Constrained Inverse Reinforcement Learning

Challenges and Solutions for Real World Implementation

Pierre Chassagne

Master thesis

Under the supervision of:
Andreas Schlaginhaufen, Dr. Tony A. Wood, Kai Ren and Prof. Maryam Kamgarpour

14.07.2023

École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

Abstract

This thesis explores the challenges and solutions linked to the implementation of Constrained Inverse Reinforcement Learning (CIRL) for real world application. To this end we study two algorithms, one utilizes stochastic gradient descent ascent (SGDA-CIRL), while the other incorporates IQ-Learn, an advanced imitation learning algorithm. Findings reveal that the Q-CIRL algorithm shows potential and succeeds in recovering a reward for which the expert is optimal but fails to generalize to new transitions dynamics. Meanwhile, the SGDA-CIRL algorithm demonstrates fast convergence and results comparable to CIRL with known dynamics. Additionally, an open-source framework for CIRL is developed, providing a versatile platform for implementing and extending CIRL algorithms and reinforcement learning techniques.

Contents

Acknowledgement

I would like to extend my sincere appreciation and gratitude to the following individuals who have contributed significantly to the completion of my master's thesis:

First and foremost, I express my deepest gratitude to Professor Maryam Kamgarpour for once again entrusting me with this project. Her confidence in my abilities has been a constant source of motivation and inspiration throughout my months at Sycamore Lab.

I am extremely grateful to Andreas Schlaginhaufen, my supervisor, for his unwavering support, invaluable guidance, and remarkable insights. His expertise and knowledge have greatly contributed to shaping my research and enhancing the quality of my work.
I would also like to express my sincere gratitude to Kai Ren, my co-supervisor for its valuable feedback and support throughout the duration of this project. Their input and guidance have been extremely valuable in shaping the development and progress of it.

I would like to extend my heartfelt thanks to Dr. Tony A. Wood for believing in me from the very beginning and for his exceptional supervision. His guidance, wisdom, and knowledge have been instrumental in the successful completion of this thesis and my past works.

I am indebted to my friends and family, who have been a constant source of love and encouragement throughout this demanding journey. Although I may not have always been available during times of intense work, their unwavering support and understanding have meant the world to me.

I would like to express my gratitude to Antoine Schmider, Vianney Mellerio, Thomas Peeters, Patrick Geiger, Sebastian Jeanfavre, Dario Bolli, David Heim, Lucien Pierrejean, Jonathan Scheidegger, and James Germanier, dear friends whose presence has played an integral role in my success at EPFL. Their friendship, encouragement, and intellectual discussions have enriched my academic experience in countless ways.

Special thanks are also due to Patrick Geiger and Antoine Schmider for their invaluable insights and contributions to my work. Their perspectives and feedback have been decisive in shaping the final outcome of this master thesis.

Chapter 1

Introduction

When applying reinforcement learning to safety-critical systems such as autonomous driving and robotics, it becomes necessary to incorporate constraints to prevent the agent from taking actions that could be harmful or violate safety regulations based on the learned policy. One approach to formulating these constraints would be to penalize all state-action pairs that the agent should avoid. Although this approach is feasible, it does not guarantee that the constraints will not be violated. In fact, if violating the constraints leads to a higher expected reward compared to an alternative path, the agent would choose the violating path. While it may seem straightforward to design the reward function to prevent this, it often proves to be a challenging task in real-world scenarios. Constrained reinforcement learning addresses this issue by explicitly incorporating constraints into the optimization problem, ensuring that they are not violated.

The problem of designing a reward function is in fact a well known drawback of the reinforcement learning approach. It becomes particularly difficult for complex goals and if the rewards are inaccurately specified, it can lead to poor performance and potentially unsafe behavior. Inverse reinforcement learning tackles this problem by incorporating the reward function into the learning process itself, rather than treating it as a predefined hyperparameter before training. However, just because the goal is learned does not mean that it is safer. This is why we investigate constrained inverse reinforcement learning (CIRL).

CIRL not only incorporates the reward function into the learning process but also explicitly considers safety constraints to ensure that the learned policy adheres to safety guidelines [Schlaginhaufen and Kamgarpour, 2023]. By imposing constraints on the optimization problem, the agent is trained to make decisions that not only maximize the cumulative reward but also satisfy the specified safety requirements. This approach provides a framework for ensuring the achievement of goals with the assurance of safety, making it particularly valuable in safety-critical domains such as autonomous

driving and robotics.

Contributions The main contributions of this work include the design, implementation, and evaluation of two novel algorithms inspired by the work of Schlaginhaufen and Kamgarpour [2023] and Garg et al. [2021]. The report also presents the development of a Python framework dedicated to CIRL and RL algorithms, providing researchers and practitioners with a user-friendly platform for experimentation and development. Additionally, the existing Sycabot framework has been extended to support CIRL with the integration of jetbots, enabling the application of CIRL algorithms in real-world scenarios. This work further explores the challenges, requirements, and limitations associated with implementing CIRL in discrete cases, offering valuable insights into the practical considerations and potential limitations of these techniques. Overall, it advances the field of CIRL by introducing novel algorithms, providing a dedicated framework, extending existing frameworks, and contributing to the understanding of the complexities involved in deploying CIRL algorithms.

Chapter 2

Methodology

2.1 Background

2.1.1 Constrained Markov Decision Process

A Constrained Markov Decision Process [Altman, 1999] provides a comprehensive mathematical framework to study stochastic decision making problems with safety constraints. A CMDP is defined by a tuple $M = (\mathcal{S}, \mathcal{A}, \boldsymbol{P}, \boldsymbol{\nu}_0, \boldsymbol{r}, \boldsymbol{\Psi}, \boldsymbol{b}, \gamma)$, where $\mathcal{S}$ denotes the state space with $|\mathcal{S}| = n$; $\mathcal{A}$ the action space with $|\mathcal{A}| = m$; $\boldsymbol{P} : \mathcal{S} \times \mathcal{A} \to \Delta_{\mathcal{S}}$ the transition law where $\Delta_{\mathcal{S}}$ denotes the probability simplex over $\mathcal{S}$ and $\boldsymbol{P}(s'|s,a)$ the probability of transitioning from state s to s' following action a; $\boldsymbol{r} \in \mathrm{R}^{nm}$ the reward; $\boldsymbol{\nu}_0$ is the initial state distribution; $\boldsymbol{\Psi} := [\boldsymbol{\Psi}_1, \cdots, \boldsymbol{\Psi}_k] \in \mathbb{R}^{nm \times k}$ the matrix of safety constraint costs and $\boldsymbol{b} \in \mathbb{R}^k$ the corresponding threshold. Starting from some initial state $s_0 \sim \boldsymbol{\nu}_0$, the agent can at each step in time t, choose an action $a_t \in \mathcal{A}$, will arrive in some state $s_{t+1} \sim \boldsymbol{P}(\cdot|s_t, a_t)$, and receives reward $\boldsymbol{r}(s_t, a_t)$ and safety cost $\boldsymbol{\Psi}(s_t, a_t)$. The agent's goal is then to find a policy $\boldsymbol{\pi} : \mathcal{S} \to \Delta_{\mathcal{A}}$ which optimizes the CMDP problem

$$\begin{aligned} &\max_{\boldsymbol{\pi} \in \Pi} \ \boldsymbol{r}^\top \boldsymbol{\mu}^{\boldsymbol{\pi}} + \beta \mathcal{H}(\boldsymbol{\pi}) \\ &\text{s.t.} \quad \boldsymbol{\Psi}^\top \boldsymbol{\mu}^{\boldsymbol{\pi}} \leq \boldsymbol{b}. \end{aligned} \tag{2.1}$$

Here, $\boldsymbol{\mu}^{\boldsymbol{\pi}}$ is the occupancy measure defined as

$$\boldsymbol{\mu}^{\boldsymbol{\pi}}(s,a) := (1-\gamma) \left[\sum_{t=0}^{\infty} \gamma^t \Pr\left(s_t = s, a_t = a\right) \right], \tag{2.2}$$

and $\mathcal{H}(\boldsymbol{\pi}) = \mathbb{E}_{(s,a) \sim \boldsymbol{\mu}^{\boldsymbol{\pi}}} [-\log \boldsymbol{\pi}(a|s)]$ is an entropy regularization with inverse temperature parameter $\beta \geq 0$.[1] For notational convenience, all equations in this chapter will be derived using the occupancy measure. As shown by

[1] The entropy regularization term encourages for exploration by penalizing policies that are too deterministic.

Schlaginhaufen and Kamgarpour [2023] the regularized CMDP problem is for $\beta > 0$ equivalent to an unconstrained MDP problem of the form

$$\max_{\boldsymbol{\pi} \in \Pi} (\boldsymbol{r} - \boldsymbol{\Psi}\boldsymbol{\xi})^\top \boldsymbol{\mu}^{\boldsymbol{\pi}} + \beta \mathcal{H}(\boldsymbol{\pi}), \tag{2.3}$$

where $\boldsymbol{\xi}$ is an optimal dual variable for the safety constraints.

2.1.2 Max Entropy Reinforcement Learning

For a given reward function $\boldsymbol{r} \in \mathcal{R}$, maximum entropy RL [Haarnoja et al., 2017, Bloem and Bambos, 2014] aims to solve a given CMDP problem by learning a policy that maximizes the expected cumulative discounted reward along with the entropy in each state

$$\max_{\boldsymbol{\pi} \in \Pi} \boldsymbol{r}^\top \boldsymbol{\mu}^{\boldsymbol{\pi}} + \beta \mathcal{H}(\boldsymbol{\pi}). \tag{2.4}$$

Bloem and Bambos [2014] show that the optimal policy for (2.4) is given by

$$\boldsymbol{\pi}^*(a|s) = \frac{1}{Z_s} \exp(Q^*(s,a)/\beta), \tag{2.5}$$

where Z_s is a normalization factor given by $Z_s = \sum_{a'} \exp(Q^*(s,a')/\beta)$ and $Q^*(s,a)$ is the optimal soft Q-function [Haarnoja et al., 2017] defined as

$$Q^*(s,a) = r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V^*(s') \right], \tag{2.6}$$

where the optimal value function is given by

$$V^*(s) = \beta \log \sum_{a'} \exp\left(\frac{1}{\beta} Q^*(s,a') \right). \tag{2.7}$$

2.2 Problem formulation

Given a CMDP without reward $M \setminus \boldsymbol{r} = (\mathcal{S}, \mathcal{A}, \boldsymbol{P}, \boldsymbol{\nu}_0, \boldsymbol{\Psi}, \boldsymbol{b}, \gamma)$ and a data set of demonstrations $\mathcal{D} = \{(s_t^i, a_t^i)_{t=0}^T\}_{i=1}^N$ from some expert $\boldsymbol{\mu}^E$, CIRL aims to recover a reward function for which the expert's policy is optimal for the CMDP problem (2.1). As shown by Schlaginhaufen and Kamgarpour [2023], the max entropy CIRL problem can be cast as the following min-max optimization problem,

$$\begin{aligned} &\min_{\boldsymbol{\xi} \geq 0} \max_{\boldsymbol{r} \in \mathcal{R}} \max_{\boldsymbol{\pi} \in \Pi} && \boldsymbol{r}^\top (\boldsymbol{\mu}^{\boldsymbol{\pi}} - \hat{\boldsymbol{\mu}}_{\mathcal{D}}^E) + \beta \mathcal{H}(\boldsymbol{\mu}^{\boldsymbol{\pi}}) + \boldsymbol{\xi}^\top (\boldsymbol{b} - \boldsymbol{\Psi}^\top \boldsymbol{\mu}^{\boldsymbol{\pi}}) \\ &= \min_{\boldsymbol{\xi} \geq 0} \max_{\boldsymbol{r} \in \mathcal{R}} \max_{\boldsymbol{\pi} \in \Pi} && \boldsymbol{w}^\top \boldsymbol{\Phi}^\top (\boldsymbol{\mu}^{\boldsymbol{\pi}} - \hat{\boldsymbol{\mu}}_{\mathcal{D}}^E) + \beta \mathcal{H}(\boldsymbol{\mu}^{\boldsymbol{\pi}}) + \boldsymbol{\xi}^\top (\boldsymbol{b} - \boldsymbol{\Psi}^\top \boldsymbol{\mu}^{\boldsymbol{\pi}}) \\ &= \min_{\boldsymbol{\xi} \geq 0} \max_{\boldsymbol{r} \in \mathcal{R}} \max_{\boldsymbol{\pi} \in \Pi} && L_{\mathcal{D}}(\boldsymbol{\pi}, \boldsymbol{w}, \boldsymbol{\xi})), \end{aligned} \tag{2.8}$$

where the reward class is $\boldsymbol{r} \in \mathcal{R} := \{\boldsymbol{r}_{\boldsymbol{w}} = \boldsymbol{\Phi}\boldsymbol{w} : \boldsymbol{\Phi} \in \mathbb{R}^{mn \times d}, \|\boldsymbol{w}\| \leq c\}$ and the empirical expert occupancy measure is given by

$$\hat{\boldsymbol{\mu}}_{\mathcal{D}}^E(s,a) = \frac{1-\gamma}{N} \sum_{i=1}^N \sum_{t=0}^T \gamma^t \mathbb{1}(s_t^i = s, a_t^i = a). \tag{2.9}$$

2.3 GDA-CIRL

To solve the min-max problem (2.8), recent work by Schlaginhaufen and Kamgarpour [2023] proposes to use a gradient descent-ascent algorithm, where the policy and rewards are updated within a single optimization loop. The policy is updated using Natural Policy Gradient (NPG)[2] with softmax parametrization of the policy [Cen et al., 2020], while the dual variables $\boldsymbol{w}$ and $\boldsymbol{\xi}$ are updated using projected sub-gradient descent. The pseudo-code of the algorithm is presented in Algorithm 1. This algorithm has been shown to provably converge by the work of Renard [2023]. In the

Algorithm 1
Gradient descent-ascent for constrained entropy-regularized IRL

Input: Expert data $\mathcal{D}$, learning rate η, *max_iter*
Output: Learned reward $\boldsymbol{r}$, learned policy $\boldsymbol{\pi}$
Initialize: $\boldsymbol{\pi} \in \Pi$, $\boldsymbol{\xi} = 0$, $\boldsymbol{w} = 0$

while not *max_iter* **do**

Do primal update with:

$$\begin{aligned} \boldsymbol{r} &\leftarrow \boldsymbol{\Phi}\boldsymbol{w}^{(t)} - \boldsymbol{\Psi}\boldsymbol{\xi}^{(t)} \\ Q^{\boldsymbol{\pi}^{(t)}} &\leftarrow \text{Soft-Q evaluation}(\boldsymbol{r}, M \setminus \boldsymbol{r}) \\ \boldsymbol{\pi}^{(t+1)} &\leftarrow NPG(Q^{\boldsymbol{\pi}^{(t)}}, \eta_{\boldsymbol{\pi}}) \end{aligned}$$

Do dual update with:

$$\begin{aligned} \boldsymbol{w}^{(t+1)} &\leftarrow P_{B_C}(\boldsymbol{w}^{(t)} - \eta_{\boldsymbol{w}} \nabla_{\boldsymbol{w}} L_{\mathcal{D}}(\boldsymbol{\pi}^{(t+1)}, \boldsymbol{w}^{(t)}, \boldsymbol{\xi}^{(t)})) \\ \boldsymbol{\xi}^{(t+1)} &\leftarrow P_{[0,\infty)}(\boldsymbol{\xi}^{(t)} - \eta_{\boldsymbol{\xi}} \nabla_{\boldsymbol{\xi}} L_{\mathcal{D}}(\boldsymbol{\pi}^{(t+1)}, \boldsymbol{w}^{(t)}, \boldsymbol{\xi}^{(t)})) \end{aligned}$$

end while

Algorithm 1, as shown by Cen et al. [2021], the NPG update is given by the following update rule in the policy space,

$$\boldsymbol{\pi}^{(t+1)}(a|s) \propto \left(\boldsymbol{\pi}^{(t)}(a|s)\right)^{1-\frac{\beta\eta_{\boldsymbol{\pi}}}{1-\gamma}} \exp\left(\frac{\eta_{\boldsymbol{\pi}} Q^{\boldsymbol{\pi}^{(t)}}(s,a)}{1-\gamma}\right), \tag{2.10}$$

and the gradients for $\boldsymbol{w}$ and $\boldsymbol{\xi}$ are given by

$$\begin{aligned} \nabla_{\boldsymbol{w}} L_{\mathcal{D}}(\boldsymbol{\pi}^{(t)}, \boldsymbol{w}, \boldsymbol{\xi}) &= \boldsymbol{\Phi}^\top(\boldsymbol{\mu}^{\boldsymbol{\pi}^{(t)}} - \hat{\boldsymbol{\mu}}_{\mathcal{D}}^E) \\ \nabla_{\boldsymbol{\xi}} L_{\mathcal{D}}(\boldsymbol{\pi}^{(t)}, \boldsymbol{w}, \boldsymbol{\xi}) &= (\boldsymbol{b} - \boldsymbol{\Psi}^\top \boldsymbol{\mu}^{\boldsymbol{\pi}^{(t)}}), \end{aligned} \tag{2.11}$$

where P_{B_c} denotes the projection onto the ball $B_c := \{\boldsymbol{w} : \|\boldsymbol{w}\| \leq c\}$ and $P_{[0,\infty)}$ the trivial projection onto the non-negative orthant.

[2] Haarnoja et al. [2018] show that for a step $\eta_{\boldsymbol{\pi}} = (1-\gamma)/\beta$ NPG is completely equivalent to soft policy iteration.

Moreover, to perform soft-Q evaluation for a given policy π, we recursively compute,

$$Q^{(t)}(s,a) = r(s,a) + \sum_{s'} P(s'|s,a)V^{(t-1)}(s'), \tag{2.12}$$

where

$$V^{(t-1)}(s') = \sum_{a'} Q^{(t-1)}(s',a')\pi(s'|a'), \tag{2.13}$$

and then stop when the error $||Q^{(t)} - Q^{(t-1)}||_1 \leq 10^{-9}$ [Sutton and Barto, 2018].

While the implementation of this algorithm is relatively straightforward, the soft-Q evaluation step assumes that we can exactly evaluate the value of a policy, which requires complete knowledge and a clear understanding of the system's dynamics. The dual update in Eq. (2.11) also requires the dynamics to evaluate the occupancy measure $\boldsymbol{\mu}^{\pi^{(t)}}$. This mean that we require to know the probability distributions that govern the movement of our robot. However, in reality, this is often not the case, and the best we can do is approximate a model of our robot's dynamics. Furthermore, when dealing with large or even continuous state and action spaces, exact value evaluation methods become prohibitively expensive. Instead, Sutton and Barto [2018] offer to learn from experiences to overcome this limitation. Learning from actual experience doesn't require any prior knowledge about the environment's dynamics, yet it can still achieve optimal behavior with the added benefits of not depending on a model anymore.

2.4 SGDA-CIRL

In the previous section, we discussed the need to develop an algorithm that can learn without having precise knowledge of the environment's dynamics. Instead, our algorithm should learn from a replay buffer $\mathcal{B}_{\pi} = \{(s_t^i, a_t^i, {s'}_t^i)_{t=0}^T\}_{i=1}^N$ which contains N rollouts obtained by following the policy π until time T. To this end, we estimate the Q-function in the primal update via temporal difference learning and the gradients in the dual update via Monte Carlo sampling. Moreover, in practice we use a done signal `d` which is 1 when the agent reaches a terminal state. This approach optimizes iteration time by permitting the truncation of episodes before reaching time T, while still being aware of the episode's end time T_d in order to later recover the discounted reward[3].

[3]Refer to chapter 3 for further details on this.

2.4.1 Primal update

For the primal NPG update, we need an estimate of the soft-Q function $Q^{\boldsymbol{\pi}}$. For this purpose, we use soft Q-learning [Haarnoja et al., 2017] to approximately evaluate the Q-function of the policy $\boldsymbol{\pi}$. In this algorithm, we aim to minimize the cost

$$J_Q = \mathbb{E}_{(s,a)\sim\boldsymbol{\mu}^{\pi}}\left[\frac{1}{2}\left(Q(s,a) - \mathbb{E}_{s'\sim\boldsymbol{P}(\cdot|s,a)}\left[y(s,a,s')\right]\right)^2\right], \tag{2.14}$$

where $y(s,a,s')$ is the target Q-value given by

$$y(s,a,s') = r(s,a) + \gamma V^{\boldsymbol{\pi}}(s'), \tag{2.15}$$

and

$$V^{\boldsymbol{\pi}}(s) = \sum_a \boldsymbol{\pi}(a|s) Q^{\boldsymbol{\pi}}(s,a). \tag{2.16}$$

In practice, problem (2.14) is approximately solved using stochastic gradient descent. Therefore, the Monte Carlo estimate of the gradient of $J_Q(s_i,a_i)$ at iteration j becomes

$$\hat{\nabla}_Q J_Q(s,a) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\left(Q^{\boldsymbol{\pi}^{(j-1)}}(s_t^i,a_t^i) - y^{\boldsymbol{\pi}^{(j-1)}}(s_t^i,a_t^i,s'^i_t)\right)\mathbb{1}(s=s_t^i, a=a_t^i). \tag{2.17}$$

2.4.2 Dual update

A straightforward approach to address this is by replacing explicit gradient descents on the dual variables with stochastic gradient descent (SGD) algorithms. In this approach, the expectations are estimated by the sample averages

$$\hat{\nabla}_{\boldsymbol{w}} L_{\mathcal{D}}(\boldsymbol{\pi},\boldsymbol{w},\boldsymbol{\xi}) = \frac{1-\gamma}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\gamma^t\Phi(s_t^i,a_t^i) - \Phi^{\top}\hat{\boldsymbol{\mu}}_{\mathcal{D}}^{E}, \tag{2.18}$$

$$\hat{\nabla}_{\boldsymbol{\xi}} L_{\mathcal{D}}(\boldsymbol{\pi},\boldsymbol{w},\boldsymbol{\xi}) = \boldsymbol{b} - \frac{1-\gamma}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\gamma^t\psi(s_t^i,a_t^i).$$

Then, the Q-function associated with the reward can be learned by Q-learning and used to update the policy with Natural Policy Gradient (NPG).

2.4.3 Algorithm

This gives us the following practical algorithm:

Algorithm 2 CIRL with stochastic GDA

Input: Expert state occupancy measure σ_E, reward feature matrix $\boldsymbol{\Phi}$, constraint cost $\boldsymbol{\Psi}$, learning rates η, *max_iter*.
Output: Learned reward $\boldsymbol{r}$, learned policy $\boldsymbol{\pi}$.
Initialize: $\boldsymbol{\pi} \in \Pi$, $\boldsymbol{\xi} = 0$, $\boldsymbol{w} = 0$, $Q = 0$, $\boldsymbol{\pi} = \mathcal{U}_{\mathcal{S}\times\mathcal{A}}$.

for step j in $\{1, \cdots, N\}$ **do**
Reset environment to a random initial state $s_0 \sim \nu_0$.
for trajectory in $\{1, \cdots, T\}$ **do**
while not done **do**
Observe state s *and take random action* $a \sim \boldsymbol{\pi}(\cdot|s)$
Observe next state s'*, and done signal* d.
Store (s, a, s', d) *in replay buffer* $\mathcal{B}_{\boldsymbol{\pi}}$.
end while
end for
Calculate reward as: $\boldsymbol{r} \leftarrow \boldsymbol{\Phi}\boldsymbol{w}^{(j)} - \boldsymbol{\Psi}\boldsymbol{\xi}^{(j)}$
Update Q *by one step of Q-learning using (2.17):*

$$Q^{\boldsymbol{\pi}^{(j)}}(s,a) = Q^{\boldsymbol{\pi}^{(j-1)}}(s,a) - \eta_Q \hat{\nabla}_Q J_Q(s,a) \tag{2.19}$$

Update $\boldsymbol{\pi}$ *by one step of NPG:*

$$\boldsymbol{\pi}^{(j+1)} \leftarrow NPG(Q^{\boldsymbol{\pi}^{(j)}}, \eta_{\boldsymbol{\pi}}) \tag{2.20}$$

Update $\boldsymbol{\xi}$ *and* $\boldsymbol{w}$ *by one step of gradient descent using (2.18).*

$$\begin{aligned} \boldsymbol{w}^{(j+1)} &\leftarrow P_{B_C}(\boldsymbol{w}^{(j)} - \eta_{\boldsymbol{w}} \hat{\nabla}_{\boldsymbol{w}} L_{\mathcal{D}}(\boldsymbol{\pi}^{(j+1)}, \boldsymbol{w}^{(j)}, \boldsymbol{\xi}^{(j)})) \\ \boldsymbol{\xi}^{(j+1)} &\leftarrow P_{[0,\infty)}(\boldsymbol{\xi}^{(j)} - \eta_{\boldsymbol{\xi}} \hat{\nabla}_{\boldsymbol{\xi}} L_{\mathcal{D}}(\boldsymbol{\pi}^{(j+1)}, \boldsymbol{w}^{(j)}, \boldsymbol{\xi}^{(j)})), \end{aligned} \tag{2.21}$$

end for

2.5 Q-CIRL

The key idea of IQ-Learn [Garg et al., 2021], a recent state of the art imitation learning algorithm, is to exploit the bijection between rewards and soft-Q functions, proved by Garg et al. [2021], to reformulate the IRL objective in terms of Q functions. Similarly, we can reformulate the original CIRL problem (2.8) in terms of Q as follows

$$\min_{\boldsymbol{\xi}\geq 0} \max_{\boldsymbol{r}\in\mathcal{R}} \max_{\boldsymbol{\pi}\in\Pi} \quad \mathbb{E}_{(s,a)\sim\boldsymbol{\mu}^{\pi}}[(\mathcal{T}^{\boldsymbol{\pi}}Q)(s,a)] - \mathbb{E}_{(s,a)\sim\hat{\boldsymbol{\mu}}_{\mathcal{D}}^{E}}[(\mathcal{T}^{\boldsymbol{\pi}}Q_r)(s,a)] - \mathcal{H}(\boldsymbol{\mu}^{\boldsymbol{\pi}}) + \boldsymbol{\xi}^{\top}\boldsymbol{b} \tag{2.22}$$

where $(\mathcal{T}^{\pi} q)(s,a) = q(s,a) - \gamma \mathbb{E}_{\substack{s' \sim \boldsymbol{P}(.|s,a) \\ a' \sim \boldsymbol{\pi}(.|s')}} \left[q(s',a') - \beta \log \boldsymbol{\pi}(a'|s') \right]$ and

$$\begin{aligned} Q(s,a) &= Q_{\boldsymbol{r}}(s,a) - \boldsymbol{\xi}^{\top} Q_{\boldsymbol{\Psi}}(s,a), \\ Q_{\boldsymbol{r}}(s,a) &= \boldsymbol{r}(s,a) + \gamma \mathbb{E}_{s' \sim \boldsymbol{P}(.|s,a)} \left[V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s') \right], \\ Q_{\boldsymbol{\Psi}}(s,a) &= \boldsymbol{\Psi}(s,a) + \gamma \mathbb{E}_{s' \sim \boldsymbol{P}(.|s,a)} \left[\sum_{a'} Q_{\boldsymbol{\Psi}}(s',a') \boldsymbol{\pi}(a'|s') \right]. \end{aligned} \tag{2.23}$$

Note that only the $Q_{\boldsymbol{r}}$ function is regularized and that $\boldsymbol{r}$ is the same as in (2.3). For a fixed policy, the reformulated problem is entirely equivalent to the original problem, this can be demonstrated through the bijection property established by Garg et al. [2021]. This formulation allows us to use the IQ-learn algorithm developed by Garg et al. [2021] while performing Q-learning steps to learn the Q function associated with the constraints. After training the optimal policy can be recovered using (2.5) with the overall soft-Q function

$$Q(s,a) = Q_{\boldsymbol{r}}(s,a) - \boldsymbol{\xi}^{\top} Q_{\boldsymbol{\Psi}}(s,a). \tag{2.24}$$

2.5.1 IQ-learn

Formally, the inverse problem is solved by maximizing [Garg et al., 2021]:

$$\max_{Q_{\boldsymbol{r}} \in \Omega} L^*(Q_{\boldsymbol{r}}) = \mathbb{E}_{\boldsymbol{\mu}_E} \left[\phi \left(Q_{\boldsymbol{r}}(s,a) - \gamma \mathbb{E}_{s \sim P(\cdot|s,a)} V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s) \right) \right] - (1-\gamma) \mathbb{E}_{\boldsymbol{\mu}_0} \left[V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s_0) \right] \tag{2.25}$$

where $V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s)$ is defined as,

$$V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s) = \sum_{a'} Q_{\boldsymbol{r}}(s,a') \boldsymbol{\pi}(a'|s) - \beta \log \boldsymbol{\pi}(a'|s). \tag{2.26}$$

The objective forms a variant of soft-Q learning, where we try to learn the optimal Q-function given an expert distribution and where the optimal policy is given by equation (2.5). In practice, for discrete cases, Garg et al. [2021] found that using a Chi2 divergence $\phi(x) = x + \frac{1}{4\alpha} x^2$ gave the best results and that instead of directly estimating $\mathbb{E}_{\boldsymbol{\mu}_0} \left[V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s_0) \right]$ getting a sample estimate $\mathbb{E}_{(s,a,s') \sim \boldsymbol{\mu}_E} \left[V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s) - \gamma V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s') \right]$ from the expert replay buffer improves stability for convergence. The objective can then be re-written as:

$$\max_{Q_{\boldsymbol{r}} \in \Omega} \mathbb{E}_{(s,a,s') \sim \boldsymbol{\mu}_E} \left[Q_{\boldsymbol{r}}(s,a) - V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s) + \frac{1}{4\alpha} \left(Q_{\boldsymbol{r}}(s,a) - \gamma V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s') \right)^2 \right], \tag{2.27}$$

which can be optimised with SGD using Monte Carlo sampling,

$$\begin{aligned} \hat{\nabla}_{Q_r} L_{Q_r}(s,a) = & \frac{1-\gamma}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \gamma^t \Bigg(1 - V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s_t^i) \\ & + \frac{1}{2\alpha} \left(Q_{\boldsymbol{r}}(s_t^i, a_t^i) - \gamma V_{\boldsymbol{r}}^{\boldsymbol{\pi}}(s'^i_t) \right) \Bigg) \mathbb{1}(s = s_t^i, a = a_t^i). \end{aligned} \tag{2.28}$$

Note that for this gradient derivation $V_{\boldsymbol{r}}^*$ are considered as targets and therefore kept constant with respect to $Q_{\boldsymbol{r}}$ as for discrete cases it speeds up the training while it has no visible impact on the results [Garg et al., 2021].

2.5.2 Constraint update

The constrained problem is solved exactly as in the previous algorithm and the Q-function associated to the constraint cost $\boldsymbol{\Psi}$ is approximated using classical Q-learning where the objective to minimize is given by [Sutton and Barto, 2018],

$$L(Q_{\boldsymbol{\Psi}}) = \mathbb{E}_{(s,a,s')\sim\boldsymbol{\mu}_E}\left[\frac{1}{2}\left(Q_{\boldsymbol{\Psi}}(s,a) - \boldsymbol{\Psi}(s,a) - \gamma\sum_{a'} Q_{\boldsymbol{\Psi}}(s',a')\boldsymbol{\pi}(a'|s')\right)^2\right] \tag{2.29}$$

which can also be optimised with SGD using Monte Carlo sampling,

$$\begin{aligned}\hat{\nabla}_{Q_{\boldsymbol{\Psi}}} L_{Q_{\boldsymbol{\Psi}}}(s,a) = \frac{1-\gamma}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\gamma^t\Big(Q_{\boldsymbol{\Psi}}(s_t^t,a_t^t) - \boldsymbol{\Psi}(s_t^t,a_t^t) \\ - \gamma\sum_{a'} Q_{\boldsymbol{\Psi}}(s',a')\boldsymbol{\pi}(a'|s')\Big)\mathbb{1}(s=s_t^i, a=a_t^i).\end{aligned} \tag{2.30}$$

2.5.3 Algorithm

Finally, we can put all of this together and we obtain Algorithm 3,

Algorithm 3 CIRL with SGDA and IQ-learn

Input: Expert data $\mathcal{D}_E$, learning rate η, *max_iter*
Output: Learned reward $\boldsymbol{r}$, learned policy $\boldsymbol{\pi}$
Initialize: $\boldsymbol{\pi} \in \Pi$, $\boldsymbol{\xi} = 0$, $Q_r = 0$, $Q_\psi = 0$

for step in $\{1, \cdots, N\}$ **do**
  Sample a batch of transition from $\mathcal{D}_E$.
  Reset environment to a random initial state $s_0 \sim \boldsymbol{\mu}_0$.
  for trajectory in $\{1, \cdots, T\}$ **do**
    while not done **do**
      Observe state s *and take random action* $a \sim \pi$
      Observe next state s' *and done signal* d.
      Store (s, a, s', d) *in replay buffer* $\mathcal{B}_{\boldsymbol{\pi}}$.
    end while
  end for
  Update $Q_{\boldsymbol{\Psi}}$ *by one step of Q-learning using* (2.30).

$$Q_{\boldsymbol{\Psi}}(s, a) = Q_{\boldsymbol{\Psi}}(s, a) - \eta_{Q_{\boldsymbol{\Psi}}} \hat{\nabla}_{Q_{\boldsymbol{\Psi}}} L_{Q_{\boldsymbol{\Psi}}}(s, a) \tag{2.31}$$

  Update dual variable ξ *by one step SGD as in* (2.18).

$$\boldsymbol{\xi} \leftarrow P_{[0,\infty)}(\boldsymbol{\xi} - \eta_{\boldsymbol{\xi}} \hat{\nabla}_{\boldsymbol{\xi}} L_{\mathcal{D}}(\boldsymbol{\pi}, \boldsymbol{w}, \boldsymbol{\xi})) \tag{2.32}$$

  Update $Q_{\boldsymbol{r}}$ *by one step of IQ-learn using* (2.28)*:*

$$Q_{\boldsymbol{r}}(s, a) = Q_{\boldsymbol{r}}(s, a) - \eta_{Q_{\boldsymbol{r}}} \hat{\nabla}_{Q_{\boldsymbol{r}}} L_{Q_{\boldsymbol{r}}}(s, a) \tag{2.33}$$

end for
Recover $\boldsymbol{r}$ *using* (2.23) *and* $\boldsymbol{\pi}$ *with* (2.5) *and* (2.24).

Chapter 3

Implementation

The implementation part of this master thesis holds a pivotal role in realizing the practical application of CIRL in real-world scenarios. By implementing the theoretical concepts discussed in earlier sections into software, this phase not only demonstrates the feasibility of CIRL but also provides a tool for future research and practical implementations.

The primary objective of this implementation is to develop a comprehensive Python package that enables users to effortlessly develop and train reinforcement learning algorithms and more specifically CIRL algorithms with a strong emphasis on providing a well-documented and intuitive codebase. Moreover, the package facilitates algorithmic implementation in discrete state-action spaces and was developed to be easily extendable to continuous settings in the future. Additionally, this implementation serves as a showcase of the algorithms presented in the previous section to assess their, validity strengths, weaknesses and effectiveness.

3.1 Hardware

In order to showcase the results of CIRL training in a real-world setting, we employed Jetbots, which are robust wheeled robots powered by Nvidia Jetson Nano. To facilitate tracking and localization, we utilized an Optitrack system along with Motive software, simulating a global localization system. The lab environment, as shown in Figure 3.1, consists of a maze with known obstacles positions. Additionally, a video projector is used to display more information on the floor, such as policy, reward or states.

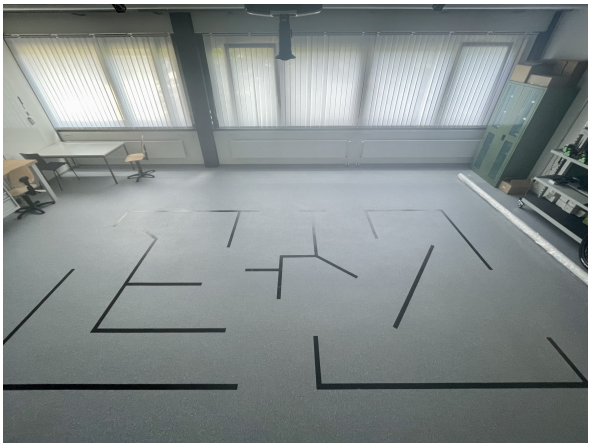

Figure 3.1: *Picture of the lab. The maze is defined by the black bands.*

3.2 Methodology for software development

The implementation relied on several key libraries and frameworks, each serving a specific purpose in the development process with a strong emphasis on minimizing unnecessary dependencies and maintaining a streamlined architecture. The principal libraries and frameworks are:

1. **NumPy**: The NumPy library [Harris et al., 2020] played a role in performing efficient array and vector operations. It provided essential mathematical functionalities required for data manipulation and calculations. This library allowed for optimized numerical computations, enhancing the overall performance of the software.

2. **SciPy**: The SciPy library [Virtanen et al., 2020] was employed for specialized mathematical operations. It offered a wide range of mathematical functions, including advanced operations like softmax and logsumexp. These functions were instrumental in implementing complex algorithms and computations required for reinforcement learning and optimization tasks.

3. **JSON**: The JSON library and format was utilized for writing configuration files. This allowed users to easily customize and adjust the behavior of the software package. JSON provided a lightweight and human-readable format for storing configuration data, ensuring flexibility and ease of use.

4. **PyTorch**: The PyTorch [Paszke et al., 2019] library offers extensive support for automatic differentiation, enabling efficient gradient computation for optimization algorithms. For continuous states actions spaces, it can also be used for the implementation of deep neural net-

works as it provides a powerful platform for building and training neural network models.

5. **Gymnasium**: Gymnasium [Towers et al.] served as the base class for defining the environment and conducting deep reinforcement learning trainings within a simulated environment. It provided a standardized interface for interacting with the environment, enabling the software to work seamlessly with the various reinforcement learning algorithms. Gymnasium facilitated rapid prototyping, development of environments and experimentation with different agents.

6. **WandB**: The WandB [Biewald, 2020] library, short for "Weights & Biases," was used for experiment tracking and visualization. It provided a suite of tools for logging experiment metrics and visualizing model or algorithm performances. Wandb allowed for efficient experiment management, enabling the tracking and comparison of different training runs, and aiding in the analysis of results.

7. **ROS2**: ROS2 [Macenski et al., 2022] provides a robust and flexible infrastructure for managing the exchange of messages, controlling robot behavior, and interacting with sensor data. It offered a standardized and scalable approach for developing and deploying the framework on the Jetbots.

In terms of design principles and architecture patterns, the implementation followed the principles of object-oriented programming (OOP). Emphasis was placed on key OOP principles, including abstraction, encapsulation, inheritance, and selective use of polymorphism for some functions. These principles were leveraged to enhance maintainability, modularity, and performance of the codebase. Additionally, the following design rules were followed to ensure a well-structured and robust architecture:

1. Each package was designed as an independent unit with a specific purpose, allowing for potential replacement or rewriting of a package without impacting the rest of the system.

2. The implementation aimed to minimize dependencies, utilizing only the necessary libraries and frameworks required for the project.

3. Circular dependencies between packages are avoided as much as possible. To ensure a clean and efficient structure where each package only relies on the necessary dependencies.

Furthermore, an important aspect of the methodology was the development of the Python package to be independent of ROS2. This decoupling was undertaken to enhance the reusability and accessibility of the software, enabling any user, regardless of their access to a proper ROS2 installation, to utilize the package effectively. A separate Python layer was built on top of

the package to integrate it with ROS2, enabling seamless integration within the Sycabot framework while preserving the package's independence from it. This approach directly aligns with the objectives stated in the introduction, where the aim was to create a user-friendly Python package that allows easy development and training of constrained inverse reinforcement learning algorithms. Furthermore, it provides notable benefits for code testing purposes. By separating the package from ROS2, it becomes simpler to isolate and test specific functionalities and components of the CIRL implementation in a controlled environment. Meanwhile, the ROS2 components can be tested separately in their own environment. This approach ensures the dependability and precision of the package's implementation.

By following these methodology and design approaches, the implementation achieved a well-structured and modular software package. The chosen libraries and frameworks supported the necessary functionalities, while adhering to OOP principles and design rules ensured a maintainable, extensible, and efficient codebase. The integration with ROS2 further expanded the software's capabilities, allowing for seamless communication with robots and real world demonstrations, while maintaining the package's reusability and independence.

3.3 System Overview

This section provides a detailed exploration of the various components comprising our system and their interconnectedness. Similar to any Python package, our system is structured into main modules, each containing submodules that implement different classes or functions. Each module is dedicated to a specific task, and its submodules or subclasses handle the associated subtasks. For instance, the Agents module encompasses the implementation of all available agents, with each agent being a submodule responsible for its own learning methods. Figure 3.2 offers a simplified overview of each system component and their dependencies, where the ROS2 overlay was omitted as it is more independent of the entire system.

3.3.1 Config module

The Config module is responsible for loading and storing the configuration and hyperparameters of the reinforcement learning problem. It handles the creation of a Config object that stores all the necessary settings and is automatically generated from a JSON file. The Config module ensures that the entire system is configured correctly and consistently while providing a centralized location to manage and modify the hyperparameters and settings of the problem, making it easier to experiment with different configurations. By separating the configuration management from the rest

of the framework, the Config module promotes modularity and reusability. It enables the user to easily customize training's hyperparameters or modify other settings such as noise or obstacles.

3.3.2 Logger module

The Logger module plays a crucial role in capturing and recording valuable information during the training process of agents. It offers a range of logger classes, each serving as an interface for different types of discrete agents. These loggers enable the collection and storage of important metrics, statistics, and other relevant data points that provide insights into the agent's performance and learning progress. By integrating the appropriate logger into the agent training pipeline, developers can easily monitor and analyze crucial training metrics such as rewards, episode lengths, and exploration rates. The Logger module serves as a valuable tool for assessing agent performance, diagnosing issues, and facilitating iterative improvements to the reinforcement learning process.

3.3.3 Expert

It is responsible for loading pre-trained optimal agents along with their training configurations. These Experts are utilized to generate trajectories using the learned policy. During the training phase, the Expert module provides access to its buffer, which contains high-quality trajectories obtained from the learned policy or expert demonstrations. This module offers flexibility and efficiency in integrating prior knowledge and expert demonstrations into the training pipeline. The module also contains all the methods to generate rollouts from an agent policy by interacting with the environment.

3.3.4 Environments module

The Environment module plays a role in the creation and management of environments where agents interact and learn. It accomplishes this by providing a base class called DiscreteEnv, which inherits from the gymnasium.Env class to ensure compatibility with the Gymnasium library. This class establishes essential methods for interacting with environments and can be easily specialized. Within the Environment module, the gridworld class is also present, it specializes the DiscreteEnv class specifically for creating Constrained Discrete grid-based environments. This gridworld class serves as the fundamental environment for all training activities in this project. In summary, the Environment module forms a solid foundation for creating and managing environments within the framework. It guarantees compatibility with Gymnasium, facilitates the creation of custom environments, and relies on the Config module for efficient configuration management.

3.3.5 Methods module

The Method module encompasses a collection of static functions to implement various algorithms of reinforcement learning (RL), including inverse RL, constrained RL, and entropy-regularized RL. By design, the Method module is independent and self-contained, allowing the functions to be easily utilized across different projects and codebases simply by importing the module. This modularity and reusability enable seamless integration of new functions and the utilization of existing ones, empowering researchers and developers to extend the capabilities of the framework effortlessly. The Method module serves as a versatile toolbox, providing a wide range of algorithms and techniques to address diverse RL scenarios while promoting code efficiency.

3.3.6 Agents module

The Agents module provides essential tools for agent creation and training. It offers developers the flexibility to design and implement discrete agents by extending the DiscreteAgent base class and utilizing the method modules. The DiscreteAgent class establishes the core methods and functionalities that are expected from any discrete agent. Within the Agents module, there are agent classes corresponding to the algorithms discussed in the previous section. Each agent relies on its associated method modules and the logger to implement specific learning algorithms and record crucial training information. Additionally, every agent is aware of its environment and interacts with it. This interaction allows agents to acquire relevant details during training.

3.3.7 Training module

The training module has two main functionalities: saving the agent's data and training configuration. It provides the capability to easily and efficiently load a pre-trained agent and its training environment and configuration for further training or to demonstrate its learning in real-world scenarios.

3.3.8 ROS2 overlay

The ROS2 Overlay module plays a role in interfacing the CIRL framework and the Sycabot framework, enabling seamless communication between the two and facilitating the interaction between them. Through the ROS2 Overlay module, users gain the ability to communicate with the robots via the Sycabot framework, allowing for real-time trajectory harvesting during training or showcasing the results of a training session. This online training capability empowers users to continuously adapt and refine their models

based on real-world robot demonstrations. Additionally, the module offers a convenient means of displaying essential information using a video projector, enhancing the visual feedback and facilitating demonstration and comprehensive understanding of the training process.

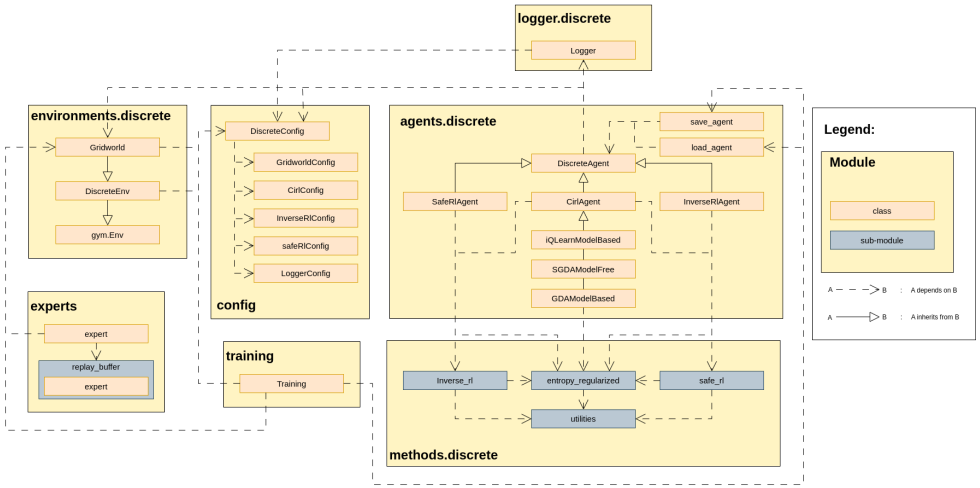

Figure 3.2: *Simplified view of the system and its dependencies.*

3.4 Implementation details

3.4.1 Training of an agent

The functions responsible for training an agent are all located in the root directory of the Python package and all start with the prefix "**train**_*.py". Their purpose is to orchestrate the various steps involved in the training process. A simplified version of Python code for such function is given below. Its workflow can be broken down as follow:

1. **[Line 1] Loading Settings**: The Config module is used to load the settings from the specified JSON file in the settings folder. These settings contain various configuration parameters required for the training process, such as environment specifications, batch size, maximum number of iterations, etc...

2. **[2] Environment creation**: The environment is immediately created using gymnasium **make** command after that.

3. **[4-6] Sanity Check**: A sanity check is performed using linear programming (LP) to solve the constrained problem and verify the feasibility of the problem. This check ensures that the specified environment and settings are valid and can be solved. If the sanity check fails, a warning is raised and the training is stopped.

4. **[8-12] Expert Creation**: An expert is created using the provided environment and expert data. The expert is responsible for generating a buffer of trajectories and eventually sampling transitions from it during the training process.[1]

5. **[14-20] Training**: The logger and agent objects are initialized and the training loop starts, it iterates until convergence or the maximum number of iterations is reached. Depending on the agent which is being trained, a buffer of experiences is generated using the agent policy and passed to it.

6. **[22,23] Saving**: If logging is enabled, the relevant information from the training (e.g. learned reward) are saved in the **training** module.

7. **[25] Visualization**

[1] It is important to "humanly" check that the expert was trained in the same environment as the one that will be used for the training. Nothing was made to check this as it was hard to determine which feature to use.

3.4.2 Model Based GDA

The implementation of the model-based GDA approach as seen in section 2.3 is relatively straightforward and does not require extensive discussion. The use of the gradient descent algorithm for updating the dual variables is a well-established and widely used optimization technique, and no specific methods were employed to optimize the gradient descent process in this context. The formula for the natural policy gradient update, as described in the NPG Cen et al. [2020] paper, is also straightforward, and the pseudo-code provided in the paper offers a clear and informative outline of the implementation process. Nevertheless, the algorithm's structure will serve as a fundamental architecture for model-free implementations, as it consistently involves two distinct updates for the inverse and constrained problems, followed by a policy update using the combined recovered Q-values from both problems.

3.4.3 Sampling

To move towards model-free implementations, the sampling method for expert trajectories and agent replay buffer during training needs to be implemented. This is achieved through the use of a **Buffer** object, which is returned when using the **generate_until_done()** method from the **replay_buffer** module. For this work, the sampling is only done in simulation using a noisy Gridworld environment, however it can easily be extended to sample trajectories with the jetbots using the ROS2 overlay module.
The purpose of the **Buffer** object is to generate rollouts and store transition data. Where each transition is represented by a numpy array of shape `[N_transitions, 6 + n_constraints]`, where `buffer[i, :]` contains the current state *s*, action taken *a*, next state *s_next*, termination and truncated signals d[2] and *t*, reward *r*, and constraint violation indicators *psi_j* for each constraint *j*. During training, the **Buffer** object is utilized to extract data as index -numpy or torch- arrays of shape `[N_transitions, 1]` (except for *psi*, which has shape `[N_transitions, n_constraints]`). This allows to then easily index arrays to perform operations over all states and actions in the buffer.
Additionally, the **Buffer** object allows for sampling random trajectories or transitions from the buffer, it can also be used as is with Numpy functions thanks to the implementation of the `__array__` class method. The **Buffer** can also calculate and returns the vector of gammas Γ corresponding to each transition in its buffer, to account for discounting -when estimating an expectation with samples- until a fixed horizon **T** and this even if we truncated the episode when the agent reached a terminal state at **T_d** < **T**.

[2]Note that the done signal which terminates the episode is raised when we are on the terminal state, otherwise we would never collect rewards when they are defined on terminal states. If we never reach a terminal state, the truncated signal is raised.

This vector is defined as follows:

$$E_\mu[r] = \frac{1}{|B|} \sum_{i=0}^{N_{traj}} \sum_{t=0}^{T} \gamma^t r(s_t^i, a_t^i) = \frac{1}{|B|} \sum_{i=0}^{N_{traj}} \mathbf{r}_i^\top \Gamma \tag{3.1}$$

Where:

$$\Gamma = \left[1, \gamma, \gamma^2, \cdots, \frac{\gamma^{T_f} - \gamma^{T+1}}{1-\gamma}\right]$$

$$\mathbf{r}_i = \left[r(s_0^i, a_0^i), \cdots, r(s_{T_f}^i, a_{T_f}^i)\right] \text{ with } T_f = T_d \text{ if terminal state else } T_f = T$$

Examples of using the **Buffer** object in Python code can be found in Appendix A.2.
The **generate_until_done** function, which is part of the **replay_buffer** module, is responsible for generating the buffer object with trajectories that complete once the agent reaches a terminal state. It takes the environment *env*, the agent, the number of trajectories *n_traj*, and the maximum number of steps *max_steps* as input. During each iteration, the function collects the current state, selects an action, advances the environment, and records the next state, reward, termination signal, and constraint violations. Finally, the buffer is converted into a **Buffer** object and returned for further use.

3.4.4 Model Free GDA

With the Buffer object, the implementation of Model Free GDA is straightforward. Compared to Model Based GDA we want to remove the dependency on the model by approximating the gradients with the trajectories in the replay buffer and from the expert while performing soft Q learning to recover the Q values for the NPG update step. This is done in $\sim$ 15 lines of code without any problems:

```
def gda_model_free_update(policy_buffer):
    s, a, s_next, d, _, _ = policy_buffer.extract_datas()

    ## Primal update
    # soft Q learning step
    _q_grad = np.zeros_like(self._q)
    next_v = soft.pi_values(self._q, self._policy,
     ↪  self._beta)[s_next]
    targets = self.r[(s, a)] + (1 - d) * self._gamma * next_v
    np.add.at(_q_grad, (s, a), self._q[(s, a)] - targets)
    self._q = self._q - self._cfg.eta_q * _q_grad /
     ↪  policy_buffer.size

    # NPG udpate
```

```
        self._policy = npg_step(self._q, self.policy, self._beta,
        ↪  self._gamma, self._cfg.eta_pi)

        ## Dual update
        # Gradient descent
        grad_w = np.mean(self._Phi[s, a], axis=(0, 1)) -
        ↪  self._sigma_E
        self._w = self._w - self._cfg.eta_w * grad_w

        grad_xi = self._env.b - np.mean(self._env.Psi[s, a],
        ↪  axis=(0, 1))
        self._xi = self._xi + self._cfg.eta_xi * grad_xi

        # Project onto ball
        self._w = project(self._w, ball="l2-ball")
        self._xi = project(self._xi, ball="linf-ball")

        # soft Q learning step
        _q_grad = np.zeros_like(self._q)

        next_v = soft.pi_values(self._q, self._policy,
        ↪  self._beta)[s_next]
        targets = self.r[(s, a)] + (1 - d) * self._gamma * next_v

        np.add.at(_q_grad, (s, a), self._q[(s, a)] - targets)
        self._q = self._q - self._cfg.eta_q * _q_grad /
        ↪  policy_buffer.size

        # NPG udpate
        self._policy = npg_step(self._q, self.policy, self._beta,
        ↪  self._gamma, self._cfg.eta_pi)
```

Where `sigma_E` corresponds to $\phi^T \hat{\mu}_{\mathcal{D}}^E$ where $\hat{\mu}_{\mathcal{D}}^E$ is calculated as in (2.9).

3.4.5 IQ-learning

The implementation of IQ-learn update presented challenges, primarily due to the limited explanations provided in the original papers [Garg et al., 2021] and the absence of code to replicate their gridworld experiments. Various aspects, including gradient calculations, convergence problems, and sampling methods, were either not addressed or only partially discussed, particularly for the discrete case. Nonetheless, by adapting the implementation to leverage the automatic differentiation capabilities of PyTorch, it was possible to achieve comparable results, at least for the imitation learning task as we will see in the next chapter.

The main reason for this is that initially, the gradient calculations for the objective were performed explicitly. However, due to the complexity involved, such as the presence of the `logsumexp` function, softmax operations, and handling discrete values, the calculation process became error-prone. It became challenging to determine whether observed discrepancies or issues stemmed from the implementation itself or the lack of information in the paper. Consequently, after weeks of implementation efforts and encountering numerous difficulties, it was decided to utilize the capabilities of PyTorch for gradient calculations. Additionally, attempts were made to replicate the code workflow provided in the paper as closely as possible, aiming to reduce the likelihood of implementation errors. This adaptation, as well as the willingness to replicate the results presented in their papers, made this part of the code the longest and hardest part of this work. On the other hand, it really contributed to enhance the capabilities of the framework as many parts of the code had to be re-designed, re-thought and/or fixed when they were suspected to be causing issues.
All the methods to perform one step of the IQ_learn algorithm are available as static methods in the `IQ_learning` module inside the `methods` module of the framework. Here is a brief explanation of the workflow of the code to perform one step of the q_update for the IQ_learn algorithm:

1. **Line 10-11:** In order to use torch's gradient calculation, it is necessary to convert all the NumPy arrays that will be used as torch tensors. The `requires_grad` should be set to **True** for the loss as well as the `q_values` since we want to propagate the backward pass through them. By performing the necessary transformations using torch operations and calling the `backward()` method on the loss, we can then obtain the `grad` attribute of `q_values`, which contains the gradient computation.

2. **Line 14-24:** If the user requires using the policy buffer for the approximation of the second term of the IQ loss, the policy buffer is appended to the expert buffer. The index arrays are then extracted from the resulting buffer.

3. **Line 35-62:** The loss is computed, following the same workflow as in Garg et al. [2021].

4. **Line 63-64:** The gradient is obtained using torch's autograd, and the gradient ascent is performed to update the `q_values`. Note that the losses are saved during the calculations and returned for logging purposes.

The full implementation of the update function with PyTorch is available in Appendix A.3.

3.4.6 Gymnasium environment

Gymnasium based its implementation on the classic "agent-environment loop" depicted in Figure 3.3. While it is a simplified representation of reinforcement learning it allows to set a standard for RL's interaction implementation. This loop can be easily implemented using the following code:

```
import gymnasium as gym
env = gym.make("<registered_env>")
observation, info = env.reset()
is_done = False

# Generate one trajectory
while not is_done:
    action = agent.sample_action(observation)  # Agent policy
    ↪  that uses the observation
    observation, reward, is_done, _, _ = env.step(action)

    if is_done:
        observation, _ = env.reset()

env.close()
```

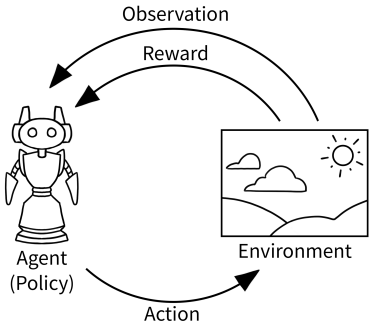

Figure 3.3: *Schema of the "agent-environment loop". Source:* https://gymnasium.farama.org/content/basic_usage/

In order to register a user-defined environment to be able to call it with Gymnasium's make function, two requirements must be met. First, the environment class should inherit from the base class `gymnasium.Env` and implement the main API methods: `step()`, `reset()`, `render()`, and `close()`. The `step()` method updates the environment state based on the provided action and returns the next observation, the reward received, a boolean flag indicating if the episode is done, a boolean flag which is not used in this implementation and any additional information. The `reset()` method resets the environment to its initial state and returns the initial observation.

The `render()` method can be implemented to visually represent the environment state, although it is not available in the current implementation. The `close()` method can be used to release any resources associated with the environment.
The second requirement is to register the user-defined environment using the `gymnasium.register` function, which allows the environment to be accessed using the `gymnasium.make` function by providing the registered environment ID.
Here's an example of registering a user-defined environment:

```
from gym.envs.registration import register

register("<env_name>",
 ↪  entry_point="<entry_point_of_user_defined_env>")
```

3.4.7 DiscreteEnv and Gridworld

The DiscreteEnv class is responsible for creating discrete environments that are compatible with gymnasium. It requires three arguments: the initial state distribution ν_0, the transition dynamics P, and the state-action reward r. With this information, it automatically infers the state and action spaces and implements the necessary methods for compatibility with Gymnasium. One specific environment created using DiscreteEnv is the gridworld environment, registered in gymnasium as `gridworld-v0`. This environment represents a constrained discrete environment where the world is structured as a grid of cells. Fig. 3.5 below depicts the lab's maze represented as a gridworld environment.

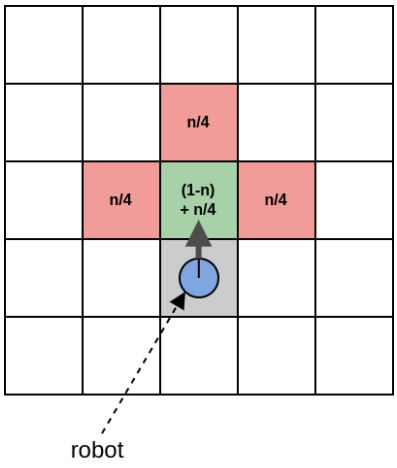

Figure 3.4: *Schema of the model of the noise with probability written in the cells, n corresponds to the noise of the environment. GREEN: Anticipated final state. RED: Noisy final state.*

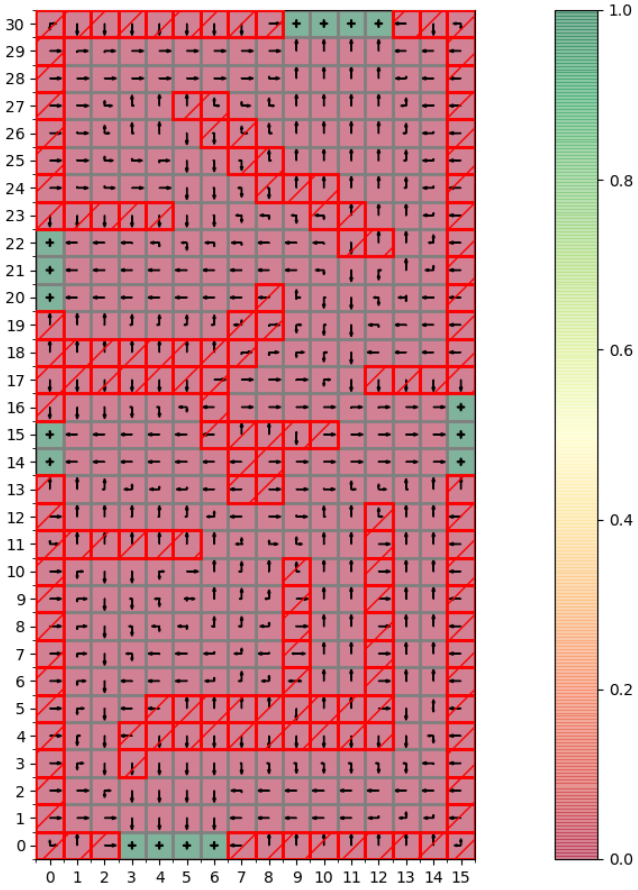

Figure 3.5: *View of the lab represented as a gridworld. Arrows represent a policy, cells' color the reward, hatched cells represent the obstacles.*

In this environment the noise adds a probability that the agent ends up in the wrong terminal state. It allows to model a bit finer what could really happen in reality. The model of the noise is depicted in the Fig. 3.4.
The Gridworld class, built as a specialization of the DiscreteEnv, is responsible for handling the creation of this gridworld environment. It offers great flexibility in creating constrained environments, allowing users to specify various parameters in the settings file. Some of the key parameters that can be specified include:

- A dictionary of obstacles, where each obstacle is defined as a list of lines' endpoints.
- A list of exit states, where each exit is represented as a the two endpoints of a line.
- The dimensions (width and height) in meters, which can be specified or automatically inferred based on the obstacles and exits.

- The grid size in meters, corresponding to the length of one side of the square grid.
- The noise level, which represents the probability of ending up in a non-desired state.
- The list of possible actions that agents can take in the environment.
- The way the rewards must defined in this world among the predefined reward classes.

3.4.8 Sweeps and hyperparameters search

WandB offers a powerful feature called sweeps, which allows for automated hyperparameter search and exploration of the model space. Sweeps can be performed using different methods, including random search, grid search, and Bayesian tuning. Grid search involves defining a grid of hyperparameters along with lists of acceptable values to try for each hyperparameter, and the algorithm exhaustively tries every combination on the grid. On the other hand, random search involves sampling hyperparameter values from specified distributions, instead of trying all possible combinations.
Grid search can be computationally expensive as it requires trying all combinations, while random search provides a good overview with fewer runs by sampling hyperparameter configurations independently. However, both approaches treat hyperparameter configurations as independent, which may not always be optimal. Bayesian hyperparameter tuning addresses this limitation by building a probabilistic model for the objective function and using it to find the best hyperparameters for model training[3].
After many different tries the grid search was finally employed for the sweeps as it was hard to define a cost function for the Bayesian hyperparameter search and the grid search gave a good overview of parameters importance.

3.4.9 Config and settings files

To facilitate experimentation, tuning, and variation, it is possible to specify settings before training, as described in Section 3.3. These settings can be defined in a JSON file following and encompass various objects and their corresponding parameters. They are declared as shown in the code below. The main objects that can be specified include `gridworld`, `cirl`, `safe_rl`, `inverse_rl`, and `logger`. Each of these objects has its own set of parameters, an inclusive list under the form of JSON file of these parameters along with a brief description of their respective functions can be found in Appendix A.4.

[3]Further details on these methods can be found in the WandB sweeps documentation https://docs.wandb.ai/guides/sweeps

```
"object":{
    "parameter1": value,
    "parameter2": value
}
```

3.4.10 ROS2 interface

The integration of the CIRL framework with the Sycabot ROS2 framework was a straightforward process. The ROS2 framework can be treated as a blackbox that advertises the position of the jetbots and awaits commands for the MPC or the jetbot's wheels from external sources via ROS2 topics. The simplified diagram below illustrates the interconnection between the ROS2 framework, the ROS2 interface, and the CIRL framework:

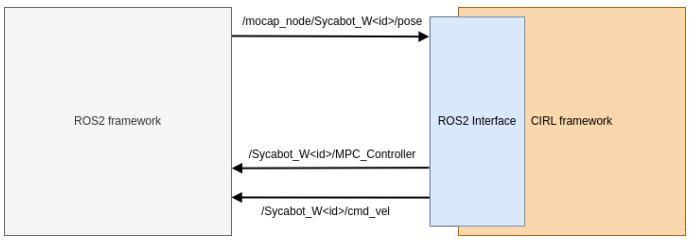

Figure 3.6: *Simplified overview of the interconnections between the frameworks.*

The ROS2 interface acts as a bridge, facilitating the communication and data exchange between the CIRL framework and the Sycabot ROS2 framework. It allows the CIRL framework to access the pose information of the jetbots and send commands to the MPC controller or directly control the jetbot's wheels through the ROS2 topics provided by the ROS2 framework.

Chapter 4

Experiments and discussion

The following section presents the results and comparisons of the three algorithms - GDA-CIRL, SGDA-CIRL, Q-CIRL - developed in the methodology section. To ensure clarity and consistency, all experiments are conducted and compared in the 6x6 gridworld environment shown in Figure 4.1. The optimization results in the laboratory environment are showcased in Appendix B.2 for the model-free algorithm but not discussed further as the 6x6 gridworld setting allows for quick and fast testing and analysing while the discussions and observations made in this settings should remain valid for the laboratory settings (16x31 gridworld).

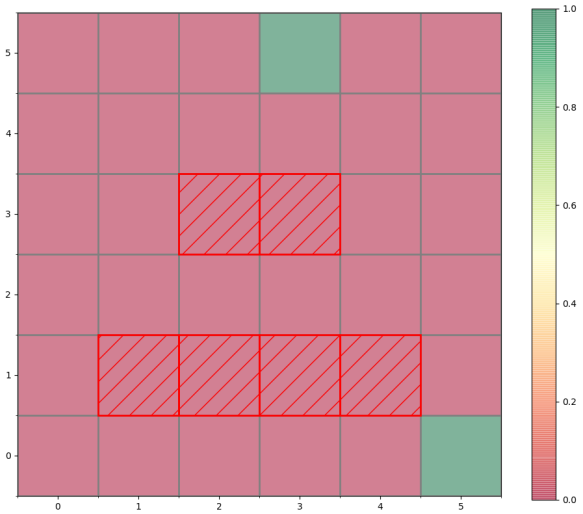

Figure 4.1: *6x6 gridworld environment used for the trainings. Cells' color represents the reward and red hatched cells represent the constrained states.*

4.1 Training setup

4.1.1 Environment

The gridworld environment was initialised with 4 actions UP, DOWN, LEFT, RIGHT, with noise of 0.05, gamma of 0.9 and beta of 0.1. The constraint threshold was set to 5e-3 for every constraint and the reward to be +1 on every exit.

4.1.2 Expert demonstrations

We obtained expert demonstrations by training a regularized model based constrained agent in the environment. The convergence criterion was chosen as the l1-norm between the unregularized cost (2.3) of the previous iteration and the current one. The optimisation was stopped if the norm was below the tolerance threshold, set to $tol = 1e - 5$. The policy obtained after training is shown in Figure 4.2. For training in the 6x6 setup, the expert buffer is filled with 1000 trajectories, with a different seed, at the beginning of each run.

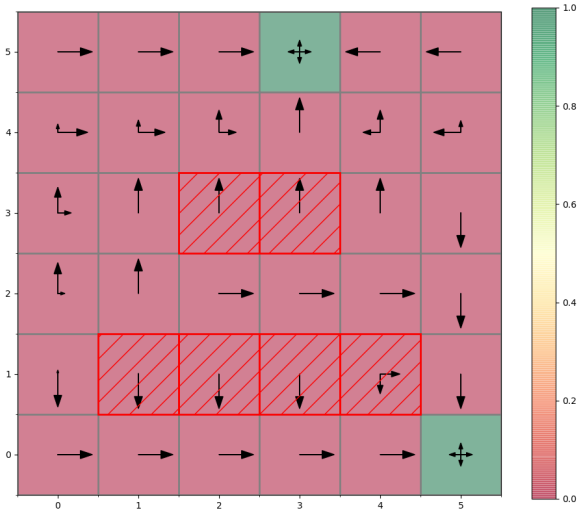

Figure 4.2: *6x6 gridworld environment where the arrows correspond to the expert policy. Cells' color represents the reward and red hatched cells represent the constrained states.*

4.1.3 Hyperparameters

The optimal hyperparameters were found using random search sweeps from WandB and empirical search. For the sweeps β[1], as well as the learning rates η were considered as a tuning parameter, all other parameters were fixed empirically. Tables B.1 in Appendix B.1 shows the hyperparameters used for each agent training.

4.1.4 Testing

Every agent are tested every 500 iterations by comparing the expected reward of the expert and the agent, the constraints violation, the l1-norm to the expert occupancy measure and the l2-norm to the expert reward up to a constant. These metrics are calculated using the exact occupancy measure[2] of the agent's policy and the Monte Carlo estimate of the expert one as defined in (2.9).

4.1.5 Other comments

The feature class of the GDA and SGDA-CIRL algorithms has been designed to not account for constrained state and enforce the reward to zero at these states. This adjustment had no apparent effect on the training process and results. It is reasonable to assume that since the reward cannot be determined within the constraints, assigning it a value of zero is a fair choice. By implementing this approach, it is possible to enhance the generalizability of rewards by avoiding abnormal values for the reward in these states as they are almost never visited.

4.2 Results

In this section, we will showcase the training results of the three agents on the 6x6 gridworld. Each algorithm is trained 10 times for 200.000 iterations, the results are then averaged and given along with their standard deviation.

4.2.1 Optimisation time and convergence time

Table 4.1 presents the algorithmic performances of the three algorithms. The convergence time is defined as the point at which the l1-norm to the expert occupancy measure, stabilizes within a range of $\pm 5\%$ of its final value. Also note that the average time per iteration takes into account the total training time, including the generation of any required buffers and logging, which occurs every 500 iterations.

[1]Here beta is considered a tunable parameter as it can be viewed as a way to control the randomness of the learned policy and therefore drastically impact the training results.

[2]The occupancy measure is calculated using the environment's dynamics during logging. This is the only time the dynamics are used for the model free agents.

Agent	Average time per iteration (s)	Convergence time (#iterations)
GDA-CIRL l1 ball	0.00186 ± 0.0001	51.328 ± 37.518
GDA-CIRL l2 ball	0.00186 ± 0.0001	6.328 ± 1.951
SGDA-CIRL	0.00337 ± 0.00017	9.217 ± 2.106
Q-CIRL	0.00204 ± 0.00005	48.843 ± 10.761

Table 4.1: *Comparison of algorithmic performances of the three algorithms.*

Among the three algorithms, the SGDA-CIRL algorithm shows slightly lower performance in term of time per iteration. This can be attributed to the fact that SGDA-CIRL needs to generate a policy buffer at each iteration and perform four different gradient steps over the buffer elements. In contrast, the GDA-CIRL algorithm does not require any buffer for training and mainly involves quick array operations. Moreover, the Q-CIRL algorithm also generates a buffer, but only performs three gradient steps and utilizes the optimized gradient calculations provided by PyTorch. Moreover, it is important to note that the focus of this study was not on optimizing time complexities, so these differences could also be attributed to that aspect.
Additionally, the SGDA-CIRL and GDA-CIRL agents converges approximately five times faster in average compared to the other two algorithms.

4.2.2 Training

The primary focus of this section will be to compare the SGDA-CIRL and Q-CIRL algorithms, as they represent the novel contributions in this study. Additionally, the GDA-CIRL algorithm will be utilized as a benchmark to assess the performance of the other two algorithms.

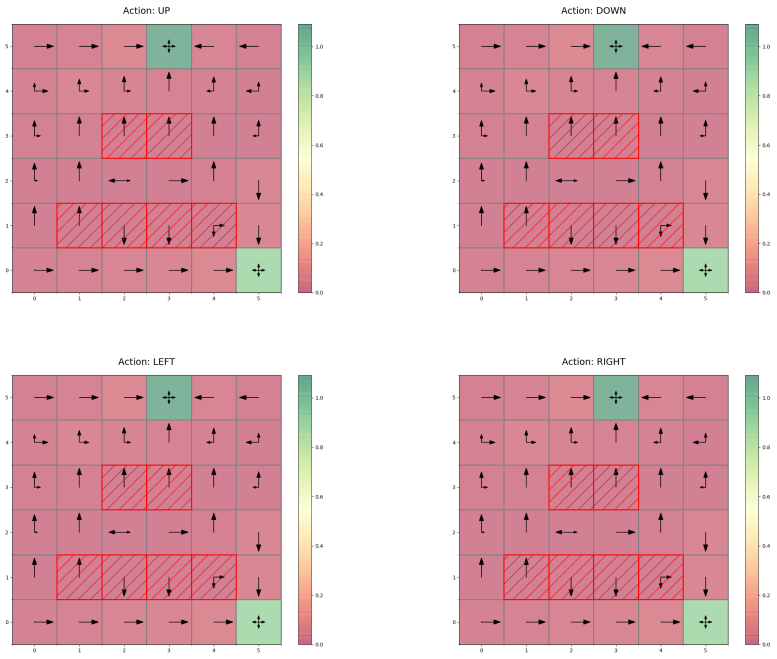

Figure 4.3: *Result of the training of the GDA-CIRL algorithm with reward featured projected onto the l2 ball. The color of the cells represents the reward value for the specified action, the red hatched cells the constrained states and the arrows the learned policy. The learned reward is state dependent only.*

Fig. (4.3, 4.6, 4.4), display the results of the training for each agent. Results for the l1-ball projection of the SGDA and GDA algorithms are shown in Appendix B.3. Table 4.2 show the comparison of the performances of the three agent.

Agent	$\mathbb{E}_{\mu^{\boldsymbol{\pi}}}[\boldsymbol{r}] - \mathbb{E}_{\mu_E}[\boldsymbol{r}]$	$\|\mu^{\boldsymbol{\pi}} - \mu_E\|_1$	$\|\boldsymbol{r} - (\boldsymbol{r}_E + k) \cdot \hat{\boldsymbol{n}}\|_2$
GDA-CIRL l1 ball	0.0171 ± 0.0003	5.769 ± 0.83	0.1857 ± 0.004
GDA-CIRL l2 ball	0.092 ± 0.005	0.144 ± 0.025	0.427 ± 0.083
SGDA-CIRL l2 ball	0.119 ± 0.008	0.112 ± 0.036	0.4048 ± 0.097
Q-CIRL	-3.172 ± 0.008	0.058 ± 0.013	2.687 ± 0.001

Table 4.2: *Averaged training results.*

It appears that while the Q-CIRL algorithm outperforms the other in imitating the expert policy, its efficiency in recovering rewards that are close to the expert one is lower than what was claimed in the original paper. This behavior has a direct impact on the generalizability of the reward to new sets of constraints, which is the main motivation behind CIRL. However,

the original paper proposes a modification to the optimization function that aims to learn state-only rewards. Unfortunately, all attempts to implement this modification successfully have failed thus far, this remains open for further investigation.

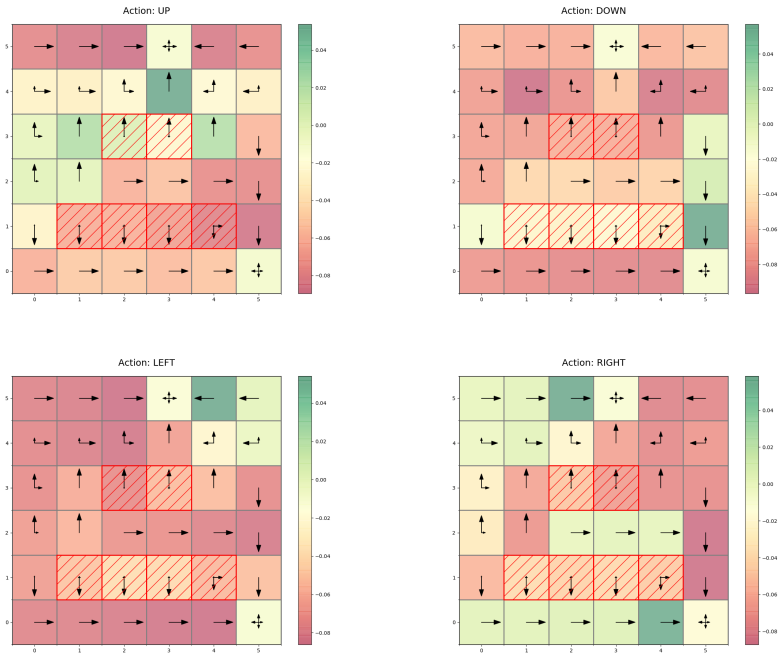

Figure 4.4: *Result of the training of the Q-CIRL algorithm. The color of the cells represents the reward value for the specified action, the red hatched cells the constrained states and the arrows the learned policy.*

In contrast, the SGDA-CIRL algorithm demonstrates impressive performance in recovering both the reward and policy of the expert, producing results that are close to optimal compared to the GDA-CIRL algorithm. Although the recovered reward may exhibit some slight imbalance, it appears to possess effective generalizability to a new set of constraints. This intuition is supported by the visualization in Figure 4.5, which displays the policies obtained by training a new agent using the learned reward for both the SGDA and Q-CIRL agents[3]. Notably, for states (2,2), (3,2), and (4,2), the policy obtained for the Q-CIRL reward continues to avoid the previous obstacles, while the one for the SGDA-CIRL reward directly heads towards the exits.

[3]In this case, the CMDP problem was solved using an LP solver.

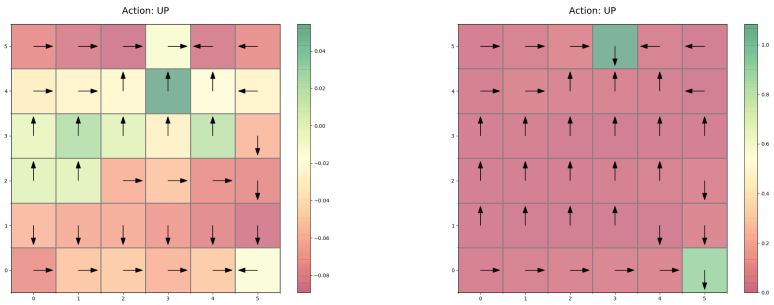

Figure 4.5: *Comparison of the policy obtained after training a new agent with the reward obtained for the Q-CIRL (LEFT) and for the SGDA-CIRL (RIGHT) algorithms. An LP solver was used to solve the unregularized CMPD, that is why policies are deterministic. Policy on the terminal state (5,1) and (3,5) should be ignored as the agent will stay there anyway.*

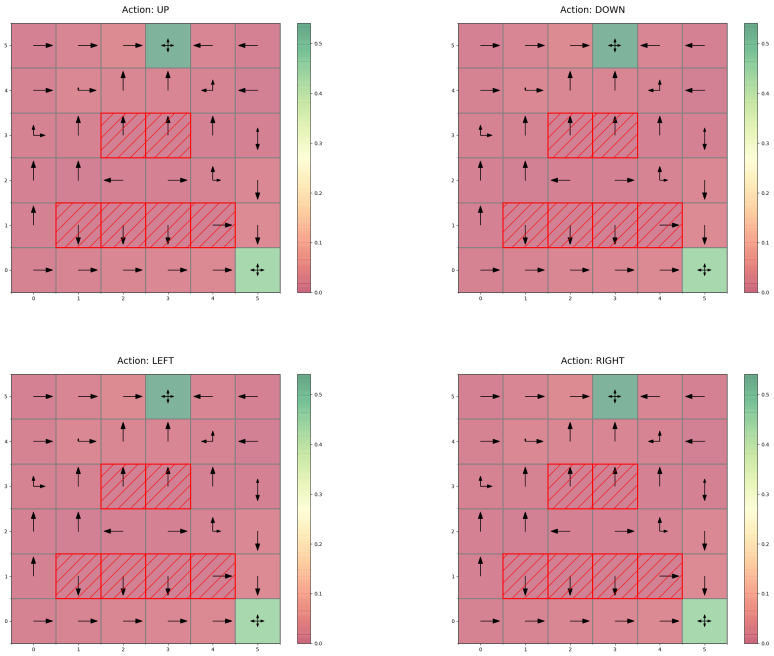

Figure 4.6: *Result of the training of the SGDA-CIRL algorithm. The color of the cells represents the reward value for the specified action, the red hatched cells the constrained states and the arrows the learned policy. The learned reward is state dependant only.*

4.3 Going further

In the previous section, we assessed the performance of three algorithms in a noisy gridworld. However, we did not explore how well the robot would

perform if it followed the learned policy in a real-world scenario, as everything was conducted in simulation. In our experiments, the robot adhered to the policy precisely due to the use of a highly reliable MPC controller, ensuring it moved to the next cell accurately. Nevertheless, we recognize that this approach could be perceived as somewhat biased, as the agent's noise is notably diminished when using the controller. Nonetheless, incorporating a controller in conjunction with a discrete algorithm could offer a viable real-world solution, especially considering the ease of training algorithms in discrete settings. However, it is important to note that implementing this approach for complex tasks with thousands of cells would demand substantial memory and computational capacity to store the necessary arrays.
An alternative approach, which we did not investigate in this study, involves operating in a continuous action space, where actions are represented as velocity commands provided to the robot's wheels. To achieve this, the algorithms would need to undergo some modifications, incorporating deep neural networks and soft actor-critic methods to approximate policy and value functions. In such cases, relying solely on simulations for learning may not suffice, as accurately modeling the robot's dynamics becomes challenging. Fine-tuning the algorithms using real-world trajectories might become necessary to achieve optimal performance in these scenarios.

Chapter 5

Conclusion

In conclusion, this scientific report presented the successful implementation, design, and evaluation of two novel algorithms for Constrained Inverse Reinforcement Learning (CIRL), one based on stochastic gradient descent ascent (SGDA-CIRL) and the other based on IQ-Learn, a recent state-of-the-art imitation learning algorithm.
Our findings have shown promising results for the Q-CIRL algorithm, but it also revealed certain limitations that may pose challenges for its further exploitation in CIRL. Notably, the algorithm struggled to recover easily generalizable rewards and faced difficulties in enforcing reward classes. Despite these limitations, there are still unanswered questions regarding its implementation, particularly in recovering state-only rewards. On the other hand, the SGDA-CIRL algorithm surpassed expectations, delivering results that approached the model-based approach and demonstrating rapid convergence compared to Q-CIRL.
The development of an open-source framework for CIRL was a significant contribution of this study, offering a versatile platform for implementing and extending CIRL algorithms and other reinforcement learning (RL) techniques. The availability of this framework as open-source software encourages collaborative contributions and paves the way for future advancements in the field.
To further advance the research in this area, additional experiments could be conducted by exploring new sets of hyperparameters and creating novel gridworld configurations. The framework's emphasis on code reusability and adaptability makes such experiments easy to conduct. Moreover, new algorithms based on various CIRL and RL methods implemented in the framework could be proposed and tested, providing valuable insights into the effectiveness of different approaches.
However, practical considerations and limitations need to be acknowledged. Model-based algorithms, while successful in controlled environments, is not really realistic in real-world implementations and sampling based methods seems more appropriate. While discrete implementation is well suited to quickly experiment and validate theoretical results in a lab settings, the

memory and computation requirements of this method limit its applicability to complex real-world problems. For continuous state-action space applications, methods like soft actor-critic would be necessary for policy estimation. Addressing noise in system dynamics through regularization is essential for any real-world application, and careful consideration of the link between a CMDP and the real world is vital, which can be facilitated using practical frameworks like ROS2, as utilized in this study.
In conclusion, this research contributes significantly to the advancement of CIRL algorithms, shedding light on their strengths, limitations, and potential avenues for future research. By addressing the outlined considerations and limitations and conducting further experiments, the field of Constrained Inverse Reinforcement Learning can continue to progress and offer solutions to complex real-world problems

Bibliography

E. Altman. *Constrained Markov Decision Processes*. Chapman and Hall, 1999.

L. Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

M. Bloem and N. Bambos. Infinite time horizon maximum causal entropy inverse reinforcement learning. In *53rd IEEE Conference on Decision and Control*, pages 4911–4916, 2014. doi: 10.1109/CDC.2014.7040156.

S. Cen, C. Cheng, Y. Chen, Y. Wei, and Y. Chi. Fast global convergence of natural policy gradient methods with entropy regularization. *arXiv*, 2020. doi: 10.48550/ARXIV.2007.06558. URL https://arxiv.org/abs/2007.06558.

S. Cen, C. Cheng, Y. Chen, Y. Wei, and Y. Chi. Fast global convergence of natural policy gradient methods with entropy regularization, 2021.

D. Garg, S. Chakraborty, C. Cundy, J. Song, and S. Ermon. Iq-learn: Inverse soft-q learning for imitation. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=Aeo-xqtb5p.

T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement learning with deep energy-based policies. 2017. doi: 10.48550/ARXIV.1702.08165. URL https://arxiv.org/abs/1702.08165.

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. 2018. doi: 10.48550/ARXIV.1801.01290. URL https://arxiv.org/abs/1801.01290.

C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. URL https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

T. Renard. Provable convergence guarantees for constrained inverse reinforcement learning, 2023.

A. Schlaginhaufen and M. Kamgarpour. Identifiability and generalizability in constrained inverse reinforcement learning. 2023. doi: 10.48550/ARXIV.2306.00629. URL https://arxiv.org/abs/2306.00629.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. 2018.

M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. de Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. J. S. Tan, and O. G. Younis. Gymnasium. URL https://github.com/Farama-Foundation/Gymnasium.

P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

Appendix A

A.1 Simplified implementation of the training process

```
def training():
    cfg = Config.load_config("settings_filename")
    env = make("gridworld-v0", cfg=cfg.env)

    success, message, _ = LP.solve(env, cfg)
    if not success:
        raise RuntimeWarning("some error message")

    expert = Expert(
        env,
        datas_path="expert_datas_filename",
        n_expert_trajectories=cfg.n_expert_trajectories,
    )

    logger = AgentLogger(args, cfg.Logger)
    agent = AgentClass(args, [expert], cfg.AgentCfg, logger)

    it = 0
    while it < cfg.max_iter or convergence_criterion:
        buffer = replay_buffer.generate(env, agent,
        ↪  cfg.batch_size, cfg.max_steps)
        agent.update([expert.buffer.get_batch(cfg.batch_size),
        ↪  buffer], n_iter=it)
        it += 1

    if logging:
        save(relevant_information)

    visualize(env, infos)
```

A.2 Buffer code examples

Here are some code examples using the Buffer object:

```
# Generate a new Buffer
buffer = replay_buffer.generate_until_done(env, agent,
    cfg.batch_size, cfg.max_steps)

# Returns index arrays as numpy arrays.
s, a, s_next, d, r, psi = buffer.extract_datas(as_torch=False)
# Useful to index array easily.
q[(s,a)] = r[(s,a)] + gamma*values[(s,a)]

# Transform the Buffer object into a numpy array of shape
    [N_transitions, 5 + n_constraints].
buffer_arr = np.array(buffer)

# Get a batch of random transitions or traj, also a Buffer.
batch_transitions: Buffer =
    buffer.get_batch_random_tranisitions(batch_size)
batch_traj = buffer.get_batch_random_traj(batch_size)

# Get vector Gamma
gammas = buffer.get_gamma_vec(gamma)
```

A.3 iQ learn implementation

The following code shows the implementation of one update step of the iQ-learn algorithm.

```
def q_update_torch_grad(
    expert_buffer: np.ndarray,
    q_r: np.ndarray,
    gamma: float,
    beta: float,
    cfg: DiscreteConfig.inverseRlConfig,
    policy_buffer: np.ndarray = None,
):
    # Cast numpy array to tensor and create loss
    q_values = torch.tensor(q_r, requires_grad=True)
    loss = torch.tensor(0.0, requires_grad=True)

    # Add the policy buffer to the expert buffer only if the
        calculations of the second term use policy states
```

```
    if iQLossType(cfg.loss_type) == iQLossType.VALUE:
        buffer = Buffer(np.append(expert_buffer, policy_buffer,
        ↪  axis=0))
        is_expert = np.append(
            np.ones(np.array(expert_buffer).shape[0],
            ↪  dtype=bool), np.zeros(policy_buffer.shape[0],
            ↪  dtype=bool)
        )
    else:
        buffer = expert_buffer
        is_expert = np.ones(np.array(expert_buffer).shape[0],
        ↪  dtype=bool)

    # Get index array from the buffer
    s, a, s_next, d, _, _ = buffer.extract_datas(as_torch=True)

    # Compute optimal values
    values = beta * torch.logsumexp(q_values / beta, axis=1)

    current_q = q_values[s, a]
    next_v = values[s_next]
    current_v = values[s]

    #  calculate 1st term for IQ loss
    #  -E_(mu_E)[Q(s, a) - gamma*V(s')]
    y = (1 - d) * gamma * next_v
    if cfg.use_targets:
        with torch.no_grad():
            y = (1 - d) * gamma * next_v

    r = (current_q - y)[is_expert]
    with torch.no_grad():
        grad_phi = get_grad_phi(r, cfg.div_method,
        ↪  cfg.alpha_chi2)

    loss = (grad_phi * r).mean()
    loss_dict = {"softq_loss": loss.item()}

    # calculate 2nd term for IQ loss, we show different
    ↪  sampling strategies
    if iQLossType(cfg.loss_type) == iQLossType.VALUE:
        # sample using expert and policy states (works online)
        # E_(mu)[V(s) - gamma*V(s')]
        value_loss = (current_v - y).mean()
```

```
    elif iQLossType(cfg.loss_type) == iQLossType.VALUE_EXP:
        # sample using only expert states (works offline)
        # E_(mu_E)[V(s) - gamma*V(s')]
        value_loss = (current_v - y)[is_expert].mean()
    elif iQLossType(cfg.loss_type) == iQLossType.V0:
        # alternate sampling using only initial states (works
        ↪  offline but usually suboptimal than `value_expert`
        ↪  startegy)
        # (1-gamma)E_(mu0)[V(s0)]
        value_loss = (1 - gamma) * current_v[is_expert].mean()
    loss -= value_loss
    loss_dict["value_loss"] = value_loss.item()
    loss_dict["total_loss"] = loss.item()
    loss.backward()
    q_r -= cfg.eta_qr * np.array(-q_values.grad)
    return q_r, loss_dict
```

A.4 Config object declaration

The following shows the different objects which can be declared in a JSON setting file as well as their respective parameters. Note that each cirl agent defined by its `class_type` parameter has its own mandatory attributes, that is why three different versions of the object are shown.

```
{
    "gamma": 0.9,
    "beta": 0.001,
    "batch_size": 5,
    "n_expert_trajectories": 1000,
    "max_steps": 10,
    "expert_filename": "safe_rl_exp_6x6",

    "cirl":{
        "class_type": "gda_model_based",
        "eta_w":1e-2,
        "eta_xi":0.01,
        "eta_policy":0,
        "proj_type": "l1_ball",
        "ball_radius": 8,
        "feature_class": 1,
        "max_iter":20000,
        "tol": 1e-9,
        "q_threshold": 100000,
```

```
        "xi_threshold": 100000
    },

    "cirl":{
        "class_type": "gda_model_free",
        "eta_w":1e-5,
        "eta_xi":0.01,
        "eta_q":0.5,
        "eta_policy":1e-4,
        "proj_type": "",
        "ball_radius": 8,
        "feature_class": 1,
        "max_iter":150000,
        "tol": 1e-9,
        "q_threshold": 100000,
        "xi_threshold": 100000
    },

    "cirl":{
        "class_type": "iq_learn",
        "eta_qr":1e-4,
        "eta_qc":1e-4,
        "eta_xi": 1e-2,
        "div_method": "chi2",
        "alpha_chi2": 0.5,
        "loss_type":"value_expert",
        "use_targets": true,
        "state_only_reward":false,
        "torch_grad": false,
        "max_iter":200000,
        "tol": 1e-9,
        "q_threshold": 100000,
        "xi_threshold": 100000
    },

    "inverse_rl":{
        "eta_w": 0.9,
        "eta_qr":0.001,
        "eta_policy":0,
        "lambda_w": 0,
        "feature_class": 0,
        "n_traj_feature_exp":10000,
        "proj_type": "l1_ball",
        "ball_radius": 2,
```

```
        "div_method": "chi2",
        "alpha_chi2":0.5,
        "max_iter": 200000,
        "tol": 1e-16,
        "q_threshold": 200000,
        "use_targets": true,
        "loss_type": "value_expert",
        "state_only_reward":false,
        "torch_grad": true
    },

    "safe_rl":{
        "eta_xi":0.001,
        "eta_qc": 0,
        "eta_policy": 0,
        "max_iter": 100000,
        "tol":1e-5,
        "xi_threshold": 999999999999
    },


    "logger":{
        "project_name": "cirl_gda_model_free",
        "configs_to_log":["cirl"],
        "hyperparam_to_log": ["beta", "gamma",
↪   "gridworld.noise", "batch_size", "max_steps"],
        "log_freq": 500,
        "logging": true
    },

    "gridworld":{
        "noise":0.05,
        "actions":[[0,-1],[1,0],[0,1],[-1,0]],
        "gridsize":1,
        "dimensions":[5,5],
        "infer_dimensions":false,
        "reward_class":0,
        "offset": [2.5,2.5],
        "b_values":{
            "default":5e-3
            "1": 0.1
        },
        "exits":[
            [[3,5],[3,5]],
```

```
            [[5,0],[5,0]]
        ],
        "obstacles":{
            "1":[
                [[1,1],[4,1]],
                [[2,3],[3,3]]
            ]
        }
    }
}

```

Parameter	Description
`gamma`	The discounting factor.
`beta`	The temperature parameter.
`batch_size`	Number of batches to generate for training.
`n_expert_trajectories`	Number of expert trajectories to generate and store in the expert buffer.
`max_steps`	Maximum number of steps per episode.
`expert_filename`	Filename of the pre-trained expert to use for training.
`class_type`	Agent to use for CIRL (options: "iq_learn", "gda_model_based", "gda_model_free").
`eta_<var>`	Learning rate for the gradient algorithm corresponding to <var>.
`proj_type`	Projection to use for projecting the feature vector w.
`ball_radius`	Radius of the L1_ball.
`feature_class`	Feature class for the reward.
`max_iter`	Maximum number of iterations for training.
`tol`	Tolerance for convergence.
`q_threshold`	[DEPRECATED]
`xi_threshold`	[DEPRECATED]
`use_targets`	Whether to take targets into account in the gradient update for iQ learning.
`loss_type`	Method to estimate the second term of the loss in iQ learning.

`state_only_reward`	Whether to recover state-only rewards in iQ learning.
`torch_grad`	Whether to compute gradients using torch or the explicit method.
`project_name`	Project name for logging in WandB.
`configs_to_log`	Configurations of the object to log as config for the WandB run.
`hyperparam_to_log`	Specific hyperparameters to log from the configuration file.
`log_freq`	Frequency of logging.
`logging`	Whether to enable logging or not.
`noise`	Noise level of the environment.
`actions`	List of available actions in the environment.
`gridsize`	Length of one side of the squares in the grid.
`dimensions`	Dimensions of the world in meters.
`infer_dimensions`	Whether to infer dimensions from obstacles and exits or not.
`reward_class`	Predefined reward class to use.
`offset`	Offset to change the centering of the gridworld.
`b_values`	Dict defining the constraint cost for each constraint.
`exits`	List defining the endpoints of the exit lines in real-world coordinates.
`obstacles`	Dict specifying the obstacles as lists of line endpoints.

Table A.1: *Settings parameters and their usage*

Appendix B

B.1 Hyperparameters used

Table B.1 shows the hyperparameter configuration used for the trainings. Note that for the GDA-CIRL agent, there are two sets of parameters, one set for each of the projection used.

Agent	Hyperparameter	Value
Q-CIRL	beta	1e-2
	eta_qr	1e-3
	eta_qc	1e-6
	eta_xi	1e-2
	batch_size	20
	max_steps	10
	loss_type	"value_expert"
	div_method	"chi2"
	alpha_chi2	0.5
	use_targets	True
SGDA-CIRL	beta	1e-2
	eta_q	5e-2
	eta_w	1e-3
	eta_xi	1e-2
	eta_pi	5e-4
	ball_radius	$\sqrt{2}$
	batch_size	20
	max_steps	10
	feature_class	Everywhere
	proj_type	"l2_ball"
GDA-CIRL	beta	1 | 1e-1
	eta_w	1e-3 | 1e-3
	eta_xi	10 | 1e-2
	ball_radius	2 | $\sqrt{2}$
	proj_type	"l1_ball" | "l2_ball"
	eta_pi	0 (soft policy iter) | 5e-4
	feature_class	Everywhere

Table B.1: *Hyperparameters used for the training of the agents.*

B.2 Training in the lab settings

Figure B.1 illustrates the training results of the SGDA-CIRL algorithm in the lab settings. The obtained rewards demonstrate satisfactory performance, indicating potential for generalization. However, a drop in performance is observed in this settings. It is important to note that the hyperparameters utilized in this training phase were suboptimal, suggesting that alternative parameter configurations may yield improved results. Moreover, the infrequent visitation of certain states by the expert, such as (5,1), (5,2), (5,3), or exits (4,0) and (5,0), presents challenges for the agent to recover their associated rewards. Consequently, the agent may assign higher significance to more frequently visited states. However, addressing this issue remains an open question for future research.

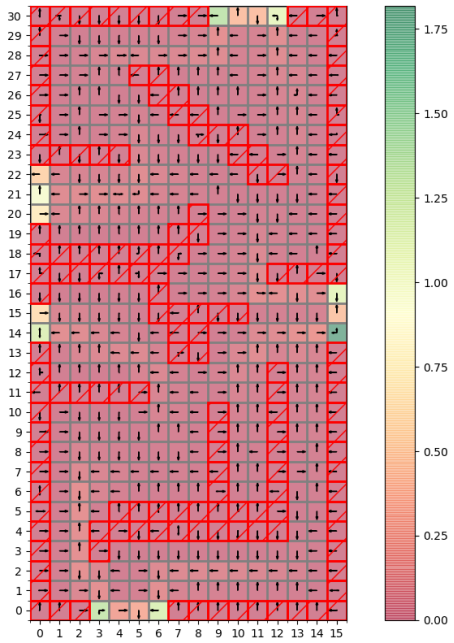

Figure B.1: *Result of the training of the SGDA-CIRL algorithm (RIGHT) in the lab environment. The expert used is depicted in Fig.B.2. The reward features are projected onto the l2 ball.*

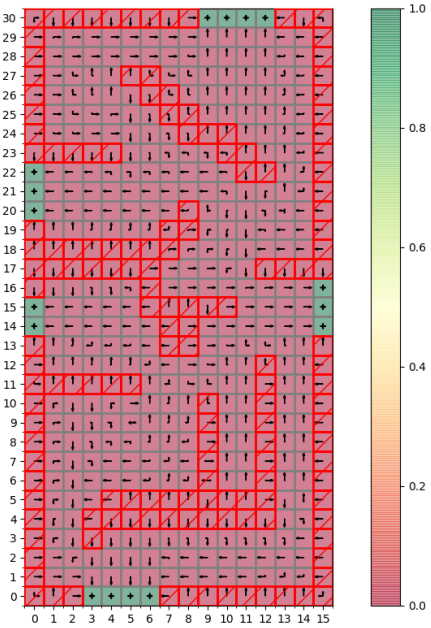

Figure B.2: *Expert used for the training in the lab settings.*

B.3 l1-ball feature projection

The suboptimal results observed for the SGDA-CIRL agent on Fig. B.4 when using the projection onto the l1-ball raise an open question for future investigation as the underlying reasons for this outcome warrant further exploration and analysis.

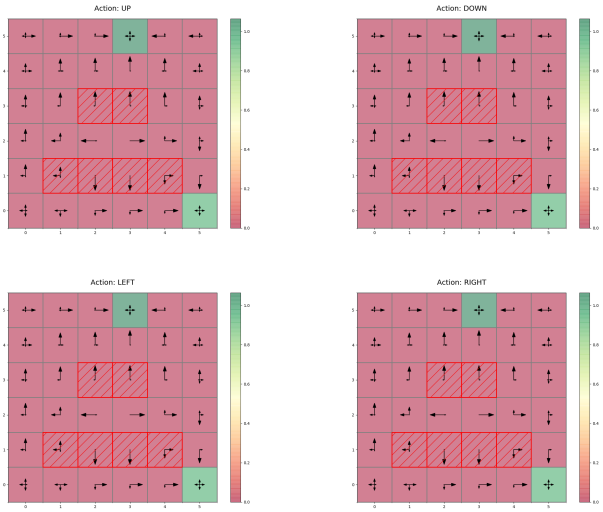

Figure B.3: *Result of the training of the GDA-CIRL algorithm with reward features projected onto the l1 ball.*

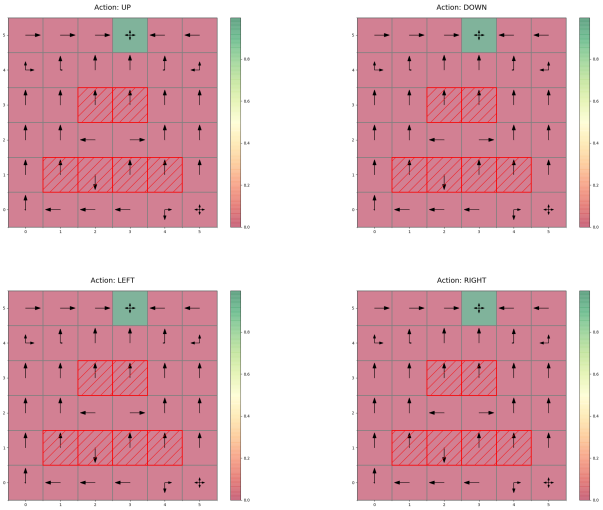

Figure B.4: *Result of the training of the SGDA-CIRL algorithm with reward features projected onto the l1 ball.*