximate the original reward function (think of trying to approximate a sinus with a single second order polynomial). The number of approximating functions d also plays a role in this (with enough second order polynomials, we could approximate a sinus rather well in a closed interval).

Given that we do not know which basis function will work "properly", it might be a good idea to be lenient with the condition 4 and instead of searching for strict solutions, penalise those that violate it. To this end we introduce the function p as a penalisation term:

$$p(x) = \begin{cases} x & \text{if } x \geq 0 \\ 2x & \text{if } x < 0 \end{cases} \tag{6}$$

This function makes the negative terms more negative, so it should work as expected. The authors of the paper mention that the results were not sensitive to the arbitrary factor of 2.

All of this leads to the following linear programming formulation:

$$\mathit{maximise} \sum_{s \in S_0} \mathit{min}_{a \in A \setminus a_1} \{p(E_{s' \sim P_{sa_1}}[V^\pi(s')] - E_{s' \sim P_{sa}}[V^\pi(s')])\}$$
$$\mathit{such\ that}\ |\alpha_i| \leq 1, \quad i = 1, ..., d \tag{7}$$

Where the value function $V^\pi(s)$ is a function of the α_i: $V^\pi = \sum_{i=1}^{d} \alpha_i V_i^\pi$ and V_i^π denotes the value function of the policy π when the reward function is only ϕ_i.

2.3 IRL from Sampled Trajectories

The two algorithms discussed above make a very strong assumption: the expert policy is available. Unfortunately this is not always the case, the more realistic case is that where we have demonstrations of the expert. In this section we discuss this more general scenario and we will provide a pseudo-code for the algorithm (something that is not available in the original paper).

For this algorithm we will again use the linear approximation 5 of the reward function R. We then need to solve the following linear programming:

$$\begin{aligned} &maximise \sum_{i=1}^{k} p(\hat{V}^{\pi^*}(s_0) - \hat{V}^{\pi_i}(s_0)) \\ &such\ that\ |\alpha_i| \leq 1, \quad i = 1, ..., d \end{aligned} \tag{8}$$

This task resembles the one in 7 and we also use the penalisation function p as defined in 6. However, instead of the expectations $E_{s' \sim P_{sa_1}}[V^{\pi}(s')]$ we use the estimated value function $\hat{V}(s_0)$.

The key difference to the previous scenarios is that now we do not have access to the policy, and therefore we cannot compute its value function for all states. However we can estimate it. To do so we use the following tricks:

1. For simplicity and because the state space is infinite, we set the initial state to always be the "dummy" s_0, whose next state is uniformly distributed over the whole state space.

2. We estimate the value function of s_0 by $\hat{V}^{\pi}(s_0) = \sum_{i=1}^{d} \alpha_i \hat{V}_i^{\pi}$ where $\hat{V}_i^{\pi}$ denotes the average empirical return of the expert trajectories when the reward function is only ϕ_i. This takes the following form:

$$\hat{V}_i^{\pi}(s_0) = \frac{1}{J} \sum_{j=1}^{J} \phi_i(s_0) + \gamma \phi_i(s_1^j) + \gamma^2 \phi_i(s_2^j) + \ldots + \gamma^{l_j} \phi_i(s_{l_j}^j) \tag{9}$$

   Where J is the number of trajectories, s_1^j denotes the state visited by trajectory j after s_0, and l_j is the length of trajectory j (which is determined by either an early stopping or reaching an absorbing final state).

Finally, and to clarify the meaning of $\pi_i \ for\ i = 1, \ldots, k$ in equation 8, we present the pseudo-code for the algorithm:

Algorithm 1: IRL from Sampled Trajectories

Data: sampled (optimal) trajectories
Result: estimated reward function
1. compute value estimates for the sampled trajectories $\hat{V}_i^{\pi^*}(s_0)\ for\ i = 1, \ldots, d$;
2. generate a random policy π_1;
3. initialize $k = 1$;
while *stopping condition not met* **do**
    4. generate trajectories for π_k and compute the it's value estimates $\hat{V}_i^{\pi_k}(s_0)$;
    5. obtain α_i^{new} by solving the LP problem 8;
    6. construct a reward function R^{new} with α_i^{new};
    7. find the policy π_{k+1} that optimizes R^{new} (via RL);
    8. increment k by 1;
end
return R^{new}

The stopping condition in the IRL Algorithm is left undefined, we have opted to perform the loop for a fixed number of iteration, but other criteria could have been used.

3 Paper 2: Bayesian Inverse Reinforcement Learning

This paper introduces a particularly interesting approach to IRL. The authors propose to use Bayes' Theorem (10), combining the observations from an expert agent with prior knowledge of how the reward function should be like to obtain a posterior distribution.

$$P(Reward|Observations) = \frac{P(Observations|Reward) \cdot P(Reward)}{P(Observations)} \tag{10}$$

In the following we will discuss how to compute the likelihood $P(Observations|Reward)$ and how to generate samples from the posterior with the proposed algorithm.

3.1 Likelihood

To obtain the posterior distribution the first step is to determine the likelihood of the observations given a reward function.

In contrast to the sampled trajectories we discussed in the previous section 2, here we assume that we have observation of the state-action pairs from the expert. We also make two important assumptions about the policy that the expert is using:

1. The policy aims to maximise the expected return and is not exploratory (no ϵ-greedy allowed)

2. The policy is the same for all trajectories and it does not change within trajectories.

Given the assumption in 2 we can apply independence to obtain:

$$P(\{(s_1, a_1), (s_2, a_2), \ldots, (s_k, a_k)\}|R) = P((s_1, a_1)|R)P((s_2, a_2)|R) \cdots P((s_k, a_k)|R) \tag{11}$$

The choice of the $P((s_i, a_i)|R)$ is now "free". However, since the expert is maximising the expected return: the larger the $Q^*(s_i, a_i)$, the more likely it should be to observe that state-action pair. The following function fulfills this condition (and it will make our life easier):

$$P((s_i, a_i)|R) = \frac{1}{Z_i} e^{\alpha Q^*(s_i, a_i, R)} \tag{12}$$

With which we can now rewrite the likelihood as:

$$P(\{(s_1, a_1), (s_2, a_2), \ldots, (s_k, a_k)\}|R) = \frac{1}{Z} e^{\alpha \sum_i^k Q^*(s_i, a_i, R)} \tag{13}$$

3.2 Posterior

Applying Bayes' Theorem we obtain a posterior distribution of the reward functions that can be used:

$$P(R|\{(s_1, a_1), (s_2, a_2), \ldots, (s_k, a_k)\}) = \frac{1}{Z'} e^{\alpha \sum_i^k Q^*(s_i, a_i, R)} P(R) \tag{14}$$

Here Z is just a normalising constant, α is a parameter that weights the confidence in the optimality of the expert and $P(R)$ is the prior distribution of the reward functions.

One could attempt now to find the MAP (Maximum a Posteriori) or the mean estimate of the posterior analytically, but that is hard task and we are better off with other (Monte Carlo) methods as we will now see.

3.3 PolicyWalk as sampling algorithm

We can now talk about the main contribution of this paper: the PolicyWalk sampling algorithm to obtain samples from the posterior distribution. This is similar to a Markov Chain Monte Carlo (MCMC) method. The key idea is to start off with a random reward function, and then check if moving slightly away from it translates into an increase of the posterior probability. If so, then we move in that direction, if not, we stay with some probability.

As with all MCMC methods, it is necessary that the Markov Chain converges rapidly enough to it's invariant distribution, so that we might sample from it

Algorithm `PolicyWalk`(Distribution P, MDP M, Step Size δ)

1. Pick a random reward vector $\boldsymbol{R} \in \mathbb{R}^{|S|}/\delta$.
2. $\pi :=$ `PolicyIteration`$(M, \boldsymbol{R})$
3. Repeat
   (a) Pick a reward vector $\tilde{\boldsymbol{R}}$ uniformly at random from the neighbours of $\boldsymbol{R}$ in $\mathbb{R}^{|S|}/\delta$.
   (b) Compute $Q^{\pi}(s, a, \tilde{\boldsymbol{R}})$ for all $(s, a) \in S, A$.
   (c) If $\exists (s, a) \in (S, A), Q^{\pi}(s, \pi(s), \tilde{\boldsymbol{R}}) < Q^{\pi}(s, a, \tilde{\boldsymbol{R}})$
      i. $\tilde{\pi} :=$ `PolicyIteration`$(M, \tilde{\boldsymbol{R}}, \pi)$
      ii. Set $\boldsymbol{R} := \tilde{\boldsymbol{R}}$ and $\pi := \tilde{\pi}$ with probability $\min\{1, \frac{P(\tilde{\boldsymbol{R}}, \tilde{\pi})}{P(\boldsymbol{R}, \pi)}\}$
      Else
      i. Set $\boldsymbol{R} := \tilde{\boldsymbol{R}}$ with probability $\min\{1, \frac{P(\tilde{\boldsymbol{R}}, \pi)}{P(\boldsymbol{R}, \pi)}\}$
4. Return $\boldsymbol{R}$

Figure 2: Figure 3. from the paper showing the pseudo-code. (Original paper)

efficiently. This is covered in enough detail by the authors and proved with Theorem 4 so we refer the curious reader to the original paper for the proof.

3.4 Reward and policy learning

With the ability to sample from the posterior, we now ask ourselves how to handle these samples to obtain the most meaningful results. Is the MAP or the mean the more relevant result?

Section 4.1 of the paper (and specifically Theorem 2) deals with the task of reward learning. Theorem 2 states that the MAP of the posterior (when using a Laplacian prior) is actually equivalent to the result of the IRL algorithm we have seen in the previous section 2. However, in the more general case where the posterior might present multiple local maxima, it might be more informative to report the mean.

Section 4.2 of the paper focuses on policy learning, and for this case the authors prove with Theorem 3 that the policy that optimizes the mean reward of the posterior will be the optimal policy.

4 Our implementation

Hopefully by now we have conveyed the essence of the problem and how these two papers try to solve it. We now jump to a "guide" of our work within the

scope of this project. Here we will just give an overview of the most relevant (and painful) aspects of the implementation, which is now publicly available in this GitHub repository.

4.1 Overview

Early in this project we decided to stick to a rather simple and easy to interpret environment on which we would test the different algorithms: a Gridworld. So this is the only environment we have used along the project.

We also implemented two RL algorithms, value iteration and Q-learning. These are regarded as "agents" that learn the optimal policy when the reward is given. We also implemented a similar agent for the IRL task. For BIRL the agent is only included in a jupyter notebook for now.

4.2 Solving the LP problem

For the IRL algorithm 1 the core part is solving the LP problem. And it turned out to be one of the trickiest parts to formulate. Fortunately, we now have a rather clean function implementing it:

```
def solve_lp(self, target_estimate, candidate_estimates):
    """Solve the Linear programing task at hand:
    Maximize: sum over i of p(V_target - V_candidate_i) s.t. |alpha_i| <= 1
    Where V_target and V_candiate_i are the product of the estimates times the alphas
    And where p is x if x >= 0 and 2x if x < 0
    """

    target_value = np.dot(np.array(target_estimate), self.alphas)  # this returns a scalar
    candidate_values = np.dot(np.array(candidate_estimates), self.alphas)  # this returns an array of len(candidate_estimates)

    lp_input = np.zeros(len(self.d_centers))

    for i in range(len(candidate_estimates)):
        if target_value - candidate_values[i] >= 0:
            lp_input += np.array(candidate_estimates[i]) - np.array(target_estimate)
        else:
            lp_input += self.penalty_factor * (np.array(candidate_estimates[i]) - np.array(target_estimate))

    res = linprog(lp_input, bounds=(-1, 1), method="simplex")

    self.alphas = np.array(res.x)
    return self.alphas
```

Figure 3: Function for solving the LP problem in the IRL algorithm.

Ultimately we are using a package solver (from scipy).

4.3 Computing the posterior

In the PolicyWalk algorithm 2, in step 3.(c) the probability distributions P take two arguments; R and π. The implementation of uidilr helped a lot for better understanding as the paper does not fully elaborate this.

Actually, the PolicyWalk is (as the name properly states) exploring both the reward and the policy space in a rather intertwined way. Moreover, in our implementation we are computing the evidence (the sum of Q values for the

observations) with the Q function of the current policy, not the optimal one. By applying this we achieve rather promising results. Unfortunately the paper does not provide any proof or further details on this and there is not enough time within this project to research this in detail.

```python
def compute_log_posterior(gw_env:Grid_World,
                          observations: List[List[Tuple[Any]]],
                          reward: np.ndarray,
                          policy: Dict[Any, Any],
                          prior: Callable,
                          alpha: float,
                          gamma: float
                         ) -> float:

    gw_env.set_board(new_board=reward)
    q = perform_q_function_evaluation(gw_env=gw_env, policy=policy)

    log_p = 0

    for observation in observations:
        log_p += np.sum([alpha * q[s][a] - np.log(np.sum(np.exp(alpha * np.array(list(q[s].values()))))) for s, a in observation])
        #### Should we use np.mean() here? our obs are not all of the same length
        # log_p += np.mean([alpha * q[s][a] - np.log(np.sum(np.exp(alpha * np.array(list(q[s].values())))))) for s, a in observation])

    log_p += np.log(prior(reward))
```

Figure 4: Function for calculating the ratio between the posterior of the proposal state and the current state

4.4 Generating samples of the posterior

The PolicyWalk algorithm 2 provides a way of generating samples from the posterior, however it only returns one sample. This is why we have gone one step further and implemented a way to generate multiple samples of the posterior within one iteration.

The idea is: as we are running the PolicyWalk at some point we will reach the invariant distribution (in this case the posterior) of the Markov Chain. This means that all the states we will explore from then on are going to be distributed according to the invariant distribution. After this point (which we call burn-in phase) we will start taking samples every N_{out} steps. The idea of taking samples that are not directly after each other is to allow for a certain independence between them. All in all, since most of the computation is dedicated to reaching the invariant distribution, this approach allows us to obtain more samples efficiently.

Nevertheless, we are running multiple iterations, which restarts the search and allows the PolicyWalk to explore different parts of the space.

At the end, as discussed in 3 we take the mean of the posterior samples as it is the more informative estimate.

5 Results

Finally we present the results we have obtained by running the IRL and BIRL algorithms for GridWorld environments of increasing size. With the goal of

recreating the results presented in Figure 4 and 5 of the BIRL paper (see 5), we can confirm that we observe the same trend (exponential growth and BIRL outperforming IRL).

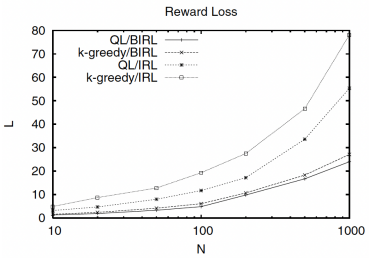

Figure 4: Reward Loss.

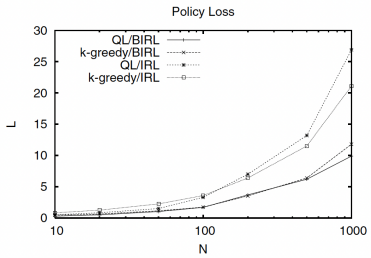

Figure 5: Policy Loss.

Figure 5: Function for calculating the ratio between the posterior of the proposal state and the current state

The scale of the plots is not the same because we have used different environments. In the BIRL paper they just state that they used a "random MDP", which probably implies much higher state connectivity and therefore lower losses overall.

We also have less lines, that is because we have only trained one agent for each algorithm instead of two.

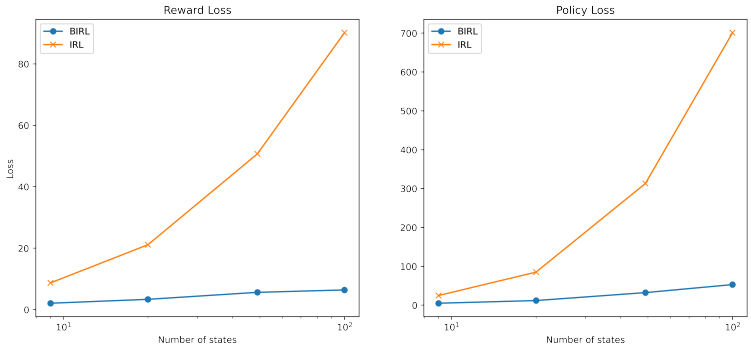

Figure 6: Function for calculating the ratio between the posterior of the proposal state and the current state

All in all, these results are very satisfactory as we have been able to replicate the work of both papers.

6 Open topics and further steps

Some topics are still open and there are questions that could guide further steps in this project.

- In the BIRL paper, the authors use a "k-greedy" RL agent, which is not explained any further in the paper and therefore requires additional research.
- In the BIRL paper, different prior distribution are proposed and it would be interesting to compare their performance. Moreover, it would be very interesting to empirically validate Theorem 2 (refer to 3.4).
- We have spent a lot of time running and waiting for the algorithms. It would be interesting to make a comparison on efficiency. We have seen that BIRL will incur lower losses, but at what cost?

7 References

The main references we followed during this project are:

- Ng, Andrew Y., and Stuart Russell. "Algorithms for Inverse Reinforcement Learning" Icml. Vol. 1. 2000.
- Ramachandran, Deepak, and Eyal Amir. "Bayesian Inverse Reinforcement Learning" IJCAI. Vol. 7. 2007

We also had the following repositories as reference implementations:

- ShivinDass repository on IRL
- cobriant repository on IRL
- uidilr repository on BIRL