Final Report for MSc Machine Learning Individual Project

Inverse Reinforcement Learning using Generative Planning Models in Trajectory Space

Student Number: 23077650

Supervisors: Dr Ilija Bogunovic, Lorenz Wolf and William Bankes

September 2024

Submitted in partial fulfilment of the requirements for the award of MSc in Machine Learning from University College London

Contents

Abstract

Reinforcement Learning has been applied to great success to tasks where a reward signal reward is clearly defined or can be hand-crafted. However, its application to tasks such as alignment to ethical standards has been limited by the inability to craft a reward function that can balance multiple (and often subjective) preferences. A possible solution is Inverse Reinforcement Learning (IRL), a class of problems in which one learns a reward function from observed agent behaviour. In this work we propose a method for learning a reward function using diffusion models. Recent work has proposed using diffusion models to learn high-reward policies in sequential decision-making tasks. The general method involves training a diffusion model on a dataset of trajectories in order to learn a model of the environment dynamics, and then using the classifier guidance property of diffusion models to steer their output towards high-return policies. In this work we hypothesise that for a choice of trajectory similarity metric, and given a diffusion model trained on arbitrary trajectories in an environment, and example trajectories of a behaviour we wish to imitate, one can learn a proxy reward function of the desired behaviour (IRL). This learnt reward function can be used to steer the diffusion process towards the behaviour distribution, making our method learn a reward function while also imitating behaviour. We study the performance of our method across three different environments, evaluating both the quality of the reward function learnt, as well as the quality of the output behaviour. We show our method learns a reward function that induces optimal behaviour in simple environments, outperforming state of the art IRL methods. We extend this method to more complex environments, showing that its performance lags behind in such settings. Finally, we present reasons for the failure modes of our method, and propose possible fixes.

Code: The code for this project is publicly available at: `https://github.com/Sam-Oliveira/diffuser_irl`. This repository is a fork from the original Diffuser repository [1]. Development of this project for all environments in this report was done in the 'maze2d' branch, and this branch should be used for CPU execution. The 'cluster' branch was created for running the project using machines with CUDA.

[1]`https://github.com/jannerm/diffuser`

1 Introduction

Reinforcement Learning (RL) is a machine learning paradigm that excels at solving sequential decision-making tasks [1, 2, 3]. The general assumption is that such tasks can be formulated with an agent that interacts with its surrounding environment, and that learns an optimal way to behave based on rewards signals received from the environment. While RL has been applied to great success to settings such as games [4, 5, 6], where there is a clear reward signal to guide learning (e.g. game score, or win/lose), its application in real-world tasks is often limited by the lack of a natural reward function. Tasks such autonomous driving, creative generation, or alignment to human preferences are either highly subjective, or require considering multiple preferences that can't easily be condensed into one scalar function.

To solve the RL problem in the absence of a reward signal, a possible approach is to collect examples of optimal behaviour in an environment, and learn to act in an identical way. This paradigm is called imitation learning, and has been extensively used in areas such as robot control [7, 8, 9]. However, policies are highly sensitive to changes in environment dynamics [10], and thus any learnt behaviour is often rendered unusable when either environment dynamics change, or when aiming to transfer previously acquired knowledge to new tasks.

An alternative approach is to instead use the collected examples of optimal behaviour to learn the reward function under which such behaviour is optimal. Inverse Reinforcement Learning (IRL) [10, 11] is the problem of learning a reward function from observed agent behaviour, and is often used as an intermediate step in imitation learning [7, 8, 12]. Learning a reward function (as opposed to directly learning to imitate the observed behaviour) provides better robustness and generalization to environment changes [10], as the reward function provides a more succinct representation of a task [12, 13] that can successfully be transferred to new environments. Furthermore, reward learning can help better understand a system's preferences, with applications in alignment and safety [14, 15]. Most IRL methods learn by alternating between reward learning, and policy optimisation based on the learnt reward. Early methods focused on parameterising the reward function using a linear combination of features, and learning based on Maximum Entropy IRL [16]. Guided Cost Learning [8] was introduced as an adaption of the maximum entropy IRL framework to reward functions parameterised by neural networks, significantly expanding the complexity of the tasks considered. Finn et al [17] showed a close link between IRL and GAN [18] training, inspiring further work on using adversarial training methods for IRL [9, 19].

IRL suffers from many of the same issues as typical Reinforcement Learning (RL), namely high computational costs [20] and poor sample complexity [21, 22]. Its high computational complexity stems from its typical two-step iterative optimisation, where firstly a MDP is solved for the current reward function (in order to find the optimal policy in such a MDP), and secondly the value function estimate is updated based on some distance between the current policy and the expert policy. Solving a MDP is exponential in the state dimension [12], limiting IRL's application to real-world (often continuous) problems. A large state dimension also leads to a larger number of samples being required for learning. RL's well documented sample inefficiency [23, 24, 25] is also true in an IRL setting, making it the more valuable to create methods that can learn even with few expert demonstrations.

In this work, we introduce a method for extracting a reward function from diffusion models [26, 27]. Diffusion models have emerged in recent years as a valuable class of generative models.

This popularity stems, in part, from their superior ability to generate high-quality samples, as well as their superior training stability, when compared to previous generative models such as GANs [18] and VAEs [28]. Diffusion models have been applied to fields as distinct as natural language processing [29, 30], drug discovery [31, 32], and computer vision [27, 33, 34]. One of the main advantages of diffusion models is the ability to steer their output based on a classifier: either by classifier-free guidance [35], or by classifier-guided sampling [26]. While the former requires classifier-aware training of the diffusion model, the latter can be used to fine-tune the output of an arbitrary diffusion model.

Recent work [36, 37] has applied diffusion models to sequential decision-making. Janner et al [36] proposed training diffusion models to model a prior distribution over offline trajectories, in a method called Diffuser. Using classifier-guided sampling, the Diffuser's output trajectories can be steered with a reward function, as shown in Figure 1. Firstly, the base diffusion model is trained on a dataset of reward agnostic trajectories. Secondly, a reward function is parameterised by a neural network, trained in a supervised fashion using a dataset of trajectory-return pairs, and then used to steer the diffusion model to output trajectories that maximise the return under the learnt reward model. This work shows an alternative approach to generating high-reward behaviour: instead of training a diffusion model directly on high-reward trajectories, which might be scarce, one can train on lower-quality but abundant trajectories, and then steer the model's behaviour with a known reward function. Furthermore, one can use multiple value functions, but the same base trained diffusion model, to obtain distinct behaviours at test time. The guided sampling approach suggests that if one has access to a base trained Diffuser, and expert trajectories that can be interpreted as the output of guided-sampling of this Diffuser, the reward function "used" for such behaviour can be extracted from the guiding process of the Diffuser. That is, one can learn the reward function required to guide the diffusion process from a base Diffuser's distribution towards the expert trajectories' distribution.

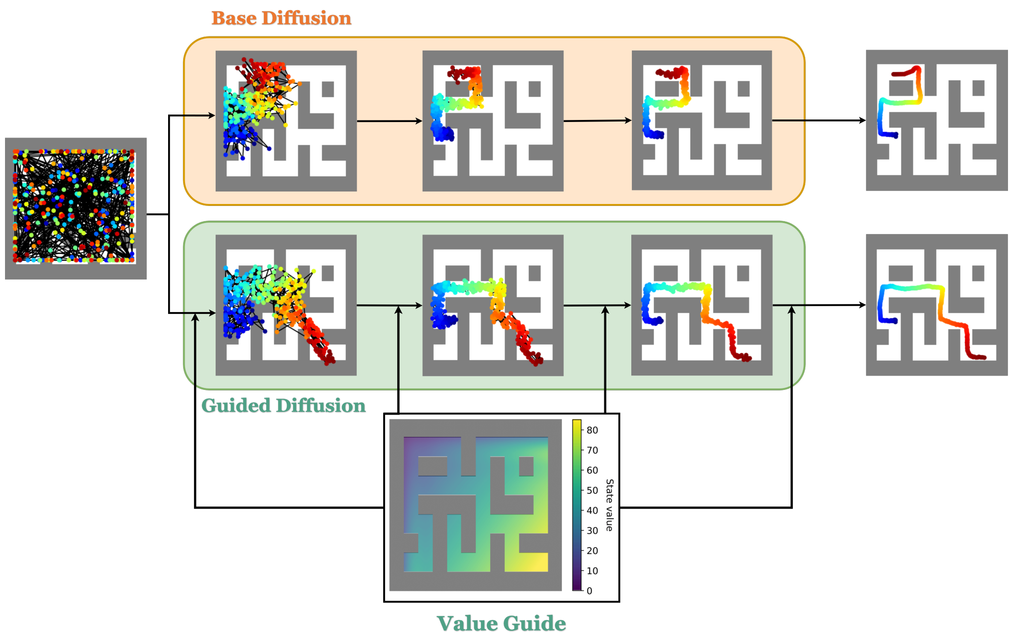

Figure 1: Schematic of classifier-guided sampling using the Diffuser. Trajectories in a maze are shown as a collection of dots representing the agent's location, where the colour shows the temporal progression of its location (from blue to red). Two identical base diffusion models are used to denoise a noisy trajectory, with fixed starting state. A reward guide is added to the bottom model, resulting in a trajectory that moves towards states with high reward.

We introduce a method for extracting a reward function from the Diffuser. Alike the typical Diffuser setting, where learning of the environment dynamics and planning are intertwined, we suggest a method that can learn a value function, and optimise behaviour based on said function without solving two separate problems. Thus, our method circumvents explicitly solving the computationally slow MDP-solving step, while also implicitly using samples for both problems. Learning is thus more sample efficient, as expert trajectories are used to both learn a value function and to optimise behaviour based on the learnt function. Our method does not require environment access, and is applicable to any classifier-guided diffusion model that has been trained to learn an environment's dynamics.

Our method resembles Nutti et al. [38]. Critically, we make two important distinctions. Firstly, we only require one diffusion model, and use the expert trajectories to guide this diffusion model, as opposed to training two diffusion models on different data distributions. Secondly, we aim to learn an actual reward function, instead of a relative reward function[2]. These distinctions mean our method only uses the expert trajectories to learn a reward model, as opposed to an entire diffusion model, making our method more sample efficient and thus more applicable to low data regimes. Furthermore, the training of a base diffusion model and of a single reward model is expected to be faster than the serial training of two diffusion models, reducing computational costs.

We propose a practical algorithm, with a loss function based on distance metrics in trajectory space, for extracting a reward function from a Diffuser model. It does so by learning the guide required to steer the base Diffuser towards outputting samples from the same distribution as the expert trajectories. We evaluate our method's performance by evaluating both the quality of the guided behaviour, as well as the quality of the learnt reward function. We evaluate behaviour quality by calculating the average reward of trajectories resulting from guided sampling of the base Diffuser with the learnt reward model. We calculate the episode return correlation distance (ERC) [39] as a proxy for the distance between a learnt reward functions and the true reward. We compare our method's performance to two imitation learning baselines (Behaviour Cloning and GAIL [9]) and one IRL baseline (AIRL [19]).

We evaluate our method across three different environments, in different experimental setups. In two Maze2D [40] environments (U-Maze and Large Maze), we train a base diffusion model on exploratory reward-agnostic trajectories, and then learn a reward function by guiding the base diffusion model towards outputting trajectories that move towards the bottom right corner of the maze. We observe that our learnt reward model results in a guided diffusion process that outputs samples that indeed mimic the expert dataset and move to this corner in the case of the U-Maze. For the Large Maze, we show that issues with the learnt environment dynamics prevent us from obtaining high-quality expert trajectories. We regardless train our method on sub-par expert trajectories, leading to poor learning. In the HalfCheetah locomotion environment [40], we train the base diffusion model on low-reward trajectories. We obtain a dataset of expert high-reward trajectories, and learn a reward function by steering the base model towards the high-reward behaviour. We observe unstable training, and our method fails to learn a reward function that properly guides towards the expert trajectory dataset. For all environments, we observe our method learns a more meaningful reward function (that is, a reward function that is closer to the true reward) than AIRL.

This thesis is structured as follows: Section 2 presents background and related work on RL,

[2]For technical details on the definition of a relative reward see Section 2.6 or Nutti et al [38].

IRL, diffusion models, and their intersection. Two different classes of metrics relevant to this problem are introduced: in Section 2.2.4 we introduce reward function similarity metrics (metrics to assess how similar two reward functions are), whereas in Section 2.3 we introduce and discuss trajectory space metrics (metrics that asses how different two trajectories are). Section 3 formally introduces the problem statement, and presents our proposed practical algorithm. Our method is then evaluated on the U-Maze [40] environment as a proof-of-concept. In Section 4 we study the empirical performance of our method in more complex environments, in particular the Large Maze and HalfCheetah [40] environments. Section 5 sums up our contribution, highlighting areas of future work.

2 Background

This section summarises presents introductions to each of the fields necessary for the understanding of our method, both providing some theoretical background as well as surveying current methods in each of the fields. We start by introducing RL and the main algorithms in the field (Section 2.1). We then present the IRL and imitation learning problems, motivating them, and surveying not only the first methods used to solve them, but also the current state of the art methods (Section 2.2). In this section we also present metrics to assess similarity between value functions (Section 2.2.4), which will be used to evaluate our method's performance. We then present metrics to assess similarity between trajectories (Section 2.3), which are used as loss functions in our method, as seen in Algorithm 4. Finally, we present a thorough explanation of diffusion models (Section 2.4), and then review the application of such methods to RL (Section 2.5) and IRL (Section 2.6).

2.1 Reinforcement Learning

Reinforcement Learning is a sub-field of machine learning focused on sequential decision-making. The general framework assumes the existence of an agent that interacts with its surrounding environment, and that learns an optimal behaviour in this environment based on reward signals received from the environment. While RL techniques have evolved from tabular settings [41] to deep-learning based approximations [2], the most commonly used model has remained the Markov decision process (MDP) [42].

2.1.1 Markov Decision Process (MDP) and Bellman Equations

A MDP is defined as a tuple $M \coloneqq \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, where its elements are such that

- $\mathcal{S}$ represents the set of states. S_t represents the state at time t.
- $\mathcal{A}$ represents the set of actions. A_t represents the action taken at time t.
- $\mathcal{T}$ represents the transition function $\mathcal{S} \times \mathcal{A} \to \text{Prob}(\mathcal{S})$, where its output defines a probability distribution over the set of next states.

- $\mathcal{R}$ represents the reward function, which can be defined in either of the following ways: $\mathcal{S} \to \mathbb{R}$, $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$ or $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, depending on whether one considers the reward of a single state, the reward of a state-action tuple, or the reward of a specific transition to the next state respectively.
- $\gamma \in [0, 1]$ is the discount factor, representing the decreasing weight given to future rewards.

A MDP is based on the following two environment assumptions: the environment is Markovian, such that the state at time step $t + 1$ is fully determined by the state at time t, and fully independent of the previous states given the current state S_t; and the agent can observe the entire environment at any point (fully observable environment).

The RL problem amounts to finding how the agent should behave under this MDP so that it maximises the cumulative discounted reward, also known as return, expressed for an episode with T steps as:

$$G_t = \sum_{i=0}^{T} \gamma^i R_{t+i+1} \tag{1}$$

The return is a random variable, and highly depends on the state S_t, transition function $\mathcal{T}(S_{t+1}|S_t, A_t)$ and policy $\pi(A_t|S_t)$ (that is, the choice of actions by the agent). Thus, one can define the state value function $V(s)$ as the expected return from a state s, such that:

$$V(s) = \mathbb{E}_{S,A\sim\mathcal{T}}\left[G_t\right] = \mathbb{E}_{S,A\sim\mathcal{T}}\left[\sum_{i=0}^{T} \gamma^i R_{t+i+1}\right] \tag{2}$$

$$= \mathbb{E}_{\pi}\left[R_{t+1} + \gamma V(S_{t+1})|S_t = s\right] \tag{3}$$

Similarly, one can define the state-action value function $Q(s, a)$ as the expected return from a state s taking action a:

$$Q(s, a) = \mathbb{E}_{\pi}\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})|S_t = s, A_t = a\right] \tag{4}$$

The recursive equations (3) and (4) are the Bellman equations [43]. Similarly, one can define the Bellman optimality equations for the optimal state value function $V^*(s) = \max_{\pi} V_{\pi}(s)$ and optimal state-action value function $Q^*(s, a) = \max_{\pi} Q(s, a)$ such that:

$$V^*(s) = \mathbb{E}_{\pi}\left[R_{t+1} + \gamma V^*(S_{t+1})|S_t = s\right] \tag{5}$$

$$Q^*(s, a) = \mathbb{E}_{\pi}\left[R_{t+1} + \gamma Q^*(S_{t+1}, A_{t+1})|S_t = s, A_t = a\right] \tag{6}$$

For any MDP, there is always at least one optimal policy, and the value function of all optimal policies is the same. Thus, there is always at least one optimal deterministic function, obtained from picking the action that maximises the state-action value function at each state.

If both $\mathcal{T}$ and R are known, then the environment is fully known. Under this condition, the Bellman equations can be solved via dynamic programming [44]. The two main algorithms used for solving the Bellman equations are policy iteration and value iteration [45].

However, the application of such methods is limited to problems with relatively small state/action spaces (under 100 thousand). Furthermore, in many problems the environment is not fully known. In that case, one can use Monte Carlo (MC) methods. These consist of taking samples of the discounted return from a state, and averaging such samples to obtain an estimate of the value of a state.

TD Learning [46] is an alternative to MC methods that takes inspiration from human learning. TD methods take a limited number of steps from a state, and then bootstrap from the estimated value function of the state that has been reached. Its simplest form, TD(0) [45], can be described by the update

$$V(s_t) = V(s_t) + \alpha \left(R_{t+1} + \gamma V(S_{t+1}) - V(s_t)\right) \tag{7}$$

where α is a learning rate. Transferring the general TD Learning update to a state-action value function setting yields the SARSA [45] algorithm. Transferring SARSA to off-policy learning (a learning setting where the agent's behaviour policy differs from the policy it is trying to evaluate) yields Q-learning [47].

Both MC and TD based methods sample from the environment, and thus do not directly aim to learn a model of the environment. This is the case for a large part of current RL methods, although section 2.1.4 presents some of the work being done on model-based RL. Model-free methods are often divided in value-based and policy-based methods.

2.1.2 Value-based RL

Value-based methods rely on representing the value function, and aim to find the optimal value function V^*. Both SARSA and Q-Learning are examples of such methods. One of the first main RL breakthroughs was Deep Q-Learning (DQN) [2], which combines neural networks and Q-Learning. The use of a neural network to model the state-action value function allows for non-finite state/action spaces. DQN's success was largely due to the incorporation of two seemingly simple ideas: experience replay, and the use of a target network. The former reduced correlation between training samples, whereas the latter reduced the effects of the "deadly triad" [48], both resulting in more stable training.

DQN was the first RL system to achieve above-human performance in most Atari games. Further work introduced the Double DQN (DDQN) [49], where the target network, previously obtained by freezing parameters, was replaced by an entirely separate network, and optimisation of the two networks was done in turns. This helped solve the issue of overestimation of Q values in Q-Learning.

Much work has resulted from extensions to the original DQN. Hester et al [50] extended on the method to allow for multi-step TD Learning, whereas boostrap DQN [51] combined DQN with Thompson sampling to increase exploration. Rainbow [52] combined many of the aforementioned extensions (experience replay, multi-step learning), as well as dueling networks [53], noisy networks [54], and distributional DQN [55], to achieve state of the art performance in Atari Games, beating both DQN and DDQN.

2.1.3 Policy-based RL

A distinct class of methods bypasses the need to calculate a value function, and instead directly aims to optimise the agent's policy. These methods are commonly labeled policy gradient methods. The general idea is that one can parameterise the policy $\pi_\theta(a|s)$, and directly optimise it to maximise the expected return:

$$J(\theta) = \mathbb{E}_{\pi_\theta(s)}\left[R_t\right] = \mathbb{E}_{\pi_\theta(s)}\left[\sum_a Q(s,a)\pi_\theta(a|s)\right] \tag{8}$$

By using the log-derivative trick, this optimisation procedure can easily be turned into a typical gradient descent approach by following the gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta(s)}\left[R_t \nabla_\theta \log \pi_\theta(A_t)\right] \tag{9}$$

The REINFORCE [45] algorithm breaks the expectation operator in (9) by taking Monte-Carlo samples of the return and following this stochastic gradient direction. The Policy Gradient Theorem [45] shows that the policy gradient for an average reward formulation is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[q_{\pi_\theta}(S_t, A_t)\nabla_\theta \log \pi_\theta(A_t|S_t))\right] \tag{10}$$

This has led to the development of Actor-Critic methods, which incorporate aspects of both value-based and policy-based RL. For the case of the Q-function Actor-Critic, a Critic is used to estimate $Q(S_t, A_t)$ using any of the algorithms described in section 2.1.2, whereas an Actor uses such an estimate to update θ by following the policy gradient. Further work has extended this by using a Critic that estimates an advantage function $A(S_t, A_t) = Q(S_t, A_t) - V(S_t)$ so as to reduce the variance of the learning process. Mnih et al [56] suggested the asynchronous advantage Actor-Critic (A3C), where data collection happens asynchronously, leading to faster training. The method also introduces N-step returns and adds the policy's entropy to the objective function, in order to increase exploration. A2C [56] was introduced as its synchronous version, overall achieving better performance.

Policy-based methods are inherently optimisation procedures, where the (possibly stochastic) gradient direction is followed. As with other gradient descent procedures, the learning rate α can largely affect learning. Intuitively, the return should monotonically increase as the optimisation proceeds. Schulman et al [57] proposed TRPO, an algorithm using a trust-region constrain to ensure that in each policy update the old and new policies did not differ too much, improving training stability and achieving theoretical guarantees on a monotonic return increase as training proceeds. Later work introduced PPO [58], an improved version of TRPO, which introduced a truncated objective function leading to more stable policy updates.

2.1.4 Model-based RL

With the exception of dynamic programming methods, all algorithms presented thus far are model-free, as they skip an explicit modeling of the environment and instead focus on learning optimal value functions/policies. However, explicit learning a model of the environment allows the rest of the decision-making to be taken care of by more mature planning methods (recall the two dynamic programming methods introduced earlier). Furthermore, sample efficiency increases, as the (imperfect) model of the environment can be used to generate synthetic transition data used to improve planning.

Dyna-Q [59] learns a model from real agent experience, and then learns a value function from both real data and data simulated based on the learnt environment. As the model becomes more accurate, the data used for learning the value function also gets better, resulting in a more accurate value estimate while requiring significantly less data than typical model-free methods. Janner et al [60] introduced a model-based policy optimization algorithm (MBPO) which uses a learnt reward to create rollouts from previously encountered data. Hafner et al [61] proposed Dreamer, an algorithm that learns a model from experience, and then uses an Actor-Critic to learn an optimal policy by interacting with the learnt environment.

A different class of methods focuses on simulation-based search. In such algorithms, the next action is selected by looking ahead (forward search) and building a search tree according to a model of the MDP. This MDP is ”solved” to find the best new action, and the agent takes a step in the environment. At the next time-step, the forward search is once again repeated. In such a setting, planning is much simpler, as there is no need to ”solve” the whole MDP: the agent only has to solve the sub-MDP starting from its current state. This is similar to the Diffuser [36] setting, in which we develop our method. In typical simulation-based search methods, to break the curse of dimensionality associated with very large MDPs, the algorithm instead simulates/samples episodes of experience from its learnt model, and then uses model-free methods to evaluate all possible state-action pairs. This setting is often called Monte-Carlo Tree Search [62], a method that blurs the boundary between model-free and model-based RL, as while it keeps an explicit model of the environment, it resorts to sampling from it (as in typical model-free methods) when planning ahead.

2.2 Inverse Reinforcement Learning

The goal of Inverse Reinforcement Learning (IRL) [10, 11] is to infer the reward function of an agent, given observations of its behaviour. While in a typical RL setting learning amounts to learning a policy that maximises a reward signal, in IRL the setting is inverted and we aim to find the reward signal for which the observed policy/behaviour is optimal. Note that this setting is very similar to imitation learning [7]. However, in the latter, we circumvent learning a reward function and instead aim to directly learn how to create behaviour that matches some dataset of observed behaviour. Section 2.2.3 presents further discussion on imitation learning. Methods such as the one we propose in this report, as well as AIRL [19], blur the boundary between these two types of methods, and output both a value function and imitated behaviour.

While it might not be clear at first-sight why one would aim to solve this inverse problem, learning a reward function is actually key to applying RL to real-world settings. The benefits behind learning a reward function can be informally divided into three categories:

1. **Circumvent manual specification of a reward function**
    - RL has been applied, to great success, to game-like settings [2, 52], where a clear reward signal is often available or can be crafted. However, the application of RL methods in the real world is often limited by the lack of a natural reward function.

Let us think of a self-driving car A aiming to overtake another car B on a highway: an hypothetical reward function would have to not only take into account the distance of car A to car B (as well as their velocities/accelerations), but also the number of cars on the road, their positions with respect to cars A and B, the weather conditions influencing the traction on the road, as well as the topology of the road. In short, it is not realistic to create a reward function that encompasses all factors required for decision-making. While in certain applications roughly specified rewards can be enough to learn (near-)optimal behaviour, in real-world applications, especially those where safety is paramount, it is unlikely optimal behaviour can be obtained.

2. **Improved Generalization**

   - Due to RL's documented low sample efficiency [23, 25], it is paramount to find ways to transfer knowledge already acquired: be it for an agent whose environment partly changes, or for knowledge to be transferred between different agents to increase the knowledge pool of the group. The most straightforward way to do so would be by transferring an agent's policy. However, even slight changes in the environment can dramatically change the optimal behaviour [10], leading to an inoperable policy. On the other hand, the reward function has been shown to be much more robust to such changes [10]. Thus, IRL provides a way for not only increased robustness, but also for knowledge transfer (be it for multi-task or multi-agent settings).

3. **Imitation Learning**

   - As further discussed in section 2.2.3, imitation learning methods aim to learn a policy that matches an available behaviour dataset. While this is not IRL's goal, IRL can be used as an intermediate step in imitation learning frameworks: one can learn the reward function that explains the observed data, and then use the learnt value function to generate behaviour that matches the observed data. Such frameworks can be applied to behaviour prediction tasks.

In a formal setting, the IRL problem can be defined as:

Definition 1 (IRL Problem) *Let us define a non-reward MDP $\mathcal{M} := \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \gamma \rangle$. Let D be the dataset of trajectories $D = \left[\langle (s_0, a_0), (s_1, a_1), ..., (s_T, a_T) \rangle^1, ..., \langle (s_0, a_0), (s_1, a_1), ..., (s_T, a_T) \rangle^N \right]$ with N trajectories, $s \in \mathcal{S}, a \in \mathcal{A}$. The IRL problem amounts to determining $\mathcal{R}_D$, the reward model that best explains dataset D.*

There are challenges that make IRL a particularly challenging problem to solve. Firstly, it does not have a single solution, as there is no single value function that explains the observations. Ng and Russel [11] showed that many reward functions can explain a dataset of observations. Furthermore, assessing similarity between value functions is not straightforward. Firstly, a direct comparison of rewards is not accurate, as an MDP's optimal policy is invariant to affine transformations of the reward function [63]. Secondly, reward functions that differ only in only very few state-action pairs can actually result in largely different optimal policies. Section 2.2.4 presents a more thorough discussion on this issue, introducing metrics for value function comparison.

IRL also suffers from many of the same issues that typical RL faces, including high computational complexity and poor sample complexity. Regarding the former, typical IRL algorithms rely on a two-step iterative process. Firstly, the MDP is solved for the current reward function, generating behaviour samples. Secondly, the value function parameters are updated to minimise some metric between the sampled behaviour and the observed behaviour (dataset). Solving a MDP is exponential in the number of components of the state vector (curse of dimensionality), and this issue is further worsened if the state space is continuous, as is the case in applications such as robotics. Regarding the latter, as the state dimension grows, a larger number of samples is required to cover the entire state space. RL's inherent poor sample complexity translates to the IRL setting, making algorithms particularly data-hungry as state/action space dimensionality increases.

2.2.1 Canonical Methods

Most of the canonical methods in IRL can be divided in three categories: Maximum Margin methods [11, 64, 65], Maximum Entropy methods [16, 9, 8] and Bayesian methods [66].

Maximum Margin Methods

Maximum margin methods aim to learn a reward function that explains the observed behaviour better than any other policy by a certain margin. Ng and Russel [11] suggested maximising the difference between the expected state-action value of the optimal action from state s and the expected value of the second-best action:

$$\sum_{s\in\mathcal{S}} Q(s,a^*) - \max_{a\in\mathcal{A}\setminus a^*} Q(s,a) \tag{11}$$

with a^* being the optimal action for state s. With a feature-based linear reward function $Q(s,a) = \boldsymbol{w}^T\boldsymbol{\phi}(s,a)$, this can be formulated into a linear programming problem, where the obtained policy maximises the margin (11) while also producing the policy represented by the given trajectory dataset. Ratliff et al [64] proposed maximum margin planning (MMP), which assumes that each trajectory in the dataset is associated with a distinct MDP and represents a different policy. These different MDPs are assumed to share the same reward model, and the linearly parameterised reward function is solved via quadratic programming, using the margin in (11) and a regularisation term that encourages the behaviour derived from the learnt value function to match the observed trajectories.

Maximum Entropy Methods

Maximum entropy IRL focuses on solving the IRL problem while obtaining a policy with the most entropy possible. High entropy policies allow for better exploration, and avoid many of the failure cases of deterministic policies. One possible formulation aims to find a reward function for whom the distribution over all trajectories has the maximum entropy, that is

$$\max_{\Delta} H(\tau) = \max_{\Delta} - \sum_{\tau\in(\mathcal{S}\times\mathcal{A})^T} P(\tau)\log P(\tau) \tag{12}$$

over the simplex Δ of T-length trajectories τ. Ziebart et al. [16] introduced MaxEntIRL, which finds the IRL solution that maximises the entropy (12) while introducing two additional constraints: matching of feature counts between learnt and demonstration trajectories, and ensurance that the distribution over trajectories is a probability distribution. This is a convex non-linear optimisation problem, and it has been shown that its solution lies in the exponential family [16]. Thus, one can parameterise $P(\tau) = e^{\sum_{(s,a)\in\tau} \boldsymbol{w}^T \boldsymbol{\phi}(s,a)}$ and solve the optimisation procedure by finding the maximum of:

$$\arg\max_{\boldsymbol{w}} \sum_{\tau\in\mathcal{D}} \log P(\tau;\boldsymbol{w}) \tag{13}$$

Deep MaxEntIRL [67] generalises this framework to non-linear reward functions using a neural network. To do so, the network follows the gradient of the likelihood in (13). PI-IRL [68] extended the same objective to continuous state spaces.

Bayesian Methods

Bayesian inference can also be applied to an IRL problem setting. In such cases, we place a prior on the reward function $\mathcal{R}$, and define a likelihood function $P(\tau|R)$. This yields a typical Bayesian posterior:

$$P(\mathcal{R}|\tau) \propto P(\tau|\mathcal{R})P(\mathcal{R}) \tag{14}$$

where the likelihood is often factorised such that $P(\tau|\mathcal{R}) = \prod_{(s,a)\in\tau} P((s,a)|\mathcal{R})$

A common choice for the likelihood function is a Boltzmann distribution with the energy function being the state-action value function, such that

$$P(s,a|\mathcal{R}) \propto e^{\frac{Q(s,a)}{k}} \tag{15}$$

with a parameter k. BIRL [66] uses such a parameterisation choice, and presents different possible priors over $\mathcal{R}$. Other approaches such as GPIRL [69] model the reward function using a Gaussian process with a learnt kernel.

2.2.2 GAN-GCL and AIRL

In recent years, inverse RL methods have evolved from hand-crafted features to deep learning methods. Finn et al [17] showed that the learning of a cost function by a discriminator in typical GAN set-ups is akin to the learning of a reward model in maximum entropy IRL, such that a reward function can be extracted from the discriminator if it is trained to optimality (please see section 2.3.2 for more detail on GANs). Their analysis interprets the IRL problem as solving the maximum likelihood problem:

$$\max_{\theta} \mathbb{E}_{\tau\sim\mathcal{D}} [\log p_\theta(\tau)] \tag{16}$$

with $p_\theta(\tau) \propto p(s_0) \prod_{t=0}^{T} e^{\gamma^t \mathcal{R}_\theta(s_t,a_t)} p(s_{t+1}|s_t, a_t)$. In such work, they introduce GAN-GCL, which optimises (16) using a GAN [18]. In such a setting, the discriminator takes the form:

$$D_\theta(\tau) = \frac{e^{h(\tau)}}{e^{h(\tau)} + \pi(\tau)} \tag{17}$$

where $h_\theta(\tau)$ is a learnt function. The policy π, akin to the generative model, is trained to maximise $\mathcal{R}(\tau) = -\log(D(\tau)) + \log(1 - D(\tau))$, by a policy optimisation method of choice. As in other IRL methods, optimisation consists of two alternating optimisation procedures: in a first step, the reward function is updated (here equivalent to updating the discriminator), and in a second step the MDP is "solved" for the current reward function, outputting a new policy (here equivalent to updating the policy). If this set-up is trained to optimality, the optimal reward function is $R^*(\tau) = h^*(\tau) - const.$, and π is the optimal policy.

Fu et al [19] introduced AIRL, a method based on adversarial learning. AIRL identifies that the trajectory-level framework that GAN-GCL assumes is prone to high variance and consequently poor learning. Thus, AIRL adapts the discriminator to a single transition setting. The discriminator and consequent reward function then take the form:

$$D_{\theta,\phi}(s, a, s') = \frac{e^{h_{\theta,\phi(s,a,s')}}}{e^{h_{\theta,\phi(s,a,s')}} + \pi(a|s)} \tag{18}$$

$$\mathcal{R}_{\theta,\phi}(s, a, s') = \log D_{\theta,\phi}(s, a, s')) - \log(1 - D_{\theta,\phi}(s, a, s')) \tag{19}$$

with $h_{\theta,\phi(s,a,s')} = f_\theta(s, a) + \gamma g_\phi(s') - g_\phi(s)$. The term $g_\phi(s)$ mitigates the effects of unwanted reward shaping, and leads to more robust reward functions. The policy $\pi(a|s)$ is updated in each iteration with respect to the new reward function using a policy optimization algorithm. AIRL produces rewards that are disentangled from the environment dynamics observed during training, producing reward functions that are robust to changes in environment dynamics.

2.2.3 Imitation Learning: Behaviour Cloning, Guided Cost Learning and GAIL

As previously explained, imitation learning is a problem class which aims to learn a policy that matches available agent behavioural data. While some methods directly aim to learn a suitable policy, others take intermediate steps (such as learning a reward function) to help in learning a policy. The simplest method, Behaviour Cloning (BC), reduces to using supervised learning on (s, a) pairs. Most implementations use either the log likelihood of the policy, or the MSE between actions for learning. While it can learn to mimic behaviour quite well, BC tends to generalise very poorly, and lacks robustness to changes in environment dynamics.

Finn et al [8] introduced Guided Cost Learning (GCL), using the learning of a reward function as an intermediate step to learn behaviour from demonstrations. GCL introduced the learning of a reward function via a neural network. It aims to maximise the log likelihood of the expert trajectories under the model $p(\tau)$, assuming $p(\tau)$ is a Boltzmann distribution with its energy function being the negative of the reward function. The method introduces a way to estimate this log-likelihood via sampling (suitable for unknown dynamics), and then uses the algorithm presented by Levine [70] to perform policy optimization under unknown dynamics.

GAIL [9] (partly) circumvents the learning of a reward model, aiming to directly learn a policy from the demonstration data. It does so by having a discriminator $D(s, a)$ that aims to distinguish between expert and generated trajectories. The policy improvement step uses TRPO [57] with cost function $\log(D(s, a))$. In doing so, GAIL obtains a policy as if it were obtained by policy improvement on an IRL learnt value function, without explicitly requiring an intermediate IRL step. As for GAN-GCL, GAIL is shown to be akin to learning with a GAN [9].

2.2.4 Value Function Similarity

One of the roadblocks for a better understanding of IRL methods is the difficulty in assessing similarity between reward functions. It is particularly hard to assess the failure modes of IRL algorithms, and to compare the performance of different algorithms. This difficulty is often bypassed by instead evaluating the policy obtained from following the learnt value function. However, this mixes the reward learning error with possible policy optimisation errors, and might not be a good proxy for the quality of the learnt reward. Simple metrics such as the L2-distance are not satisfactory, as two reward functions which induce the same ordering of policies can have large L2-distance, or alternatively two reward functions which induce opposite ordering of policies can have small L2-distance [39].

Gleave et al [39] introduce ERC as a baseline, analyse where it fails, and introduce Equivalent-Policy Invariant Comparison (EPIC) as the first distance that closely matches the desiderata for a valid value function distance. The authors also introduce NRC, and we present a small description of it in Appendix E.1. Episode return correlation (ERC) is defined as the Pearson distance between the episode returns of two reward functions.

Definition 2 (Pearson distance) *The Pearson distance between two reward functions $\mathcal{R}_A$ and $\mathcal{R}_B$ is defined to be $D_\rho(\mathcal{R}_A, \mathcal{R}_B) = \sqrt{1 - \rho(\mathcal{R}_A, \mathcal{R}_B)}/\sqrt{2}$, with $\rho(\mathcal{R}_A, \mathcal{R}_B)$ being the Pearson coefficient between the two random variables.*

Definition 3 (ERC) *Let D be a distribution over trajectories (denoted coverage distribution). Let τ be the randomly sampled trajectory from D. Let $G(\tau, \mathcal{R}_A)$ be the episodic return of trajectory τ according to reward model $\mathcal{R}_A$. The Episode Return Correlation (ERC) distance between two reward functions $\mathcal{R}_A$ and $\mathcal{R}_B$ is defined as the Pearson distance between their episodic returns on coverage distribution D, that is $ERC(\mathcal{R}_A, \mathcal{R}_B) = D_\rho\left(G(\tau, \mathcal{R}_A), G(\tau, \mathcal{R}_B)\right)$.*

If the return of reward function $\mathcal{R}_A$ is a positive affine transformation of the return of reward function $\mathcal{R}_B$, then they both share the same set of optimal policies. Since the Pearson distance is invariant to positive affine transformations, it follows as a suitable metric. However, ERC is shown to be susceptible to reward shaping, and is not robust to the choice of coverage distribution (the distribution of transitions for which the reward function is evaluated). EPIC is presented as an alternative to previous sub-par distances. Importantly, EPIC satisfies several of the desiderata: it is a pseudo-metric, it is invariant to potential shaping and positive rescaling, it is computationally efficient, it shows robustness to the choice of coverage distribution, and its value is predictive of similarity between trained policies.

Equivalent-Policy Invariant Comparison (EPIC) [39] is a pseudo-metric defined as the Pearson distance between two canonically shaped reward functions.

Definition 4 (Canonically shaped reward function) *Let $\mathcal{D}_{\mathcal{S}} \in \Delta(\mathcal{S})$ be the distribution over states, and $\mathcal{D}_{\mathcal{A}} \in \Delta(\mathcal{A})$ the distribution over actions. Let S, A, S' be independent samples from their respective distributions. The canonically shaped reward function $\mathcal{R}$ is:*

$$C_{\mathcal{D}_{\mathcal{S}},\mathcal{D}_{\mathcal{A}}}(\mathcal{R})(s,a,s') = \mathcal{R}(s,a,s') + \mathbb{E}\left[\gamma\mathcal{R}(s',A,S') - \mathcal{R}(s,A,S') - \gamma\mathcal{R}(S,A,S')\right]. \tag{20}$$

Definition 5 (EPIC) *Let $\mathcal{D}$ be the coverage distribution over transitions (s,a,s'), from which we jointly sample (S,A,S'). Let $\mathcal{D}_{\mathcal{S}}$ and $\mathcal{D}_{\mathcal{A}}$ be the distributions over states and actions respectively. The EPIC distance between two reward functions is*

$$D_{EPIC}(\mathcal{R}_A,\mathcal{R}_B) = \mathcal{D}_{\rho}(C_{\mathcal{D}_{\mathcal{S}},\mathcal{D}_{\mathcal{A}}}(\mathcal{R}_A)(S,A,S'), C_{\mathcal{D}_{\mathcal{S}},\mathcal{D}_{\mathcal{A}}}(\mathcal{R}_B)(S,A,S')) \tag{21}$$

EPIC is shown to be bounded, invariant to potential shaping, and it bounds the regret of optimal policies. The metric outperforms NPEC and ERC empirically, and proves to be a good predictor of subsequent quality of policies optimized for the learnt reward.

Wulfe et al [71] present Dynamics-Aware Reward Distance (DARD), which unlike EPIC, takes into account the dynamicso f the environment. This allows it to achieve better theoretical guarantees in settings where the transition dynamics are known, but naturally decrases its robustness to changes in environment dynamics.

STAndardised Reward Comparison (STARC) [72] is currently the state-of-the-art class of metrics for value function comparison. Alike DARD, STARC takes into account the transition dynamics. The following definitions from the STARC paper explain this class of metrics:

Definition 6 (Canonicalisation function) *A function $f : \mathbb{R} \to \mathbb{R}$ is a canonicalisation function if f is linear, $f(R)$ and R only differ by potential shaping and S'-redistribution [73] for all $\mathcal{R}$, and if for all $\mathcal{R}_A, \mathcal{R}_B \in \mathcal{R}$, $f(\mathcal{R}_A) = f(\mathcal{R}_B)$ iff $\mathcal{R}_A$ and $\mathcal{R}_B$ only differ by potential shaping and S'-redistribution.*

Definition 7 (Admissible metric) *An admissible metric $m : \mathcal{R} \times \mathcal{R} \to \mathbb{R}$ is a metric for which there exists a norm p and two positive constants u, l such that $l \cdot p(x,y) \leq m(x,y) \leq u \cdot p(x,y)$ for all $x, y \in \mathbb{R}$.*

Definition 8 (STARC metric) *A distance between two reward functions is a STARC metric if there a canonicalisation function f, a function h that is a norm on $Im(f)$, and a metric m that is admissible on $Im(s)$ such that $D_{STARC}(\mathcal{R}_A,\mathcal{R}_B) = m(s(\mathcal{R}_A,\mathcal{R}_B))$, with $s(\mathcal{R}_A) = f(R)/h(f(R))$ if $h(f(R)) \neq 0$, and $s(R) = f(R)$ otherwise.*

STARC metrics all standardise the reward in various ways (in particular eliminating equivalences originating from potential shaping and S'-redistribution, as neither affects the policy ordering) to collapse certain equivalences between reward functions, with the final calculation step calculating the distance in a space without redundancy. STARC metrics provide very strong theoretical guarantees, inducing an upper and lower bound on the worst-case regret. Furthermore, the authors show that any pseudometric on the reward function space $\mathcal{R}$ with the same theoretical guarantees as STARC metrics must be equivalent to them, thus showing STARC metrics provide a solid answer to the problem of measuring reward function similarity.

2.3 Trajectory-Space Metrics

We now introduce two possible metrics to calculate the similarity between two trajectories. These are considered as possible loss functions for our method.

2.3.1 Maximum Mean Discrepancy (MMD)

Maximum Mean Discrepancy is often used as the test statistic in kernel-based two-sample test [74], aiming to detect whether two distributions are the same. It has been used as the loss function when training generative models [75, 76].

MMD considers the distance between distributions as the distance between the mean embeddings of features, for a choice of kernel. The feature space is determined by the chosen kernel, and thus MMD can be computed for different kernels. If the features induce a universal RKHS, the MMD is asymptotically 0 if and only if $P = Q$ [74]. Good choices of kernel, particularly for high-dimensional settings, have been previously studied [77]. However, there is yet to be work on how such theoretical findings can be translated into a practical setting.

We now define MMD. For a kernel-defined feature space $\phi : \mathcal{X} \to \mathcal{H}$, where $\mathcal{H}$ is a Hilbert space, we have that the MMD between two distributions P and Q is

$$MMD^2(P,Q) = \left|\left|\mathbb{E}_{X\sim P}\left[\phi(X)\right] - \mathbb{E}_{Y\sim Q}\left[\phi(Y)\right]\right|\right|^2_{\mathcal{H}} \tag{22}$$

$$= \left\langle \mathbb{E}_{X\sim P}\left[\phi(X)\right], \mathbb{E}_{X'\sim P}\left[\phi(X')\right] \right\rangle_{\mathcal{H}} + \left\langle \mathbb{E}_{Y\sim Q}\left[\phi(Y)\right], \mathbb{E}_{Y'\sim Q}\left[\phi(Y')\right] \right\rangle_{\mathcal{H}}$$

$$- 2\left\langle \mathbb{E}_{X\sim P}\left[\phi(X)\right], \mathbb{E}_{Y\sim Q}\left[\phi(Y)\right] \right\rangle_{\mathcal{H}} \tag{23}$$

$$= \mathbb{E}_{X,X'\sim P}\langle\phi(X),\phi(X')\rangle_{\mathcal{H}} + \mathbb{E}_{Y,Y'\sim Q}\langle\phi(Y),\phi(Y')\rangle_{\mathcal{H}} - 2\mathbb{E}_{X\sim P,Y\sim Q}\langle\phi(X),\phi(Y)\rangle_{\mathcal{H}} \tag{24}$$

$$= \mathbb{E}_{X,X'\sim P}[k(X,X')] + \mathbb{E}_{Y,Y'\sim Q}[k(Y,Y')] - 2\mathbb{E}_{X\sim P,Y\sim Q}[k(X,Y)] \tag{25}$$

When considering finite data settings, equation (25) can be empirically estimated via

$$MMD^2(X,Y) = \frac{1}{m(m-1)}\sum_i\sum_{j\neq i}k(\mathbf{x}_i,\mathbf{x}_j) - \frac{2}{m^2}\sum_i\sum_j k(\mathbf{x}_i,\mathbf{y}_j) + \frac{1}{m(m-1)}\sum_i\sum_{j\neq i}k(\mathbf{y}_i,\mathbf{y}_j) \tag{26}$$

for dataset of size m for X and size n for Y. MMD has been shown to provide more stable training than the adversarial GAN loss [75, 76]. However, the sample quality is also shown to be generally worse. For this reason, Li et al. [76] calculates the MMD on a latent generative space, and then places the latent state through a decoder to obtain the desired output. Levine et al. [78] use MMD alongside policy gradients to guide policy search. In particular, this work considers the MMD between the distributions of state-action pairs.

2.3.2 GAN-based Discriminator

GANs [18] were introduced as a new type of generative models, where a generative model G is trained alongside a discriminative model D. The models are trained in a minimax two-player game setting: D learns to estimate the probability that a sample came from the training data rather than the model's distribution, whereas the generator G is trained to maximise the probability of D making a mistake in its discriminative task.

In the original setting, both D and G are chosen to be multi-layer perceptrons, the generator G takes noise and aims to output a valid sample, and the discriminator D outputs the probability that a sample came from the data distribution rather than the generative model. Taking the prior on the input noise $p_{\boldsymbol{z}}(\boldsymbol{z})$, D is trained to maximise the probability of assigning the correct label to both training examples and samples from G, whereas G is trained to minimize $\log(1 - D(G(\boldsymbol{z})))$, which intuitively means it fools the discriminator. Thus, D and G take the role of two players in a minimax game with value function $V(G, D)$, solving the problem

$$\min_G \max_D V(G, D) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})} [\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})} [\log(1 - D(G(\boldsymbol{z})))] \tag{27}$$

In practice, this is solved by optimising D and G in alternate fashion. Importantly, optimising D to completion without updating G results in overfitting, and thus typical training consists of alternating between k (usually $= 2$) steps of optimising D for each step of optimising G.

This adversarial approach has been applied to imitation learning , as part of the previously introduced GAIL [9]. Generative moment matching networks (GMMNs) [76] combine GANs with Maximum Mean Discrepancy In this setting, the discriminator is replaced by a two-sample MMD test. Thus, the generator G is trained to minimise the MMD distance. Further work has extended on this idea by considering adversarially learnt kernels [79]. This idea, entitled MMD-GAN, also aims to optimise two networks G and D in a minimax fashion, but the objective is different. Whereas in GANs the discrimator D is a binary classifier trained to distinguish between two distributions, in MMD-GANs the discriminator D is an adversarially learnt kernel. This method shows better sample efficiency and similar performance to GANs.

2.4 Diffusion Models

Diffusion models can broadly be classified into Diffusion probabilistic models (DPMs) and Score-Based generative models. While our method uses the former, initial idealizations of this project considered the use of the latter, and thus a small review of score-based methods is shown in Appendix E.2. Regardless, note we will show throughout this section that these two classes are often equivalent.

Diffusion probabilistic models (DPMs) were first introduced in 2015 as generative models, using ideas from non-equilibrium thermodynamics [26]. The general idea accounts to using a Markov chain to slowly corrupt data x at each diffusion step t until it gets converted into a target distribution (usually isotropic Gaussian noise). Learning then accounts to learning the reverse diffusion process, that transforms the target Gaussian noise $x^{(T)}$ into the original data $x^{(0)}$ distribution, via variational inference. This model can fit to any data distribution, with its training still remaining tractable.

Labelling the data distribution as $q(x^{(0)})$, the forward process joint distribution can be expressed as

$$q(x^{(0:T)}) = q(x^{(0)}) \prod_{t=1}^{T} q(x^{(t)}|x^{(t-1)}) \tag{28}$$

where $q(x^{(t)}|x^{(t-1)})$ is typically set as a Gaussian distribution. In a similar manner, the reverse process can be modelled as

$$p(x^{(0:T)}) = p(x^{(T)}) \prod_{t=1}^{T} p(x^{(t-1)}|x^{(t)}) \tag{29}$$

If the variance of the forward process $q(x^{(t)}|x^{(t-1)})$ is sufficiently small, the true reverse process $q(x^{(t-1)}|x^{(t)})$ will have the same functional form [80]. This means that learning amounts to learning the means and covariances of the Gaussian reverse process model $p(x^{(t-1)}|x^{(t)})$.

While the likelihood of the data under the reverse model $p(x^{(0)})$ is intractable, the authors derive a lower bound using cues from annealed importance sampling. In particular, the log likelihood of the generative model under the original data distribution (from here onwards denoted by ℓ) can be bounded by:

$$\ell = \int q(x^{(0)}) \log p(x^{(0)}) dx^{(0)} \geq \int q(x^{(0:T)}) \log \left[p(x^{(0)}) \prod_{t=1}^{T} \frac{p(x^{(t-1)}|x^{(t)})}{q(x^{(t)}|x^{(t-1)})} \right] dx^{(0:T)} \tag{30}$$

$$= -\sum_{t=2}^{T} \int q(x^{(0)}, x^{(t)}) \times D_{KL} \left(q(x^{(t-1)}|x^{(t)}, x^{(0)}) || p(x^{(t-1)}|x^{(t)}) \right) dx^{(0)} dx^{(t)}$$
$$+ H_q(X^{(T)}|X^{(0)}) - H_q(X^{(1)}|X^{(0)}) - H_p(X^{(T)}) \tag{31}$$

This is a combination of entropies and KL divergences which can be analytically computed. Learning of the reverse process means and covariances is done by maximising this lower bound.

Beyond their generative capabilities, DPMs also allow for easy multiplication of distributions, which will occur when calculating posteriors. This is because sampling from a new distribution $\tilde{p}(x^{(0)}) \propto p(x^{(0)}) r(x^{(0)})$, resulting from disturbing with some function $r(x^{(0)})$, amounts to perturbing each diffusion step of the DPM trained to approximate $\tilde{p}(x^{(0)})$. For the Gaussian case, and assuming $r(x^{(t)})$ is sufficiently smooth, it is shown [26] that sampling from the disturbed distribution $\tilde{p}(x^{(0)})$ corresponds to a reverse diffusion process with:

$$p(x^{(t-1)}|x^{(t)}) = \mathcal{N} \left(f_\mu(x^{(t)}, t) + f_\Sigma(x^{(t)}, t) \left. \frac{\partial \log r(x^{(t-1)'})}{\partial x^{(t-1)'}} \right|_{x^{(t-1)'} = f_\mu(x^{(t)}, t)}, f_\Sigma(x^{(t)}, t) \right) \tag{32}$$

where f_μ and f_Σ are the learnt means and covariances of the original reverse process. This is the guided-sampling property of DDPMs that the Diffuser [36] and our method rely on.

More recent work [27] aims to perform learning in diffusion models using ideas from variational Bayesian methods, introducing a new diffusion model parameterization which yields Denoising Diffusion Probabilistic Models (DDPMs). In this work, the authors start from a different bound on the log likelihood:

$$\mathbb{E}_q \left[\underbrace{D_{KL}\left(q(x^T|x^{(0)})||p(x^{(T)})\right)}_{L_T} + \sum_{t>1} \underbrace{D_{KL}\left(q(x^{t-1}|x^{(t)}, x^{(0)})||p_\theta(x^{(t-1)}|x^{(t)})\right)}_{L_{t-1}} \underbrace{- \log p_\theta(x^{(0)}|x^{(1)})}_{L_0} \right] \tag{33}$$

also formed by a combination of entropies and KL divergences which can be analytically computed for the Gaussian case. The key lies in a different parameterization of the L_{t-1} term.

Firstly, by setting the forward process variances to constants, we note that q has no learnable parameters, and thus L_T can be disregarded during training. Secondly, L_0 is disregarded initially, and instead learnt using a separate decoder. Approximating the reverse process with a diagonal Gaussian such that $p_\theta(x^{(t-1)}|x^{(t)}) = \mathcal{N}\left(\mu_\theta(x^{(t)}, t), \sigma_t^2 I\right)$, it follows that:

$$L_{t-1} = \mathbb{E}_q \left[\frac{1}{2\sigma_t^2} ||\tilde{\mu}_t(x^{(t)}, x^{(0)}) - \mu_\theta(x^{(t)}, t)||^2 \right] + C \tag{34}$$

However, using the parameterization $x^{(t)}(x^{(0)}, \epsilon) = \sqrt{\bar{\alpha}_t} x^{(0)} + \epsilon\sqrt{1 - \bar{\alpha}_t}$ for $\epsilon \sim \mathcal{N}(0, I)$, the objective L_{t-1} can be expressed as:

$$L_{t-1} = \mathbb{E}_{x^{(0)},\epsilon} \left[\frac{\beta_t^2}{2\sigma_t^2 \alpha_t (1 - \bar{\alpha}_t)} ||\epsilon - \epsilon_\theta\left(\sqrt{\bar{\alpha}_t} x^{(0)} + \sqrt{1 - \bar{\alpha}_t}\epsilon, t\right)||^2 \right] \tag{35}$$

which is similar to denoising score matching over multiple noise scales [81]. Thus, instead of creating a model that predicts $\tilde{\mu}_t$, the forward process posterior mean, one can instead create a model that predicts ϵ from $x^{(t)}$ and t. Training amounts to predicting the amount of noise that has been added to the current input $x^{(t)}$. It is thus intuitive that most DDPMs are chosen to have denoising architectures, such as U-Nets [82], which learn to find the noise in an input. Sampling from a trained diffusion model then amounts to computing:

$$x^{(t-1)} = \frac{1}{\sqrt{\alpha_t}} \left(x^{(t)} - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right) + \sigma_t z$$

with $z \sim \mathcal{N}(0, I)$. This expression is similar to the typical Langevin Dynamics update:

$$x_{t+1} = x_t + \frac{\delta}{2} \nabla_x \log p(x_t) + \sqrt{\delta}\epsilon_t$$

with $\epsilon_t \sim \mathcal{N}(0, I)$. Thus, $\epsilon_\theta(x_t, t)$ plays the same role as the score function $\nabla_x \log p(x_t)$.

Despite the theoretical background of the objective (35), it was found that a simpler objective:

$$L_{\text{simple}} = \mathbb{E}_{x^{(0)},\epsilon} \left[||\epsilon - \epsilon_\theta\left(\sqrt{\bar{\alpha}_t} x^{(0)} + \sqrt{1 - \bar{\alpha}_t}\epsilon, t\right)||^2 \right] \tag{36}$$

without the weighting resulted in better empirical sample quality.

Algorithms 1 and 2 show pseudo-code for the training and post-training sampling of diffusion models. Line 8 of Algorithm 2 can be adapted according to (32) for guided sampling.

Algorithm 1: DDPM - Training

```
  Input: Dataset $q(\mathbf{x}^0)$, N Training Epochs
1 for epoch $i \leftarrow 0$ to $N$ do
2     $\mathbf{x}^0 \sim q(\mathbf{x}^0)$
3     $t \sim \mathcal{U}(\mathbf{0}, \mathbf{I})$
4     $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5     Take opt. step in direction:
6     $\nabla_\theta ||\epsilon - \epsilon_\theta \left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}, t\right) ||_2^2$
7 end
```

Algorithm 2: DDPM - Sampling

```
1 $\mathbf{x}^T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2 for $t \leftarrow T, ..., 1$ do
3     if $t > 1$ then
4         $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5     else
6         $\mathbf{z} = 0$
7     end
8     $\mathbf{x}^{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}^t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(\mathbf{x}^t, t)\right) + \sigma_t \mathbf{z}$
9 end
  Result: $\mathbf{x}^0$
```

2.5 Diffusion Models in RL

Diffusion models, in particular DDPMs, have emerged in recent years as a valuable class of generative models. Applications of diffusion models span fields as distinct as natural language processing [29, 30], drug discovery [31, 32], and computer vision [27, 33, 34]. Naturally, diffusion models have also been applied to the sequential decision-making tasks, and in particular to offline decision making. As it turns out, there are multiple challenges in RL that can be partly addressed by diffusion models.

Firstly, offline RL [83, 84] has been used to counteract the sample efficiency of its online counterpart. However, off-policy learning in an offline setting is known to suffer from poor extrapolation [83]. Existing approaches have opted to employ conservative value estimates for OOD samples [84], or to regularise the policy to be close to the policy used to generate the dataset [85]. However, such approaches tend to restrict the expressiveness of the policies obtained [86]. Diffusion models, which are capable of representing any normalisable distribution [87], hold the potential to improve the expressiveness of such policies.

Secondly, experience replay [2] has been extensively used to counteract RL's poor sample efficiency. This has often been combined with data augmentation, so as to improve the amount of data fed into the model. Rudimentary augmentation techniques such as random cropping have previously been applied to vision-based RL [88], whereas generative models such as VAEs and GANs have been used to create synthetic RL data [89]. With their improved sample quality [27], diffusion models have the potential to generate richer datasets, providing a more accurate representation of the true data source.

2.5.1 Diffuser

The first instance of the use of diffusion models for RL is the Diffuser [36], a trajectory-level DDPM. The Diffuser can be trained in an offline manner using a dataset of reward-agnostic trajectories. The Diffuser learns to predict all timesteps of a plan instead of the typical autoregressive planning found in most RL approaches. Its training mirrors Algorithm

1, minimising the simplified objective

$$L_{\text{simple}} = \mathbb{E}_{\boldsymbol{\tau}^{(0)},\epsilon,i}\left[||\epsilon - \epsilon_\theta\left(\boldsymbol{\tau}^i, i\right)||_2^2\right] \tag{37}$$

where i is the diffusion step and $\boldsymbol{\tau}^i$ is the noise-corrupted trajectory obtained from adding noise ϵ to $\boldsymbol{\tau}^0$. Importantly, the trajectories are now not only representative of the environment dynamics, but also of the policy used to generate the dataset, since:

$$\boldsymbol{\tau} = \begin{bmatrix} s_0 & s_1 \dots s_T \\ a_0 & a_1 \dots a_T \end{bmatrix} \tag{38}$$

The Diffuser's biggest strength lies in the strong coupling between prediction and planning, and how the former can be executed via diffusion model guided sampling. Recalling Section 2.4, and specifically equation (32), it is straightforward to perform guided sampling on a previously trained diffusion model according to some likelihood $r(\boldsymbol{\tau})$.

To adapt this idea to a RL setting, we refer to the control-as-inference graphical model [90]. Let O_t be a binary RV denoting whether timestep t of a trajectory is optimal, and let us set the function $r(\boldsymbol{\tau})$ to be the likelihood $p(O_{1:T}|\boldsymbol{\tau})$. Subbing this into equation (32), we obtain the gradient guiding term

$$\left.\frac{\partial \log r(\boldsymbol{\tau}^{(t-1)'})}{\partial \boldsymbol{\tau}^{(t-1)'}}\right|_{\boldsymbol{\tau}^{(t-1)'}=f_\mu(\boldsymbol{\tau}^{(t)},t)} = \left.\frac{\partial \log p(O_{1:T}|\boldsymbol{\tau}^{(t-1)'})}{\partial \boldsymbol{\tau}^{(t-1)'}}\right|_{\boldsymbol{\tau}^{(t-1)'}=f_\mu(\boldsymbol{\tau}^{(t)},t)} \tag{39}$$

$$= \left.\frac{\partial \sum_{t=0}^{T} j(s_t, a_t)}{\partial \boldsymbol{\tau}^{(t-1)'}}\right|_{\boldsymbol{\tau}^{(t-1)'}=f_\mu(\boldsymbol{\tau}^{(t)},t)} \tag{40}$$

$$= \nabla_{\boldsymbol{\tau}} J\left(f_\mu(\boldsymbol{\tau}^{(t)}, t)\right) \tag{41}$$

where $j(s_t, a_t)$ is the reward for a (state,action) tuple. Taking into account we simplify the notation in the last step to emphasise that this is simply the gradient, with respect to the trajectory, of a reward model J. Thus, one simply requires the gradient of the reward model J to guide the sampling process to achieve optimal trajectories. The pseudo-code for guided planning using the Diffuser is shown in Algorithm 3. Note we change notation from the cumbersome $f_\mu(\boldsymbol{\tau}^{(t)}, t)$ and $f_\Sigma(\boldsymbol{\tau}^{(t)}, t)$ to the simpler $\mu_\theta(\boldsymbol{\tau}^{(t)}, t)$ and Σ^i.

Algorithm 3: Diffuser - Guided Planning

```
Input: Trained Diffuser $\mu_\theta$, Reward Model $J$, $N$ diffusion steps, Initial state $s$,
       Learning rate $\alpha$;
1 Initialize $\boldsymbol{\tau}^N \sim \mathcal{N}(0, I)$
2 for $i = N$ to 1 do
3     $\mu \leftarrow \mu_\theta(\boldsymbol{\tau}^i, i)$ ;                    /* Diffuser Parameters */
4     $\boldsymbol{\tau}^{i-1} \sim \mathcal{N}(\mu + \alpha\Sigma\nabla_{\boldsymbol{\tau}} J(\mu), \Sigma^i)$ ; /* Sampling guided by gradient of return */
5     $\boldsymbol{\tau}_0^{i-1} \leftarrow s$ ;          /* Condition each diffusion step on initial state */
6 end
```

As previously mentioned, the Diffuser plans entire trajectories, instead of the typical single-step autoregressive predictions of RL algorithms. Due to this difference, the Diffuser does not suffer from the compouding rollout errors commonly found in typical RL models [91], and thus allows for larger horizons without sacrificing rollout quality.

On a different note, the connection between sampling from the Diffuser and planning with it (these simply differ due to the addition of a reward model gradient which acts as a guide for the latter) means that the Diffuser is particularly well suited to a multi-task setting. In such tasks, the Diffuser model can learn an environment model from some particular task, and then at test time the model can plan for novel tasks using the same base diffusion model, simply by specifying different reward models for each task.

2.5.2 Other Approaches

Subsequent works have extended on the Diffuser framework. LatentDiffuser [92] learns to encode trajectories into a latent space, and then performs diffusion-based planning on this latent space, adding a separate decoder to recover trajectories. The use of a more compact planning space resulted in better performance in high-dimensional tasks. SafeDiffuser [93] incorporates invariance into the diffusion process, to ensure safe planning for safety-critical applications. AdaptDiffuser [94] utilises the Diffuser's generated samples to augment its training dataset, creating richer datasets which allow for better generalization across a variety of both seen and unseen tasks. The Decision Diffuser [37] frames the problem as one of conditional generative modelling, modeling the policy as a return-conditional diffusion model.

In the case of the Diffuser, the diffusion model takes the role of the planner, outputting trajectories that are both realistic under the learnt model, but also likely under the likelihood function $r(\boldsymbol{\tau})$. This approach is akin to model-based RL, as the environment dynamics are learnt during the diffusion model training. However, diffusion models have also been extensively used in model-free approaches, with the diffusion model being used as the policy model. The high expressiveness of diffusion models means this method circumvents the issues with low expressiveness of RL policies previously mentioned.

Diffusion-QL [86] employs a diffusion model that simply takes the current state s_t as input, and learns to output an optimal action a_t. The training is guided via a jointly-learnt Q-value function, guiding the policy samples towards the optimal policy. That is, the diffusion model learns a noise predictor $\epsilon_\theta(a^i, i, s_i)$ for the state s_i at time i. The guiding via Q values amounts to including a weighted Q maximization term in the diffusion training loss.

This approach does not allow for guided sampling with new objectives post-training, as the Diffuser does. Thus, recent work [95] instead constructs the guided diffusion policy as

$$\pi_\theta(a|s) \propto \mu_\theta(a|s)\exp(Q(s,a)), \tag{42}$$

where $\mu_\theta(a|s)$ represents the mean output of the diffusion model. This formula allows for the training of a base diffusion model, followed by guided sampling with Q function guides. This idea has since been extended to allow for sampling from more general, potentially unnormalised, energy-guided distributions [96].

2.6 Inverse RL with Diffusion Models

To the best of our knowledge, there is only one work to date on learning a reward function using diffusion models. Nutti et al [38] introduce a method to learn a relative reward function from two Diffusers. This method works for any two diffusion models, regardless of whether they are unguided, or guided with either classifier-guided or classifier-free guidance. Nutti et al. introduce the notion of a relative reward function between two diffusion models:

Definition 9 (ϵ-Relative Reward Function) *Let $\boldsymbol{s}^1_\theta(\cdot, t)$ and $\boldsymbol{s}^2_\phi(\cdot, t)$ be the score functions of two diffusion models, and $t \in (0, T)$ the diffusion step index. Let $\boldsymbol{h}_t$ denote the optimal relative reward gradient [38] of $\boldsymbol{s}^1_\theta(\cdot, t)$ and $\boldsymbol{s}^2_\phi(\cdot, t)$ at time t. Then, for $\epsilon > 0$, the ϵ-relative reward function $\rho : \mathbb{R}^n \times [0, T]$ is such that:*

$$\int_{\mathbb{R}^n} ||\nabla_{\boldsymbol{x}} \rho(\boldsymbol{x}_t, t) - \boldsymbol{h}_t(\boldsymbol{x})||_2^2 \leq \epsilon \ , \quad \forall t \in (0, T] \tag{43}$$

To approximate Definition 9, a neural network parameterised approximation ρ_ψ is learnt according to the training objective:

$$L(\psi) = \mathbb{E}_{t \sim \mathcal{U}[0,T], \boldsymbol{x}_t \sim p_t} \left[||\nabla_{\boldsymbol{x}} \rho_\psi(\boldsymbol{x}_t, t) - (\boldsymbol{s}^2_\phi(\cdot, t) - \boldsymbol{s}^1_\theta(\cdot, t))||_2^2 \right] \tag{44}$$

where p_t is the marginal at diffusion step t of the forward noising diffusion process. A common use setting consists of having two diffusion models, a non-expert one and an expert one, and using this algorithm to align/steer the non-expert diffusion model towards the expert model.

This method succeeds in extracting relative reward functions on the Maze2D [40] environments. Furthermore, it is shown to be able to align behaviour between diffusion models trained on data of different quality, as the authors show that the learnt relative reward is able to (partly) steer the behavioural performance of a non-expert diffusion model towards an expert model.

Our method, presented in the next section, differs from this work as it aims to extract a reward function from a single diffusion model, instead of two. In doing so, we aim to obtain an actual reward function, as opposed to a relative reward [38]. This makes our method more sample efficient, as we only use expert trajectories to learn the reward model, instead of training an entire diffusion model on them. Finally, we do not assume diffusion step t to be an input to the reward model. Thus our guide is not diffusion step-specific, as opposed to Nutti et al.'s [38]. The effects of this choice will be discussed in the experimental sections of this report.

3 Methodology

In Section 2 we started by introducing the RL (Section 2.1) and IRL (Section 2.2) problems. We introduced metrics with which we can compute the similarity between two reward functions (Section 2.2.4), which will be used to evaluate our method's performance. We also introduced a class of metrics to compute similarity between agent trajectories (Section 2.3), which will be used as loss functions in our algorithm. We concluded by introducing diffusion models (Section 2.4), and presenting a survey of their applications in RL (Section 2.5) and IRL (Section 2.6). We are now in a position to introduce our proposed method.

3.1 Formal Problem Specification

We start by formally specifying the problem of IRL in a trajectory-level diffusion model.

Let us consider a base diffusion model $\boldsymbol{\epsilon}_\theta(\boldsymbol{\tau}^t, t)$ which has been trained on non-expert trajectories from dataset $\mathcal{D}_b$. Let $p_b(\boldsymbol{\tau}^0)$ be the base Diffuser's generative distribution, learnt according to the empirical distribution $q_b(\boldsymbol{\tau})$of dataset $\mathcal{D}_b$. Assume we have an expert trajectory dataset $\mathcal{D}_e$ for trajectories in the same state-action space as those in $\mathcal{D}_b$. Let $q_e(\boldsymbol{\tau})$ be the empirical (expert) trajectory distribution according to $\mathcal{D}_e$. The IRL problem amounts to finding the reward function $R(\boldsymbol{\tau}^t)$ such that:

$$q_e(\boldsymbol{\tau}) = p_e(\boldsymbol{\tau}^0) = \frac{1}{Z^0} p_b(\boldsymbol{\tau}^0) R(\boldsymbol{\tau}^0) \tag{45}$$

$$p_e(\boldsymbol{\tau}^1) = \frac{1}{Z^1} p_b(\boldsymbol{\tau}^1) R(\boldsymbol{\tau}^1) \tag{46}$$

$$\vdots$$

$$p_e(\boldsymbol{\tau}^T) = \frac{1}{Z^T} p_b(\boldsymbol{\tau}^T) R(\boldsymbol{\tau}^T) \tag{47}$$

where t is the diffusion step index, and Z^t is the normalization constant for each intermediate distribution. That is, given the base Diffuser, the IRL problem consists of finding the reward function $R(\boldsymbol{\tau}^t)$ that can guide the guided-diffusion output distribution towards the empirical expert trajectory distribution $q_e(\boldsymbol{\tau})$.

3.2 Learning Rewards for Classifier-Guided Diffusion

We now introduce our method to learn a reward function using a base diffusion model trained to learn environment's dynamics, samples of some expert behaviour, and a metric in trajectory space.

Ho et al. [27] shows that for a learnt diffusion model $\boldsymbol{\epsilon}_\theta(\boldsymbol{\tau}_t, t)$, one can obtain samples via the backward diffusion (denoising) process:

$$\boldsymbol{\tau}^{t-1} = \frac{1}{\alpha^t}\left[\boldsymbol{\tau}^t - \frac{1-\alpha^t}{\sqrt{1-\bar{\alpha}^t}} \epsilon_\theta(\boldsymbol{\tau}^t, t)\right] + \sigma^t \boldsymbol{z} \tag{48}$$

where t denotes the diffusion step. α^t,$\bar{\alpha}^t$ and σ^t are diffusion parameters which can either be fixed or learnt. They are not relevant to our derivation. As presented in Section 3.1, we aim to obtain a diffusion model described by a posterior distribution $p_e(\boldsymbol{\tau}^t)$. Sohl et al. [26] showed that, given a likelihood $p_\phi(y|\boldsymbol{\tau}^t)$, we can guide a (prior) diffusion model's distribution towards the respective posterior distribution by performing each step of the reverse diffusion process according to:

$$\boldsymbol{\tau}^{t-1} = \frac{1}{\alpha^t}\left[\boldsymbol{\tau}^t - \frac{1-\alpha^t}{\sqrt{1-\bar{\alpha}^t}} \boldsymbol{\epsilon}_\theta(\boldsymbol{\tau}^t, t)\right] + s\Sigma\nabla_{\boldsymbol{\tau}^t} \log p_\phi(y|\boldsymbol{\tau}^t) + \sigma^t \boldsymbol{z} \tag{49}$$

where s is a learning rate-like parameter. This is an equivalent formulation of the previously encountered equation (32). Thus, the system of equations (45), (46) and (47) can all be solved by independently shifting each diffusion step.

The original derivation was for a non RL related likelihood $p_\phi(y|\boldsymbol{\tau}^t)$. However, this likelihood can be adapted to a RL setting. Referring to the control-as-inference graphical model [90], one can express the likelihood of a state-action pair being optimal as

$$p(O_t = 1) = \exp(r(s_t, a_t)) \tag{50}$$

Under this assumption, [36] shows that:

$$\begin{aligned} \nabla_{\boldsymbol{\tau}} \log p_\phi(O_{1:N}|\boldsymbol{\tau}) &= \nabla_{\boldsymbol{\tau}} \sum_{t=0}^{N} r(s_t, a_t) && (51) \\ &= \nabla_{\boldsymbol{\tau}} R(\boldsymbol{\tau}) && (52) \end{aligned}$$

where $R(\boldsymbol{\tau})$ is the return of the entire trajectory $\boldsymbol{\tau}$. Thus, the guided reverse process for an output trajectory $\boldsymbol{\tau}_0$ can be obtained by T reverse diffusion steps, each according to:

$$\boldsymbol{\tau}^{t-1} = \frac{1}{\alpha^t}\left[\boldsymbol{\tau}^t - \frac{1-\alpha^t}{\sqrt{1-\bar{\alpha}^t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{\tau}^t, t)\right] + s\Sigma\nabla_{\boldsymbol{\tau}^t} R(\boldsymbol{\tau}^t) + \sigma^t \boldsymbol{z} \tag{53}$$

Our algorithm proposes parameterising the return $R(\boldsymbol{\tau})$ with a neural network $R_\phi(\boldsymbol{\tau})$. We first train the base diffusion model $\epsilon_\theta(\boldsymbol{\tau}^t, t)$ on a dataset of reward-agnostic trajectories to learn the dynamics of the environment, and initiate the trajectory return model $R_\phi(\boldsymbol{\tau})$. Then, our algorithm consists of repeatedly performing guided sampling (with fixed first state according to an expert trajectory) with the current reward function, calculating a loss between the sample obtained and an expert trajectory, and backpropagating this error so as to update the parameters of the reward network. A diagram explaining this process is shown in Figure 2.

Our algorithm thus learns the guide necessary to guide the base diffuser distribution towards the expert trajectory distribution. The pseudo-code for our method is shown in Algorithm 4. Appendix A contains a more detailed derivation of classifier guidance in the Diffuser framework, as well as a derivation of the gradient of the MSE loss with respect to the guide's parameters.

Algorithm 4: IRL with Diffuser

```
   Input: Base (Trained) Diffuser $\boldsymbol{\epsilon}_\theta(\boldsymbol{\tau}^t, t)$ with $N$ diffusion steps, Expert Trajectories
          Dataset $\mathcal{D}_e$, Number of Steps $K$, Distance metric $M$, Initialised Reward
          model/guide $R_\phi$
1  for epoch $i \leftarrow 0$ to $N$ do
2      for $\boldsymbol{\tau}_e$ in $\mathcal{D}_e$ do
3          $\boldsymbol{\tau}^N \sim \mathcal{N}(0, I)$ ;                      /* Sample diffusion start */
4          $\tau_0^N \leftarrow \boldsymbol{\tau}^e$ ;              /* Fix first state according to target $\boldsymbol{\tau}_e$ */
5          $\boldsymbol{\tau}^0 \leftarrow \boldsymbol{\epsilon}_\theta(\boldsymbol{\tau}_e)$ ;        /* Reverse guided diffusion, fix first state */
6          $\boldsymbol{\tau}^0 \leftarrow \boldsymbol{\tau}^0[: K]$ ;        /* Get first K steps of each plan in batch */
7          $L = M(\boldsymbol{\tau}^0, \boldsymbol{\tau}^e)$ ;             /* Calculate Loss according to metric */
8          opt.step() ;                  /* Backprop grads and do opt step on $\phi$ */
9      end
10 end
   Result: Reward Model $R_\phi$
```

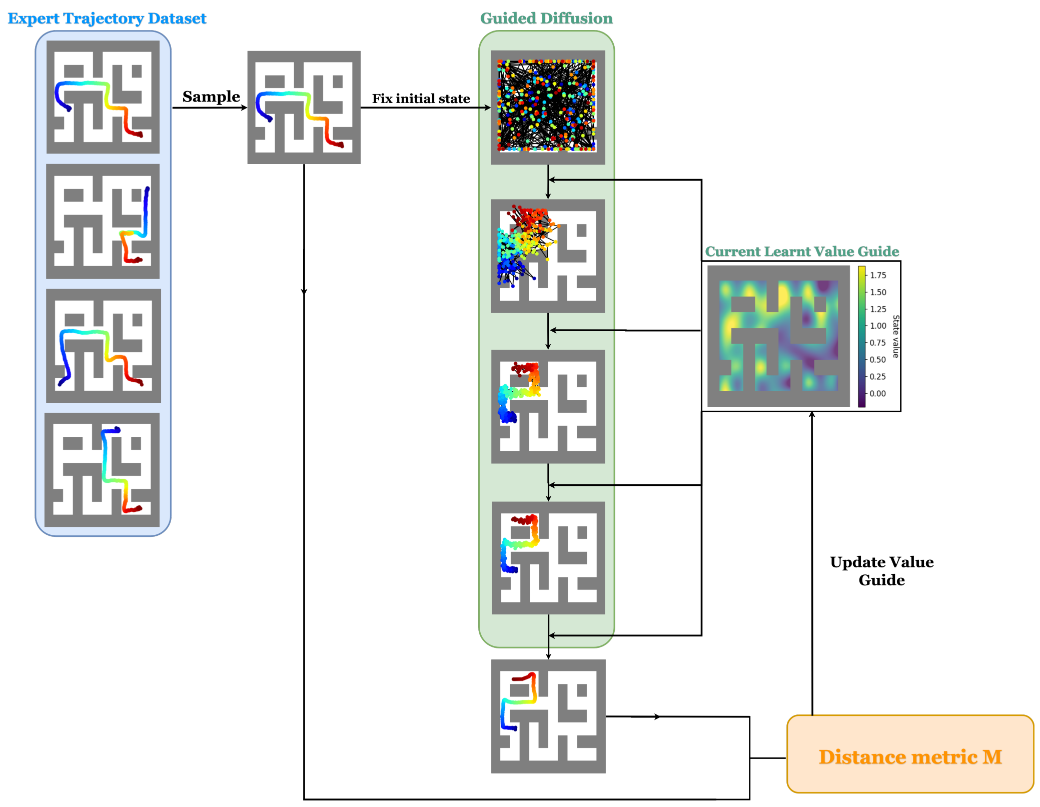

Figure 2: Diagram explaining the workings of a one sample version of our method. A trajectory is sampled from the expert trajectory dataset. This trajectory fixes the first state of the diffusion process. The reverse diffusion process samples a trajectory with fixed current state. A trajectory-distance metric is calculated between the sampled trajectory and the expert trajectory, with the value network's parameters being updated accordingly.

3.3 Practical Considerations

Distinction between plans and rollouts. A plan is simply the output trajectory of the diffusion model, and it reflects the model's long term planning capabilities (and thus indirectly its knowledge of the environment dynamics, as the agent never actually interacts with the environment when planning). A rollout is defined as the trajectory obtained from iteratively planning from the current state, taking the first action according to this plan in the environment, moving to a new state according to the true environment dynamics, and planning again from this state (and so on, until a certain rollout length is reached). Plans are significantly noisier than rollouts, and as previously mentioned incorporate errors from the Diffuser's learnt environment model. Thus, expert trajectories for IRL are ideally rollouts, so they reflect the true environment dynamics. As the output of the diffusion model is a plan, it was deemed fairer to the algorithm to compare only the initial portions of its plan to the expert rollout (this is because plans are notoriously noisier and of poorer quality after the first 10 to 20 steps when compared to rollouts). The number of steps K considered is chosen depending on the environment.

Due to the distinction highlighted above, our method assumes a pre-processing step that creates a dataset of K-length sections of expert trajectories. This can be obtained by splitting longer trajectories into K-length sections using a sliding window approach on longer trajectories. Algorithm 5 describes this approach. Furthermore, when planning in Algorithm 4, we truncate the Diffuser's sampled plan so as to only consider its first K steps.

Algorithm 5: Pre-processing trajectories into K-step trajectories

```
  Input: Dataset of full length expert trajectories $\mathcal{D}_f$, Number of Steps $K$
1 $\mathcal{D}_e \leftarrow \emptyset$ ;                                   /* Initialise dataset */
2 for each expert trajectory $\boldsymbol{\tau}^f$ in $\mathcal{D}_f$ do
3     step $\leftarrow$ 0
4     while step < len($\boldsymbol{\tau}^f$) do
5         $\mathcal{D}_e$.append($\boldsymbol{\tau}^f$[step:step+K]) ;     /* Get dataset of K-step sections */
6     end
7     step $\leftarrow$ step+$K$
8 end
  Result: Expert trajectory dataset $\mathcal{D}_e$
```

Trajectory-space metrics The loss function M is chosen to be a metric in trajectory space, thus representing the difference between two trajectories. As the diffusion process approximates each step of the denoising process as a Gaussian, the mean squared error (MSE) is a natural (albeit naive) choice for the metric M. While we do not present such an analysis in this report, future work should study whether by choosing a MSE loss we are indeed matching the means of the Gaussians in each of the denoising steps.

If the loss function is defined as the mean squared error between a sampled plan $\boldsymbol{\tau}^0$ and an expert trajectory $\boldsymbol{\tau}_e$ such that

$$L = ||\boldsymbol{\tau}_e - \boldsymbol{\tau}^0(\phi)||_2^2 \tag{54}$$

then we have that the gradient of this loss with respect to the reward model's parameters ϕ is:

$$\nabla_\phi L = \nabla_\phi \left|\left| \boldsymbol{\tau}_e - \frac{1}{\sqrt{\alpha_1}}\boldsymbol{\tau}^1 + \frac{1-\alpha^1}{\sqrt{\alpha^1}\sqrt{1-\bar{\alpha}^1}}\boldsymbol{\epsilon}_\theta(\boldsymbol{\tau}^1, 1) + s\Sigma\nabla_{\boldsymbol{\tau}^1} R_\phi(\boldsymbol{\tau}^1) \right|\right|_2^2 \tag{55}$$

$$= 2||\cdot|| \left[\boldsymbol{\tau}_e - \frac{1}{\sqrt{\alpha^1}}\frac{\partial \boldsymbol{\tau}^1}{\partial \phi} + \frac{1-\alpha^1}{\sqrt{\alpha^1}\sqrt{1-\bar{\alpha}^1}}\frac{\partial \boldsymbol{\epsilon}_\theta(\boldsymbol{\tau}^1, 1)}{\partial \phi} + s\Sigma\frac{\partial}{\partial \phi}\frac{\partial}{\partial \boldsymbol{\tau}^1} R_\phi(\boldsymbol{\tau}^1) \right] \tag{56}$$

$$= 2||\cdot|| \left[\boldsymbol{\tau}_e - \frac{1}{\sqrt{\alpha^1}}\frac{\partial \boldsymbol{\tau}^1}{\partial \phi} + \frac{1-\alpha^1}{\sqrt{\alpha^1}\sqrt{1-\bar{\alpha}^1}}\frac{\partial \boldsymbol{\epsilon}_\theta(\boldsymbol{\tau}^1, 1)}{\partial \phi} + s\Sigma\frac{\partial}{\partial \phi}\frac{\partial}{\partial \boldsymbol{\tau}^1} R(\boldsymbol{\tau}^1) \right] \tag{57}$$

Note that this involves a second derivative of the reward model. Furthermore, the recursive nature of the first and second terms indicates how the gradient calculation flows through each diffusion step. Possible parallel work should study whether this is desirable, or whether stopping gradient flow after the first noising step could provide better training stability. Regarding the choice of loss, MSE is a rather naive choice for a distance metric between trajectories, scaling poorly with dimension, and not taking into account the generative nature of a diffusion model. Furthermore, especially in complex environments, the mean between two trajectories might not even be feasible, or desirable. In Section 2.3 we introduce two main classes of possibly more appropriate metrics for trajectory similarity: Maximum Mean Discrepancy (MMD), and discriminators such as those used in GAN setups. We leave a GAN-like discriminator as possible future work, as this direction was not prioritised due to the instabilities in training described in Section 4.2. Since GAN training is known to be unstable [97, 98], we hypothesise that it is likely any instabilities in our reward model training would be exacerbated by the incorporation of an adversarial set-up.

Thus, to consider more complex trajectory distance metrics, we also implement a variation of our method based on a loss function defined as the MMD between the empirical distribution of expert trajectories and the empirical distribution of sampled plans. As discussed in Section 2.3.1, the empirical MMD between two datasets X and Y (with sizes i and j respectively) can be computed, for a choice of kernel $k(x_1, x_2)$, as:

$$MMD^2(X,Y) = \frac{1}{m(m-1)}\sum_i \sum_{j\neq i} k(\mathbf{x}_i, \mathbf{x}_j) - \frac{2}{m^2}\sum_i \sum_j k(\mathbf{x}_i, \mathbf{y}_j) + \frac{1}{m(m-1)}\sum_i \sum_{j\neq i} k(\mathbf{y}_i, \mathbf{y}_j) \tag{58}$$

We consider two choices of kernel, Gaussian and Matern, and study the impact of the choice of trajectory-space metric on the performance of our method. The choice of these two different kernels allows for different assumptions on the smoothness of the trajectories. More specifically, a Gaussian kernel assumes infinite differentiability, whereas the Matern kernel, as it is chosen with a smoothness parameter of 0.5, is identical to an absolute exponential kernel, and thus does not assume the trajectories can be differentiated even once.

Note that despite Algorithm 4 calculating the loss sample-wise, we implemented this algorithm in a way such that it takes data in a batch. While optional for the MSE, this is necessary for the MMD cases, so we create batches of expert trajectories, and batches of sampled plans (one for each starting point of its respective expert trajectory), and calculate the MMD between the two empirical distributions. We would not be able to simply calculate the MMD if each distribution had only 1 sample.

Evaluation metrics. While being classed as a IRL method, our method aims to solve not only the IRL problem (learn a reward function), but also the imitation learning problem (imitate behaviour according to the expert trajectories). Thus, we assess our method's performance on two different axes: behaviour quality and reward function quality. This allows us to (partly) separate errors in learning a value function from errors in learning an optimal policy in the environment. The quality of behaviour is assessed by the return obtained by a trajectory in the environment (this is either available in the environment, or is given by a "true" reward function that we hand craft, depending on the experiment). The quality of the learnt value functions is given by the Episode Return Correlation (ERC) distance, introduced in Section 2.2.4, between the return given by the environment (or by the handcrafted "true" reward function) and the return according to the learnt value function. Note that despite the name, the ERC is defined as a Pearson distance (instead of a Pearson coefficient), and thus a smaller ERC (which can take values in the interval [0, 1]) is associated with more similar value functions. While there are stronger value similarity metrics (as discussed in Section 2.2.4), we did not find viable already existing implementations.

Baselines. We compare our method to three baselines: Behaviour Cloning (BC), GAIL and AIRL (introduced in Sections 2.2.2 and 2.2.3), all using off-the-shelf implementations [3] and trained on the same expert trajectories. BC and GAIL are imitation learning methods, and

[3]`https://imitation.readthedocs.io/en/latest/`

thus we compare the quality of behaviour generated by our method to the behaviour generated by these baselines. As previously described, we assess the quality of behaviour based on the return obtained by a trajectory in the environment. On the other hand, AIRL is closer to our method, in that it learns a reward function and uses it to create imitative behaviour. Thus, for AIRL we compare both the quality of the learnt value function as well as the quality of behaviour. As previously mentioned, the quality of behaviour is assessed via ERC.

3.4 U-Maze environment

We begin by evaluating our method on the U-Maze (seen in Figure 3). More details about this environment can be found in Appendix B. This is a simple environment with low dimensionality, used to provide a proof of concept for our method.

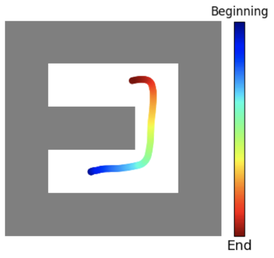

Figure 3: Trajectory of an agent in the U-Maze environment. Maze in white, walls in grey.

3.4.1 Experimental setup

Base Diffuser Training. The base Diffuser was trained on 1566 reward-agnostic trajectories, each of length 300. Example datapoints are shown in Appendix D.1. The planning horizon was chosen as 128, and the number of diffusion steps as 64. Training details are shown in Appendix C.

True Reward. We designed a reward model according to the formula $J_\phi = 5x + 5y$, where x and y are the coordinates in the maze (horizontal and vertical respectively, starting from the top left corner, with $x \in [0, 5]$ and $y \in [0, 5]$ including the maze walls). Thus, our reward model gives the highest reward to the bottom right corner of the maze. Figure 4 shows this reward function.

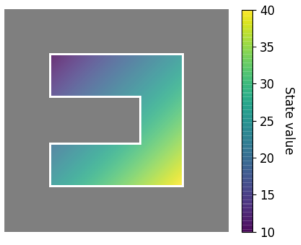

Figure 4: Left: True value function used to generate expert trajectories.

Expert trajectories. The expert trajectories used for reward model learning were obtained by guided sampling on the trained base Diffuser using the true reward function in Figure 4 as the guide. Nine (9) expert trajectories of length 300 were generated by rollouts on the environment, starting from 3 different start points (3 trajectories per start point). Naturally, the generated trajectories move towards the bottom right corner, and do not leave it. Figure 5 shows one of the expert trajectories per starting point.

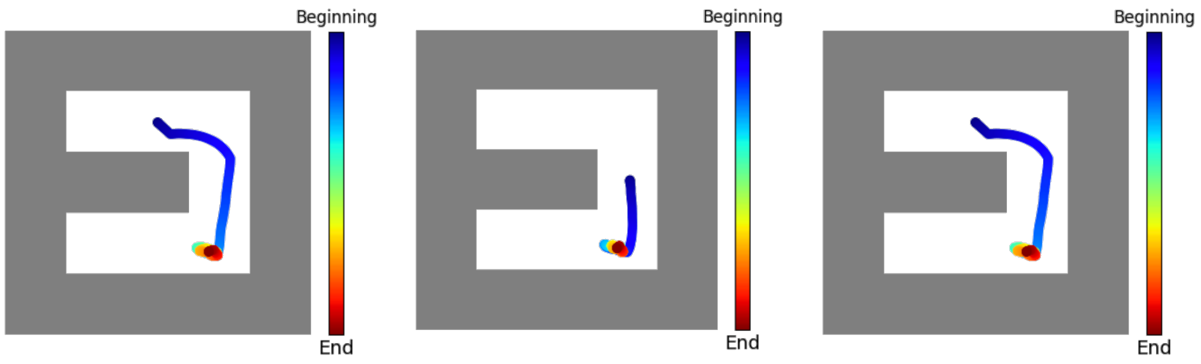

Figure 5: Example expert trajectories used for learning in the U-Maze. Each figure corresponds to a different starting point.

Experimental details. We choose a number of steps $K = 10$, which is deemed to be small enough that plans and rollouts have comparable amounts of noise, but large enough that variations in trajectories result in significantly different outcomes. We consider the distance metric M between a 10-step section of an expert trajectory and the first 10 steps of a sampled plan with start state the same as the expert trajectory's section. For the MSE, each individual sample in a batch gets compared to its respective sample in the expert trajectory batch. For the MMD losses, the MMD between the two batches is calculated. The 9 expert trajectories thus get split into 270 10-step sections. We parameterise the reward model as a 4-layer Multi-Layer Perceptron (MLP). Details of the architectural characteristics of the network, as well as hyperparameter values, are given in Appendix C. This reward model is trained for 100 epochs. The loss function curves are shown in Figure 6.

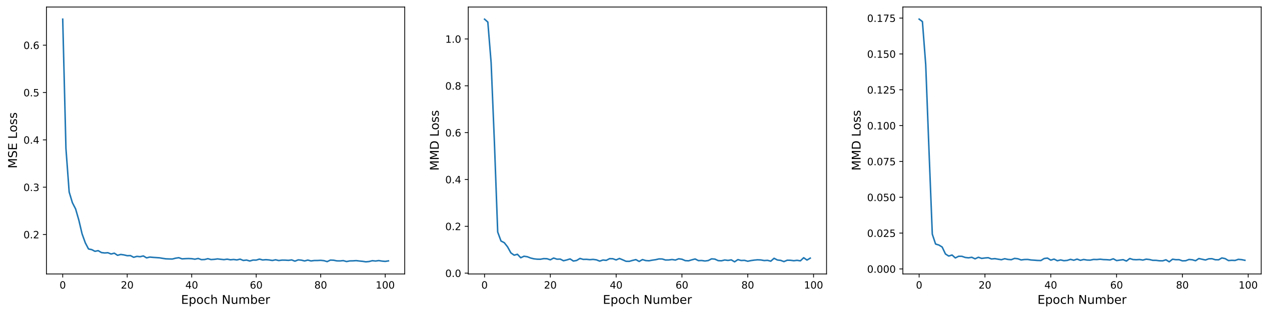

Figure 6: Loss functions for the three different variations of our method as a function of training epoch for the U-Maze training. From left to right: MSE, MMD with Gaussian kernel, MMD with Matern kernel.

Evaluation. We evaluate the quality of the rollouts obtained by guided planning of the base Diffuser with the learnt value function, from 5 different starting points. We measure the quality of each rollout based on its return under the true reward model. To evaluate the similarity between value functions, we randomly sample trajectories from the dataset used to train the base Diffuser (assumed to accurately cover the state space), and calculate the ERC distance (see Section 2.2.4) between the return of these trajectories according to the learnt reward model, and the return according to the true reward model. In Table 1, ”Base Diffuser” represents the trained base Diffuser without any guide and ”Diffuser (True Reward)” represents the trained base Diffuser with the true reward function as a guide.

3.4.2 Results and discussion

Table 1 shows the return and ERC for each method, averaged over 100 rollouts. Note the ERC is naturally only calculated for the methods which learn an explicit value function. Figure 7 shows example output trajectories for the three variations of our method and the three baselines, from 5 different starting points.

Table 1: Reward of guided rollouts and ERC of base dataset (averaged over 100 rollouts) for U-maze training.

Method	Return ↑	ERC ↓
Base Diffuser	2709.87 ±199.81	———
Diffuser (True Reward)	3569.82 ±243.84	———
Diffuser (Learnt MSE Reward)	3578.47 ±237.46	0.689
Diffuser (Learnt MMD-Gauss Reward)	3563.60 ±262.15	0.652
Diffuser (Learnt MMD-Matern Reward)	3557.16 ±247.75	0.457
Behaviour Cloning	3535.96 ±248.75	———
GAIL	3782.45 ±272.66	———
AIRL	3724.23 ±328.94	0.719

Firstly, we observe that as expected the true reward properly guides the base Diffuser towards high-reward behaviour. Importantly, we observe that both GAIL and AIRL outperform the base Diffuser guided by the true reward. While counter-intuitive, this can be attributed to the base Diffuser training: despite using the true guide, the output distribution of the diffusion model is still dependent on the dynamics learnt during the base Diffuser training. Moving directly to the bottom right corner and sticking to that corner (as seen in the expert trajectories in Figure 5) is not behaviour present in the dataset that the base Diffuser was trained in, and thus is out-of-distribution behaviour. The Diffuser with the true guide sticks to the bottom right corner as it learns strong enough of a guide to do so, but it achieves this ”goal” while moving as much as possible in a way that is consistent with the base reward-agnostic dataset (for example, always being significantly away from walls). GAIL and AIRL, on the other hand, are not constrained by these learnt environment dynamics.

All three variations of our method achieve similar returns to the Diffuser with the true reward as a guide. Thus we can conclude our method learns how to imitate behaviour. However, the quality of output behaviour of our method depends both on its learnt environment dynamics and on its learnt guide. The return does not directly tell us that the learnt value function is

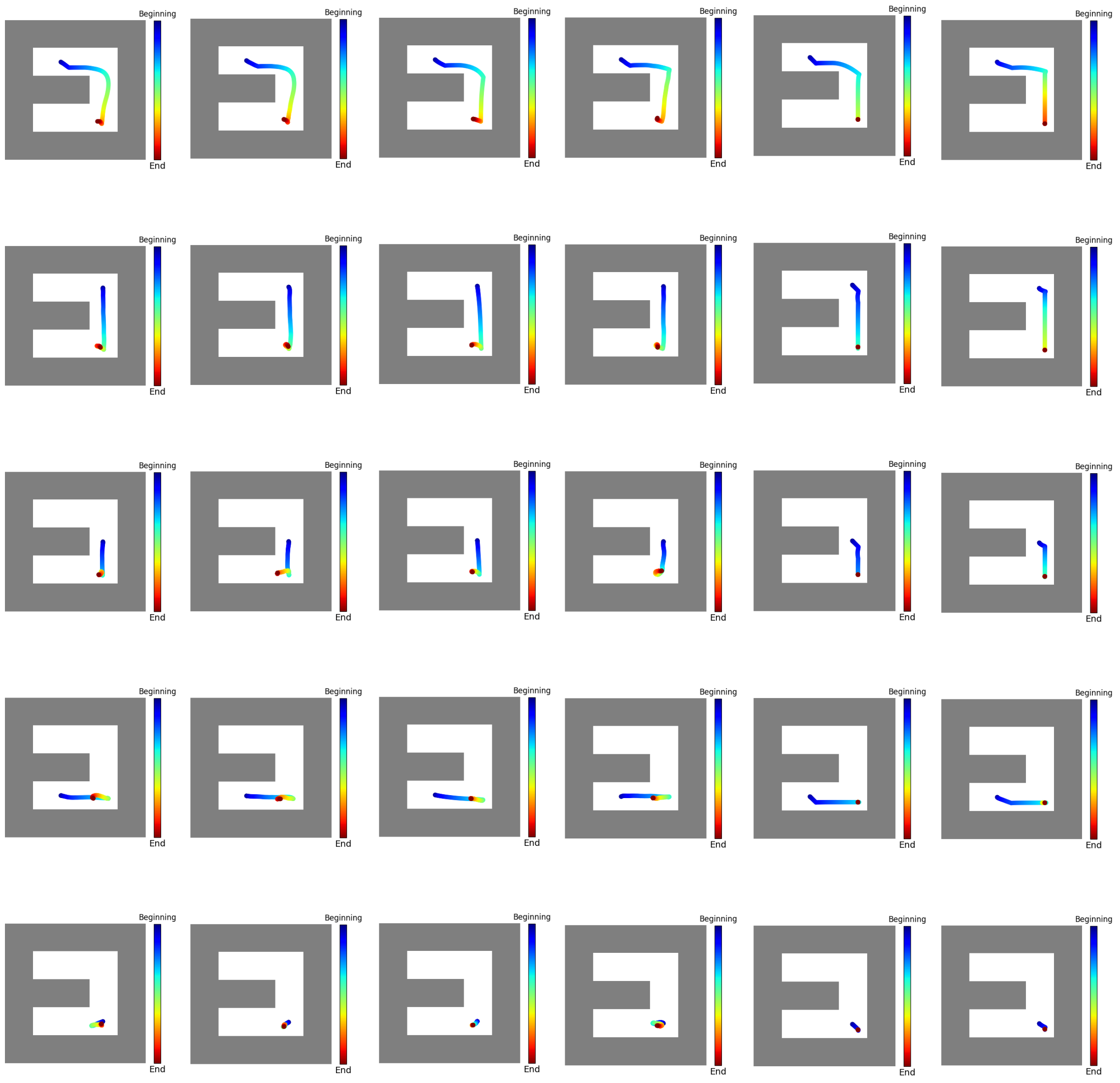

Figure 7: Example output trajectories after learning by different algorithms. Each row is a different starting point. Each column is a different method, from left to right: MSE, MMD with Gaussian kernel, MMD with Matern Kernel, BC, GAIL and AIRL.

similar to the true value function. It simply tells us our method learns a value function that results in similar behaviour as a base Diffuser guided by the true guide. We observe insignificant differences (well within one standard deviation) between the return achieved by the different distance metrics considered.

The main advantage of our method is reflected on the significantly lower values of ERC, especially for the MMD with Matern kernel, which mean our method learns a value function that is more similar to the true reward function. This presents an interesting scenario: despite learning a significantly better value function, all three variations of our method achieve lower return than AIRL. This can be attributed to the previously discussed influence of the base Diffuser training distribution. The Diffuser with a learnt guide does output trajectories that stick to the bottom right corner, but these are generated in a way that is (somewhat) consistent with the base reward-agnostic dataset. AIRL, on the other hand, is not constrained by these learnt environment dynamics, and can optimise its behaviour to single handedly optimise its learnt reward function. However, recall from Section 2.2.4 that ERC is susceptible to reward shaping, and not robust to the choice of coverage distribution (the distribution from which the transitions are sampled to evaluate ERC). Thus, stronger conclusions regarding the quality of the learnt value function should only be taken based on STARC metrics (see Section 2.2.4). Evaluation of such metrics is taken as future work, due to the lack of any open-source implementations.

4 Experiments on Complex Environments

We have observed that our method learns near-optimal behaviour in the U-Maze environment, and that the learnt value function has a smaller ERC than state-of-the-art methods (AIRL). However, the U-Maze is a simple and low dimensional environment. In this section, we implement our method for two more complicated environments. Firstly, the Large Maze (seen in Figure 8) is a more challenging maze where long-term planning is much trickier, and where maximising long-term return often involves sacrificing immediate reward. Furthermore, the state space is much harder to cover, making it harder to recover a reward function that is accurate across the entire trajectory space. Secondly, we consider HalfCheetah (seen in Figure 9), a Mujoco Locomotion environment with a high dimensional state and action space, where moving forward (the goal) requires the agent to learn the subtle interactions between the different positions, velocities and accelarations of different parts of its body. More details about these environments can be found in Appendix B.

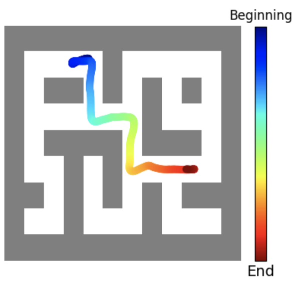

Figure 8: Example trajectory of an agent in the Large Maze environment. The walls of the maze are shown in grey and the area in which the ball can move in white.

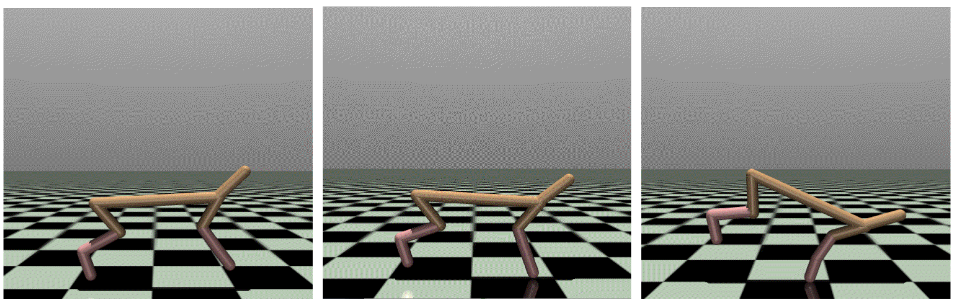

Figure 9: Example frames from a trajectory in the HalfCheetah environment.

4.1 Large Maze

The Large Maze, due to its added complexity, is an appropriate testbed to assess our method's performance on long term planning tasks. We first present an issue with generating expert trajectories in this environment, and how we partially attempt to fix it. We then assess the performance of our learnt-guide Diffuser method on this environment. Finally, we study the base Diffuser's performance in this environment, and show it has considerable flaws, thus rendering our choice of expert trajectories as sub-optimal.

Generating rollouts is non-trivial. In the U-Maze, the expert trajectories used were rollouts obtained from the Diffuser guided with the true reward function, and they all reached the part of the maze with the largest reward (bottom right corner). However, generating such rollouts in the large maze is particularly challenging due to the dataset that the base Diffuser was trained on. As a matter of fact, rollouts resulting from planning (either via impainting or guided planning) consistently get stuck if required to reach the bottom right corner. Figure 10 shows the rollouts obtained from the base Diffuser with the true reward function as guide. As observed, none reach the bottom right most part of the maze, and many of them get fully stuck in their sub-section of the maze. This shows that the Diffuser with the true value function does not necessarily induce optimal behaviour in the environment. As ultimately we evaluate agent behaviour using the reward obtained from rollouts, and not plans, we can conclude that it is entirely possible our method could theoretically learn the true reward function, and yet its rollouts would not be optimal. This issue will be discussed further after our method's performance is presented.

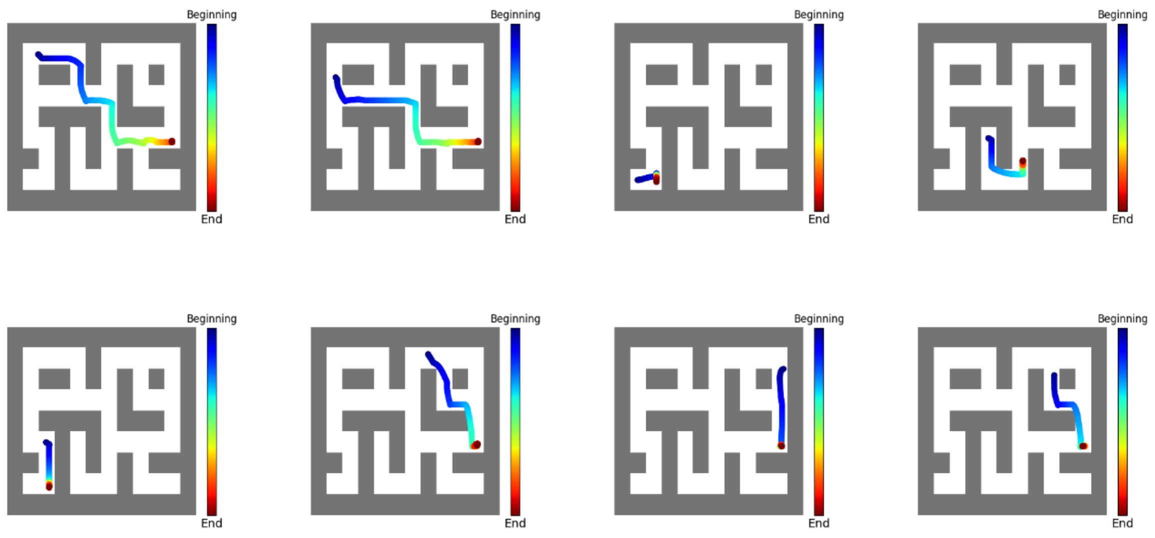

Figure 10: Example rollouts of the base Diffuser using the true reward as guide in the Large Maze environment.

Generating sub-optimal expert trajectories. Due to the inability to generate expert behaviour with rollouts, the expert trajectories used for reward model learning were obtained by planning via impainting on the trained base Diffuser, so as to force all plans to reach the bottom right corner. Importantly, note that unlike for the U-Maze case, these are plans output by the Diffuser, and not actual rollouts in the environment. Eight expert trajectories of length 380 were generated by planning on the environment, starting from 8 different start points (1 trajectory per start point). Figure 5 shows the 8 expert trajectories. Two of these trajectories (bottom right) are not examples of perfect behaviour as the agent does not reach the goal as fast as possible, but the delay was deemed to be small enough that this could be mostly ignored. As they are simply plans, our expert trajectories reflect the environment dynamics as learnt by the Diffuser, and not necessarily the true environment dynamics (which only impact actual rollouts). There is thus a potential environment dynamics gap built into our expert trajectories.

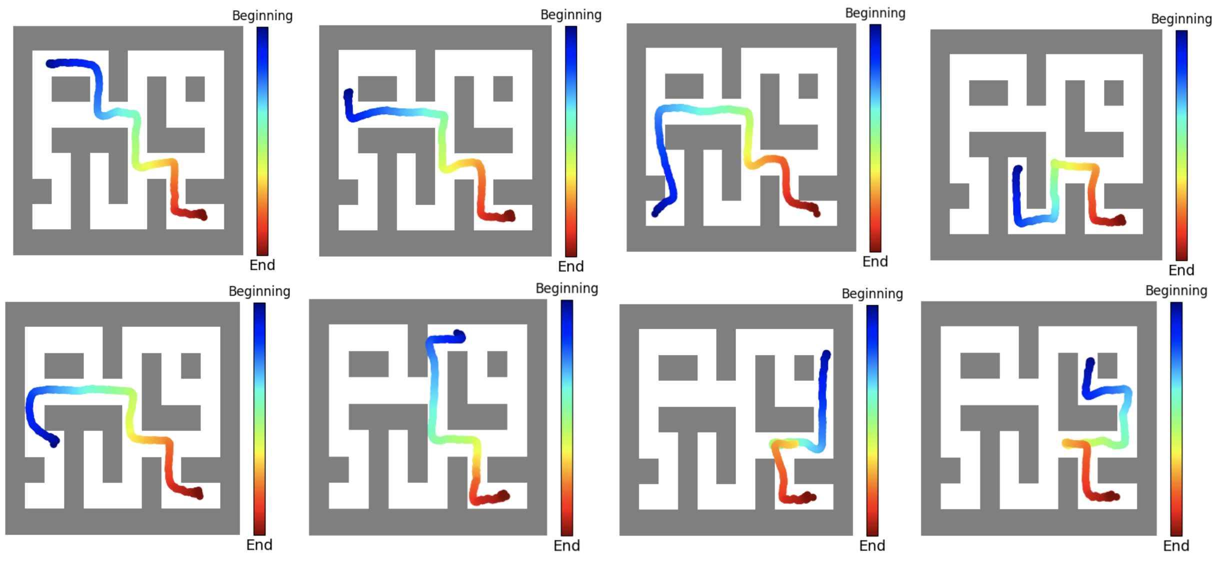

Figure 11: Plans used as expert trajectories for reward learning in the Large Maze environment. Each starts in one of 8 starting points, so as to ensure appropriate state space coverage.

We will further analyse the issue with using plans (instead of rollouts) as expert trajectories in Section 4.1.2. For now, we present our method's performance when the expert trajectories are the plans in Figure 10.

4.1.1 Base experiment

Base Diffuser Training. The base Diffuser was trained on 1062 reward-agnostic trajectories, each of length 384. Example datapoints are shown in Appendix D.2. The planning horizon was chosen as 384, and the number of diffusion steps as 256, as in the original Diffuser [36]. Training details are shown in Appendix C. Example samples of trained base Diffuser are shown in Appendix D.2.

True Reward. As was the case for the U-Maze, we designed a reward model according to the formula $J_\phi = 5x + 5y$, where x and y are the coordinates in the maze (horizontal and vertical respectively, starting from the top left corner, with $x \in [0, 12]$ and $y \in [0, 9]$ including the maze walls). Thus, our reward model gives the highest reward to the bottom right corner of the maze, as seen in Figure 12.

Experimental details. We choose a number of steps $K = 10$ as for the U-Maze. We consider the distance metric M between a 10-step section of an expert trajectory and the first 10 steps of a sampled plan with start state the same as the expert trajectory's section. For the MSE, each individual sample in a batch gets compared to its respective sample in the expert trajectory

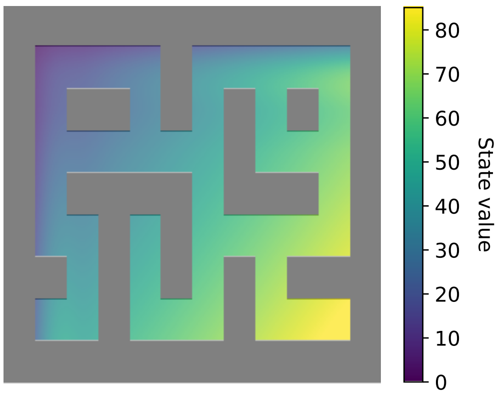

Figure 12: True reward function for the Large Maze experiment. The walls of the maze are shown in grey, whereas states where the agent can move to have their reward shown according to the colourbar.

batch. For the MMD losses, the MMD between the two batches is calculated. The 8 expert trajectories of length 380 get split into 304 10-step sections. We parameterise the reward model as a 4-layer Multi-Layer Perceptron (MLP). Details of the architectural characteristics of the network, as well as hyperparameter values, are given in Appendix C. This reward model is trained for 500 epochs. The loss function curves are shown in Figure 13.

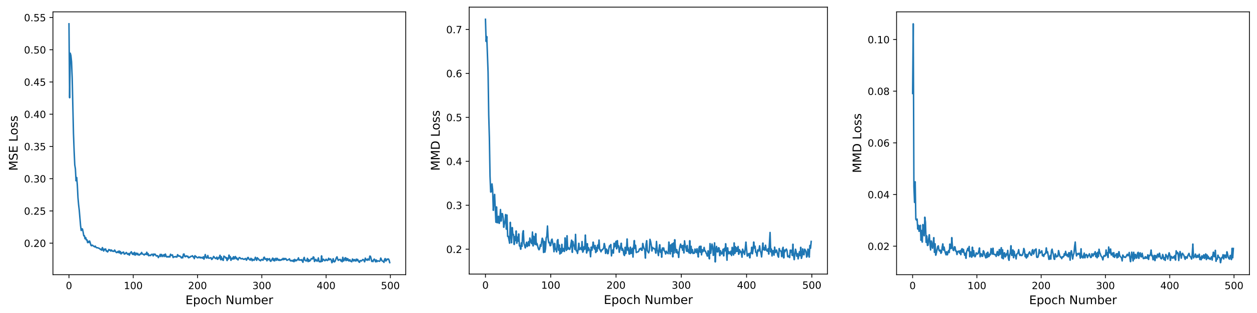

Figure 13: Distance metric for the three different variations of our method as a function of training epoch for the Large Maze training. From left to right: MSE, MMD with Gaussian kernel, MMD with Matern kernel.

Evaluation. We evaluate the quality of the rollouts obtained by guided planning of the base diffuser with the learnt value function, starting from the 8 starting points. We measure the quality of the rollout based on its return under the true reward model. To evaluate the similarity between value functions, we randomly sample trajectories from the dataset used to train the base diffuser, and calculate the ERC (see Section 2.2.4) between the return of these trajectories according to the learnt reward model, and the return according to the true reward model. In Table 2, "Base Diffuser" represents the trained base Diffuser without any guide and "Diffuser (True Reward)" represents the trained base Diffuser with the true reward function as a guide. Note the ERC is only calculated for the methods which learn an explicit value function.

Results. Table 2 shows the return of rollouts according to each of the methods after learning, as well as the ERC between the return of trajectories according to the learnt reward model, and the return according to the true reward. Figure 14 shows example output trajectories for each method.

Table 2: Reward of guided rollouts (averaged over 32 rollouts) and ERC (calculated using 100 rollouts of the base Diffuser dataset) for Large Maze training.

Method	Return ↑	ERC ↓
Base Diffuser	14156.37 ±3849.58	———
Diffuser (True Reward)	22693.68 ±4109.22	———
Diffuser (Learnt MSE Reward)	14639.58 ±4367.54	0.662
Diffuser (Learnt MMD-Gauss Reward)	16150.08 ±5149.77	0.653
Diffuser (Learnt MMD-Matern Reward)	14958.32 ±5275.93	0.663
Behaviour Cloning	17917.14 ±4074.49	———
GAIL	16616.25 ±3506.86	———
AIRL	18575.12 ±5033.36	0.696

Discussion. Firstly, despite Figure 10 showing that rollouts with the true reward do not reach the optimal part of the maze, Table 2 shows that the true reward does guide the Diffuser towards areas of higher reward, even if not to the optimal ones. While we observe learning in all variations of our method, improvements in performance for MSE and MMD with Matern kernel are quite limited, and well within standard deviation range. MMD with Gaussian kernel shows more significant improvements in performance, but still remains far from the performance of the Diffuser guided by the true reward. The overall high values of ERC show our method has failed to learnt a reward function that is similar to the true reward. We hypothesise this is due to the environment dynamics gap present in our expert trajectories, and analyse whether this gap is present in Section 4.1.2. The high standard deviations across all results have three possible explanations: firstly, the small number of rollouts over which performance was averaged (32 instead of 100) due to computational constraints, but also the high variance in behaviour: for specific start points, different seeds will result in either the agent getting completely stuck around its starting point, or in the agent almost making it to the optimal part of the maze. Finally, as our reward function is coordinate-based, and we choose 8 different testing starting points, the return of each trajectory is highly dependent on its starting point.

Figure 14 shows example output trajectories after learning for the different algorithms. Note that AIRL learns near-optimal behaviour for certain starting points, but gets completely stuck in one place for others. On the other hand, Behaviour Cloning's trajectories do not go as directly to the optimal section of the maze, but the method is also less likely to get "stuck". Regarding the three variations of our method, none achieve behaviour that seems optimal. Regardless, MMD with Gaussian Kernel is the only one to obtain behaviour that somewhat resembles expert behaviour, and its trajectories, when not stuck, move in the correct direction to achieve the optimal section of the maze (unlike the MSE trajectories).

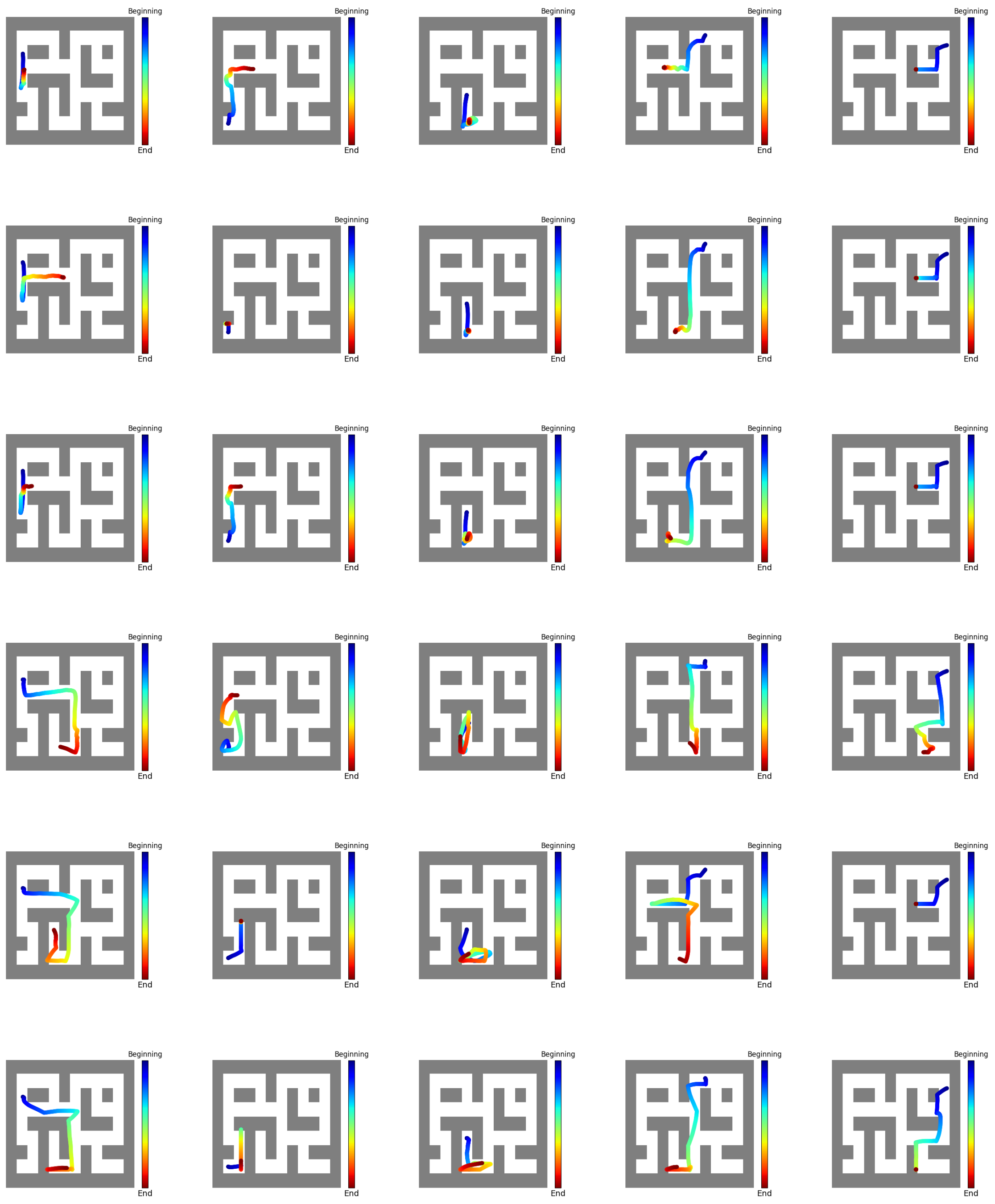

Figure 14: Example output trajectories in the Large Maze after learning by different algorithms. Each row is a different starting point. Each column is a different method, from top to bottom: MSE, MMD with Gaussian kernel, MMD with Matern Kernel, BC, GAIL and AIRL.

Resembling the result obtained for the U-Maze, our method once again obtains a value function with lower ERC than AIRL. However, it is valuable to note a value of ERC around 0.65 still only shows a small correlation between returns, and thus all value functions obtained are poor (for this choice of evaluation metric) approximations of the true reward function. We hypothesise that the smaller values of ERC for our method in the Large Maze, as well as the U-Maze, are a benefit of the Diffuser set-up: while for AIRL the algorithm must learn both the environment dynamics and the value function, for our method the algorithm can use the expert data to only alter the value function, thus extracting more information from the data. Further work should study the effects on performance of reduced or increased expert dataset sizes. As for the U-Maze, we hypothesise that the base Diffuser learnt environment dynamics damage our method's performance, as the output trajectories are still impacted by the base Diffuser's training distribution, which is not at all optimal under our reward function. Notwithstanding, a stronger value function similarity metric should be used before stronger conclusions are drawn.

All three baselines outperform our method. Importantly, none of them reach similar performance to the True Reward. Thus we hypothesise that our expert trajectories (which are plans, not rollouts) are not appropriate for either imitation learning or reward function learning.

4.1.2 Analysis of the Diffuser's environment model

As previously discussed, plans were used as expert trajectories for learning in the Large Maze. Both our method and the three baselines struggled to learn the true reward function (or to act according to it), and we hypothesised that this is because the plans used as expert trajectories are sub-optimal, in the sense that they reflect the environment dynamics according to the base Diffuser, and not the true environment dynamics. Thus, learning to replicate such plans does not actually result in rollouts in the environment that are optimal. In this section we study the environment dynamics learnt by the Diffuser, and their impact on the quality of plans.

Firstly, we aim to find how the base Diffuser's knowledge of the true environment dynamics varies across the large maze. To do so, we select 500 points uniformly across the maze. For each point, the agent plans, and we select the first action a and next state s' of that plan. Thus, for each of the 500 points, we have a tuple (s, a, s'). Note that this reflects what next state s' the agent thinks it will be if it takes action a from state s. For each tuple, we set the environment state as s, take the action a, and receive a true next state s^T. We then compare the MSE between the vectors s' and s^T. This is a measure of the difference between the true next state, and the next state according to the diffuser dynamics. As the action a is the same for both cases, there is no effect of the policy in this metric. We plot a heatmap of the MSE for different points in the large maze in Figure 15.

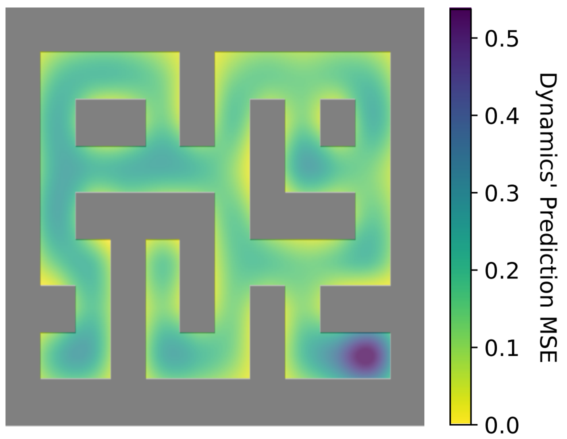

Figure 15: Dynamics' prediction MSE as a function of the maze coordinates, averaged over the first step of 500 plans with uniformly sampled start points.

Firstly, note that the bottom right corner (exactly the optimal section of the maze according to our true reward model) is where the Diffuser's dynamics are the most wrong. This is to be expected: reward-agnostic trajectories in the base Diffuser's training dataset very rarely reach this specific part of the maze. Secondly, we observe a few specific zones where the learnt dynamics are particularly wrong, as seen in Figure 16. These coincide with some of the start points of our expert plans. Thus, while one may think the expert plans in Figure 11 would reach the maze (as it looks like they would), they actually seem to get stuck (as seen in Figure 10, partly because the environment dynamics have not been fully learnt in certain sections of the maze.

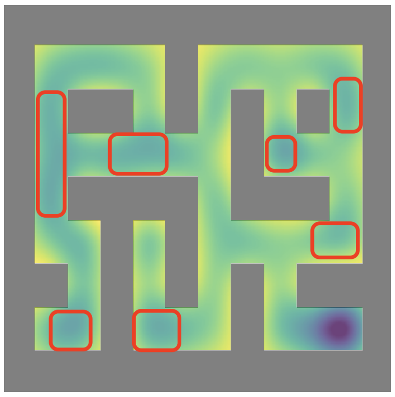

Figure 16: Dynamics' prediction MSE heatmap, with highlighted (in red) high MSE areas which coincide with areas where rollouts get stuck.

In Figure 15, we analysed the quality of the learnt environment dynamics in the first step of a plan. As the Diffuser does not plan autoregressively, it does not in theory suffer from the typical compounding error in planning. However, we decided to study the quality of the plan as a function of the planning step. As for the previous case, we sample 500 starting points uniformly in the large maze. For each starting point, we plan 384 steps ahead (typical horizon used in the large maze). We separate each plan in (s_t, a_t, s'_{t+1}) tuples, for plan step t. We then calculate the MSE between (s_t, a_t, s'_{t+1}) and (s_t, a_t, s^T_{t+1}). Figure 17 shows the MSE as a function of the planning step t.

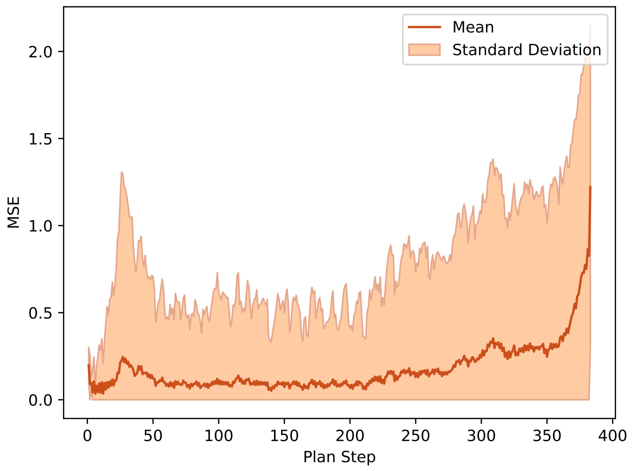

Figure 17: Dynamics' prediction MSE as a function of the plan step, averaged over 500 plans with uniformly sampled start points. The planning horizon for the Large Maze is 384.

Firstly, note that the error on the first step of a plan is consistently larger than any of the other first 25 steps. Secondly, except for a spike at $t = 30$ (for which we have not found a particular explanation), the dynamics error remains somewhat stable for the first 225 planning steps. However, this error grows for the last 150 steps of a plan, and in particular in the last 100 steps. This is an interesting phenomenon that is quite unexpected for the Diffuser, as it is expected that the method does not suffer from compounding planning error. While we leave as future work to understand the underlying cause for this growing error, we note that, as per Algorithm 5, we split expert plans into K-step trajectories ($K = 10$ in this case). Thus, the latter parts of a plan get split into expert trajectory sections with very large environment dynamics' MSE. Thus, it is expected that these sections will be very poor expert datapoints, and can mislead learning.

Overall, our method fails to accurately learn a value function and behave according to it in the Large Maze. We show that our expert plans are not suitable for this task, and contain significant gaps in their perceived environment dynamics. However, it is important to note that while AIRL did not achieve optimal behaviour, it achieved significantly better performance than our method, despite being trained on the same flawed plans. Future work should study the reason behind this gap in performance. Nevertheless, as AIRL does not achieve optimal behaviour (and neither does the Diffuser with the true reward function as guide), it is paramount that a more careful design of expert trajectories is performed. While different reward functions were informally attempted in the aim to fix this issue (without success), it is possible that smarter reward models could generate optimal rollouts. In particular, a non-dense reward function (that is, a reward function that only gives reward at a certain state, as opposed to giving coordinate-based rewards at any point of the maze) could potentially generate appropriate expert trajectories. However, such non-dense reward functions are commonly harder (and much less sample efficient) to learn. Finally, we note that our reward network was a simple 4-layer MLP. Previous work [38, 36] used much more complex networks, in particular value networks where the value guide is dependent on the diffusion step. While learning diverged when we attempted this direction, it is possible a more complex network could unlock performance in this much more complicated maze. We propose these two directions as future work, but highlight the importance behind obtaining expert trajectories with better quality before reaching stronger conclusions about any other aspects of learning in our method.

4.2 HalfCheetah

Our experimental setup is significantly different for the HalfCheetah environment. In the maze environments, we hand-crafted a true reward function, built trajectories that would be considered as optimal (or near optimal) under this reward, and used our method to learn an approximation of the true reward function. For the HalfCheetah environment, we train a base diffusion model on a dataset of low-reward trajectories, use a dataset of high-reward trajectories as expert trajectories, and aim to learn a reward function that can align the Diffuser samples towards the high-reward distribution. Thus, we evaluate the quality of the trajectories based on the reward signal that is inherent to the environment. In the HalfCheetah case, this reward is positive the further the agent moves forward, and has a negative penalty term for actions with large cost. For more information about this environment see Appendix B.

Datasets. The Half-Cheetah environment has different datasets, with different quality of trajectories. "Expert" is a 1 million transition dataset, obtained from sampling from a policy trained to completion with Soft Actor-Critic [99]. ”Medium” is a 1 million transition dataset, obtained from a policy trained to achieve below half of the return of the expert. ”Medium-Replay” uses the replay buffer of the training a policy up to the performance of ”Medium”. Thus, initial samples have very low reward, and reward improves towards the latter parts of the dataset as the training agent improved its performance and incorporated good trajectories in its replay buffer. Regardless, we only randomly sample from this dataset, discarding any order. Table 3 shows the average return of trajectories in each of the three datasets. Furthermore, the table shows the average L2 norm of the action vector of trajectories in each dataset. This metric will be analysed in the discussion.

Table 3: Average return of 1000-step trajectories, and average action vector L2 norm, in different HalfCheetah datasets.

Dataset	Return	Action L2 Norm
Medium-Replay	285.54 ±179.67	2.37 ±0.63
Medium	442.01 ±106.52	2.35 ±0.69
Expert	1001.77 ±226.37	2.40 ±0.48

All datasets in the HalfCheetah environment are automatically divided in K-step sub-trajectories, where K is the horizon used for the base Diffuser. As we use $K = 4$ [38], we automatically get datasets of 4-step sub-trajectories. Thus, we can skip the pre-processing step of our method, described in Algorithm 4. Note that Table 3 was created taking into account 1000-step trajectories, as this will be the length used for evaluation.

4.2.1 Base experiment

Base Diffuser Training. The base Diffuser was trained on 200 reward-agnostic trajectories, each of length 1000, part of the Medium-Replay dataset. The planning horizon was chosen as 4, and the number of diffusion steps as 20. Nutti et al. [38] showed this was an appropriate choice under a guidance scale of 0.001, and thus we also chose this scale. Training details are shown in Appendix C.

Expert trajectories. The expert trajectories were chosen as 100 thousand randomly sampled 4-step sub-trajectories from the Expert HalfCheetah dataset [40]. The three baselines were trained exclusively on 1600 1000-step trajectories from the Expert HalfCheetah dataset, ensuring the same number of total training transitions as for our method.

Experimental details. As the diffusion horizon is chosen as 4, we automatically choose the number of steps of our method to be $K = 4$ (as this cannot be larger than the diffusion horizon). We consider the distance metric M between a 10-step section of an expert trajectory and the first 10 steps of a sampled plan with start state the same as the expert trajectory's section. For the MSE, each individual sample in a batch gets compared to its respective sample in the expert trajectory batch. For the MMD losses, the MMD between the two batches is calculated. We parameterise the reward model as a 4-layer Multi-Layer Perceptron (MLP). Details of the architectural characteristics of the network, as well as hyperparameter values, are given in

Appendix C. The reward model is trained for 250 epochs. The loss function curves are shown in Figure 18.

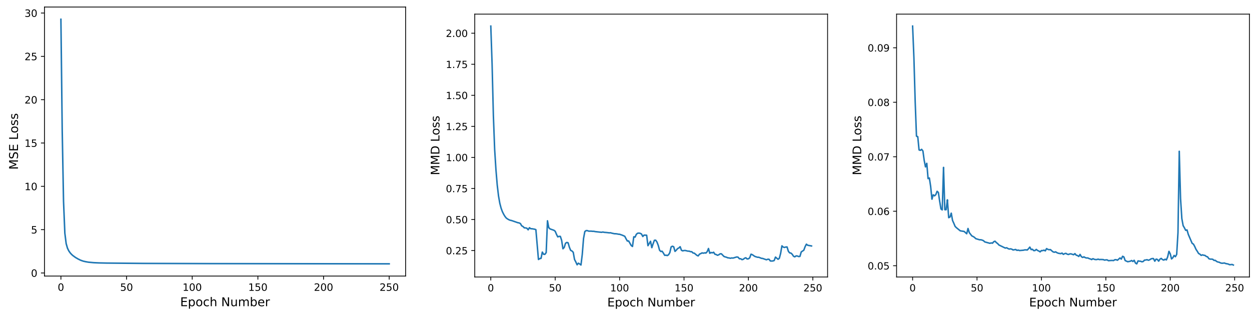

Figure 18: Distance metric for the three different variations of our method as a function of training epoch for the HalfCheetah environment. From left to right: MSE, MMD with Gaussian kernel, MMD with Matern kernel.

Evaluation. We evaluate the quality of the rollouts obtained by guided planning of the base Diffuser with the learnt value function. As this is the case in the HalfCheetah environment, the starting point (barring some added noise) is always the same. We measure the quality of the rollout based on the return given by the environment. To evaluate the similarity between value functions, we calculate the ERC between the value estimate for a trajectory, and the return given by the environment for that trajectory (which one can think of as the true reward function). We use a mixed dataset with 50% medium-replay samples, and 50% expert samples, and guarantee these were not seen during either base Diffuser or reward model training. In Table 4, "Base Diffuser" represents the trained (on the medium-replay Dataset) base Diffuser without any guide. Note the ERC is only calculated for the methods which learn an explicit value function.

Results. Table 4 shows the return of rollouts according to each of the methods after learning, as well as the ERC between the return of trajectories according to the learnt reward model, and the return according to the true reward.

Table 4: Reward of guided rollouts and ERC of base dataset (averaged over 100 rollouts) for HalfCheetah training.

Method	Return ↑	ERC ↓
Base Diffuser	276.49 ±74.28	———
Diffuser (Learnt MSE Reward)	-39.15 ±0.95	0.758
Diffuser (Learnt MMD-Gauss Reward)	-32.50 ±1.08	0.807
Diffuser (Learnt MMD-Matern Reward)	-19.80 ±18.00	0.698
Behaviour Cloning	1657.78 ±45.62	———
GAIL	998.21 ±231.52	———
AIRL	1298.25 ±152.04	0.757

Discussion. Firstly, we observe that the base Diffuser, trained on the Medium-Replay dataset, achieves very similar performance to its training data. This implies that the environment dynamics seem to be properly learnt by the base Diffuser. However, recall that the Medium-Replay dataset is far from optimal, and thus the base Diffuser learns dynamics that can, in part, be corrupted due to the sub-par examples it is trained on.

We observe in Figure 18 that learning is very unstable in this setting, in particular for the MMD-based losses. Note that these loss curves are already the result of an informal parameter tuning process, and thus the instability is unlikely to be fully due to hyperparameter choices. Gradients and weight values were monitored throughout the training process (to monitor possible gradient or weight exploding) and appeared, for the most part, to be standard. However, future work should provide a proper analysis of how these values vary during training, and in particular whether either of these explode when the loss function spikes. The results in Table 4 represent the best performance over checkpoints (every 10 epochs) for each loss. We see that our method consistently fails to learn high-reward behaviour. Furthermore, its performance is far below the base Diffuser's performance, implying that the guide learnt actually severely worsens its performance. Note in particular the very low variances for both the MSE and MMD with Gaussian kernel distances.

All ERC values are particularly high, showing that neither our method nor AIRL learn a value function that is close to the environment reward. It is thus concluded that there is no significant correlation between the two for any method, and thus the IRL task is not succefful in such a setting. Regardless, MMD with Matern kernel clearly outperforms the other methods in terms of ERC, which seems to also result in a higher return than other variations of our method.

Regarding the baselines, we see GAIL achieves similar performance to the expert dataset. Quite surprisingly, both AIRL and Behaviour Cloning significantly outperform the dataset they were trained on. We hypothesise that the particularly good performance of BC is due to the HalfCheetah always having the same starting point (barring some noise): as the agent always starts in the same state, the algorithm learns to mimic the high reward behaviours in the dataset, without having to worry about exploration and out-of-distribution performance, areas where BC naturally struggles. In the AIRL case, we hypothesise that the learning of a value function allows the agent to clearly learn which states are valuable, and deterministically act according to this value function.

As a first step in an analysis to understand the failure of our method, we hypothesised that the very negative reward our method obtains could be due to the agent choosing actions with very large norm. Appendix B.2 explains how the reward in the HalfCheetah environment is composed of two terms: one positive term for how far the agent moves forward, and a negative penalty for actions with large norm. Thus, we calculated the average L2 norm of the action vector in the rollouts output by each method (the same rollouts used for Table 4), and compared the L2 norm to those in Table 3.

The obtained results do not support our hypothesis. Firstly, there is no significant variation of norm between the different HalfCheetah datasets. Secondly, it seems that the version of our method that does best (MMD Matern) is also the only one to have L2 norm close to those in the HalfCheetah dataset, which is significantly higher than the L2 norm obtained by the other variations of our method. It seems, thus, that for all variations of our method, the agent simply falls and tries to act to get back up but fails to do so, resulting in a negative-only reward signal from any actions it does to attempt to get back up. Note, however, that the action L2 norm of the base Diffuser is significantly lower than the action L2 norm of the Medium-replay dataset it was trained on. This mismatch should be investigated further, even if the base Diffuser seems to achieve the same return as the medium-replay dataset (Table 4).

Table 5: Action vector L2 norm averaged over 100 rollouts for methods learning on the Halfcheetah environment.

Metric	Action L2 Norm
Base Diffuser	1.77 ±0.33
Diffuser (Learnt MSE Reward)	1.45 ±0.01
Diffuser (Learnt MMD-Gauss Reward)	1.25 ±0.38
Diffuser (Learnt MMD-Matern Reward)	2.48 ±0.64

Further analysis ought to be conducted to understand why our method fails in such a setting. Importantly, both the Diffuser [36] and Nutti et al. [38] (partly) succeeded in aligning behaviour in such a setting. Both methods used a reward guide that also depended on the diffusion step t. We attempted this but observed no learning, even after some informal hyperparameter tuning. Regardless, this direction should be more carefully studied in the future, to clarify whether a very simple reward model is the reason our method fails. A second possible reason is due to the settings chosen for the Diffuserfor the Halfcheetah environment, where one uses a planning horizon of only 4 steps (as opposed to 128 in the U-Maze and 384 in the Large Maze). This was chosen due to Nutti et al. [38] showing this short planning horizon worked if the guidance scale was changed to 0.001. It is possible that this change actually worsens learning in our environment, and that the original planning horizon of 32 (and larger guidance scale) in the Diffuser [36] is necessary to properly learn the long-term return in this environment. Furthermore, the guidance scale is a multiplicative term on the gradient of the reward model, and thus a larger guidance scale allows for the guide to have a stronger effect on the diffusion process.

5 Conclusion

In this report we introduce a method to learn a reward function using the Diffuser, an offline trajectory-level diffusion model. We assume we have access to a base Diffuser trained on reward-agnostic trajectories in an environment, and a dataset of expert trajectories that we wish to imitate. We derive our method from a classifier-guided diffusion perspective, and parameterise the diffusion model's guide as a neural network that we can learn. We propose an algorithm to learn this reward/guide network based on a loss function (between sampled Diffuser plans and expert trajectories) in trajectory space, and present three possible loss functions. In learning how to guide the diffusion process, we not only obtain a reward function, but also behaviour that imitates the expert trajectories given.

We evaluate our method on three environments, with varying degrees of complexity. In the U-Maze [40], we observe that our method achieves near optimal behaviour, matching the performance of the baselines (Behaviour Cloning, GAIL and AIRL). Despite the similarities in behaviour, we conclude via an analysis of the Episodic Return Correlation distance [39] that our method learns a reward function that is closer to the true reward than our baseline (AIRL).

When extending our method to the Large Maze [40], we face difficulties in obtaining high-quality expert trajectories. We attempt an imperfect fix to this issue by considering expert plans instead of expert rollouts, but our method fails to both learn expert behaviour and to learn the true reward function. We further analyse the environment dynamics learnt by the Diffuser, concluding these lead to low quality plans that consequently lead to our method's poor performance. We finish by extending our method to the HalfCheetah environment, where our method fully fails to not only match the baselines, but learn altogether. We face large instabilities during training of the reward function, which we hypothesise is due to two main factors: the very simple reward network used (in particular one whose output does not depend on the diffusion step) and the small number of diffusion steps (which does not allow for proper guide contribution throughout a sufficient number of diffusion steps).

In the future, the main focus should be on studying the viability of our method in these more complex environments. For the Large Maze, obtaining near-optimal expert trajectories should be the focus, whereas we hypothesise that our method's failure in HalfCheetah can be fixed by tuning of training details (hyperparameters and architectures) and analysis of the current training instabilities.

References

[1] Xu Wang, Sen Wang, Xingxing Liang, Dawei Zhao, Jincai Huang, Xin Xu, Bin Dai, and Qiguang Miao. Deep Reinforcement Learning: A Survey. *IEEE Transactions on Neural Networks and Learning Systems*, 35(4):5064–5078, April 2024. ISSN 2162-2388. doi: 10.1109/TNNLS.2022.3207346. Conference Name: IEEE Transactions on Neural Networks and Learning Systems.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning, December 2013. arXiv:1312.5602 [cs].

[3] Lei Tai, Giuseppe Paolo, and Ming Liu. Virtual-to-real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Mapless Navigation, July 2017. arXiv:1703.00420 [cs].

[4] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 0028-0836, 1476-4687. doi: 10.1038/nature16961.

[5] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, December 2017. arXiv:1712.01815 [cs].

[6] OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with Large Scale Deep Reinforcement Learning, December 2019. arXiv:1912.06680 [cs, stat].

[7] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation Learning: A Survey of Learning Methods. *ACM Computing Surveys*, 50(2):1–35, March 2018. ISSN 0360-0300, 1557-7341. doi: 10.1145/3054912.

[8] Chelsea Finn, Sergey Levine, and Pieter Abbeel. Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization, May 2016. arXiv:1603.00448 [cs].

[9] Jonathan Ho and Stefano Ermon. Generative Adversarial Imitation Learning, June 2016. arXiv:1606.03476 [cs].

[10] Stuart Russell. Learning agents for uncertain environments (extended abstract). In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 101–103, Madison Wisconsin USA, July 1998. ACM. ISBN 978-1-58113-057-7. doi: 10.1145/279943.279964.

[11] Andrew Y. Ng and Stuart J. Russel. Algorithms for Inverse Reinforcement Learning. 2000. Meeting Name: ICML.

[12] Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence*, 297:103500, August 2021. ISSN 00043702. doi: 10.1016/j.artint.2021.103500.

[13] Stephen Adams, Tyler Cody, and Peter A. Beling. A survey of inverse reinforcement learning. *Artificial Intelligence Review*, 55(6):4307–4346, August 2022. ISSN 0269-2821, 1573-7462. doi: 10.1007/s10462-021-10108-x.

[14] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete Problems in AI Safety, July 2016. arXiv:1606.06565 [cs].

[15] Iason Gabriel. Artificial Intelligence, Values, and Alignment. *Minds and Machines*, 30(3): 411–437, September 2020. ISSN 0924-6495, 1572-8641. doi: 10.1007/s11023-020-09539-2.

[16] Brian D Ziebart, Andrew Maas, J Andrew Bagnell, and Anind K Dey. Maximum Entropy Inverse Reinforcement Learning.

[17] Chelsea Finn, Paul Christiano, Pieter Abbeel, and Sergey Levine. A Connection between Generative Adversarial Networks, Inverse Reinforcement Learning, and Energy-Based Models, November 2016. arXiv:1611.03852 [cs].

[18] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks, June 2014. arXiv:1406.2661 [cs, stat].

[19] Justin Fu, Katie Luo, and Sergey Levine. Learning Robust Rewards with Adversarial Inverse Reinforcement Learning, August 2018. arXiv:1710.11248 [cs].

[20] Gokul Swamy, David Wu, Sanjiban Choudhury, J Andrew Bagnell, and Zhiwei Steven Wu. Inverse Reinforcement Learning without Reinforcement Learning.

[21] Abi Komanduru and Jean Honorio. On the Correctness and Sample Complexity of Inverse Reinforcement Learning, June 2019. arXiv:1906.00422 [cs, stat].

[22] Abi Komanduru and Jean Honorio. A Lower Bound for the Sample Complexity of Inverse Reinforcement Learning, March 2021. arXiv:2103.04446 [cs, stat].

[23] Fuxiang Zhang, Junyou Li, Yi-Chen Li, Zongzhang Zhang, Yang Yu, and Deheng Ye. Improving Sample Efficiency of Reinforcement Learning with Background Knowledge from Large Language Models, July 2024. arXiv:2407.03964 [cs].

[24] Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. RL$ˆ2$: Fast Reinforcement Learning via Slow Reinforcement Learning, November 2016. arXiv:1611.02779 [cs, stat].

[25] Yang Yu. Towards Sample Efficient Reinforcement Learning.

[26] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep Unsupervised Learning using Nonequilibrium Thermodynamics, November 2015. arXiv:1503.03585 [cond-mat, q-bio, stat].

[27] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models, December 2020. arXiv:2006.11239 [cs, stat].

[28] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes, 2013. arXiv:1312.6114 [cs, stat].

[29] Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured Denoising Diffusion Models in Discrete State-Spaces, February 2023. arXiv:2107.03006 [cs].

[30] Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. Diffusion-LM Improves Controllable Text Generation, May 2022. arXiv:2205.14217 [cs].

[31] Minkai Xu, Lantao Yu, Yang Song, Chence Shi, Stefano Ermon, and Jian Tang. GeoDiff: a Geometric Diffusion Model for Molecular Conformation Generation, March 2022. arXiv:2203.02923 [cs, q-bio].

[32] Arne Schneuing, Yuanqi Du, Charles Harris, Arian Jamasb, Ilia Igashov, Weitao Du, Tom Blundell, Pietro Lió, Carla Gomes, Max Welling, Michael Bronstein, and Bruno Correia. Structure-based Drug Design with Equivariant Diffusion Models, June 2023. arXiv:2210.13695 [cs, q-bio].

[33] Shitong Luo and Wei Hu. Diffusion Probabilistic Models for 3D Point Cloud Generation, June 2021. arXiv:2103.01458 [cs].

[34] Andreas Lugmayr, Martin Danelljan, Andres Romero, Fisher Yu, Radu Timofte, and Luc Van Gool. RePaint: Inpainting using Denoising Diffusion Probabilistic Models, August 2022. arXiv:2201.09865 [cs].

[35] Jonathan Ho and Tim Salimans. Classifier-Free Diffusion Guidance, July 2022. arXiv:2207.12598 [cs].

[36] Michael Janner, Yilun Du, Joshua B. Tenenbaum, and Sergey Levine. Planning with Diffusion for Flexible Behavior Synthesis, December 2022. arXiv:2205.09991 [cs].

[37] Anurag Ajay, Yilun Du, Abhi Gupta, Joshua Tenenbaum, Tommi Jaakkola, and Pulkit Agrawal. Is Conditional Generative Modeling all you need for Decision-Making?, July 2023. arXiv:2211.15657 [cs].

[38] Felipe Nuti, Tim Franzmeyer, and João F. Henriques. Extracting Reward Functions from Diffusion Models, December 2023. arXiv:2306.01804 [cs].

[39] Adam Gleave, Michael Dennis, Shane Legg, Stuart Russell, and Jan Leike. Quantifying Differences in Reward Functions, March 2021. arXiv:2006.13900 [cs, stat].

[40] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4RL: Datasets for Deep Data-Driven Reinforcement Learning, February 2021. arXiv:2004.07219 [cs, stat].

[41] Marvin Minsky. Steps toward Artificial Intelligence. *Proceedings of the IRE*, 49(1):8–30, January 1961. ISSN 0096-8390. doi: 10.1109/JRPROC.1961.287775.

[42] Richard Bellman. A Markovian Decision Process. *Indiana University Mathematics Journal*, 6(4):679–684, 1957. ISSN 0022-2518. doi: 10.1512/iumj.1957.6.56038.

[43] R. Bellman. Dynamic programming. *Science (New York, N.Y.)*, 153(3731):34–37, July 1966. ISSN 0036-8075. doi: 10.1126/science.153.3731.34.

[44] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, May 1998. ISSN 00043702. doi: 10.1016/S0004-3702(98)00023-X.

[45] Richard S. Sutton and Andrew Barto. *Reinforcement learning: an introduction.* Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts London, England, second edition edition, 2020. ISBN 978-0-262-03924-6.

[46] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, August 1988. ISSN 0885-6125, 1573-0565. doi: 10.1007/BF00115009.

[47] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. Q-Learning Algorithms: A Comprehensive Classification and Applications. *IEEE Access*, 7:133653–133667, 2019. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2941229. Conference Name: IEEE Access.

[48] Hado van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep Reinforcement Learning and the Deadly Triad, December 2018. arXiv:1812.02648 [cs].

[49] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-learning, December 2015. arXiv:1509.06461 [cs].

[50] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep Q-learning from Demonstrations, November 2017. arXiv:1704.03732 [cs].

[51] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep Exploration via Bootstrapped DQN, July 2016. arXiv:1602.04621 [cs, stat].

[52] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning, October 2017. arXiv:1710.02298 [cs].

[53] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning, April 2016. arXiv:1511.06581 [cs].

[54] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy Networks For Exploration. 2018.

[55] Marc G. Bellemare, Will Dabney, and Rémi Munos. A Distributional Perspective on Reinforcement Learning, July 2017. arXiv:1707.06887 [cs, stat].

[56] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning, June 2016. arXiv:1602.01783 [cs].

[57] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization, April 2017. arXiv:1502.05477 [cs].

[58] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. arXiv:1707.06347 [cs].

[59] Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4):160–163, July 1991. ISSN 0163-5719. doi: 10.1145/122344.122377.

[60] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to Trust Your Model: Model-Based Policy Optimization.

[61] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to Control: Learning Behaviors by Latent Imagination, March 2020. arXiv:1912.01603 [cs].

[62] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, H. Jaap Van Den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games*, volume 4630, pages 72–83. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-75537-1 978-3-540-75538-8. doi: 10.1007/978-3-540-75538-8_7. Series Title: Lecture Notes in Computer Science.

[63] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: a modern approach*. Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, 3rd ed edition, 2010. ISBN 978-0-13-604259-4.

[64] Nathan D. Ratliff, J. Andrew Bagnell, and Martin A. Zinkevich. Maximum margin planning. In *Proceedings of the 23rd international conference on Machine learning - ICML '06*, pages 729–736, Pittsburgh, Pennsylvania, 2006. ACM Press. ISBN 978-1-59593-383-6. doi: 10.1145/1143844.1143936.

[65] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Twenty-first international conference on Machine learning - ICML '04*, page 1, Banff, Alberta, Canada, 2004. ACM Press. doi: 10.1145/1015330.1015430.

[66] Deepak Ramachandran. Bayesian Inverse Reinforcement Learning.

[67] Markus Wulfmeier, Peter Ondruska, and Ingmar Posner. Maximum Entropy Deep Inverse Reinforcement Learning, March 2016. arXiv:1507.04888 [cs].

[68] Gergely Neu and Csaba Szepesvari. Apprenticeship Learning using Inverse Reinforcement Learning and Gradient Methods.

[69] Sergey None Levine, Zoran Popovic, and Vladlen Koltun. Nonlinear Inverse Reinforcement Learning with Gaussian Processes.

[70] Sergey Levine and Pieter Abbeel. Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics. 2014.

[71] Blake Wulfe, Ashwin Balakrishna, Logan Ellis, Jean Mercat, Rowan McAllister, and Adrien Gaidon. Dynamics-Aware Comparison of Learned Reward Functions, January 2022. arXiv:2201.10081 [cs].

[72] Joar Skalse, Lucy Farnik, Sumeet Ramesh Motwani, Erik Jenner, Adam Gleave, and Alessandro Abate. STARC: A General Framework For Quantifying Differences Between Reward Functions, March 2024. arXiv:2309.15257 [cs].

[73] Joar Skalse, Matthew Farrugia-Roberts, Stuart Russell, Alessandro Abate, and Adam Gleave. Invariance in Policy Optimisation and Partial Identifiability in Reward Learning, June 2023. arXiv:2203.07475 [cs, stat].

[74] Arthur Gretton, Karsten Borgwardt, Malte J. Rasch, Bernhard Scholkopf, and Alexander J. Smola. A Kernel Method for the Two-Sample Problem, May 2008. arXiv:0805.2368 [cs].

[75] Gintare Karolina Dziugaite, Daniel M. Roy, and Zoubin Ghahramani. Training generative neural networks via Maximum Mean Discrepancy optimization, May 2015. arXiv:1505.03906 [cs, stat].

[76] Yujia Li, Kevin Swersky, and Richard Zemel. Generative Moment Matching Networks, February 2015. arXiv:1502.02761 [cs, stat].

[77] Arthur Gretton, Dino Sejdinovic, Heiko Strathmann, Sivaraman Balakrishnan, Massimiliano Pontil, Kenji Fukumizu, and Bharath K Sriperumbudur. Optimal kernel choice for large-scale two-sample tests.

[78] Sergey Levine, Nolan Wagener, and Pieter Abbeel. Learning Contact-Rich Manipulation Skills with Guided Policy Search, February 2015. arXiv:1501.05611 [cs].

[79] Chun-Liang Li, Wei-Cheng Chang, Yu Cheng, Yiming Yang, and Barnabás Póczos. MMD GAN: Towards Deeper Understanding of Moment Matching Network, November 2017. arXiv:1705.08584 [cs, stat].

[80] William Feller. On the Theory of Stochastic Processes, with Particular Reference to Applications. In René L. Schilling, Zoran Vondraček, and Wojbor A. Woyczyński, editors, *Selected Papers I*, pages 769–798. Springer International Publishing, Cham, 2015. ISBN 978-3-319-16858-6 978-3-319-16859-3. doi: 10.1007/978-3-319-16859-3_42.

[81] Yang Song and Stefano Ermon. Generative Modeling by Estimating Gradients of the Data Distribution, October 2020. arXiv:1907.05600 [cs, stat].

[82] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation, May 2015. arXiv:1505.04597 [cs].

[83] Scott Fujimoto, David Meger, and Doina Precup. Off-Policy Deep Reinforcement Learning without Exploration, August 2019. arXiv:1812.02900 [cs, stat].

[84] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative Q-Learning for Offline Reinforcement Learning, August 2020. arXiv:2006.04779 [cs, stat].

[85] Ilya Kostrikov, Ashvin Nair, and Sergey Levine. Offline Reinforcement Learning with Implicit Q-Learning, October 2021. arXiv:2110.06169 [cs].

[86] Zhendong Wang, Jonathan J. Hunt, and Mingyuan Zhou. Diffusion Policies as an Expressive Policy Class for Offline Reinforcement Learning, August 2023. arXiv:2208.06193 [cs, stat].

[87] Radford M. Neal. MCMC Using Hamiltonian Dynamics. In *Handbook of Markov Chain Monte Carlo*, pages 113–162. Chapman and Hall/CRC, New York, 1 edition, May 2011. ISBN 978-0-429-13850-8. doi: 10.1201/b10905-6.

[88] Michael Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement Learning with Augmented Data, November 2020. arXiv:2004.14990 [cs, stat].

[89] Daesol Cho, Dongseok Shim, and H. Jin Kim. S2P: State-conditioned Image Synthesis for Data Augmentation in Offline Reinforcement Learning, September 2022. arXiv:2209.15256 [cs].

[90] Sergey Levine. Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review, May 2018. arXiv:1805.00909 [cs, stat].

[91] Chenjun Xiao, Yifan Wu, Chen Ma, Dale Schuurmans, and Martin Müller. Learning to Combat Compounding-Error in Model-Based Reinforcement Learning, December 2019. arXiv:1912.11206 [cs, stat].

[92] Wenhao Li. Efficient Planning with Latent Diffusion, September 2023. arXiv:2310.00311 [cs].

[93] Wei Xiao, Tsun-Hsuan Wang, Chuang Gan, and Daniela Rus. SafeDiffuser: Safe Planning with Diffusion Probabilistic Models, May 2023. arXiv:2306.00148 [cs, eess].

[94] Zhixuan Liang, Yao Mu, Mingyu Ding, Fei Ni, Masayoshi Tomizuka, and Ping Luo. AdaptDiffuser: Diffusion Models as Adaptive Self-evolving Planners, May 2023. arXiv:2302.01877 [cs].

[95] Huayu Chen, Cheng Lu, Chengyang Ying, Hang Su, and Jun Zhu. Offline Reinforcement Learning via High-Fidelity Generative Behavior Modeling, February 2023. arXiv:2209.14548 [cs].

[96] Cheng Lu, Huayu Chen, Jianfei Chen, Hang Su, Chongxuan Li, and Jun Zhu. Contrastive Energy Prediction for Exact Energy-Guided Diffusion Sampling in Offline Reinforcement Learning, May 2023. arXiv:2304.12824 [cs].

[97] Monica Welfert, Gowtham R. Kurri, Kyle Otstot, and Lalitha Sankar. Addressing GAN Training Instabilities via Tunable Classification Losses, October 2023. arXiv:2310.18291 [cs, math, stat].

[98] Monica Welfert, Kyle Otstot, Gowtham R. Kurri, and Lalitha Sankar. (α_d,α_g)-GANs: Addressing GAN Training Instabilities via Dual Objectives, May 2023. arXiv:2302.14320 [cs, math, stat].

[99] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, August 2018. arXiv:1801.01290 [cs, stat].

[100] Aapo Hyvarinen. Estimation of Non-Normalized Statistical Models by Score Matching.

[101] Yang Song, Sahaj Garg, Jiaxin Shi, and Stefano Ermon. Sliced Score Matching: A Scalable Approach to Density and Score Estimation, June 2019. arXiv:1905.07088 [cs, stat].

[102] Pascal Vincent. A Connection Between Score Matching and Denoising Autoencoders. *Neural Computation*, 23(7):1661–1674, July 2011. ISSN 0899-7667, 1530-888X. doi: 10.1162/NECO_a_00142.

A Gradient Backpropagation - Derivation

Let us define the forward diffusion process as:

$$q(\tau_t|\tau_{t-1}) = \mathcal{N}(\tau_t; \sqrt{1-\beta_t}\tau_{t-1}, \beta_t\mathbb{I}) \tag{59}$$

where t denotes the diffusion step. Using the reparameterization trick, and defining $\alpha_t = 1 - \beta_t$ and $\hat{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, we have that:

$$\begin{aligned} \tau_t &= \sqrt{\alpha_t}\tau_{t-1} + \sqrt{1-\alpha_t}\epsilon_{t-1} && (60) \\ &= \sqrt{\bar{\alpha}_t}\tau_0 + \sqrt{1-\bar{\alpha}_t}\epsilon && (61) \end{aligned}$$

Thus, it follows that:

$$\tau_{t-1} = \frac{1}{\sqrt{\alpha_t}}\tau_t - \frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}}\epsilon_{t-1}\tau_0 = \frac{1}{\sqrt{\alpha_t}}\tau_t - \frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}}\epsilon \tag{62}$$

We aim to learn an approximation $p_\theta(\tau_{t-1}|\tau_t) = \mathcal{N}(\tau_{t-1}; \mu_\theta(\tau_t, t), \Sigma_\theta(\tau_t, t))$ to $q(\tau_{t-1}|\tau_t, \tau_0)$. As shown in [27], this is equivalent to predicting:

$$\mu_\theta(\tau_t, t) = \frac{1}{\alpha_t}(\tau_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_t) \tag{63}$$

It is shown in [27] that one can instead parameterise ϵ_θ and thus predict:

$$\mu_\theta(\tau_t, t) = \frac{1}{\alpha_t}\left[\tau_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\tau_t, t)\right] \tag{64}$$

Once $\epsilon_\theta(\tau_t, t)$ has been learnt, one can obtain samples via the equation:

$$\tau_{t-1} = \frac{1}{\alpha_t}\left[\tau_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\tau_t, t)\right] + \sigma_t z \tag{65}$$

Based on [26], we can guide this sampling process of a prior towards sampling from a posterior distribution obtained by the likelihood $p_\phi(y|\tau_t)$ by performing each step of reverse process as:

$$\tau_{t-1} = \frac{1}{\alpha_t}\left[\tau_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\tau_t, t)\right] + s\Sigma\nabla_{\tau_t}\log p_\phi(y|\tau_t) + \sigma_t z \tag{66}$$

where s is a learning rate-like parameter.

Referring to the control-as-inference graphical model [90], one can express the likelihood of a time-step being optimal as

$$p(O_t = 1) = \exp(r(s_t, a_t)) \tag{67}$$

Under this assumption, [36] shows that:

$$\nabla_\tau \log p_\phi(O_{1:T}|\tau) = \nabla_\tau \sum_{t=0}^{T} r(s_t, a_t) \tag{68}$$

$$= \nabla_\tau R(\tau) \tag{69}$$

where $R(\tau)$ is the return of the entire trajectory τ.

Thus, the guided reverse process is:

$$\tau_{t-1} = \frac{1}{\alpha_t}\left[\tau_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\tau_t, t)\right] + s\Sigma\nabla_{\tau_t}R(\tau_t) + \sigma_t z \tag{70}$$

The loss function is defined as a mean squared error such that:

$$L = ||\tau_e - \tau_0||_2^2 \tag{71}$$

Under this loss function, its gradient with respect to the reward model's parameters ϕ is:

$$\nabla_\phi L = \nabla_\phi \left|\left|\tau_e - \frac{1}{\sqrt{\alpha_1}}\tau_1 + \frac{1-\alpha_1}{\sqrt{\alpha_1}\sqrt{1-\bar{\alpha}_1}}\epsilon_\theta(\tau_1, 1) + s\Sigma\nabla_{\tau_1}\log p_\phi(y|\tau_1)\right|\right|_2^2 \tag{72}$$

$$= 2||\cdot||\left[\tau_e - \frac{1}{\sqrt{\alpha_1}}\frac{\partial\tau_1}{\partial\phi} + \frac{1-\alpha_1}{\sqrt{\alpha_1}\sqrt{1-\bar{\alpha}_1}}\frac{\partial\epsilon_\theta(\tau_1, 1)}{\partial\phi} + s\Sigma\frac{\partial}{\partial\phi}\frac{\partial}{\partial\tau_1}\log p_\phi(y|\tau_1)\right] \tag{73}$$

$$= 2||\cdot||\left[\tau_e - \frac{1}{\sqrt{\alpha_1}}\frac{\partial\tau_1}{\partial\phi} + \frac{1-\alpha_1}{\sqrt{\alpha_1}\sqrt{1-\bar{\alpha}_1}}\frac{\partial\epsilon_\theta(\tau_1, 1)}{\partial\phi} + s\Sigma\frac{\partial}{\partial\phi}\frac{\partial}{\partial\tau_1}R(\tau_1)\right] \tag{74}$$

B Environments

B.1 Maze2D: U-Maze and Large Maze

Both the U-Maze and the Large Maze are part of the Maze2D collection of environments [40]. They share the same state and action spaces. The only difference between the two environments is the maze configuration. The respective maze configurations are shown in Figure 19.

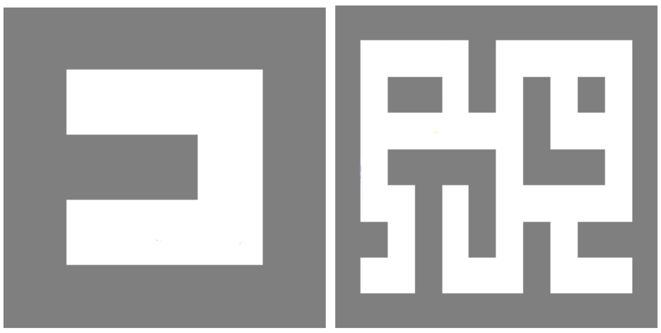

Figure 19: Layout for the U-Maze (left) and Large Maze (right) environments. Walls are shown in grey. Sections in which agents can move are shown in white. X-coordinate is horizontal, Y-coordinate is vertical. Origin at top left.

The Maze2D environments consist of moving a force-actuated ball in a 2D (coordinates x and y) environment to reach some user-defined goal. The state and action spaces are shown in Table 6 and Table 7 respectively.

Table 6: State/observation space of the Maze2D environments. "Index" refers to the index of the respective observation in the state array expected by the environment.

Index	Action	Min value	Max value
0	x-axis coordinate	-∞	∞
1	y-axis coordinate	-∞	∞
2	Velocity in the x-axis coordinate	-∞	∞
3	Velocity in the y-axis coordinate	-∞	∞

Table 7: Action space of the Maze2D environments. "Index" refers to the index of the respective action in the action array expected by the environment.

Index	Action	Min value	Max value
0	Force in the x-axis direction	-1	1
1	Force in the y-axis direction	-1	1

Note that while the coordinate elements of the state vector are each allowed to take any value in the range $(-\infty, \infty)$, the limits of each individual maze mean the agent sticks to specific ranges of coordinates, seen in Table 8.

Table 8: Coordinate limits for each of the considered Maze2D environments.

Environment	x-axis	y-axis
U-Maze	(0,5)	(0,5)
Large Maze	(0,12)	(0,9)

Episodes are usually taken to have a maximum number of 300 steps. Typical implementations terminate an episode when the agent reaches its goal state. However, as we do not consider one specific goal state, we run all episodes to 300 steps. The start state can be selecte by the user, in any of the allowed state vector elements' allowed ranges.

B.2 HalfCheetah

The HalfCheetah environment [40] consists of controlling a 2 dimensional robot, with 8 joints and 9 links. The agent can control the amount of torque applied to 6 of its joints (its torso and head are fixed) across its (front and back) thighs, feet and shins. The state of the agent consists of the positions and velocities of its different body parts. The state and action spaces are shown in Tables 9 and 10

Table 9: State/observation space of the HalfCheetah environment. "Index" refers to the index of the respective observation in the state array expected by the environment.

Index	Action	Min value	Max value
0	z-coordinate of front tip	-∞	∞
1	Angle of front tip	-∞	∞
2	Angle of back thigh rotor	-∞	∞
3	Angle of back shin rotor	-∞	∞
4	Velocity of the front tip in the x-axis coordinate	-∞	∞
5	Velocity of the front tip in the x-axis coordinate	-∞	∞
6	Angular velocity of front tip	-∞	∞
7	Angle of front foot rotor	-∞	∞
8	x-coordinate of front tip	-∞	∞
9	y-coordinate of front tip	-∞	∞
10	Angular velocity of front tip	-∞	∞
11	Angular velocity of back thigh rotor	-∞	∞
12	Angular velocity of back shin rotor	-∞	∞
13	Angular velocity of back foot rotor	-∞	∞
14	Angular velocity of front thigh rotor	-∞	∞
15	Angular velocity of front shin rotor	-∞	∞
16	Angular velocity of front foot rotor	-∞	∞

Table 10: Action space of the HalfCheetah environment. "Index" refers to the index of the respective action in the action array expected by the environment.

Index	Action	Min value	Max value
0	Torque applied to the back thigh rotor	-1	1
1	Torque applied to the back shin rotor	-1	1
2	Torque applied to the back foot rotor	-1	1
3	Torque applied to the front thigh rotor	-1	1
4	Torque applied to the front shin rotor	-1	1
5	Torque applied to the front foot rotor	-1	1

Figure 20 shows frames of an HalfCheetah trajectory.

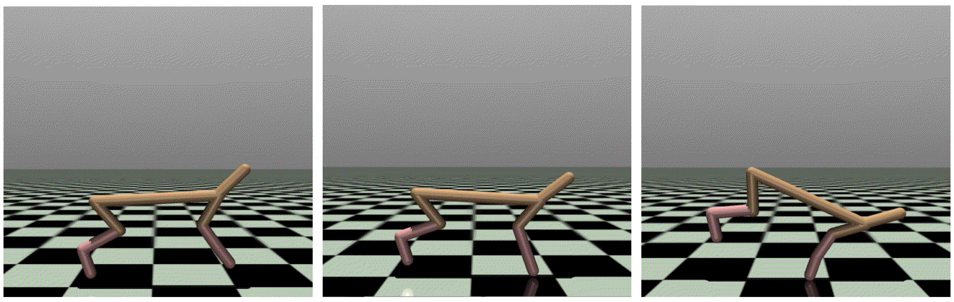
Figure 20: Example frames from a trajectory in the HalfCheetah environment. The agent initially moves forward, but eventually heads towards a fall (last frame).

The agent's goal is to move forward as much as possible, while prioritising actions with small norm. More specifically, the reward given by the environment can be described via the terms

- Forward Reward: $x_{\text{after}} - x_{\text{before}}$, where dt is the frame rate parameter (usually set to 0.05), and xx_{after} and xx_{before} represent respectively the actions at $t + 0.05$ and t.
- Action Penalty: 0.1 * sum(action^2) for each action in the action sapce vector. It is thus a term that penalises actions with large norm.

The reward is given by subtracting the action penalty from the forward reward. The initial state is the state vector initialised with all elements 0, with some added noise.

C Training Details

C.1 Base Diffuser Training

We first show the main diffusion settings. This defines the Diffuser architecture for each environment.

Table 11: Diffusion settings for different environments.

Environment	**Planning Horizon**	**Number of Diffusion Steps**
U-Maze	128	64
Large Maze	384	256
HalfCheetah	4	20

We then show the values of hyperparameters used for training of the base Diffuser in each environment. These parameters are named according to the Diffuser framework.

Table 12: Parameters for the base Diffuser training in each environment.

Parameter	**U-Maze**	**Large Maze**	**HalfCheetah**
Learning rate	2e-4	2e-4	2e-4
Batch size	32	32	32
Training steps	2e6	2e6	1e6
EMA decay	0.995	0.995	0.997
Loss	L2	L2	L2

C.2 Training of Reward Model

We show the values of hyperparameters (except for learning rate) used for learning of a reward model (our method). Several of these parameters are named according to the Diffuser framework.

Table 13: Parameters for the learning of a reward model in each environment.

Parameter	**U-Maze**	**Large Maze**	**HalfCheetah**
MLP Hidden Dimensions	(384,128,64)	(1024,512,128)	(64,32,16)
MLP Activation	ReLU	ReLU	ReLU
Batch size	270	32	256
Epochs	100	500	250
Guidance scale	0.1	0.1	0.001
Discount factor	0.995	0.995	0.99
Scale grad by standard deviation	True	True	True
t_stopgrad	2	2	0
Number of guide steps	2	2	2

Regarding the learning rate, its value was (informally) tuned depending on the distance metric M used for each environment.

Table 14: Learning rate used for reward learning using different distance metrics M in different environments.

Environment	**Distance Metric**	**Learning Rate**
U-Maze	MSE	5e-4
	MMD Gauss	2e-3
	MMD Matern	2e-3
Large Maze	MSE	1e-3
	MMD Gauss	2e-3
	MMD Matern	2e-3
HalfCheetah	MSE	2e-3
	MMD Gauss	5e-4
	MMD Matern	1e-3

D Additional Results

D.1 U-Maze Base Diffusion

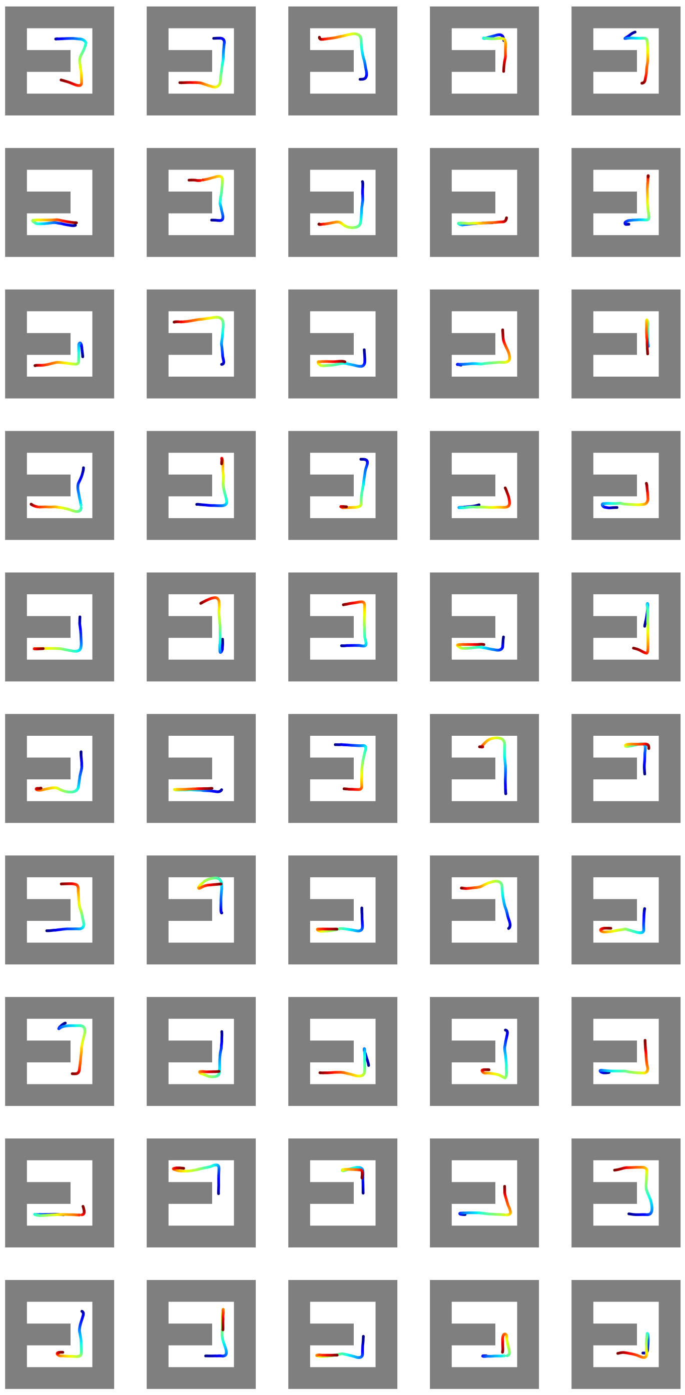

Figure 21: Example datapoints from the reward-agnostic dataset used to train the Base Diffuser in the U-Maze environment. Trajectories shown in colour, from blue (start) to red (end).

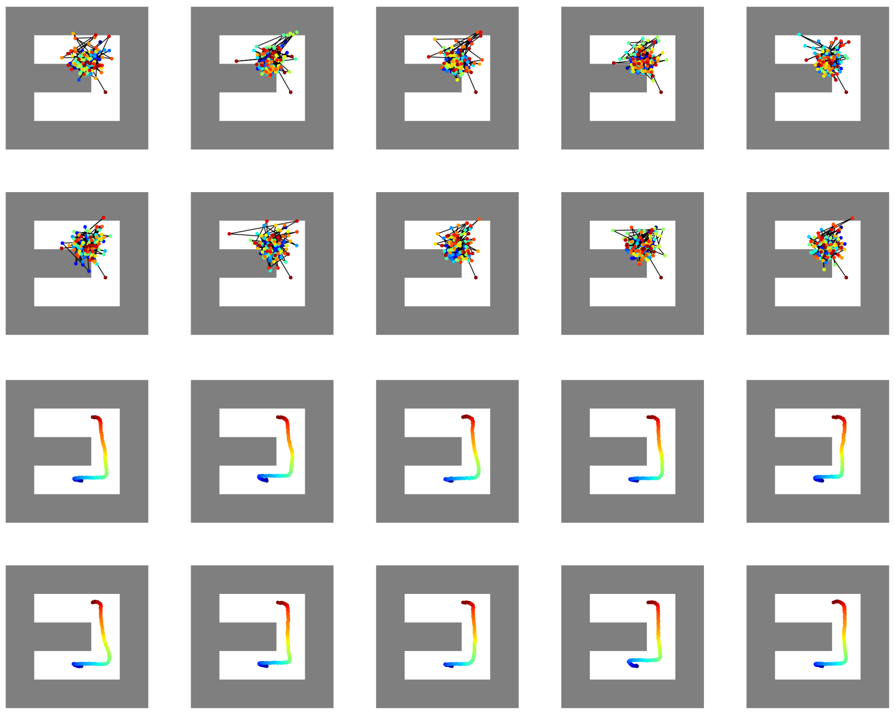

Figure 22: Example diffusion outputs after 1000 (top 2 rows) and 100000 (bottom 2 rows) training steps. Trajectories shown in colour, from blue (start) to red (end).

D.2 Large Maze Base Diffusion

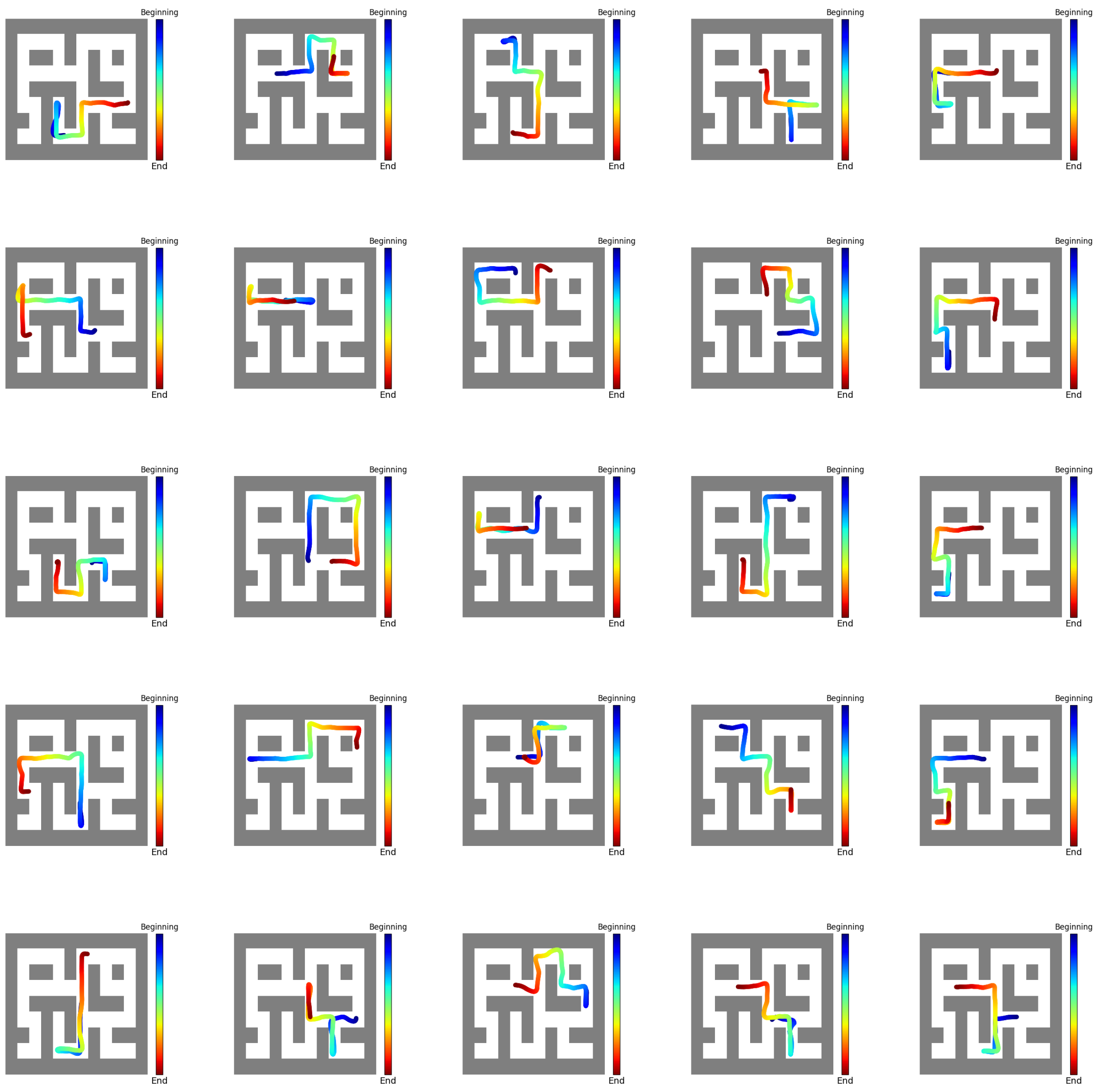

Figure 23: Example datapoints from the reward-agnostic dataset used to train the Base Diffuser in the Large Maze environment. Trajectories shown in colour, from blue (start) to red (end).

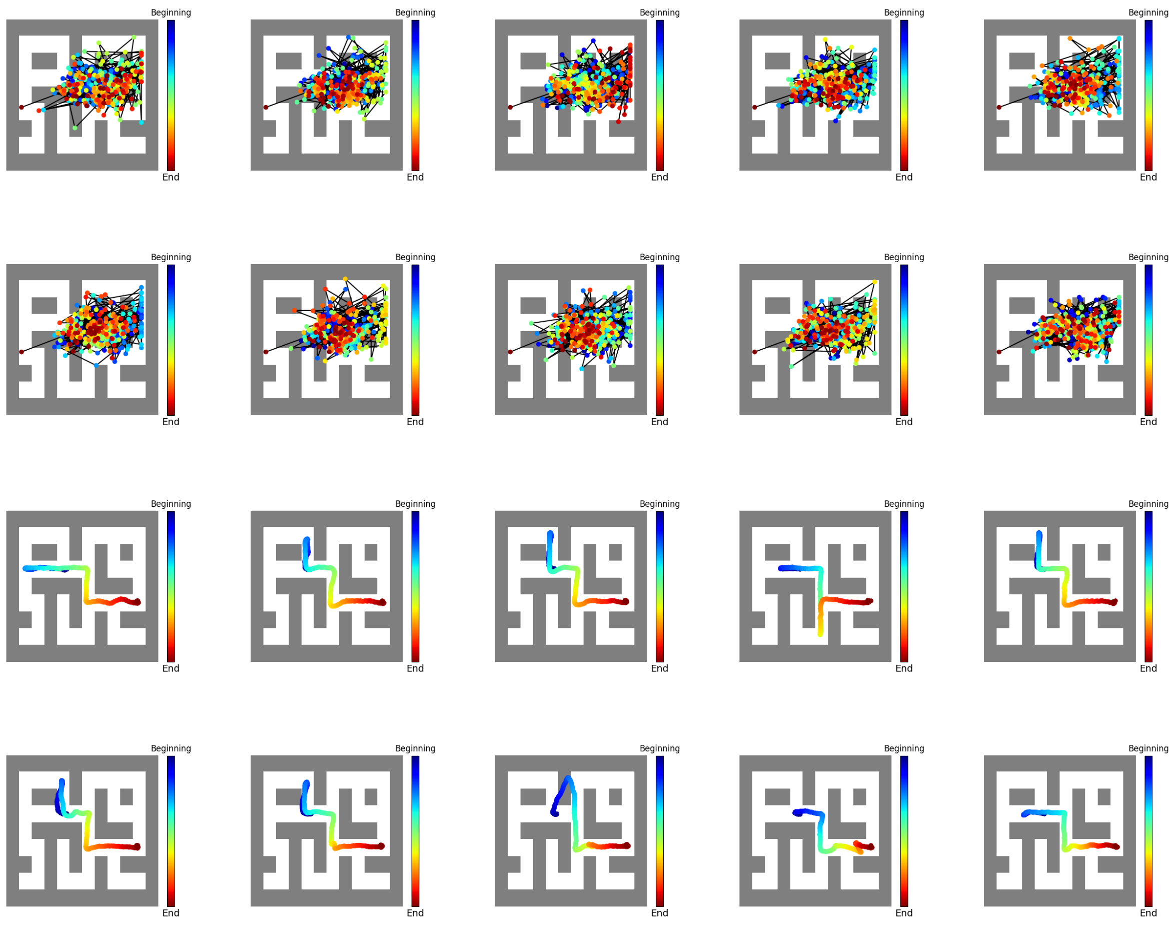

Figure 24: Example diffusion outputs after 1000 (top 2 rows) and 743000 (bottom 2 rows) training steps. Trajectories shown in colour, from blue (start) to red (end).

E Background - Extra Information

E.1 Reward Function Similarity Metrics

Gleave et al [39] also introduce NPEP as a baseline and analyse where it fails. Nearest Point in Equivalence Class (NPEC) is defined as the minimum L^p distance between equivalence classes.

Definition 10 (L^p Distance) *Let $\mathcal{D}$ be the coverage distribution over transitions (s, a, s') and let $p \geq 1$. The L^p distance between two reward functions $\mathcal{R}_A$ and $\mathcal{R}_B$ is the L^p norm of the difference between them, that is $D_{L^p}(\mathcal{R}_A, \mathcal{R}_B) = \left[\mathbb{E}_{(s,a,s')\sim\mathcal{D}}|\mathcal{R}_A(s,a,s') - \mathcal{R}_B(s,a,s')|^p\right]^{\frac{1}{p}}$.*

Definition 11 (Reward Function equivalence) *Two reward functions $\mathcal{R}_A$ and $\mathcal{R}_B$ are said to be equivalent, $\mathcal{R}_A \equiv \mathcal{R}_B$, for fixed tuple $(\mathcal{S}, \mathcal{A}, \gamma)$ iff there exists a constant $\lambda > 0$ and bounded potential function $\Phi : \mathcal{S} \to \mathbb{R}$ such that for any and all $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$:*

$$\mathcal{R}_B(s,a,s') = \lambda\mathcal{R}_A(s,a,s') + \gamma\Phi(s') - \Phi(s). \tag{75}$$

Definition 12 (NPEC) *Let $\mathcal{D}$ be the coverage distribution over transitions (s, a, s') and let $p \geq 1$ be the distance power. Let $D^U_{NPEC}(\mathcal{R}_A, \mathcal{R}_B) = \inf_{R' \equiv \mathcal{R}_A} D_{L^p,\mathcal{D}}(\mathcal{R}', \mathcal{R}_\mathcal{B})$. The nearest point in equivalence class (NPEC) between two reward functions $\mathcal{R}_A$ and $\mathcal{R}_B$ is defined as $D_{NPEC}(\mathcal{R}_A, \mathcal{R}_B) = D^U_{NPEC}(\mathcal{R}_A, \mathcal{R}_B)/D^U_{NPEC}(0, \mathcal{R}_B)$ when $D^U_{NPEC}(\mathcal{R}_A, \mathcal{R}_B) \neq 0$, and given by $D^U_{NPEC}(\mathcal{R}_A, \mathcal{R}_B) = 0$ otherwise.*

NPEC is not a pseudo-metric. Furthermore, while it is invariant to potential shaping [39], it is not robust to the choice of coverage function.

E.2 Score-Based Generative Models

E.2.1 Score Matching

Score matching [100] was first introduced as a new way to estimate statistical models where the distribution can only be evaluated up to a normalizing constant. The general idea is that for a given energy model

$$p(x) = \frac{1}{Z}e^{-f(x)} \tag{76}$$

the score function does not involve Z, that is

$$\nabla_x \log p(x) = -\nabla_x f(x) \tag{77}$$

Thus, one can model this score function without knowledge of the distribution's normalising constant. Once this model has been learnt, sampled from the distribution $p(x)$ can be obtained using Langevin dynamics.

A model $s(x, \theta)$ of this score function can be found by minimising

$$J(\theta) = \mathbb{E}_{p(x)} \left[||\nabla_x \log p(x) - s(x, \theta)||_2^2 \right] \tag{78}$$

However, computing this still requires access to the score function of the data $\nabla_x \log p(x)$. To circumvent this issue, several methods, under the umbrella term of score matching methods, have been proposed to calculate this divergence, and thus allow for the learning of score functions.

The original score matching method [100] shows that this L2-distance can be estimated with a formula using only derivatives of the model score function, thus not requiring knowledge of the data score function. More concretely, it shows that:

$$J(\theta) = C + \mathbb{E}_{p(x)} \left[\sum_{i=1}^{n} \frac{\partial s_i(x, \theta)}{\partial x_i} + \frac{1}{2} s_i(x, \theta)^2 \right] \tag{79}$$

where C is a constant with respect to θ, and thus can be disregarded in the minimisation process. However, this method is not scalable to deep networks and high dimensional data [101] due to the first term, which consists of the trace of the Jacobian of the probability distribution.

E.2.2 Variants of Score Matching

Thus, several methods have been introduced that avoid the computation of the trace term. Firstly, sliced score matching [101] aims to minimise an objective of the same form, but approximates the trace term using random projections, such that:

$$J(\theta, p_v) = \mathbb{E}_{p_v, p(x)} \left[v^T \nabla_x s(x, \theta) v + \frac{1}{2} ||s(x, \theta)||_2^2 \right] \tag{80}$$

where the random projection term can be computed using forward mode auto-diff. Alternatively, denoising score matching [102] aims to minimise a different objective. Firstly, the data x is perturbed with noise according to $q_\sigma(x'|x)$, and then the method performs score matching to estimate the score of the perturbed distribution $q_\sigma(x') = \int q_\sigma(x'|x)p(x)dx$. This is shown to be equivalent to minimising:

$$J(\theta) = \frac{1}{2} \mathbb{E}_{q_\sigma(x'|x), p(x)} \left[||s(x', \theta) - \nabla_{x'} \log q_\sigma(x'|x)||_2^2 \right], \tag{81}$$

where the gradient term is easily available for simple choices of noise distributions such as Gaussian noise.

E.2.3 Noise-conditioned score networks (NCSNs)

Score function estimation is often severely inaccurate in low density regions, where few datapoints are available. These inaccuracies very easily lead to inaccurate Langevin dynamics (the initial point is likely to be in a low density area), thus severely hindering the quality of the samples obtained. One possible solution is to perturb data with noise, and train the score model on a noisy version of the data, as done for denoising score matching [102]. Since deciding on one single noise value is not straightforward, one can instead use multiple scales of noise simulateously [81]. Thus, the data is perturbed with different levels σ_i of noise, and the score model is trained on each of the noise levels such that $s_\theta(x, i) \approx \nabla_x \log p_{\sigma_i}(x)$ for each noise level $i = 1, ..., N$. The training objective then becomes a weighted sum of L2-norms such that:

$$J(\theta) = \sum_{i=1}^{N} \lambda(i) \mathbb{E}_{p_{\sigma_i}(x)} \left[||\nabla_x \log p_{\sigma_i}(x) - s_\theta(x, i)||_2^2 \right] \tag{82}$$

By minimising it, we learn $s_\theta(x, i)$, a noise-conditioned score network [81]. After training, we can use annealed Langevin dynamics [81] on this model to obtain samples from the probability distribution.