IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

Exploring state representations for offline reinforcement learning

Supervisor:
Yiannis Demiris

Author:
Aimilios Hatzistamou

Second Marker:
Deniz Gunduz

PhD Advisor:
Angelos Filos

Submitted in partial fulfillment of the requirements for the M.Eng. degree in Electrical and Electronic Engineering of Imperial College London

January 2021

Abstract

Offline methods for reinforcement learning (RL) have the potential to turn large available datasets into strong decision-making systems and enable the use of RL in domains where learning through online interaction is too dangerous, costly or unethical. Moreover, online RL algorithms have been shown to benefit from compact and expressive state representations. To learn better representations, one can use self-supervised auxiliary objectives: they can help agents learn better policies, with fewer interactions, and with improved robustness against task-irrelevant information that is distracting. Recent research in offline methods has focused on algorithms to address the limitations of offline RL, such as distribution shift. We tackle the problem from a different angle and aim to answer the question: can current state representation learning (SRL) methods be used to learn better policies on offline data? Through a set of experiments conducted on a new proposed benchmark (classic control with distractions) and Atari games, we show offline methods can perform drastically better in certain settings when combined with auxiliary SRL objectives, but highlight their sensitivity to tuning. We also aim to provide the reader with an understanding of some of the challenges and open questions that surround offline SRL.

Acknowledgements

I would like to express my sincere gratitude to:

- **Yiannis Demiris** for his invaluable advice and encouragement regarding all aspects of this thesis.
- **Angelos Filos** for his unwavering support, invaluable guidance, and willingness to share his knowledge and expertise in RL.
- **Thanos Vlontzos** for our interesting conversations and helping me get unstuck from several encountered problems.
- **Rodrigo Chacón** for his assistance in resolving issues with lab machines.
- **My family** for being so supportive over the course of the past pandemic year.

Contents

1 Introduction **2**
1.1 Motivation 2
1.2 Contributions 4
1.3 Report layout 5

2 Preliminaries **6**
2.1 Reinforcement learning 6

3 Background **9**
3.1 State abstraction 10
3.1.1 Types of abstractions 11
3.1.2 State representation learning 13
3.1.3 Auxiliary objectives 13
3.2 Offline reinforcement learning 15
3.2.1 Overview 15
3.2.2 Distributional shift 17

4 Algorithms and representations **19**
4.1 RL algorithms 19
4.1.1 Behavioural Cloning (BC) 19
4.1.2 Deep Q-Networks (DQN) 20
4.2 SRL methods 23
4.2.1 Reconstruction vs reward 23
4.2.2 Representations considered 25
4.2.3 Reconstruction-based methods 26
4.2.4 Reward-based methods 28
4.2.5 Summary 31

5 State representation in offline RL **32**
5.1 Experimental setup 32
5.1.1 Research software & tools 32
5.1.2 Hardware 33
5.1.3 Code structure 34
5.1.4 Optimizations 35
5.2 Classic control with distractions 35
5.2.1 Cartpole 36

5.2.2 Distractors design 36
5.3 Online vs Offline Gap 40
5.3.1 Constructing an offline dataset 40
5.3.2 N-step bootstrapping considerations 42
5.3.3 Offline performance 44
5.3.4 Assessing distribution shift 45
5.4 Learning with distractions 46
5.4.1 Effect of distractors on Behavioral Cloning 47
5.5 Learning with explicit state representations 48
5.5.1 Network architecture 48
5.5.2 Considerations when training SRL methods offline 48
5.5.3 Tuning sensitivity 50
5.5.4 Convergence failures 52
5.5.5 Benchmarking results 53
5.5.6 Evaluating the representation space 53
5.5.7 Summary 55

6 Large-scale experiments 57
6.1 Atari 2600 offline 57
6.1.1 Game selection 58
6.2 Benchmarking results 58
6.3 Additional evaluation 59
6.3.1 Pong with distractions 59
6.3.2 Results 60

7 Conclusion 62
7.1 Future work 63

Appendices 70

A Manipulating deformable objects through keypoint representations 71
A.1 Environments 72
A.2 Learning keypoint representations 74
A.2.1 Building a diverse dataset from GymCloth 75
A.2.2 Training the Transporter architecture 76
A.3 Manipulating fabric from keypoints 79
A.3.1 Model-free attempt 80
A.3.2 Imitation learning 81

B Offline SRL 84
B.1 Distractor benchmark 84
B.1.1 Cartpole data generation 84
B.1.2 Online benchmarking 85
B.1.3 BC and offline DQN parameters 85
B.1.4 Representation hyperparameters 86
B.2 Atari experiments 86

Chapter 1

Introduction

A newborn knows nothing about the world. Yet, by the time human beings grow to become adults, they are prepared to set off by themselves and live a fairly complex existence. While the biological processes underlying our cognitive development are largely still a mystery to us, one thing is certain: *we learn from experience*. Over time, we interact with the world and process signals from our senses. We observe the effects of the decisions we make, as well as the effects of decisions made by others, and learn from them. Our environment's response to our actions feeds back into our brain's reward systems, and we learn to avoid our mistakes, replicate our successes. This setting whereby an agent interacts with its environment to achieve an objective has been formalized in Reinforcement Learning (RL).

To learn by trial-and-error involves processing large amounts of information coming from our bodies' sensors. Our eyes, for instance, provide us with a valuable feedback signal that we can use to inform our decisions and learn from them. Researchers in artificial intelligence strive to close the gap between human and machine and consequently look to build systems that can learn about the world through high-dimensional sensory inputs like vision and speech. We would like an intelligent machine to understand the state of its surrounding, and be able to drive a car, assist a disabled person, operate a smart prosthetic limb, etc. However, many tasks that appear trivial to us—such as manipulating a piece of cloth—often require significant engineering efforts in perception and control, if they are at all possible.

1.1 Motivation

Learning to control agents directly from high-dimensional inputs is one of the long-standing challenges of RL. While learning from images is critical for many real-world applications, it is computationally expensive. High-resolution images are made up of millions of pixels and RL algorithms do not work well with such high-dimensional data. Until recently, RL applications operating on these domains heavily relied on hand-crafted features and the quality of the feature representation. The advent of neural networks has led to Deep Reinforcement Learning methods (Deep RL) that can better deal with complex tasks (Lample and Chaplot, 2016; Lillicrap et al., 2019). Nevertheless, this success has been limited and, as a result, learning to interact from visual data has been primarily restricted to tasks where hundreds of thousands of agent actions can be taken in simulation. Real-life visual tasks face the additional challenge that our world is unstructured and full of distractions. To address these problems, one of the most promising avenues being explored is *abstraction*.

As an agent explores and interacts with its environment, it is often desirable that it builds internal representations and beliefs—just as we, humans, can build mental representations. Abstract reasoning is widely thought to be central to our ability to solve problems. At any given time, we hold many concepts in mind: e.g. "the door is closed" or "John is calling". Abstract reasoning makes it possible for us to make higher-level decisions e.g. to walk towards someone, based on the noisy signals coming from our photo-receptors, ears, etc. Using the aforementioned example of manipulating a piece of cloth, it is possible to map an agent's visual observation (e.g. of a crumpled piece of cloth) to a low dimensional underlying object state that efficiently captures all the structure information needed to complete a task (e.g. flattening the cloth on a surface). Often, leveraging the right abstractions can achieve higher levels of cognition.

While there are many ways to build representations, they are not all equally effective. Some representations can introduce important structure and useful biases. For example, it may be important to ensure that a representation can model the causal relationships that govern the world (Li et al., 2020; Zhang et al., 2020b, 2019). One of the most common approaches to learning and evaluating a representation is ensuring that it incorporates all the information required to reconstruct an agent's observation (Lesort et al., 2018)—likening representations to the problem of compression. Recent work (Zhang et al., 2020a), however, suggests that this may not always be the best approach. In our real and complex world, we receive a large amount of information—not all of which, is relevant to achieving our objectives. Think of the problem of driving a car along a highway. What we observe on the road directly affects the steering decisions we will want to make, but the exact structure and position of the clouds in the sky should not. A representation learned from a reconstruction objective will nonetheless give equal weight to everything the agent sees, no matter the relevance. It is therefore important to consider alternative methods for learning representations, methods that can capture only *task-relevant* information—potentially leading to more robust and sample-efficient training.

An additional obstacle for the widespread adoption of RL has been the fact that the agent needs to collect experience by iteratively interacting with the environment. In many domains, such interactions can be infeasible at scale (e.g. robotics), unethical or dangerous (e.g., healthcare or self-driving cars). Moreover, even when it is possible to interact, we may still like to leverage available datasets of past decisions. In recent years, *Offline Reinforcement Learning* (Levine et al., 2020)—a branch of RL that addresses these concerns—has grown in popularity. Framing the RL problem as a supervised-learning problem, it aims to capitalize on the tremendous success of data-driven methods that perform better as they are trained with more and more data. Offline RL is particularly appealing in the context of Deep RL with high-dimensional inputs, as deep networks often require large datasets.

With both abstractions and the offline RL setting showing strong promise, the focus of this Master's thesis will be to **evaluate whether offline RL can benefit from explicit representation models**—a question that is still largely unexplored to this day.

1.2 Contributions

Recent research in offline methods has focused on algorithms to address the limitations of offline RL, such as distribution shift. Recent work by Yang and Nachum (2021) investigated whether representation methods can help offline RL algorithms to learn directly from state.

To the best of our knowledge, this work is the first to explore the use of representations in offline RL tasks with pixel observations, and with distractions. The main contributions can be summarized as follows:

1. **Distractor benchmark** — we propose a benchmark environment to evaluate the robustness of different SRL methods against distractions in the observation space.

2. **Evaluation of several SRL methods on offline tasks** — we have run numerous experiments on our distractor benchmark and the Atari RL testbed (Bellemare et al., 2013) to assess the utility of explicit representation objectives in the offline setting. Our empirical evidence suggests that introducing a jointly-trained representation loss can dramatically improve the performance of offline policy learning algorithms.

3. **Insights and discussions** — we perform several studies in order to address open questions around the training of representations for offline RL. We also experimentally illustrate some of the challenges with offline RL (dataset sensitivity, distribution shift) that are relevant when training representations on offline datasets.

4. **Contrastive DBC (CDBC)** — we implement CDBC, a new reward-based representation learning method. Combining recent work by Zhang et al. (2020a) and Agarwal et al. (2021), CDBC uses a contrastive loss to embed the bisimulation metric (Ferns and Precup, 2014), effectively grouping states that are behaviorally equivalent. We show that this representation can outperform direct embedding of the bisimulation metric in some settings, and is more stable to train offline.

1.3 Report layout

Chapter 2 introduces fundamental concepts from Reinforcement Learning that are relevant throughout the report. The background and ideas from state abstraction and offline RL that drive our research are covered in the following chapter.

In Chapter 4, we select which state representation learning methods to benchmark, and also discuss some of their implementational details. We further explain the RL algorithms that we will use, in the context of offline learning.

Chapter 5 presents our main results, after first introducing the experimental setup and our new classic control benchmark. This chapter also examines various considerations regarding offline SRL, as well as issues we encountered that illustrate challenges associated with offline RL. Then, in Chapter 6, we discuss the larger-scale experiments we conducted on offline Atari data.

Finally, in Chapter 7, we summarize our work and results, highlighting a few potential directions for future work.

Chapter 2

Preliminaries

We first introduce some of the fundamental concepts from Reinforcement Learning that will be relevant throughout this work. The reader is invited to skip this chapter if already familiar with the RL paradigm.

2.1 Reinforcement learning

In AI and control, a wide range of tasks can be formulated as sequential decision-making processes. Reinforcement Learning (RL) is a subfield of machine learning that deals with such learning problems. In that set-up, we refer to the decision-maker (e.g. a humanoid robot) as the *agent* and everything outside of it as the *environment*. At every timestep t, the agent receives an observation of the environment $s_t \in \mathcal{S}$ and takes an action $a_t \in \mathcal{A}$ according to its behavioural policy π. The environment then provides a reward r_{t+1}, giving the agent useful feedback.
The goal of reinforcement learning is to use the agent's past experience (the history of previous observations, actions and rewards) to learn the optimal policy π^* that maximizes future rewards.

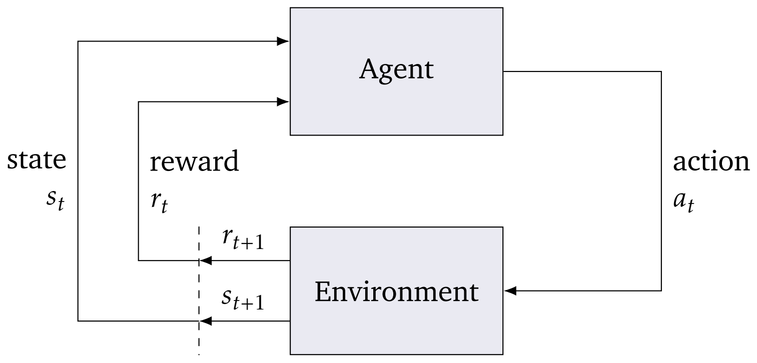


Figure 2.1: The Agent-Environment Interaction (Sutton and Barto, 1998)

An RL task is either episodic and lasts a finite amount of time (finite horizon), or continuing (infinite horizon).

- In episodic tasks, there is a terminal state. We define the return to be $R_t = \sum_{k=t}^{T} r_k$, which corresponds to the total reward accumulated from time t until time T where the terminal state is reached.
- In continuing (infinite horizon) tasks, we typically discount future rewards and look instead to maximize $R_t = \sum_{k=t}^{\infty} \gamma^k r_k$, where $0 < \gamma < 1$ is the discount factor. This ensures we place additional value on reward that is received sooner than later.

Markov Decision Processes A special class of reinforcement learning tasks occurs when the next observation and reward depend exclusively on the current observation and action, i.e.:

$$P(s_{t+1}, r_{t+1} | s_1, a_1, r_1, ..., s_t, a_t, r_t) = P(s_{t+1}, r_{t+1} | s_t, a_t) \tag{2.1}$$

These tasks are called *Markov Decision Processes* (MDP). In MDPs, the current observation summarizes all previous experience and is called the Markov state.
Most RL algorithms assume an MDP and therefore we need to ensure that our state has the Markov property. If an environment is not Markov, we can typically construct a derived MDP that has the Markov property by rolling the last k steps of the environment into a single richer state.
A task may be fully observable, in which case the agent receives a Markov state s_t at every time-step, otherwise the task is called partially observable. In the latter case, the agent must make decisions under uncertainty of the true environment state. As a consequence, optimal behaviour may often include information gathering actions taken to improve the agent's estimate of the current state, so that it can make better decisions.

Assuming a continuing task, we wish to learn the optimal policy π^* such that

$$J(\pi) = \mathop{\mathbb{E}}_{P,\pi} [\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)] \tag{2.2}$$

$$\pi^* = \operatorname*{argmax}_{\pi} J(\pi) \tag{2.3}$$

where P=$\{p(s_0), p(s_{t+1}|s_t, a_t)\}$ denotes the starting state distribution and transition dynamics of the environment. There exists a number of methods to optimize an agent's policy:

- **Policy-based methods**. These methods are arguably the most direct way to find an optimal policy. We do not use any surrogate objectives and directly optimize a parametrized π with respect to the training objective (2.3). In particular, with policy gradient methods we perform gradient ascent with respect to the training objective $J(\pi)$.
- **Value-based methods**. We use a value function that summarizes the long-term consequences of a particular policy. Specifically, $V_\pi(s) = \mathbb{E}_\pi[R_t | s_t = s]$ indicates

the expected return for the agent in state s following policy π. Similarly, the action value function $Q_{\pi}(s,a) = \mathbb{E}_{\pi}[R_t|s_t = s, a_t = a]$ is the expected return after selection action a in state s, and then following policy π. The estimated value function can then be used to optimize the agent's policy. Specifically, if we learn the action value Q_{π^*} function corresponding to π^*, the optimal policy is found by selecting actions greedily with respect to the value function.

- **Model-based methods**. In contrast, model-based approaches try to learn good approximations $\hat{P}, \hat{r}$ to the transition dynamics of the environment as well as the reward function. This is commonly done by training ML models on observed transition $\{s_t, a_t, r_t, s_{t+1}\}$. These models can then be used to approximately solve $\pi^* \approx \operatorname{argmax}_{\pi} \mathbb{E}_{\pi,\hat{P}}[\sum_{t=0}^{\infty} \gamma^t \hat{r}(s_t, a_t)]$.
- **Imitation learning**. These methods aim to learn a policy that mimics human behaviour. To this end we must provide expert examples, also called trajectories, that the agent tries to imitate. Quite often, the challenge lies in obtaining sufficient expert examples to learn the human's policy—which is why building learning algorithms that are sample efficient is critically important.

Finally, we consider the distinction between on-policy and off-policy algorithms. An RL algorithm is considered on-policy if it primarily uses samples generated from the current policy π in order to improve it. In contrast, an off-policy algorithm may use samples from any policy, including past or arbitrary policies.

Definition 2.1.1 (Block MDP). In part of this work, we will work with learning policies that solve MDPs where the agent receives rich pixel observations. In this scenario, it is relevant to introduce additional assumptions in the MDP through the block Markov decision process (BMDP) formalism (Du et al., 2019). The BMDP refers to an environment that is described by a low-dimensional and unobservable latent space $\mathcal{S}$, an action space $\mathcal{A}$ and an observable context space $\mathcal{X}$. Similar to the MDP, the dynamics are described by transition and reward functions, but in addition, we assume the existence of a "context-emission" function $q(x|s)\ \forall s \in \mathcal{S}$, which is such that each context x uniquely determines its generating state s.

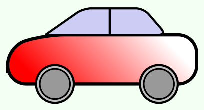
Chapter 3

Background

In this chapter, we discuss some of the background and ideas from state abstraction and offline RL that will be relevant throughout this work. Before diving in, we introduce an example RL scenario that we will repeatedly come back to in order to illustrate the various concepts discussed in later sections.

RL driving scenario

Suppose we possess a humanoid robot. This robot's hardware has been designed such that its sensors mimic human sensing. We place this robot in an automatic transmission vehicle on a highway and load its onboard computer with an RL algorithm.



We would like this robot, the "agent", to learn how to drive from a point A to a point B through trial and error.

The agent is equipped with robust motor controllers, so the algorithm must only learn to produce an **action triplet** (steering θ, brake %, accelerator %).

As driving relies primarily on visual perception, we'll assume the algorithm's only input is a **high-resolution image** received from an eye-level camera.

The agent receives an **extrinsic reward** from the environment, which is designed to reward safe driving, reasonable fuel consumption, timely arrival and penalize abrupt decisions.

Finally, we assume that any time the agent encounters a problem state (crashes) it is able to restart from a new state.

3.1 State abstraction

Environment observations can be too complicated for an agent to reason with. An agent has limited computational capabilities, such that it is not always possible to process and understand a sequence of observations down to its smallest details. If we consider the classic Pong Atari game, knowing simply the position and velocities of the three objects in the frame is sufficient to play the game. Furthermore, the set of relevant future outcomes that an agent should consider when interacting with its environment is significantly smaller than the collection of possible future observations. It is consequently of no surprise that AI and robotics algorithms greatly benefit from having compact representations of data. State abstraction aims to learn abstract low-dimensional features that represent only the most relevant properties of the world and facilitate learning. Beyond the significant theoretical appeal of using abstraction in RL, abstractions help address some of the core problems faced in practical RL today (exploration, generalization, credit assignment). In robotics, for instance, where RL is commonly used, abstraction can be leveraged to minimize the number of samples required to learn a new policy—which is critical as collecting robot data is often expensive.

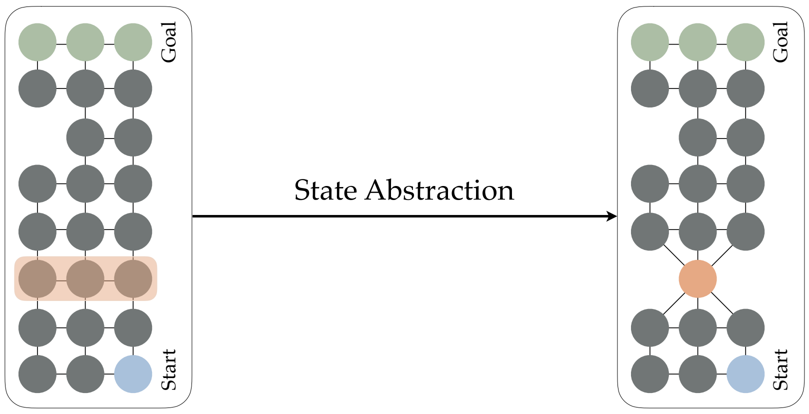


Figure 3.1: A simple grid world problem (left) and the abstracted problem induced by the state abstraction (right). Figure from *"A Theory of Abstraction in Reinforcement Learning" (Abel, 2019)*

In a standard MDP, the state fully describes the configuration of the environment. However, this definition of the state lacks structure and states can be more or less similar to each other. The goal with state abstraction is to find a representation that reduces the size of the state space by "merging" states in a way that does not significantly affect the agent's capacity to solve the MDP. Formally, we look for a function $\phi : \mathcal{S} \rightarrow \mathcal{S}_\phi$, that maps each ground environment state s into an abstract state $s_\phi \in \mathcal{S}_\phi$. This abstract state s_ϕ is the agent's representation of the world and may lose information from the true state. A central problem in state abstraction is finding what information can be discarded.

When an agent that uses abstraction receives the MDP state s_t, it runs it through ϕ and obtains the abstracted state that is then used to learn actions. Note that the abstraction ϕ is separate from the RL algorithm, and can therefore be studied separately. We can introduce the abstraction into our previous agent-environment interaction figure:

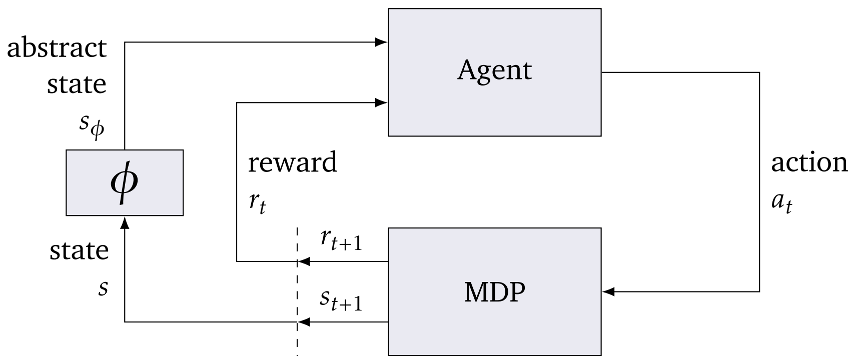


Figure 3.2: Agent-MDP Interaction with state abstraction. Every timestep, the MDP produces a new state s and the agent learns abstracted state $s_\phi = \phi(s)$.

A "good" abstraction should at least satisfy these properties (Abel, 2019):

- **Efficient creation**. Computing and learning the abstraction should not be unreasonably expensive.
- **Efficient decision-making**. The abstraction should allow the agent to efficiently make decisions, i.e. planning or learning in the abstracted state space should be faster than from the ground state.
- **Near-optimality**. The policy learned in the abstraction space should have a value that is close to the best policy that can be learned in the ground state space. This can be measured by considering the best policy that can be represented with a given state abstraction. We can compute the "value loss" incurred in using the representation, taken as the difference in value from the ground state optimal policy. The value loss bound $\tau \in \mathbb{R}$ for an abstraction satisfies:

  $$\min_{\pi_\phi \in \Pi_\phi} \max_{s \in \mathcal{S}} V^*(s) - V^{\pi_\phi}(s) \leq \tau$$

  where Π_ϕ is the set of all policies representable in the abstract state space.

3.1.1 Types of abstractions

Abstraction can be thought of as grouping states that are similar in some way into clusters. When choosing how to learn an abstraction, we may want to preserve a certain property. Li et al. (2006) presented a unifying framework for MDP state abstractions and formalized several classes of abstractions: each ensures that some property holds between true states and abstract states. For example, we may desire that Q-values are preserved for all policies, or perhaps that Q-values are preserved for the optimal policy only.

The strictest condition we may impose on an abstraction—that will preserve the most properties—is bisimulation. Bisimulation will be useful to us in learning abstractions that capture as much information that is task-relevant as possible while discarding other information. Bisimulation only groups together states that are indistinguishable in terms of rewards over all possible action sequences tested—the bisimulation relations (Givan et al., 2003) are defined as follows:

$$\phi(s_1) = \phi(s_2) \implies \begin{cases} r(s_1, a) = \quad r(s_2, a) & \forall a \in \mathcal{A} \\ P(G|s_1, a) = P(G|s_2, a) & \forall G \in \mathcal{S}_B,\ a \in \mathcal{A} \end{cases}$$

where $P(G|s,a) = \sum_{s' \in G} P(s'|s,a)$ and S_B is the set of all groups G of equivalent states under ϕ.

The bisimulation relations however, are "all or nothing": a small variation in reward results in two states being treated as though unrelated. It is therefore useful to consider bisimulation metrics that offer a way to quantify the "behavioral similarity" between states using distance $d : \mathcal{S} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$. This distance d, softens the concept of partitions of equivalent states and must account both for immediate rewards and distance between transition probability distributions, in order to address the two equations in the bisimulation relations. A mathematical method for computing the distance between probability distributions is the Wasserstein Metric (Villani, 2008). Using it, the bisimulation metric (Ferns and Precup, 2014) is then obtained, with $c \in [0, 1)$:

$$d(s_i, s_j) = \max_{a \in \mathcal{A}} (1 - c)|r(s_i, a) - r(s_j, a)| + c \cdot W_1(P^a_{s_i}, P^a_{s_j}, d) \tag{3.1}$$

Definition 3.1.1 (Wasserstein Metric)**.** The Wasserstein metric $W_d(P_i, P_j)$ between two distributions P_i and P_j, defined on a space with metric d is the minimum cost of transforming P_i into P_j . The p-th Wasserstein metric is defined as:

$$W_p(P_i, P_j; d) = \left[\inf_{\lambda \in \Gamma(P_i, P_j)} \int_{\mathcal{S} \times \mathcal{S}} d(s_i, s_j)^p \, d\gamma'(s_i, s_j) \right]^{1/p}$$

where $\Gamma(P_i, P_j)$ is the set of all couplings of P_i and P_j.

RL driving scenario: using abstraction

An abstracted representation of the agent's sensory inputs would enable it to reason at a high level and make better decisions. Instead of having to learn a mapping from pixels to steering, it would learn that e.g. if there are cars on both sides, staying centred is important.

A good representation would collapse states that are behaviorally equivalent; keeping further information would not improve the learned policy and only slow down training. For example: if there are cars on both sides, the colour of the vehicles doesn't matter when it comes to steering or braking.

3.1.2 State representation learning

We have seen that abstraction is a powerful concept to make RL on larger problems more computationally tractable: an abstraction ϕ aims to encode all information that is relevant to solving the MDP, and perhaps also satisfy certain properties such as bisimilarity.

Remark. The state abstraction definitions introduced are for discrete states and often require a complete model of the world to be computed. In most practical scenarios, the state-space is continuous, we can only learn an approximate model, and the agent receives observations that contain far too much detail for any observation to be a true state with the Markov property. Therefore an agent should not only aim to identify behaviorally equivalent states, but also the true underlying MDP state of the task being solved. This problem is similar to learning meaningful feature representations from data—which is generally approached as an unsupervised learning problem. In RL, the intersection between state abstraction and representation learning is commonly referred to as State Representation Learning (SRL). We will be using the terms *abstraction* and *representation* interchangeably throughout this thesis.

Some common approaches to learning a state representation are:

- **Auto-Encoders**: minimize the error between the state s_t and its reconstruction $\hat{s}_t$ from $s_{\phi,t}$.
- **Forward models**: predict the next abstract state $s_{\phi,t+1}$ from $s_{\phi,t}$ and a_t and minimize the error between predicted $\hat{s}_{\phi,t+1}$ and $s_{\phi,t+1}$.
- **Inverse models**: predict the action a_t given $s_{\phi,t}$ and $s_{\phi,t+1}$, minimize the error between predicted $\hat{a}_t$ and a_t.
- **Exploiting rewards**: predict $\hat{r}_{t+1}$ given $s_{\phi,t}$ and a_t, minimize the error between r_{t+1} and $\hat{r}_{t+1}$.
- **Prior knowledge**: learn a representation by using specific information about the dynamics/physics of the world, which can be incorporated into an objective function to be minimized.

We will be discussing the different SRL objectives in more detail in Section 4.2.

3.1.3 Auxiliary objectives

Reinforcement learning focuses on the maximization of reward. Yet, in many domains rewards are rarely observed, delayed or noisy—and this makes learning particularly inefficient.

In traditional ML, we use unsupervised objectives like reconstruction to accelerate the acquisition of a meaningful feature representation. Similarly, a useful practice in RL is the introduction of *auxiliary tasks* into the training objective in order to accelerate learning. The intuition is that learning to estimate quantities that are relevant to solving the main RL problem over a shared representation will speed up the progress on the main RL task.

Remark. Interestingly, auxiliary tasks were first developed for ANNs. It was demonstrated (Suddarth and Kergosien, 1990) that adding these tasks to a small neural network can effectively remove local minima—if the task has a "special relationship with the original input-output being learned".

These auxiliary objectives are typically implemented as additional cost-functions that an RL agent can optimize in a self-supervised fashion, by leveraging its stream of sensory-motor inputs and rewards. Commonly this is implemented by splitting the last layer of the agent's network into multiple heads each focusing on a different task. As the tasks propagate all errors into the shared part of the network, this forms a representation that supports all the objectives. An example objective could be predicting how close the agent is to a terminal state (terminal prediction). Note how some of the approaches to SRL mentioned before (e.g. reward prediction), can be framed as auxiliary tasks.

Hybrid objectives for SRL

Linking the concept of auxiliary tasks back to State Representation Learning, we note that SRL models often take advantage of several objective functions at the same time. For example, in the driving scenario, we could learn to represent the state by combining a pixel reconstruction error with a forward model (Figure 3.3).

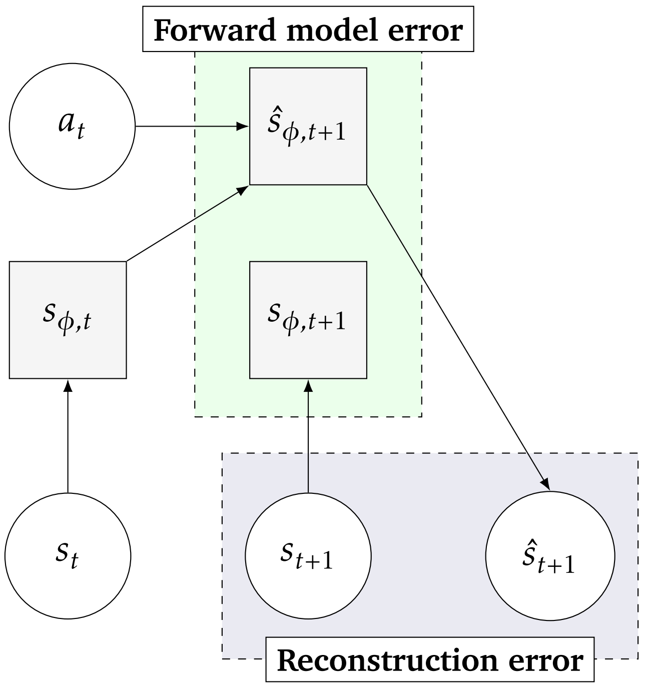


Figure 3.3: Example of a hybrid model that learns to represent a state through two auxiliary losses: a reconstruction error and a forward model error. Circles correspond to observable variables and squares to latent variables.

3.2 Offline reinforcement learning

Part of the ideas and structure for this section is based on "Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems" (Levine et al., 2020)

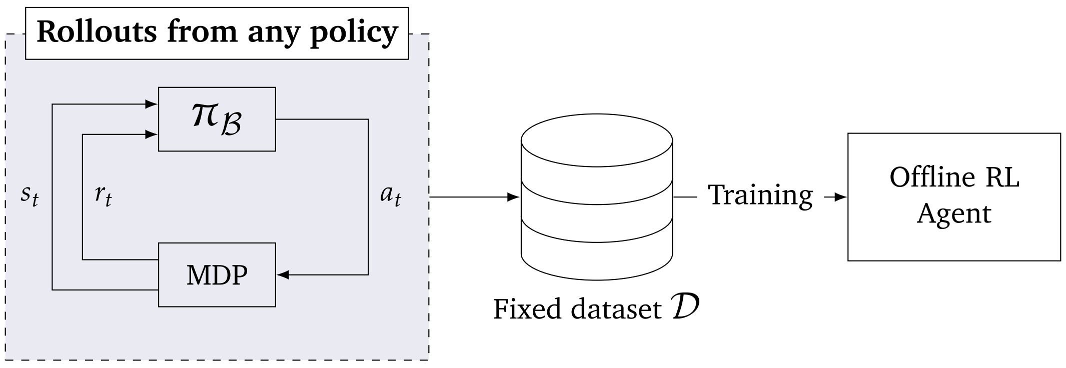


Figure 3.4: In the offline setting, an agent learns from a fixed dataset containing historical decisions and outcomes. The dataset may contain examples of both desirable and undesirable behaviour, and the policy(ies) that generated the data is typically unknown.

When an agent is trained online to solve a problem with RL, it interacts with its environment and learns to improve its behavioural policy through trial and error. This approach has been successful in many areas from robotics control to outperforming humans in games like Chess or Go (Silver et al., 2016, 2017). There are many real-life scenarios, however, where such an online trial-and-error approach is inappropriate. In medicine, for example, an agent learning to prescribe medication should not have to kill people before realizing a given prescription is wrong. Further, it is often not viable to conduct the agent-environment interaction loop at a sufficient scale (e.g. in robotics).

3.2.1 Overview

With offline RL (also referred to as 'Batch RL'), the agent tries to learn how to optimally behave in the environment by using a fixed dataset of transitions generated by unknown agents. Since the agent only learns from previously logged data, the offline setting can be powerful in contexts where exploratory behaviour is costly or dangerous, but also in situations where large collections of historical interactions are available.

Offline RL is especially appealing considering the tremendous success of supervised ML methods trained on big datasets (e.g. speech recognition), as it suggests the vision of turning data into powerful decision-making engines. Offline methods are similar in certain ways to Imitation Learning but instead of receiving data from an expert, data comes from an arbitrary policy and we try to learn behaviour that is as good and potentially better than that policy.

A strong offline RL algorithm should be able to:

- **Identify** and imitate good behaviours, in a dataset containing both good and poor behaviours.
- **Generalize**, i.e. infer good behaviours in states $\mathcal{S}_A$ based on good behaviours in $\mathcal{S}_B$.
- **Recombine** parts of behaviours seen in the dataset, e.g. perform a task without having seen the full trajectory.

RL driving scenario: offline variation

Let's now modify the driving example:

We decide (unsurprisingly) that letting the robot drive and learn through repeated failure is too hazardous for other drivers on the highway. Instead, we pick 10 human highway drivers and fit their cars with a replica of the robot's camera—ensuring the viewing angle is identical. Additionally, we install special hardware to record steering angle and pedal positions. An extrinsic reward is again assumed to be available.

Several days later we collect all data logged in those vehicles. Considering that we know nothing about the 10 drivers (some of them could have been driving under the influence), we cannot use imitation learning methods. In this scenario, an offline RL algorithm has the potential to identify which driving behaviours were most successful with regards to the extrinsic reward and combine them such that the resulting policy is even better than each individual driver.

Finally, if the offline program terminates with high confidence about its driving competence (accurate confidence estimation is an active area of research) we copy the learned policy onto our robot.

Additional notations We return to the class of Markov Decision Processes with notations from Section 2.1. As with online RL, we seek to discover the optimal policy π^* maximizing expected returns, i.e. $Q_{\pi^*}(s,a) \geq Q_\pi(s,a),\ \forall s,a,\pi$.
We are given a dataset for state transitions $\mathcal{D} = \{(s_i, a_i, s_i^{'}, r_i)\}$, that has been generated by following a policy $\pi_\mathcal{B}$ (generally unknown). We denote distribution of states in the dataset by $d^{\pi_\mathcal{B}}(s)$.

As previously mentioned, off-policy RL algorithms can use data generated by different policies. In off-policy RL, however, the behavioural policy is regularly updated to remain close to the current estimate for the optimal policy—and the agent continuously collects new data during training. Given the ability of off-policy algorithms to learn from different policies, they are naturally strong candidates in the offline setting—and are often used as starting points for offline RL algorithms (Levine et al., 2020).

Remark. Accurately evaluating the performance of an offline RL algorithm can be challenging, as we are interested in the *online* performance once the learned agent is deployed. Based on the application domain, it may be possible to leverage simulators to cheaply benchmark the algorithm's performance, or even use human expert evaluators to assess the quality of the decisions. A current active area of research is Off-policy Evaluation (OPE), which comprises methods of evaluation without online interaction.

3.2.2 Distributional shift

One of the central challenges making offline RL difficult is the distribution mismatch between the current policy and the offline data collection policy. In order to learn effective skills, the agent must deviate from behaviour in the dataset and make *counterfactual* predictions about unseen outcomes (answer "what-if" type questions). When the policy being learned enters a part of the state space that is not covered in $d^{\pi_{\mathcal{B}}}(s)$ it is impossible to accurately estimate the reward that will be received. This is further compounded by the sequential nature of RL tasks.

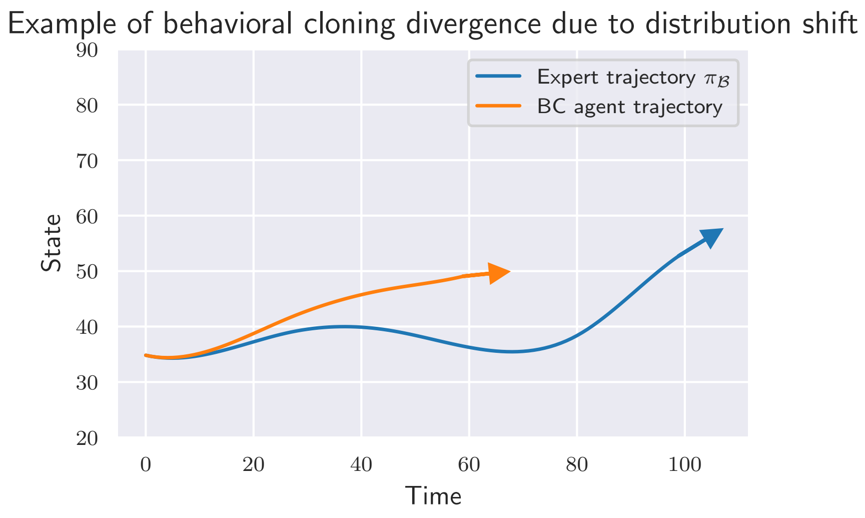


Figure 3.5: Distributional shift in the simple case of cloning an expert trajectory. The agent approximates expected behaviour and makes a small mistakes. The mistakes bring the agent into a state that is further away from the expert trajectory, further increasing the approximation error. These effects were proven to compound as $O(T^2)$.

As an illustration, imagine a simplified scenario whereby a dataset of expert actions is available and we would like to train a neural network to map states to expert actions. As illustrated in Figure (3.5), any small error that the function approximator makes will bring the agent into a state further away from the distribution of expert examples. This in turn will lead to making larger errors, and a policy that diverges from the expert policy. It has in fact been shown that the error in this scenario scales as $O(T^2)$ (Behavioral cloning error bound, Ross et al. 2011).

Extrapolation error

When the agent selects which action to take in the environment (e.g. by following the greedy policy $\pi(s) = \text{argmax}_a(Q(s,a))$) some of the state-action pairs will be out-of-distribution (OOD). In these states, the agent will need to rely on the ability of the value function to extrapolate to unseen states—resulting in extrapolation errors. When the agent is trying to act greedily, this can become especially problematic as the agent can overestimate the value of unseen states.

With Deep RL, we use neural networks as the value function approximators, and they will produce falsely optimist predictions in OOD states. In bootstrapping algorithms like Deep Q-Networks (c.f. Section 4.1.2), the value function is learned by propagating the value of nearby states. This local propagation, combined with the smoothness properties of ANNs can lead to these overly optimistic value estimates exploding and even affecting states within distribution.

In online or off-policy RL, when an agent believes that an action is high-value, it will be more likely to take that action during its environment interaction and therefore 'self-correct. This is not possible when learning entirely from offline data.

Solutions to OOD extrapolation

In recent offline RL literature, many methods have been proposed to tackle the extrapolation error (Kumar et al., 2020; Fujimoto et al., 2019; Wu et al., 2019). The simplest approach is to constrain the learning process such that distributional shift is bounded, for example by introducing a conservative bias in the Q-function (Kumar et al., 2020). While there have been significant improvements to offline RL algorithms in the last few years, we will not be covering those—we seek instead to evaluate which representation learning schemes are optimal to use in conjunction with these methods.

Chapter 4

Algorithms and representations

To assess how helpful representations can be in offline RL, the first step is to select which algorithms and representations we will seek to evaluate. This section aims to explain our choices, introduce the algorithms we will be using and some context on our implementations.

4.1 RL algorithms

There are many Reinforcement Learning algorithms that have been specifically designed to try and tackle the challenges that arise in offline RL (mainly *distributional shift*). Examples are Conservative Q-Learning (Kumar et al., 2020), which explicitly assigns lower value to unseen outcomes, Batch Constrained Deep Q-learning (Fujimoto et al., 2019), which restricts the agent's action space to be close to the behavioural policy with respect to a subset of the data.

Since our main question examines the importance of representations, it does not call for us to use a specific RL algorithm. If the high-level features captured by a representation can lead to better downstream performance by an RL algorithm, the representation is likely to be useful for other algorithms also. In fact, using a method that is new, not well-understood, or overly specific, could compromise the interpretability of our experimental results. Moreover, we are interested in whether certain representations could produce latent spaces that potentially constrain the learned policy in a similar way to CQL or BRAC. For that reason, we decide to proceed with one of the most widely used off-policy algorithms: Deep Q-Networks. We will additionally use Behavioral Cloning as an imitation-learning baseline for the offline RL problem.

We now introduce the two algorithms.

4.1.1 Behavioural Cloning (BC)

Behavioural Cloning is an imitation learning (IL) technique that we will use as a reference baseline. As previously mentioned, certain IL methods can be directly applied to offline RL.

BC is the simplest way to perform imitation learning: we learn to directly map an observation to an action. Instead of trying to find the best policy given the data, BC tries to learn to replicate the behaviour seen in the data. Therefore BC is a simple supervised learning solution to the problem and will provide a simple and interpretable baseline for our experiments.

Formally, a BC agent learns a policy $\pi_\theta(a_t|s_t)$ mapping states to actions, by minimizing the negative log-likelihood of the selected action being from the policy being cloned, i.e: minimizing $-\log p(a_t = \pi_{\mathcal{B}}(s_t)|s_t)$ across the whole dataset.

Since a BC agent aims to replicate the policy it sees—its performance should be roughly the same as that of the average policy present in the data. Just like other algorithms, BC falls prey to distributional shift. Empirically, BC is expected to work well primarily in tasks where making small mistakes is not catastrophic and where the behavioural policy is relatively good. Additionally, we note that BC will work better when samples come from a stable trajectory distribution.

RL driving scenario: BC

Behavioural cloning would be appropriate if we had selected professional drivers for the data collection. This is in fact an approach that a number of autonomous driving R&D companies are exploring (Bansal et al., 2018; Cao et al., 2020).

Remark. Various extensions and improvements to Behavioral Cloning exist (e.g. DAGGER, Ross et al. 2011). However, most of them involve some form of additional data collection—for instance by querying an expert's opinion—and are therefore not relevant to the offline RL problem.

4.1.2 Deep Q-Networks (DQN)

The DQN belongs in the space of value-based methods and the goal is to estimate the optimal action-value function:

$$Q^*(s,a) = \mathbb{E}_{\pi^*}[R_t|s_t = s, a_t = a] \tag{4.1}$$

$$Q^*(s,a) = \mathbb{E}_{\pi^*}[R_t|s_t = s, a_t = a] \tag{4.2}$$

The intuition is that if we had Q^*, then we could easily construct a policy that maximizes the expected cumulative reward as:

$$\pi^*(s) = \arg\max_a Q^*(s,a) \tag{4.3}$$

In an offline setting, given sufficient coverage of the state-action space, the DQN is able to prioritize trajectories that empirically led to higher rewards.

Q-learning

One of the early RL breakthroughs was the development of an off-policy algorithm known as *Q-learning*, defined by update rule 4.4. The algorithm is based on the Bellman equation, a well-known condition satisfied by the optimal policy in an MDP. Typically the update is performed online sample-by-sample, i.e. the value function is updated whenever a new transition is made.

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{4.4}$$

The learned action-value function Q directly approximates Q^* regardless of the policy being followed. Under mild assumptions, it has been shown that Q will converge with probability 1 to Q^* for finite state and action spaces.

The Q-learning algorithm is shown below in procedural form:

Algorithm 1: Q-learning for estimating $\pi \approx \pi_*$

```
1  Algorithm parameters: step size $\alpha \in (0,1]$, small $\epsilon > 0$;
2  Initialize $Q(s,a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, arbitrarily except that
     $Q(\text{terminal}, .) = 0$;
3  for each episode do
4      Initialize $s_t$;
5      for each step of episode do
6          Choose $a_t$ from $s_t$ using policy derived from $Q$ (e.g. $\epsilon$-greedy);
7          Take action $a_t$, observe $r_t$, $s_{t+1}$;
8          $Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$;
9          $s_t \longleftarrow s_{t+1}$;
10     end
11 end
```

where an ϵ-greedy policy amounts to taking the greedy action with probability $1-\epsilon$, and a random action with probability ϵ.

DQN Algorithm

In practice, the basic approach of Q-learning is impractical as the action-value function is estimated separately for every sequence without any generalization. Instead, the above Q-learning rule can be directly implemented by estimating the Q-function with a deep neural network. Since direct assignment as in the Q-learning update step is not possible, we define instead an error function to minimize. The error measures the difference between the current action-value and the value that should be assigned by the rule. Our implementation will be using the Huber loss ($\delta = 1$) applied to this difference (also called temporal-difference):

$$\text{Error} = L_\delta(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{4.5}$$

with the Huber loss defined as:

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases} \tag{4.6}$$

It is then possible to iteratively adjust the weights of the neural network and find an approximation of the optimal Q-function, by minimizing the error with gradient descent using backpropagation.

Note that while the original Q-learning algorithm is a tabular method restricted to discrete action and state spaces, neural networks are able to approximate action-values by taking a continuous state as an input.

The original DQN paper (Mnih et al., 2013) used two extra techniques to prevent oscillation and divergence of the network weights.

- **Experience replay.** A common assumption when training a supervised-learning model is that samples are independent and identically distributed (i.i.d). With the naive Q-learning algorithm, each transition is used to update the Q-function when the agent receives it. To ensure the neural network is able to successfully learn the Q-function and generalize, the DQN uses a replay buffer that stores a large number of previous transitions. At every gradient update step, the error function is calculated over a randomly sampled batch of transitions. This experience replay can also help ensure that the network doesn't forget past experiences.

- **Target Q-function.** Unlike Q-learning where each update only affects one state-action pair, when using a function approximator a single update will affect the value of many. To address this, we use an older set of weights to compute the targets $r_{t+1} + \gamma \max_a Q(s_{t+1}, a)$, which keeps the target function from changing too quickly.

Common improvements

Unfortunately, using a DQN as described above is often insufficient to resolve instabilities. As such many ideas have been proposed to improve performance in practice. We outline some of the improvements we will be using in our implementation of the DQN.

- **N-step bootstrapping**. Bootstrapping updates can work better if we consider multi-step transitions (such that there is a significant change in state between observations). Our online implementation of the DQN inserts 5-step transitions into the replay buffer, considering the reward collected over those timesteps, and randomly samples from the buffer.

- **Double Q-learning** (Hasselt, 2010). Q-learning is known to overestimate action values under certain conditions. Note that a DQN uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated action. The Double Q-learning algorithm decouples the selection from evaluation to prevent this. Two value functions are learned symmetrically by assigning each new sample to update only one of the functions, resulting in two sets of network parameters θ and θ'. Then, one set of weights is used to select the greedy action and the other to assess its value:

$$\text{DDQN Target} = r_{t+1} + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta'_t) \tag{4.7}$$

Training a DQN offline

When we train a DQN offline, we basically eliminate the ‘agent’ part of the off-policy algorithm and perform no additional environment rollouts. As we use a fixed-size dataset, this is equivalent to the previously introduced replay buffer—except that its size is not restricted.

All of the additions to Q-learning mentioned above are also applicable to offline training, except for N-step bootstrapping which is handled on the agent side. The offline DQN learner simply learns to estimate optimal Q-values for transitions contained in the dataset, regardless of N.

We finally note that training a DQN offline is very similar to using the Neural Fitted Q Iteration algorithm (Riedmiller, 2005), where the gradient update step is performed offline considering an entire set of transition experiences.

4.2 SRL methods

We’ve discussed how representations are necessary for an agent to learn a good policy, and how SRL can provide useful structure and help RL algorithms learn better representations faster. In this section, we will dive deeper into the different approaches for learning a state representation. We will also discuss the methods we have selected to evaluate in the offline setting.

4.2.1 Reconstruction vs reward

While we have seen that state representations can be learned in many different ways, most methods used in practice are either based on *reconstruction* or based on *rewards*. Each approach has its respective advantages. Other SRL methods we have mentioned are typically combined with reconstruction or reward. For example, it is uncommon to learn an abstraction by only using a forward model—it is easy for the learned abstraction to collapse to 0 and locally minimize the forward prediction error. To ensure robustness, we can ground the representation by using the reward or observation signal—by e.g. ensuring that s_ϕ can either reconstruct s or predict r.

Using reconstruction

In traditional ML, learning representations from reconstruction has been studied extensively and with great success. Reconstruction provides a rich training signal that can be effectively used in an unsupervised fashion. This is particularly effective with high-dimensional observations like images where it is often easy to predict the low-level details. In RL, agents often receive image observations from the environment (as in our toy driving scenario) and reconstruction methods can be very effective in learning a compressed summary representation.

The main drawback of this approach is that the reconstruction objective may not be necessarily aligned with solving the RL problem. Reconstruction objectives place equal importance on all information, regardless of their task relevance. To make an extreme example of a situation where this would be an issue, imagine an agent receives a large image at every timestep. It is possible that the reward depends exclusively on a single pixel from that image. Our optimal representation should only contain the value of that pixel, yet achieving good reconstruction requires preserving information about all other pixels too. If the size of the representation is bounded, we will be making an uninformed decision on which information to preserve.
In an early exploratory phase of this work, we investigated the application of keypoint representations (based on reconstruction) for robotic manipulation (c.f. Appendix A). This work highlighted to us some of the limitations of reconstruction-based methods and inspired us to explore reward-based methods.

Using rewards

To best satisfy the "efficient decision-making" property of a good abstraction, only task-relevant information must be included in the abstract space. While often sparse, the reward is by definition aligned with our RL objective. This means that if our representation learning loss is derived from reward, we can be certain that the information captured by our representation is important and relevant for our task. Recent work (Agarwal et al., 2021; Zhang et al., 2020a) has shown promising results with such 'RL-informed' methods. This is even more so relevant for real-life scenarios where agent observations are full of distractions:

RL driving scenario: abstraction using reward

When driving a car, we receive large amounts of detailed information from our sensory systems—which our brain maps to a low-dimensional summary understanding.

In the case of the high-resolution images captured by our robot's camera, using reconstruction requires preserving a lot of irrelevant information, like the colour of nearby cars and the shape of the clouds. Leveraging the reward signal makes it possible to learn representations that are *invariant* to distractions.

With rewards, we are also able to construct representations that are inspired by the theory of state abstractions. For example, we can use bisimulation to collapse together states that are behaviorally equivalent (as seen in Section 3.1.1). Moreover, using rewards can make it possible to learn a representation with advantageous geometries. Using reward we can construct a latent-space with better separability and curvature properties, such that e.g. a Q-function may be easier to approximate.

Summary

All in all, each approach has its merits. On the one hand, we have a rich reconstruction signal that enables faster learning of a representation, and on the other hand, we have a sparse signal that is appealing due to its natural correlation with our desired RL objective. Another possibility is to combine both signals through auxiliary losses and potentially get some of the benefits of both worlds—many popular SRL models do this (Gelada et al., 2019).

Both reconstruction and reward-based SRL methods have been shown to be a powerful tool for online RL, and are an active focus area for RL researchers. On the other hand, offline RL has just recently grown in popularity and there is currently little understanding of how it would benefit from representations.

4.2.2 Representations considered

Through our review of the relevant literature, we identified several candidate SRL methods that would be of interest to investigate in the offline setting.

We position them on the following axes: whether they use reconstruction/rewards, whether they use forward-modelling, whether the model is stochastic, whether the latent-space is discrete and whether the model considers sequences of multiple transitions.

	Training Signal		Model Architecture		Latent State	
	Recons.	Reward	Forward	Multi-Step	Stochastic	Discrete
PCA	✓					
AE	✓					
VAE	✓				✓	
VQ-VAE,	✓				✓	✓
DBC		✓				
DeepMDP		✓	✓			
VPN		✓	✓	✓		
World Models	✓		✓	✓	✓	

Table 4.1: Classification of the SRL models we considered for our experiments. Our selected methods are indicated in bold.

The primary restriction we had in selecting which methods to try was that our offline data for Atari games (Chapter 6) was arbitrarily shuffled—i.e. we could not use sequences of transitions (Oh et al., 2017; Ha and Schmidhuber, 2018). We also eliminated the VQ-VAE model (Oord et al., 2018) as discrete representations have significantly different properties—they would be nonetheless interesting to explore in future work. Since PCA (Pearson, 1901) can be considered as a particular case of auto-encoder, we will only be using the auto-encoder.

In summary After these limitations, we are left with the methods in bold. While it may be possible that e.g. discrete representations are especially suitable for the offline setting, we believe that this selection provides a sufficiently extensive picture of the current SRL landscape to assess the utility of explicit representation for offline agents. The next sections introduce them in more detail.

4.2.3 Reconstruction-based methods

This section introduces the auto-encoder (AE) and variational auto-encoder (VAE) as a state representation learning model.

Auto-encoders

Auto-encoders (Hinton and Salakhutdinov, 2006) are a tool from the wider area of representation learning that is commonly to learn state representations. An auto-encoder consists of an encoder and a decoder, and its objective is to output a faithful reproduction of its input. The encoder outputs live in a latent space that is typically of a lower dimension such that we force a compressed knowledge representation of the original input. Typically both the encoder and decoder are implemented as DNNs (or CNNs), and we refer to the output layer of the encoder as the "bottle-neck".

In the context of state representations the input to the AE is s_t, the encoder outputs the latent representation $s_{\phi,t}$ and the decoder re-projects to obtain $\hat{s}_t$. We can then select the size of the bottle-neck layer to match the dimensionality of the state representation we want to learn.

The auto-encoder will be trained by minimizing an error between s_t and $\hat{s}_t$. In our implementation, we use the L_2 norm (pixel-wise in case of image observations), but it is possible to use any loss $L(s_t, \hat{s}_t)$.

Variational Auto-encoders

Variational auto-encoders (Kingma and Welling, 2013) are autoencoders whose training is regularized to avoid overfitting and ensure that learned features are more interpretable and disentangled.

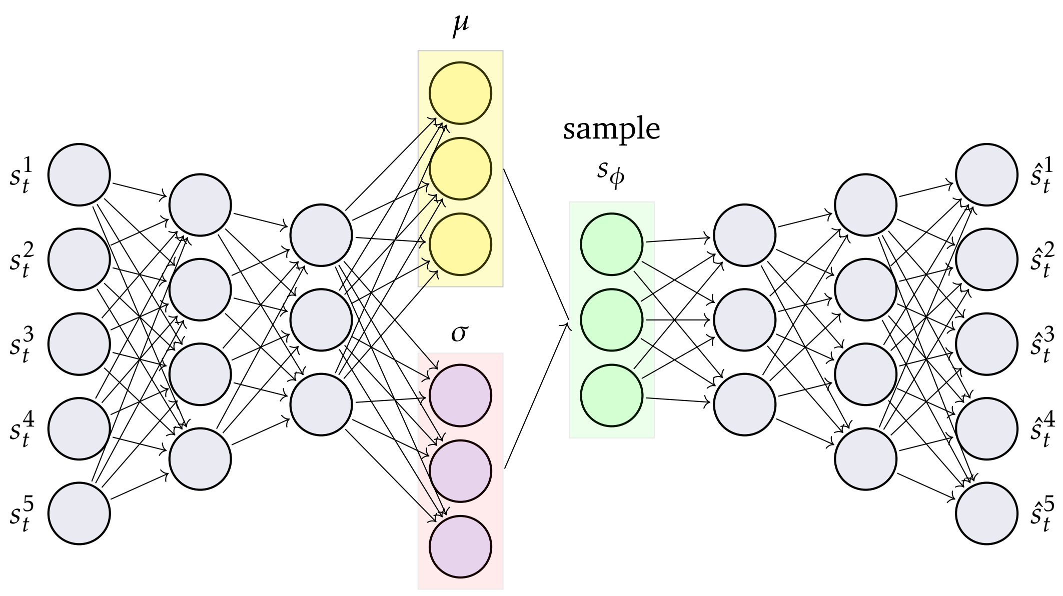


Figure 4.1: Illustration of the variational autoencoder for state representation. Note how means μ and variances σ^2 are independently learned by the encoder, and used to sample s_ϕ with the reparametrization trick. The VAE is trained by minimizing a reconstruction loss between s_ϕ and s

Instead of encoding a state into a single point in the latent space, we encode it into a distribution over the latent space. We use the most common distribution, a multivariate Gaussian with a diagonal covariance structure as this enables the encoder to simply output the means and variances of the distribution. We can then regularize the VAE by enforcing the Gaussians to be close to normal.

The decoder then learners to reconstruct the state from a sample of the encoded distribution. However, the sampling operation is stochastic and cannot be backpropagated. The VAE authors introduced a 'reparametrization trick' to address this. In the case of the multivariate Gaussian, we can transform the random variable s_ϕ as follows:

$$s_\phi \sim \phi(s_\phi|s) = \mathcal{N}(s_\phi; \mu, \sigma^2 I) \tag{4.8}$$

$$s_\phi = \mu + \sigma \odot \epsilon, \text{ where } \epsilon \sim \mathcal{N}(0,1) \tag{4.9}$$

where $\odot$ refers to the element-wise product, and μ, σ^2 refer to the means and variances of the multivariate Gaussian. The loss function of the variational autoencoder for a single state s is:

$$l(\phi, \omega) = -\mathbb{E}_{s_\phi \sim \phi(s_\phi|s)}[\log p_\omega(s|s_\phi)] + \mathbb{KL}(\phi(s_\phi|s) \| p(s_\phi)) \tag{4.10}$$

where the first term is the reconstruction loss, and the second term is the Kullback-Leibler divergence between the encoder distribution and $p(s_\phi) = \mathcal{N}(0,1)$. The decoder network is $p_\omega(s|s_\phi)$ with parameters ω.

4.2.4 Reward-based methods

We now discuss DBC and DeepMDP, two latent space models for MDPs proposed in recent work by Zhang et al. (2020a); Gelada et al. (2019), with a strong theoretical connection to bisimilarity.

Deep Bisimulation for Control (DBC)

Recent work by Monier et al. (2020) introduced a gradient-based method to learn a state abstraction with the properties of bisimulation metric: Deep Bisimulation for Control (DBC). A representation $\mathcal{S}_\phi$ is learned such that the metric corresponds to the L_1 distance between two abstract states. Training is performed by using a reward model, a dynamics model $\hat{\mathcal{P}}$ that outputs Gaussian distributions, and setting the loss function to minimize the squared difference between the bisimulation metric and the L_1 distance. Training samples are sampled in pairs from a replay buffer and both fed through the network in parallel, similar to a Siamese network (Chicco, 2021).

Recalling the bisimulation metric equation, the DBC loss can be formulated as follows:

$$J(\phi) = \left(||s_{\phi,i} - s_{\phi,j}||_1 - |r_i - r_j| - \gamma W_2(\hat{\mathcal{P}}(\cdot|\bar{s}_{\phi,i}, a_i), \hat{\mathcal{P}}(\cdot|\bar{s}_{\phi,j}, a_j)) \right)^2 \tag{4.11}$$

where r are rewards, and $\bar{s}_\phi$ denotes $\phi(s)$ with stop gradients operators (no gradient propagation).

Remark. Note how we use the 2-Wasserstein metric W_2 in equation 4.11, versus the 1-Wasserstein from equation 3.1.1. This is because W_2 has the convenient form:

$$W_2(\mathcal{N}(\mu_i, \Sigma_i), \mathcal{N}(\mu_j, \Sigma_j))^2 = ||\mu_i - \mu_j||_2 + ||\Sigma_i^{1/2} - \Sigma_j^{1/2}||_{\mathcal{F}}^2 \tag{4.12}$$

where $||\cdot||_{\mathcal{F}}$ denotes the Frobenius norm.

Zhang et al. leverage this architecture in conjunction with an online Actor-Critic and show that it succeeds in solving Mujoco tasks (Tassa et al., 2018) with highly distracting observations. Moreover, their experiments show that the latent space groups semantically similar but visually different states. When trained off-policy, the algorithm consists in iteratively updating the agent's policy or value network, the DBC encoder ϕ, and the dynamics model $\hat{\mathcal{P}}$.

In our experiments the dynamics model did not contribute to improved performance, so we simplify the loss function to directly use sample transitions. We also noticed that substituting $s_{\phi,j} = \phi(s_j)$ with $\tilde{s}_{\phi,j} = \tilde{\phi}(s_j)$ (where $\tilde{\phi}$ is the encoder from the target network) resulted in more stable learning—likely because it avoids having a moving target. Last, we enforce the metric in a latent-space constructed as a linear projection from s_ϕ, i.e. $z_\phi = h(s_\phi)$. Intuitively, this ensures the learned representation s_ϕ can

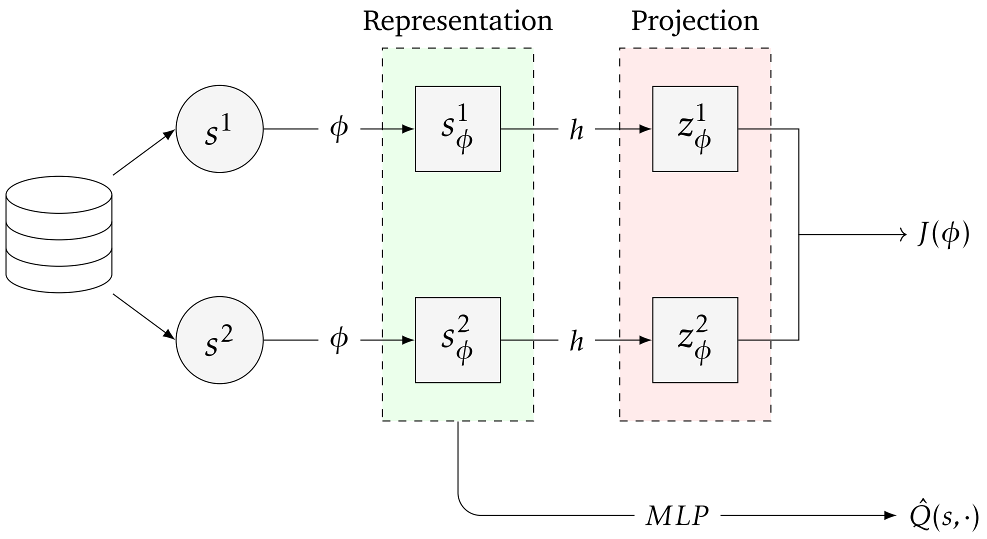


Figure 4.2: Architecture for learning DBC with an offline DQN. Given an input pair of states $\{s^1, s^2\}$, we apply encoder ϕ to produce state representations s_ϕ and project them using h to obtain embeddings z_ϕ. The representation is then trained by matching L_1 distances in the projection space to the bisimulation metric, or (next section) through a contrastive loss. DBC is trained jointly with the DQN loss, which estimates action-values from s_ϕ.

satisfy the metric within a linear transformation, but is more flexible for learning Q-values. Our final loss implements:

$$J(\phi) = \Big(||z_{\phi,i} - \tilde{z}_{\phi,j}||_1 - |r_i - r_j| - \gamma W_2(\hat{\mathcal{P}}(\cdot|\bar{z}_{\phi,i}, a_i), \hat{\mathcal{P}}(\cdot|\bar{z}_{\phi,j}, a_j))\Big)^2 \tag{4.13}$$

$$= \Big(||z_{\phi,i} - \tilde{z}_{\phi,j}||_1 - |r_i - r_j| - \gamma ||z'_{\phi,i} - z'_{\phi,j}||_2\Big)^2 \tag{4.14}$$

Algorithm 2: Training a DQN offline with DBC

```
1 for step in 1...N do
2     Sample batch $B_i \sim \mathcal{D}$;
3     Permute batch: $B_j = \text{permute}(B_i)$;
4     Train DQN: $\mathbb{E}_{B_i}[J(Q)]$                // c.f.  Algorithm 1
5     Train encoder: $\mathbb{E}_{B_i,B_j}[J(\phi)]$      // c.f.  Equation 4.14
6 end
```

Contrastive embedding of bisimilarity (Contrastive DBC)

Contrastive methods have a strong track record in representation learning. At the intuitive level, contrastive learning is an approach to formulate the task of finding similar and dissimilar samples. Various successful algorithms (SimCLR, Chen et al.

2020) have been proposed to learn self-supervised representations of images using contrastive learning.

Recent work (Agarwal et al., 2021) has proposed contrastive metric embeddings (CMEs) adapted from SimCLR to embed a state similarity metric into a representation space. This method can be used to learn a representation that ensures that states that are close under the distance metric are close in the Euclidian space. This alternative way to learn a metric-based state representation is interesting for us to explore, as contrastive embeddings can be easier to learn versus directly learning the metric: the model must only learn to predict which states are closest under the metric, as opposed to their precise distance.

We implemented CMEs to embed the bisimulation metric, resulting in a contrastive version of DBC which we call 'Contrastive DBC' (CDBC). We now introduce CMEs in the context of our implementation:

Algorithm 3: Contrastive Metric Embeddings (CMEs)

```
1 Given: State embedding $\phi(\cdot)$, Metric $d(\cdot,\cdot)$, Dataset $\mathcal{D}$ and hyperparameters:
    temperature $1/\lambda$, Scale $\beta$, Total training steps $K$.;
2 for step in k=1...K do
3     Sample a pair of batches $B_i \sim \mathcal{D}$, $B_j \sim \mathcal{D}$;
4     Update the weights of $\phi$ to minimize $\mathcal{L}_{CME}$ where
        $\mathcal{L}_{CME} = \mathbb{E}_{B_i,B_j\sim\mathcal{D}}[L_\phi(B_i,B_j)]$
5 end
```

We take the bisimilarity metric d from equation 3.1 and implement it with state transition samples as in 4.14. We use the metric d to define positive and negative pairs, and also to assign importance weights in the contrastive loss. Given two states x and y, their embedding similarity is denoted by $c_\phi = \text{sim}(\phi(x),\phi(y))$, where $\text{sim}(u,v) = \frac{u^T v}{\|u\|\|v\|}$.

We first transform distance d to a similarity measure $\Gamma \in [0,1]$. For this purpose we use the Gaussian kernel $\Gamma(x,y) = exp(-d(x,y)/\beta)$, where $\beta > 0$ is a scaling parameter controlling the sensitivity of similarity to d.

Then we produce positive and negative pairs from the two sampled batches B_i and B_j. We use states in B_j as anchors. For every anchor state $y \in B_j$, we obtain its nearest neighbour (with respect to Γ) in B_i and define positive pairs $\{(\tilde{x}_y, y)\}$, with $\tilde{x}_y = \arg\max_{x\in B_i} \Gamma(x,y)$. All other states in B_i are combined with y as negative pairs.

Finally we define the training objective, which is a soft version of the SimCLR contrastive loss, for learning our representation mapping ϕ. Given a positive pair $(\tilde{x}_y, y)$, batch B_i, and similarity measure Γ, the loss function is defined as:

$$l_\phi(\tilde{x}_y, y; B_i) = -\log \frac{\Gamma(\tilde{x}_y, y)\exp(\lambda c_\phi(\tilde{x}_y, y))}{\Gamma(\tilde{x}_y, y)\exp(\lambda c_\phi(\tilde{x}_y, y)) + \Sigma_{x' \in B_i \setminus \{\tilde{x}_y\}}(1 - \Gamma(x', y))\exp(\lambda c_\phi(x', y))} \tag{4.15}$$

Note how using Γ with a soft objective prevents penalising too much for state pairs that are selected as negative examples but are in reality are behaviorally similar.

Remark. This implementation faces a trade-off with regards to batch size. A larger batch size increases the likelihood of finding a strong positive pair, and smaller buffer sizes may increase the likelihood of collision (the same state appearing both in B_i and B_j).

Using reward and a forward-latent model (DeepMDP)

The DeepMDP paper (Gelada et al., 2019) formalizes the concept of training a latent space model through the minimization of two tractable losses: reward predictions and prediction of the distribution over next latent states.

They show theoretically that optimizing these objectives guarantees quality with regards to how the state space is represented, and with regards to the environment model. We implemented a DeepMDP representation, as an example of abstracting the state space using both reward and a forward-latent model.

For a single transition $(s_t, a_t, r_{t+1}, s_{t+1})$, our DeepMDP loss is defined as:

$$L = L_r + \alpha L_t = (r_{t+1} - \hat{r}_{t+1})^2 + \alpha \| s_{\phi,t+1} - \hat{s}_{\phi,t+1} \|_2 \tag{4.16}$$

where α is a weighting factor, $\hat{r}_{t+1} = f(s_t, a_t)$ and $\hat{s}_{\phi,t+1} = g(s_t, a_t)$. Both f and g are neural network function approximators. We use LayerNormMLPs, following previous work, and assume a deterministic transition model for simplicity.

4.2.5 Summary

In summary, we have introduced our implementation for several SRL methods that we will evaluate in the offline setting. To benchmark reconstruction, we'll train an autoencoder representation. To benchmark representations based on the reward signal, we'll 1/ learn a representation where distance in the latent space is the bisimulation metric (DBC), 2/ embed the bisimulation metric using a contrastive loss (CDBC), 3/ learn a latent-space through reward prediction and a forward-latent model (DeepMDP).

Chapter 5

State representation in offline RL

In this chapter, we discuss the experiments we conducted to benchmark the use of various state representation methods in the offline RL setting.

5.1 Experimental setup

Over the course of the project, we've run many hundreds of experiments training RL agents and representations. To this end, we leveraged a large number of tools that have made the process easier to manage. In the following sections, we document some of the practical details in conducting the experiments, as well as highlight resources that were particularly useful.

5.1.1 Research software & tools

When building our experimental codebase, we took advantage of several open-source tools that make software development and ML research more efficient.

Programming Language & Development Environment All our code, experiments, and analyses were written in Python. This was a straightforward decision: Python is widely used within the machine learning community and consequently there is a large ecosystem of powerful ML libraries built specifically for it.

We used VS Code and the Remote SSH function to develop directly from remote machines. The 'Black' code formatter helped us save time by eliminating formatting decisions. Lastly, we worked exclusively from virtual environments: our experiments must be reproducible across different machines and operating systems.

ML/RL Frameworks We used Tensorflow (TF)—a powerful symbolic math library for ML—as our primary framework. This choice was based on TF's popularity, thorough online documentation, and the fact that most research papers we reviewed had their reference code written in TF. We also extensively used Sonnet, a TF wrapper that offers higher-level APIs. We implemented our RL algorithms with Acme

(Hoffman et al., 2020), a flexible research framework that also provides easily extensible building blocks of standard RL agents. Writing a stable implementation for an RL algorithm is non-trivial and it can often be challenging to detect issues and pinpoint bugs. Acme's base implementations provided a reliable starting point for our experiments.

Experiment tracking & Version Control We used GitHub to track our codebase (the repository is accessible here). We also used 'Weights & Biases' (W&B) to monitor our training. We integrated our codebase with W&B to track parameters of every run and visualize debugging signals like Q-value distributions or renders of our online evaluation. Finally, we also set all seeds to ensure near-reproducible results.

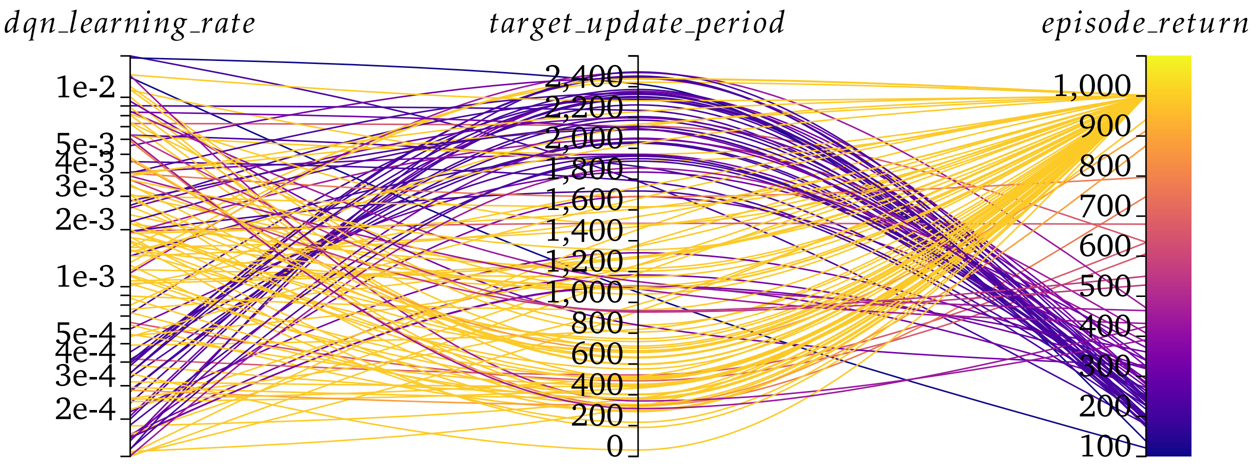


Figure 5.1: Example of a bayesian hyperparameter sweep: we tuned learning rate and target update period, to maximize the return of the offline DQN after 50k training steps.

Hyperparameter tuning Neural network models and RL algorithm can be very sensitive to their hyper-parameters. To ensure truthful evaluation of each method, we tuned whenever possible given our compute constraints various parameters such as learning rates, update frequencies for the target networks, loss-weighing factors, etc. A helpful tool in this process was again W&B. We used it to define configurations for grid or Bayesian search sweeps. Figure (5.1) shows an example bayesian sweep over the learning rate and target update period of the offline DQN.

5.1.2 Hardware

Over the course of the project, we have used various hardware setups:

- **A local development environment**. 2017 13-inch MacBook Pro, with a Dual-Core Intel Core i5, 8GB DDR4 RAM, and integrated Intel Graphics.
- **Google Colaboratory Pro (Colab)**. A Google Research product that allows anybody to run Jupyter notebooks online, with free (limited) GPU availability. We use the PRO subscription which provides preferential access to GPUs (typically an NVIDIA P100 or NVIDIA V100).

- **A remote lab machine**. The Human-Centred Robotics lab kindly provided us access to a remote machine with 16 CPU cores and an Nvidia GTX1060 GPU.
- **DOC undergraduate machines**. We also took advantage of our access to machines provided to undergraduate students by the Department of Computing. These machines were fitted with an 8-core CPU and an Nvidia Titan RTX GPU.

To conduct experiments, we initially used Colab. Unfortunately, due to the low-cost nature of the service, sessions were frequently terminated at unpredictable times to free up usage for other users. This was inconvenient, slowing down progress and causing occasional data loss. Moreover, Colab having only 2 CPU cores proved to be a severe limitation since most of our experiments were CPU-bound: simulating an environment, or loading offline data was more time-consuming than updating network weights. We were unable to run multiple small-scale experiments in parallel, and large-scale experiments would not complete with sessions being killed within the first 12h of runtime. Therefore, once we eventually got access to lab and DOC servers, we fully migrated our workflow to those machines.

5.1.3 Code structure

One of the main objectives when writing our code was to ensure a clean and flexible structure such that it would be easy to extend for new algorithms and representations. We implement a base class for BC and DQN, and for each representation we override the `_custom_init` and `_update_step` methods.

The code architecture is summarized as follows:

- **run.py**: *main script* — starts new experiments.
- **learners/**
  - **bc.py, dqn.py**: RL algorithm base classes.
  - **representations/**: _custom_init, _update_step for each representation.
- **distractors_benchmark/**: collects and loads data, adds distractors for Cartpole.
- **atari_benchmark/**: loads and pre-processes offline Atari data.
- **analysis/**: various scripts to analyse experimental results.
- **networks.py**: a collection of Sonnet modules used by agents/representations.
- **utils.py**: a collection of common helper functions.

5.1.4 Optimizations

When training representations and algorithms using pixel observations, we quickly discovered how expensive this can be. Online agents often need to observe tens of millions of frames, and while offline learning saves on simulation time, it generally requires multiple iterations over the dataset of transitions.

We outline some of the performance improvements we tried by profiling our code and following common Tensorflow guidelines, reducing our training time on offline Atari (c.f. Chapter 6) from $\sim 80+$ hours down to ~ 15 hours:

- (*Both*) Parallelizing data-transformation and data-extracting to leverage all available cores. This contributed the most to reducing training time.
- (*Both*) Pre-fetching batches while the model is training ($\sim 20\%$ reduction).
- (*Cartpole*) Vectorizing our dataset mapping transformations ($5\times$ reduction).
- (*Cartpole*) Caching the dataset in memory after the first complete cycle. This is not possible with Atari due to dataset size.
- (*Atari*) We also tried downsampling the 64×64 frames to 32×32, but crucial details were lost making learning impossible.

Note that even after all those improvements, our training time was still bound by the CPU (i.e. on the lab machine, we consistently experienced 100% utilization across all 16 CPU cores!).

5.2 Classic control with distractions

Due to the absence of well-established evaluation protocols in offline RL, we follow an approach from recent work Gulcehre et al. (2021) whereby training data is collected via an online algorithm in a simulated environment. We need to find an environment satisfying the following criteria:

- It provides **sufficiently complex dynamics** for experiments to be meaningful.
- A simulated transition is **fast** enough that we can perform our desired experiments within our compute budget.
- Has a **discrete action** space (in light of our previous algorithm choices).
- Finally, it is easy to modify, battle-tested, and well-documented.

After extensive research into various available options, we decided that this balance was struck best by classic control tasks. Specifically, we decided to work with **Cartpole**, from the classic RL literature.

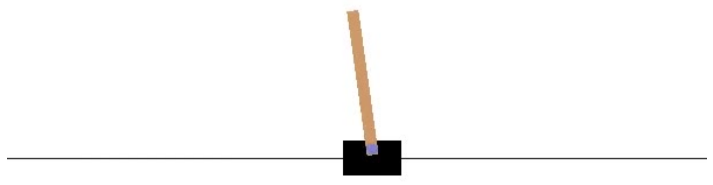


Figure 5.2: A rendered frame from the Cartpole environment.

5.2.1 Cartpole

First presented by Barto et al. (1983), the task consists in balancing a pole that is hinged to a movable cart by applying a force to the cart's base. The equations of motion of the cart-pole system are assumed unknown.

Goal: The pendulum starts upright, and the goal is to prevent it from falling over.

Reward: The only feedback signal from the environment is a +1 reward, provided every timestep that the pole remains upright. When the pole is more than $15°$ degrees from vertical, or the cart moves more than 2.4 units from the centre, the episode ends. Moreover, each episode automatically terminates after 1000 timesteps.

Action space: Discrete: ['left', 'stay', 'right'].

Observation space: 6 dimensions $(x, \dot{x}, \cos(\theta), \sin(\theta), \dot{\theta}, t)$

While several open-source implementations for this environment exist, we choose to work with the Behaviour Suite (bsuite) for Reinforcement Learning (Osband et al., 2020). Bsuite provides highly efficient implementations of several benchmark tasks, including Cartpole, for the purpose of investigating the core capabilities of an RL agent. We modify this codebase, to suit our needs and experiments.

5.2.2 Distractors design

As previously discussed, agents interacting in the real world are often faced with highly distracting environments. When tasked with a specific objective, an agent might be presented with observations that contain or are governed by variables that are task-irrelevant and should therefore be ignored. Clearly, we would like representation learning methods to be able to discern what information is important and what should be discarded.

In order to benchmark how various representations deal with distractions, we would like to build a modified Cartpole environment that introduces task-irrelevant variables in the observation received by the agent.

To this end, we design a simple scenario (it does not capture all types of real-life distractions) whereby distractions span specific dimensions of the observation space. Under this assumption, we classify ‘distractors’ (distracting dimensions) under the following main types:

- The distractor is a **random** process, cannot be predicted nor controlled, does not affect present or future rewards.
- The distractor **varies with time** in a predictable manner, but cannot be controlled and does not affect present or future rewards.
- The distractor is **action-dependent**. I.e. it will vary predictably as a function of the agent’s actions, but does not affect rewards.
- The distractor affects rewards, but **cannot be controlled**.

RL driving scenario: illustrating distractor types

Let’s consider what those distractor types could mean in a real-life scenario, by coming back to our humanoid robot example.

Example of a ***random*** *distraction*:
- Additive white additive noise observed on sensor or camera readings.

Example of a ***time-varying*** *distraction*:
- The camera sensor will produce observations whose brightness will vary depending on the time of day.

Example of ***action-dependent*** *distraction*:
- When driving, our steering actions will change the position of clouds in our field of view.

As uncontrollable distractions are less intuitive to interpret, we design three distractors (random, time-varying, action-dependent)—each introducing a different challenge that the learner algorithm must overcome. Specifically, we modify the Cartpole environment such that each observation is appended distracting variables. A given run on Cartpole is therefore parametrized by `n_distractors` (how many distracting dimensions are appended onto the state), as well as `distractor_type`.

To ensure each distractor type has a comparable effect on neural network gradient updates, we design generating functions with similar ranges of values (in our experiments, using larger amplitude distractors made learning considerably harder). An ideal abstraction should discard any task-irrelevant information and therefore throw away all appended dimensions, producing a 6-dimensional feature embedding that maps to the underlying Cartpole state 1-to-1.

Level 1: Gaussian distractors

For the simplest level of distraction, each distracting dimension is populated with noise taken from a standard normal distribution. Since each time step is uncorrelated from previous timesteps, we expect that most algorithms will be relatively robust to them—and that they will be easily ignored.

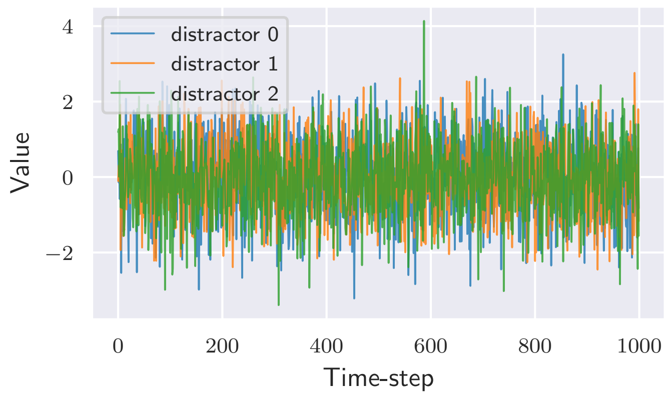


Figure 5.3: Three Gaussian distractors plotted over 1000 time-steps.

Level 2: Sine distractors

We introduce a time-varying aspect. Each distracting dimension now corresponds to an independent sinusoid that evolves in a time-consistent manner throughout the episode. This suggests that any forward model learned based on observation will be able to reconstruct the sine waves. We set all sinusoids to the same frequency, and randomly initialize the phase of each dimension when a new episode begins. The wave for an individual distractor is described by:

$$d_i(t) = \sin(2t + \phi_i) \tag{5.1}$$

where t corresponds to the elapsed time ($t = 0.01 \times \text{time-step}$).

Level 3: Action-walk distractors

Finally, we make distractor values action-dependent. We define an action-walk distractor as a random-walk trajectory where the step direction is a function of the action taken:

1. Each distracting variable is uniformly assigned a starting value $\zeta \in [-2, 2]$, and a step size $\xi \in [-0.1, 0.1]$.

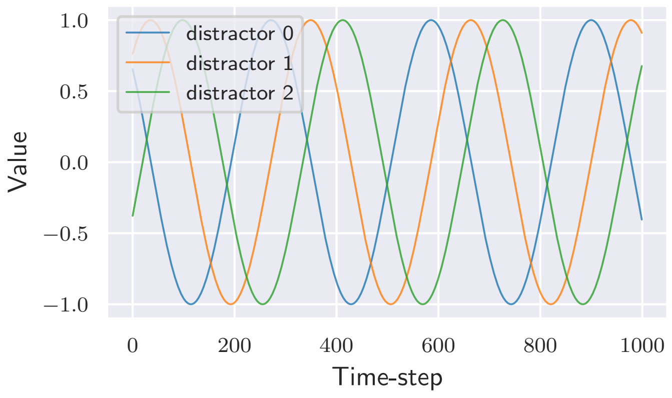


Figure 5.4: Three sine distractors plotted over 1000 time-steps.

2. When a new episode begins, the state x of the distractor is set as $x_0 = \zeta$.
3. At each time-step, the agent takes an action $a \in -1, 0, 1$. We want to update the state as $x_{t+1} = x_t + a\xi$. To ensure that distractor values remain reasonably bounded, we cycle values around $[-2, 2]$, yielding the update:

$$x_{t+1} = \mod (x_t + a\xi + 2, 4) - 2 \tag{5.2}$$

With the distraction received by the agent not only varying consistently with time but being also affected (predictably) by actions taken—it is more challenging to determine whether these variables are relevant to the task at hand (due to spurious correlations).

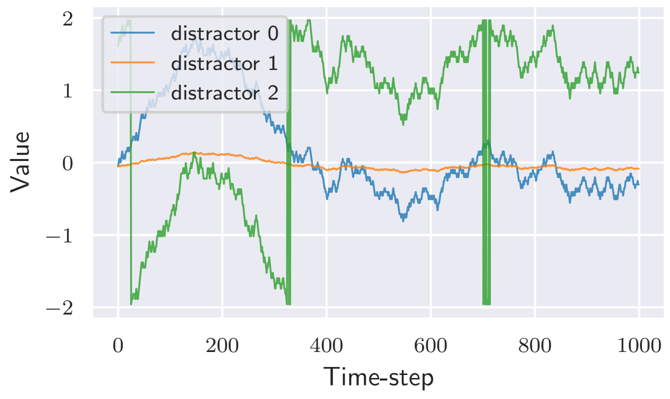


Figure 5.5: Three action-walk distractors (action-dependent random walks) plotted over 1000 time-steps, under a random policy.

5.3 Online vs Offline Gap

5.3.1 Constructing an offline dataset

For this work, we would like to evaluate methods of solving the Cartpole problem in the offline RL setting. As there is, to the best of our knowledge, no reference offline dataset for the classical control suite of tasks, we proceeded to collect our own dataset.

Data Generation The dataset is generated by running an online DQN agent and recording transitions from its replay during training. We use data from five runs with 1000 episodes each (approx. 3M transitions). To ensure that our dataset consists of both poor and expert examples (we want a diverse behavioural policy), we qualitatively lower the agent's learning rate such that expert behaviour (episode reward of ~ 1000) is achieved for fewer than 50% of transitions.

Figure (5.6) shows 5 sample online DQN runs used for data generation. Details of the Q-network and hyperparameters used are available in Appendix (B.1.1).

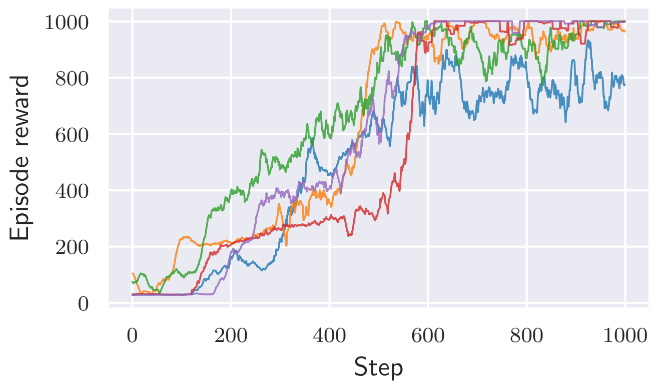


Figure 5.6: Five online DQN runs used to generate an offline Cartpole transition dataset. As the agents only converge after ~ 600 steps, the dataset contains a large proportion of suboptimal trajectories.

The final product of this process is a serialized Tensorflow dataset `.record` file, consisting of SARS examples (State-Action-Reward-State), as well as additional metadata such as episode termination (which we will use to appropriately discount states in the offline DQN agent). We additionally write a vectorized Tensorflow dataset pipeline to dynamically introduce Gaussian, sine, or action-walk distractors to individual runs.

After training agents to solve the Cartpole task based on a dataset collected with this procedure, we noted large variations in performance depending on the dataset being used. Training agents on datasets created from identical parameters would yield vastly inconsistent performance. Despite their identical parameters, different online runs naturally converge faster or slower—and explore different parts of the state-action space. With BC, for example, the agent is only as good as the average policy in the dataset—so a better performing online run is likely to lead to better offline performance (Figure 5.7). Additionally, if the agent encounters states that are not well-represented in the dataset—it is forced to rely on function approximators' extrapolation ability.

Remark. As this is not a real-life offline RL scenario, we need not be concerned with avoiding environment interactions to evaluate the performance of a policy discovered offline. All of our experiments will therefore evaluate performance by running online episodes at regular intervals throughout training. These online rollouts are purely for evaluation and do not affect any of the agent's parameters.

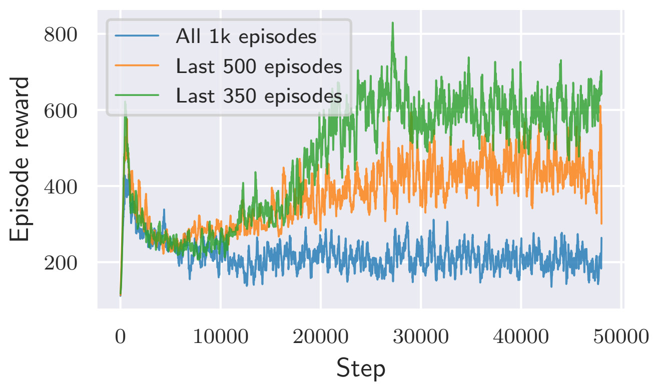


Figure 5.7: Training a behavioral cloning agent on different slices of the same dataset. The dataset contains 5 online DQN runs with 1000 episodes each. Since BC imitates the average behaviour in the data, restricting to the later episodes of each run (where the agent has already learned to achieve high reward) yields a better imitative policy.

While some agents are more robust than others, it is clear that dataset size and dataset coverage of the state-action pairs are of paramount importance to learn a strong offline RL policy. In fact, for a fixed-size dataset, offline RL presents a trade-off between trajectory quality and state-action coverage—similar to exploration vs exploitation in online RL. We need the dataset to include high-quality actions so that we can learn to achieve high reward. We also need the dataset to include unavailing exploratory actions, to better avoid low-reward scenarios.

In light of these observations, we decided to modify our data-collection DQN agent. Following a suggestion by Monier et al. (2020), we inject noise into transitions by replacing the actions from an agent with a random action with probability $\epsilon = 0.3$. The injected noise ensures better exploration of the state space and empirically resulted in more stable offline learning.

5.3.2 N-step bootstrapping considerations

Debugging an offline RL algorithm is non-trivial as the potential error sources are many. When a given implementation is dysfunctional, this could be due to anything from a code-related bug, the quality of the dataset, the value of the hyperparameters, or it may simply be that the algorithm is insufficient to tackle the given task.

After compiling the offline Cartpole dataset, we encountered an issue that took us a long time to uncover and made it impossible to learn a good offline policy. As this issue related to n-step bootstrapping, let's first expand slightly on the definition:

N-step bootstrapping Most RL methods that do not assume a perfect model of the environment fall into two categories *Monte-Carlo (MC)* and *Temporal-Difference (TD)* methods.

Monte Carlo methods solve the RL problem by sampling and averaging returns for each state-action pair. A MC update for a state is based on the entire sequence of observed rewards, from that state all the way until the end of the episode.

TD methods, on the other hand, update estimates based in part on other learned estimates without waiting for a final outcome (c.f. the DQN computation in Equation 4.4). One-step TD methods, use the one next reward and "bootstrap" from the value of the state one step later as an estimate for the remaining rewards.

An intermediate method consists of updating values based on an intermediate number of rewards (i.e. more than one reward, but less than all rewards until termination). This approach is referred to as "n-step TD" prediction. Figure (5.8) from Sutton and Barto (1998) illustrates this spectrum which ranges from one-step TD methods to MC methods at the other extreme.

Since the DQN uses Q-learning to bootstrap, it belongs to the category of TD algorithms. Depending on whether multiple observations are aggregated together to produce n-step transitions, the DQN can also perform n-step TD learning.

Choosing N Depending on the task, different values from N can perform better or worse. There is in fact a bias vs variance trade-off when choosing the value for N.

TD methods update Q-values based on estimates. At the start of learning, these estimates contain no information from real transitions and are therefore biased on the initial conditions of learning parameters. While this bias reduces asymptotically over many iterations, it can cause significant problems, especially when learning off-policy and when using function approximation. In fact, the combination of function approximation, bootstrapping and off-policy is commonly referred to as the "deadly triad" (Van Hasselt et al., 2018). MC methods do not have this bias issue, but they can often suffer from high variance and can therefore require a larger number of

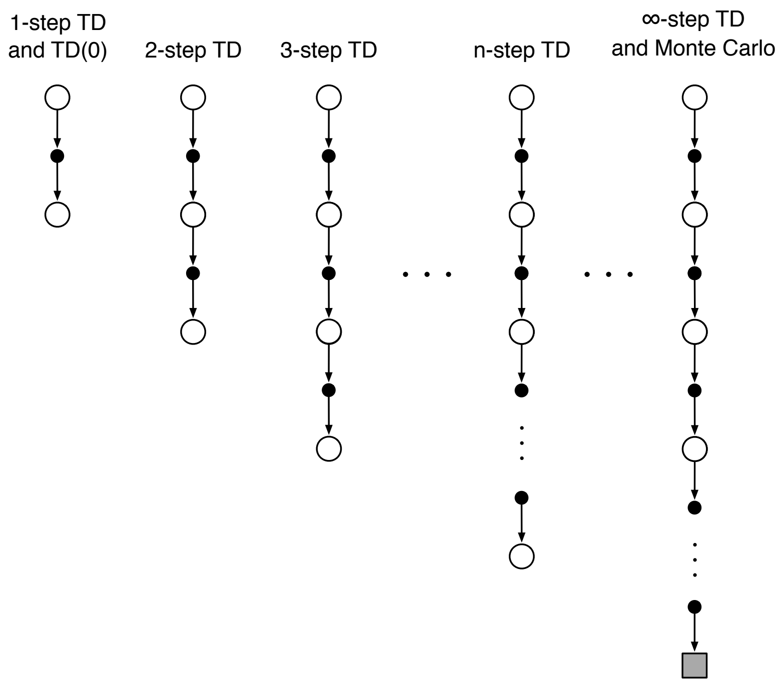


Figure 5.8: Backup diagram comparing n-step bootstrapping methods. The backup operations transfer value information back to a state. Low n: higher bias, lower variance. High n: lower bias, higher variance. Figure from *Reinforcement Learning: An Introduction* (Sutton and Barto, 1998)

samples to attain the same degree of learning.

Issue encountered As mentioned previously, we initially collected our dataset by running an online DQN agent using 5-step transitions. However, we were logging and training our offline agent on 1-step transitions. Using this dataset, we were able to achieve satisfactory performance with BC but observed highly unstable behaviour when training the offline DQN (Figure 5.9).

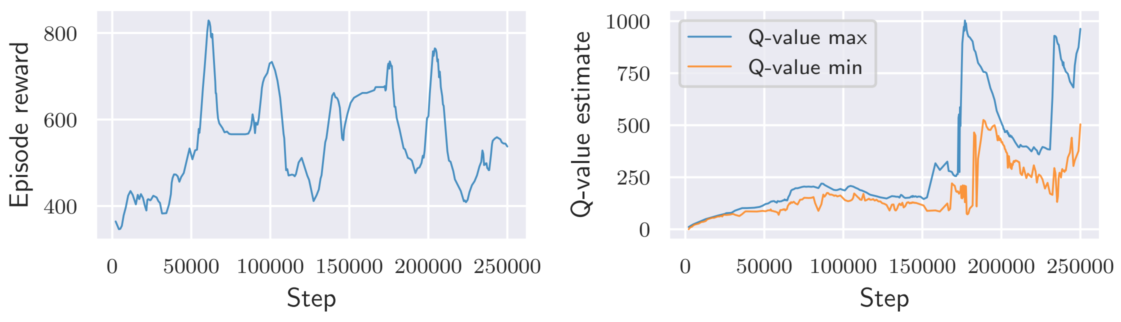


Figure 5.9: Instability when training an offline DQN on 1-step transition data. Agent performance continuously fluctuates between high and low episode reward. Note how predicted Q-values jump to over-estimated values.

After extensive investigation, we were surprised to find that an online agent had difficulty solving Cartpole when presented with 1-step transitions (Figure 5.10)—and that the difference with any $N > 1$ was significant. Taking this into account, it was simple to explain why the offline agent could not solve the task: it was trying to solve a difficult online problem, with all the extra challenges of offline learning.
In light of this result, we modified our data collection pipeline to directly capture 5-step transitions from the online agent's replay buffer.

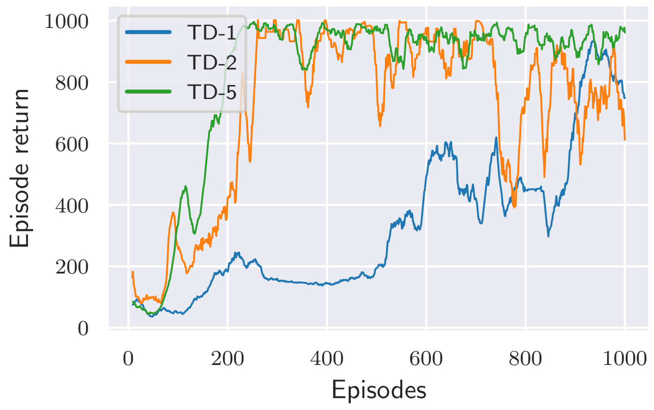


Figure 5.10: Comparison of an online DQN agent bootstrapping over different time intervals on Cartpole (averaged over 3 seeds). With 1-step transitions, the DQN is unable to learn a good policy for the first 600 episodes, likely due to the bootstrap bias. For $N > 1$, the agent converges in 200 episodes.

5.3.3 Offline performance

Figure (5.11) shows our final performance training offline algorithms on Cartpole without explicit representation.

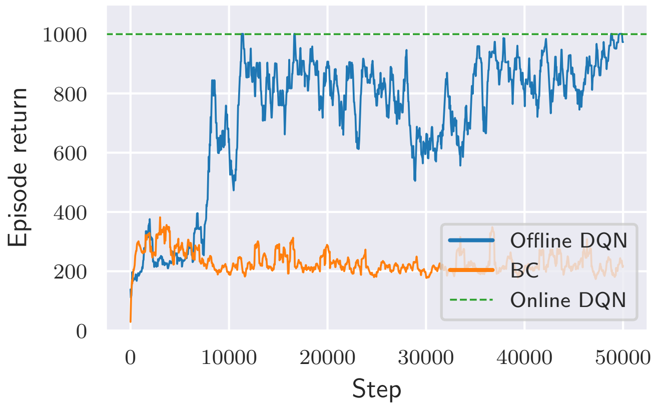


Figure 5.11: Performance of DQN and BC agents on the offline Cartpole data, without any auxiliary representation objective. The offline DQN achieves high episode returns—at times touching the optimal performance of online DQN—while the BC agent is unable to identify good behaviour and blindly learns to reproduce actions from the dataset.

5.3.4 Assessing distribution shift

In this section, we attempt to gain a better understanding of the effect of distribution shift on our DQN agent. To this end, we train an offline DQN agent and store all observations produced during its online evaluation. Using this data, we can compare state-visitation under the learned offline policy and the behavioural policy.

State visitation under the converged offline policy

We use kernel density estimation to plot estimated densities across pairs of state variables for both the final policy learned by the offline DQN and the states in the dataset (Figure 5.12).

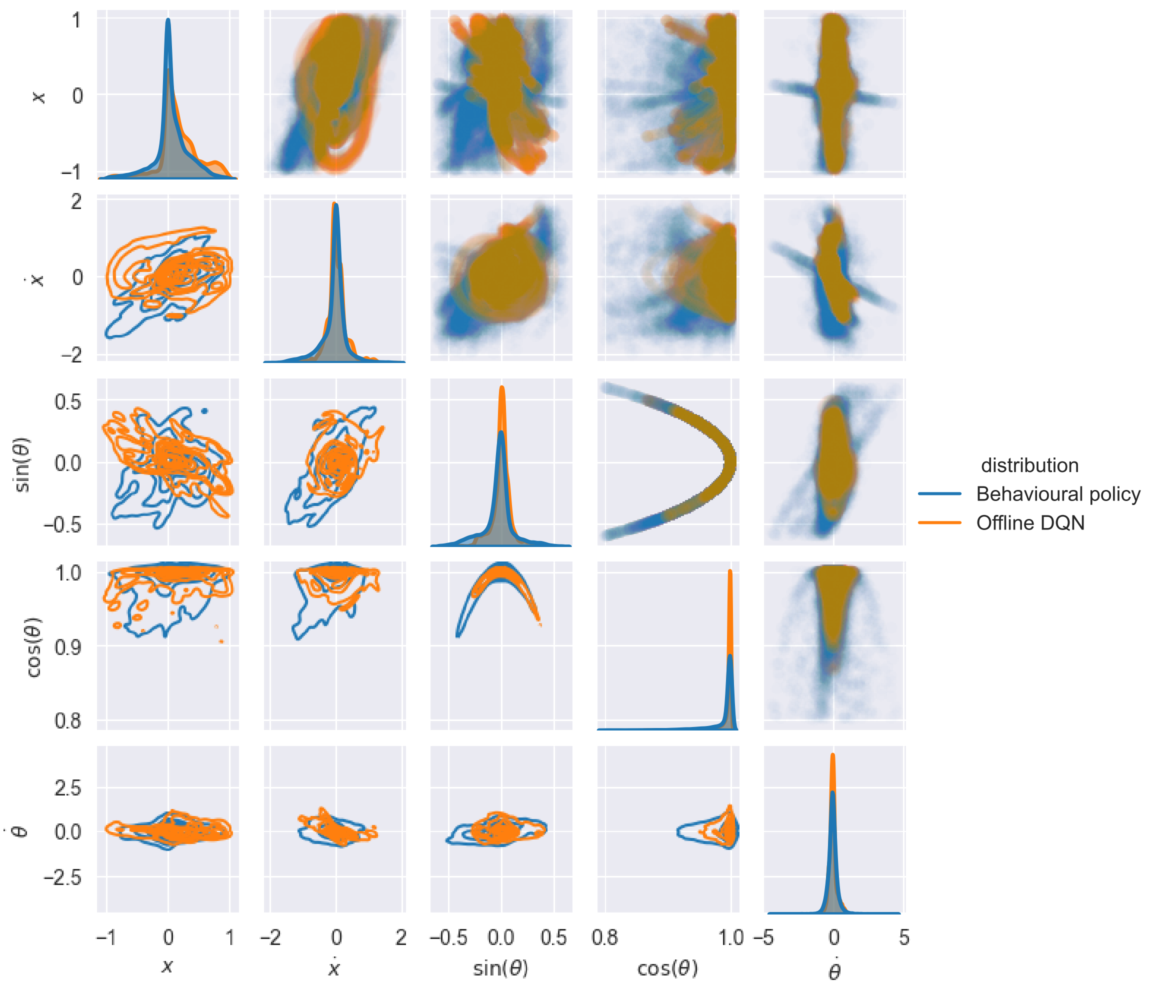


Figure 5.12: State visitation under the behavioral policy, and the policy learned by the offline DQN agent. Given that the state space is small and the offline agent learns a strong policy, we can observe no significant divergence from the behavioral policy.

We first notice that the two distributions have significant overlap. This is not unexpected, since the offline DQN reaches near-optimal performance at convergence. The offline DQN appears to be visiting states more concentrated around $\sin(\theta) = 0$.

Given that the angle θ is measured from the vertical axis, this indicates that the pole remains more upright—which makes sense as the dataset contains a proportion of 'bad' behaviour. A similar difference can be noted with $\cos(\theta)$. Secondly, we observe that the offline policy tends to visit more states on the right side of the x-axis—but on the whole, perhaps due to the small dimensionality of the state space, there seems to be no significant discrepancy.

State visitation over time

We would also like to find out whether any shift in the distribution of visited states increases throughout training, as the learner discovers new ways to achieve high reward—or begins to hallucinate.

To answer this question, we calculate the Wasserstein distance (Equation 3.1.1) between the distribution of states visited under the behavioural policy (i.e. states in the dataset), and a rolling window of the states visited when evaluating during training (Figure 5.13).

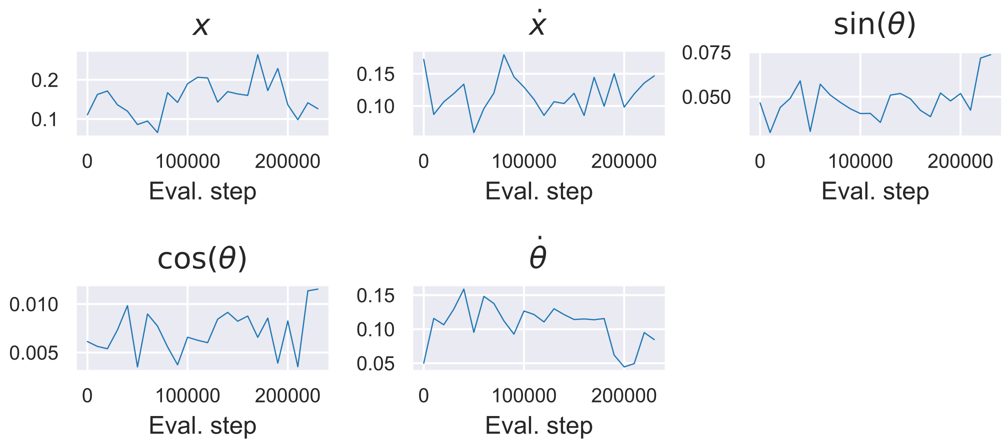


Figure 5.13: Wasserstein distance between states in the dataset and states visited by DQN throughout offline training. No clear trend can be discerned, in accordance with our previous finding from Figure (5.12) that minimal distribution shift occurs.

We note no clear pattern when it comes to diverging from states visited by the behavioural policy. We suspect this is again due to the agent's strong performance on the task, and the simplicity of the state space.

5.4 Learning with distractions

In this section, we show the impact of distractors on the ability of the BC and offline DQN agents to learn good policies. As we are not using any auxiliary loss yet, the algorithms must implicitly learn to represent the higher-dimension observations induced by the distractions. For a fair comparison, both algorithms use the same neural network to map from observations to action-values, or action probabilities in the case of BC. We find that a simple MLP with 128, 64, 64, 32 neurons in each respective hidden layer performs well. Finally, we regularize the BC network with Dropout (Srivastava et al., 2014) at each hidden layer (rate of 0.8).

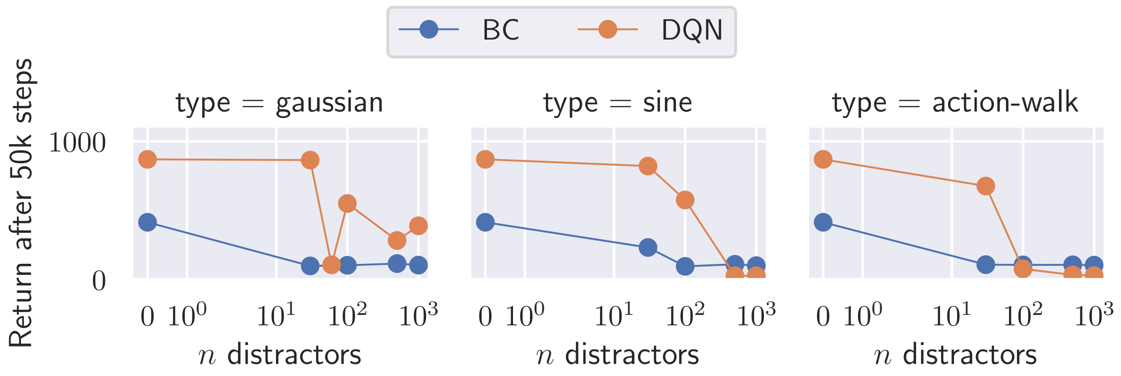


Figure 5.14: Performance of BC and DQN on our offline Cartpole dataset, for each type of distractor and varying numbers of distracting dimensions. Each data-point is averaged over 3 seeds. As anticipated, increasing distractors makes the learning problem more difficult and action-walk distractions have the largest impact on training.

Figure (5.14) illustrates certain trends that could be anticipated i.e.: increasing the number of distractors makes the learning problem more difficult, and that the distractor types are not all equally challenging to ignore.

5.4.1 Effect of distractors on Behavioral Cloning

When we initially trained the Behavioral Cloning agent, we did not regularize it and observed a result that was at first quite surprising: increasing the number of Gaussian distractors resulted in better performance. Specifically, increasing the number of distractors would lead to higher returns at convergence, and require more steps until convergence.

It turns out this uncanny trend was due to the fact that our network was over-fitting the dataset. This over-fitting would lead to achieving sub-optimal performance and, interestingly, increasing the number of distracting dimensions would act as a regularizing effect. After some research we found that adding noise to the inputs of a neural network is an (uncommon) approach to regularization. For a neural network, when we append Gaussian distractors to the 6 observation dimensions, this is somewhat equivalent to adding random noise to the input layer. Equation 5.3 shows the computation performed by a single neuron in the input layer:

$$h_1 = g(\Sigma_{i=0}^{n} w_i s_{t,i}) = g(\Sigma_{i=0}^{6} w_i s_{t,i} + \Sigma_{i=7}^{6+\text{n\_distractors}} w_i s_{t,i}) = g(\Sigma_{i=0}^{6} w_i s_{t,i} + C) \quad (5.3)$$

where g is the activation function and C is a weighted sum of the Gaussian distractors. From this, it is clear that adding more distractors would increase the magnitude of the effective regularization. By adding Dropout, we ensured the BC agent was not overfitting and that adding more distractors lowered the episode returns.

5.5 Learning with explicit state representations

In this section, we attempt to answer our primary question: can current SRL methods benefit agents learning from offline datasets? Recent work by Yang and Nachum (2021) explored pre-training objectives on offline data, but the authors used representation methods to learn directly from ground state. Instead, we are interested to find whether representations can help an offline agent learn from higher-dimensional observations that can distract from the MDP state (Block MDP setting from section 2.1.1).

We find that in our artificially constructed Cartpole MDP, SRL methods trained jointly as auxiliary objectives can dramatically improve performance of offline agents. We also find that not all SRL approaches are equally effective in learning robust representations, especially when facing distracting observations, and that they can be highly sensitive to fine-tuning of hyper-parameters.

5.5.1 Network architecture

To train the offline DQN together with auxiliary representation objectives, we used a neural network architecture consisting of:

- *(All methods)*. An encoder MLP, that transforms an observation s_t into a 64-dimensional representation $s_{\phi,t}$ (for the VAE, $s_{\phi,t}$ results from sampling).
- *(All methods)*. A MLP that maps representations to action-values (the DQN network).
- *(Autoencoder, VAE)*. A decoder MLP that reconstructs observation s_t from $s_{\phi,t}$.
- *(DBC, CDBC)*. A linear layer that projects $s_{\phi,t}$ onto $z_\phi = h(s_\phi)$.
- *(DeepMDP)*. Two deterministic transition models implemented as MLPs, one mapping $s_{\phi,t}$ to $\hat{s}_{\phi,t+1}$, the second mapping $s_{\phi,t}$ to $\hat{r}_{t+1}$.

Following previous work, the MLPs use layer normalization. More details on the architecture parameters can be found in Appendix B.1.3.

For the comparison between representations to be fair, encoders and Q-networks must have the same number of parameters across experiments. Furthermore, to fairly compare the plain DQN to the DQN with representations, we include the encoder as part of the Q-network MLP when using no representation.

5.5.2 Considerations when training SRL methods offline

While reviewing the SRL literature, we found that various implementational details of training state representations have no standard best practices and that different papers used different approaches. Some of these details, however, have the potential to significantly affect an agent's learning curve and we believe should be evaluated more rigorously. We discuss two such practical open questions that are particularly relevant when training an offline agent with an SRL objective.

Training methodology

To train a DQN agent offline with an additional state representation objective, at least three approaches are possible:

- Train the DQN and representation jointly.
- Pre-train the representation objective on the fixed dataset, then train the DQN, propagating any gradients back to the SRL encoder.
- Pre-train the representation objective on the fixed dataset, then freeze the encoder and train the DQN.

(Clearly, the two pre-training approaches are irrelevant in online learning, as we do not have access to transition data ahead of time.)

Likely, these training approaches perform better or worse depending on the representation. To improve our understanding, we try these different training methods with a DQN and autoencoder or DeepMDP representations. When pre-training, we perform 50k update steps of the SRL objective, then train the DQN for 100k steps. When training jointly, we optimize a weighted loss of the SRL and DQN objectives. We evaluate our results on Cartpole with 100 Gaussian distractors, across 4 seeds.

From Figure (5.15), the first immediate observation is that freezing the representation after pre-training is unsuitable. Indeed, freezing the encoder requires the representation space to be directly usable to learn Q-values. This may be the case for certain representations but relies on the Q-network's capacity to map action-values from the frozen representation. Allowing the gradients to propagate back to the encoder enables the DQN loss to make slight adjustments to the learned representation such that action-values are easier to estimate. These results are in contrast with recent work by Yang and Nachum (2021), who found that freezing the encoder performed better than finetuning (the authors did not try joint-training and were learning directly from the ground MDP state).

We also note that while separate pre-training enabled DQN+DeepMDP to quickly achieve high reward, the performance at convergence is comparable. As there is no clear winner between these two approaches, each performing better on one of the two representations, we decide to train our representations jointly—avoiding the additional computation associated with pre-training.

Two methods for training jointly

We found that in online SRL literature, different papers would use different approaches to training a representation jointly with an RL algorithm. Two methods

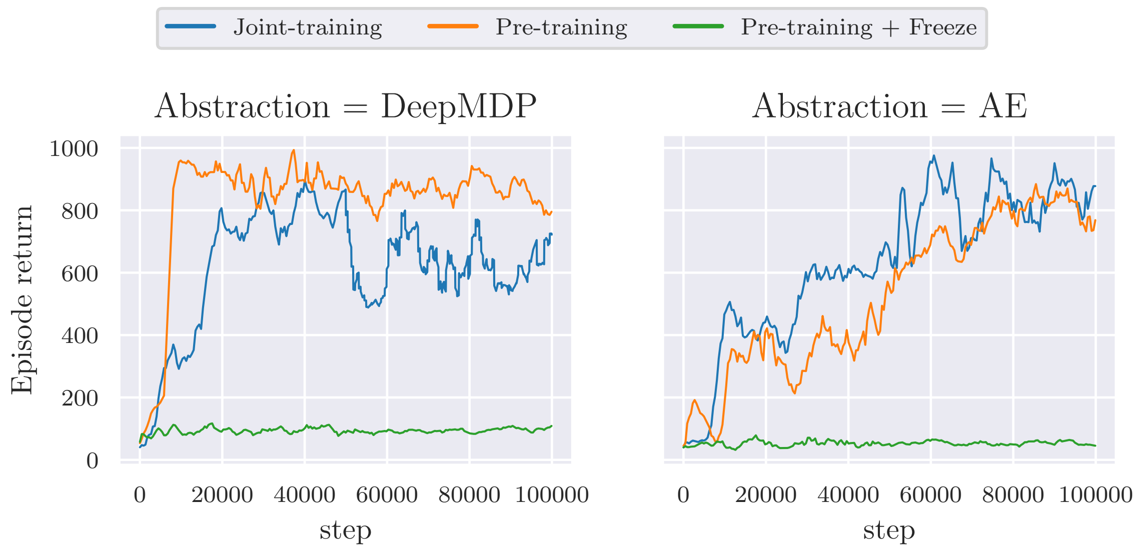


Figure 5.15: Comparing different offline SRL training methods on Cartpole with 100 Gaussian distractors (4 seeds per run). The DQN is unable to learn from a frozen pre-trained representation. Training the DQN and SRL objectives jointly achieves comparable performance to training representation and DQN in two steps without freezing the encoder. We choose the former method, as it requires fewer total training steps.

were most frequently encountered. The first is to use a single optimizer to minimize a loss consisting of a weighted sum of the two objectives. The second method uses two optimizers, one for each objective, and performs an update step for each optimizer at every iteration.

Using two separate optimizers requires relative tuning of their learning rates. Moreover, when using an optimizer that keeps a state (e.g. moving average of the gradients), the role of the interplay between the statistics of the two optimizers is unclear. We consequently opt to use the former method, as it appears more straightforward to us. We use a single Adam optimizer, as it is generally regarded to be robust against the choice of hyperparameters.

5.5.3 Tuning sensitivity

To ensure each SRL method is fairly portrayed in our evaluation, it was important to tune any significant hyperparameters. Specifically, we found that while performance tended to be insensitive to network parameters such as the embedding dimension, the learning rates and loss weighting factors were critical to tune. In fact, improper tuning of those parameters made any learning impossible.

Therefore, for every representation method, we extensively tuned the weighting factor used to combine the DQN loss with the representation objective, as well as the Adam learning rate. The final parameters discovered through bayesian search are available in Appendix B.3.

Contrastive DBC: batch size

Research in contrastive methods has shown that learning high-quality representations often requires including a large number of negative examples in the contrastive loss (Chen et al., 2020). Typically, contrastive methods obtain negative examples from the current batch (a technique called “in-batch negative sampling”). It follows that the number of negatives is increased by scaling the batch size, within the limits of available GPU memory.

We didn’t explore different batch sizes for CDBC, as we intended to keep training conditions identical across methods. We hypothesize, however, that these general findings also translate to contrastive metric embeddings—and should therefore be explored further.

AE weight scaling

In the case of the autoencoder representation, note that since we are using an L_2 reconstruction loss, the magnitude of the auxiliary loss will vary depending on the number of distracting dimensions.

To compensate, we follow the scaling approach used by Higgins et al. (2017) to compute a normalized β_{norm} factor for the β-VAE—and scale our weight linearly with the size of the observation. Specifically, we use $w_N = w^*_{100} \cdot \frac{100}{N}$, where N is the dimension of s_t and w^*_{100} denotes the weight tuned for 100 input dimensions. This ensures that the reconstruction loss is first scaled to approximately the same order of magnitude, before being weighted against the DQN loss.

VAE β sensitivity

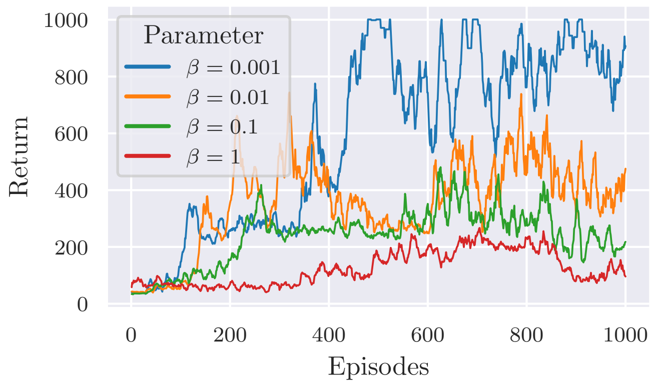


Figure 5.16: Comparing online runs of the DQN trained jointly with a β-VAE objective for different values of β. Reducing β consistently improves the learned policy, resulting in a deterministic embedding. (Trained directly from Cartpole state with no added distractions.)

In our initial experiments with the VAE, we found that the KL divergence loss severely degraded performance of the DQN. As shown in Figure (5.16), the performance of a DQN trained jointly with the VAE representation is inversely correlated with the weight β of the KL loss.

However, without KL regularization the VAE effectively behaves like a regular autoencoder (the predicted variances collapse to 0, resulting in a deterministic embedding). Since we were unable to learn a well-performing stochastic representation for this dataset, and the autoencoder consistently outperformed VAE, we decided not to include the VAE in our experimental results.

5.5.4 Convergence failures

While conducting our experiments, we found training of the DQN to be unstable in the following circumstances:

- **Many distractors, no reward signal**. Training of the DQN with either no representation or an autoencoder would diverge when subject to a large number of distractors. Predicted Q-values would grow unbounded and the agent would be unable to learn a policy. When the signal-to-noise ratio becomes very low, as is the case when there are many distractors, it becomes increasingly easy for spurious correlations to be picked up by the Q-network. Due to the offline agent's inability to 'self-correct' through interaction, the 'deadly triad' combination of function approximation, bootstrapping and off-policy learning can cause values to diverge. We also observed that this was less likely to occur with Gaussian distractions. They are easier to ignore, by design, as they cannot be predicted and there is no temporal relationship to facilitate spurious correlations with Q-values. Methods that use reward did not diverge, presumably because they better eliminate irrelevant information such that only useful correlations are learned by the DQN.

- **Directly embedding bisimulation**. When we implemented DBC, it quickly became apparent that the method is prone to instabilities when trained offline. While the sample-based approach described in Section 4.2.4 reliably converged in the online Cartpole setting (Figure B.1), training it offline proved to be challenging. The DBC loss would explode and output `NaNs` regardless of tuning. We were able to stabilize it by re-introducing a forward-latent prediction model and computing the bisimulation metric over predicted next representation. However, for certain distractor settings, the representation still diverged and the agent was unable to learn (c.f. Figure 5.17). Interestingly, our proposed contrastive embedding of the bisimulation metric did not suffer from divergence.

5.5.5 Benchmarking results

Figure (5.17) shows the performance of the various SRL methods when trained on our distractor benchmark for 70k steps. We provide for reference in Appendix (B.1) the same evaluation on an online DQN agent. Our keys insights from these results:

1. **Explicit representation objectives can help offline learners discover significantly better policies**. The offline DQN agent is unable to learn with 500 and 1k distractors—and sometimes diverges (5.5.4)—but when combined with representations it can find a good policy.

2. **Reward-informed SRL objectives consistently outperformed both reconstruction and the DQN baseline**. Without an auxiliary loss, the DQN baseline uses the encoder to find an implicit representation that is informative of future rewards (Q-values). However, due to a combination of bootstrapping, spurious correlations induced by the distractors, and the moving Q-target, the agent is unable to learn in some settings. Directly using reward in an auxiliary objective encourages the agent to focus on task-relevant aspects of environment dynamics. On the other hand, the autoencoder tries to learn a representation that can reconstruct not only the raw state dimensions but also the distractors, resulting in a necessarily lossy representation. Reconstruction objectives can be effective in compressing data that is highly redundant or spatially correlated but, in our benchmark, each distracting dimension was independently generated.

3. **CDBC is more stable to train than DBC, outperforming DBC in several settings**. Contrastive DBC equalled or surpassed the performance of DBC on every task except 500 and 1k action-walk distractors. Furthermore, unlike DBC which diverged even on the original Cartpole with no distractors, we did not face instability when training CDBC. Again, we note that we didn't investigate whether larger batch sizes could further enhance performance.

Remark. The Offline DQN predictably learned better policies than BC. As previously discussed, imitation learning methods will not work well on datasets containing non-expert data.

5.5.6 Evaluating the representation space

The best evaluation of the quality of a learned state-space representation is the performance of an RL algorithm on the target task. However, different algorithms may result in different performance for the same state representation, so in this section we discuss alternative evaluation methods.

Let's forget about the agent for a moment and focus our attention on the representation space. To directly assess the quality of an embedding, some aspects that can be considered are:

- **Suitable geometry**. The primary purpose of a representation is to embed each observation in a space that makes it easier for downstream networks (e.g. a

Q-function estimator) to extract meaningful information for completing a task. In order for a DQN to have a chance at finding the optimal policy from a state representation s_ϕ, a requirement is that it should be possible to learn the mapping $Q^*(s) = f(s_\phi)$. To find whether a given representation satisfies this requirement, we could collect pairs of representations $s_\phi = \phi(s)$ and optimal values $Q^*(s)$ and assess whether a supervised model can learn the mapping. In reality, we not only need s_ϕ to map to Q^*, but also to all Q' that must be learned before Q^* in the bootstrapping process.

- **Nearest neighbours**. One way to qualitatively evaluate a representation is to pick a state and find its nearest neighbours in the latent space, such as to confirm whether the representation has learned to group states that we consider to be similar for the ground truth state we intend to learn (Lesort et al., 2018). It is also possible to quantify this using KNN-MSE (Lesort et al., 2017).

In our case with the distracting Cartpole environment, we have direct access to the ground state that fully defines dynamics. We can therefore use it to conduct a "glass box" analysis of each representation learned and assess how state information is preserved.

We jointly train each representation method with an offline DQN agent on 500 distractors and save the encoder networks. We then collect a different dataset of online interaction with 500 distractors which we use as a test dataset. Using this dataset, we extract the raw Cartpole state (6 dimensions) and compute each representation (64 dimensions) using the saved encoders.
With these pairs of raw states and representations, we can directly assess how well ground-truth information is encoded in the latent space, and how easy it is for a downstream task to extract it. We train a model to decode the ground state from each representation and calculate the mean-squared error of the predictions on the test data. This error should vary as a function of model complexity, so we decide to find which representations are most predicting of the ground state under simple linear regression.

	Auxiliary objective				
	None	AE	DeepMDP	DBC	CDBC
Gaussian	0.046	0.025	0.041	**0.010**	0.21
Sine	0.047	0.044	0.038	**0.010**	0.045
Action-walk	N/A	0.045	0.040	0.019	**0.017**

Table 5.1: "Glass box" analysis. For each representation, we train a linear regression to predict the ground Cartpole state from s_ϕ and record the MSE on test data. Note how the ability to easily recover the ground state from a representation does not correlate with online evaluation results. 'N/A' indicates divergence of encoder weights.

From our results in Table 5.1, we see that whether a representation is directly predictive of the ground state (e.g. through linear regression) does not correlate well with task performance. This can be explained by the fact that the decoder network we use to map representations to Q-values in the DQN is larger, and that the representation need not necessarily encode the raw state; rather, it needs to discover features that explain Q-values.

The DeepMDP representation, for example, is the best performer on 500 distractors yet a linear regressor is unable to accurately recover the state. It is interesting to note how DBC and CDBC, which learn to embed a bisimulation metric produce the lowest decoding error. This makes sense as the bisimulation metric will cluster states that are behaviorally equivalent, and therefore likely to be neighbours in ground state.

5.5.7 Summary

In conclusion, the experimental results on our distractor benchmark suggest that the online benefits of explicit representation such as improved sample efficiency and generalization can translate to learning better offline policies. We also showed that, in distracting environments, representations that leverage the reward signal tend to perform better. However, we found that all SRL objectives we investigated can be unstable and tricky to tune, and proposed a contrastive representation that preserves the theoretical appeal of bisimulation metrics with empirically stabler training.

In the next chapter, we will try to scale these findings to more complex MDPs and high-dimensional pixel observations.

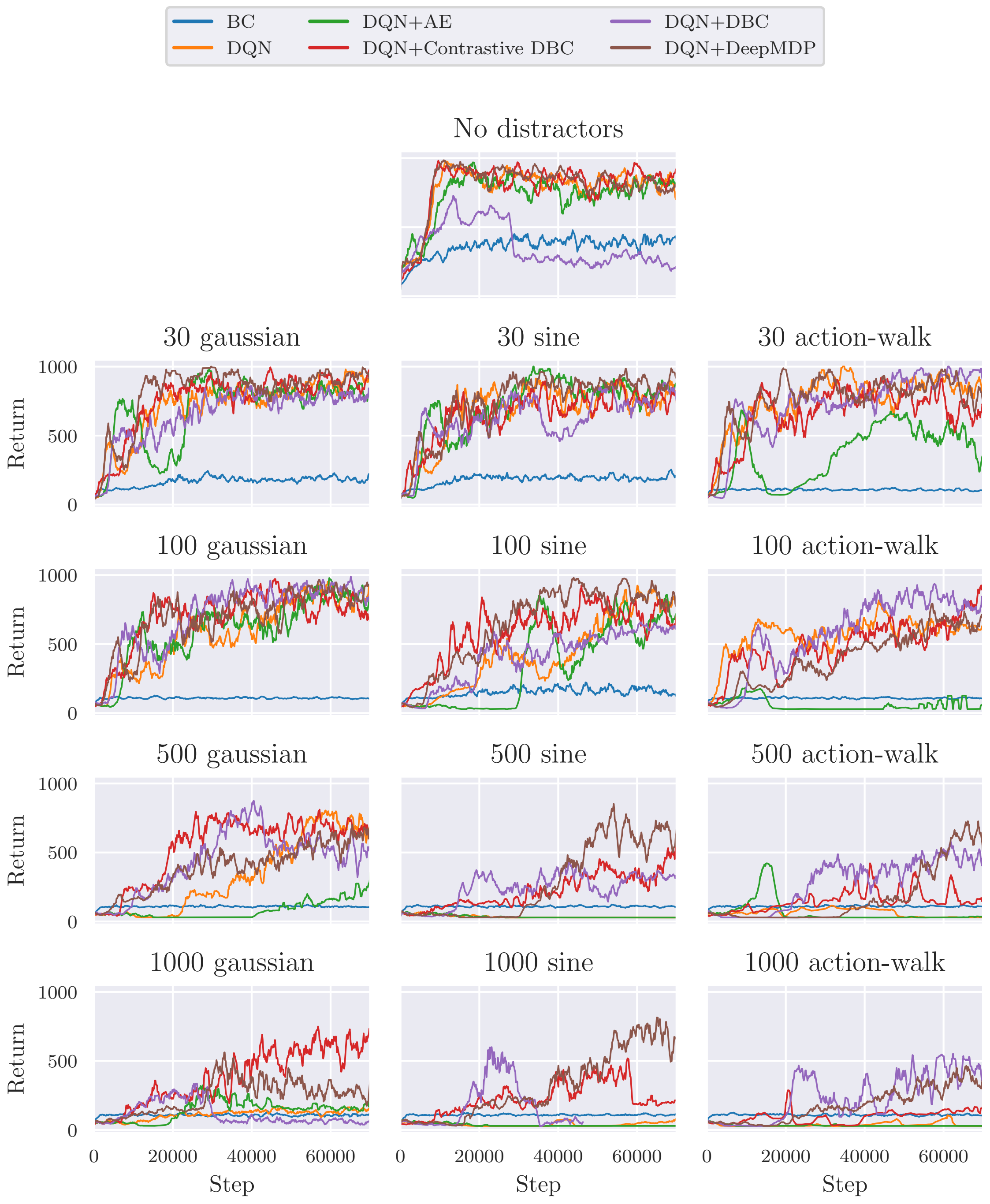


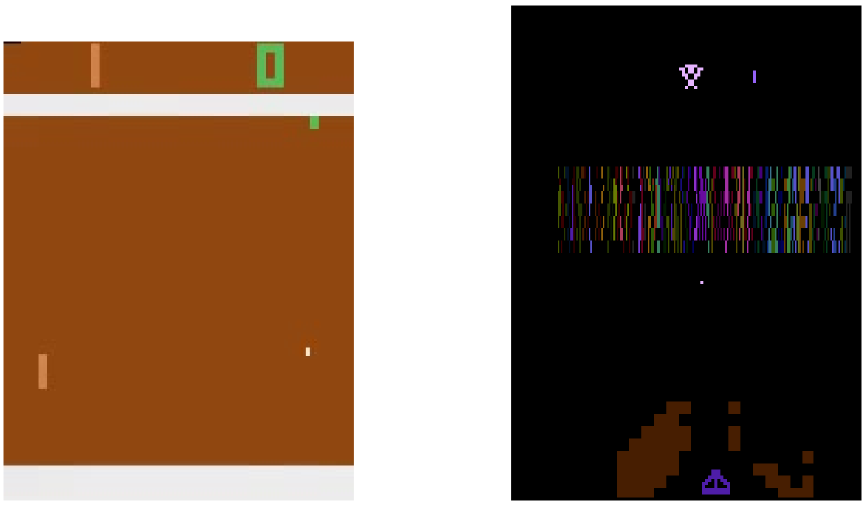
Figure 5.17: Benchmarking results. Training curves for each SRL method discussed over different numbers and types of distractors, average across 3 seeds. We outline our conclusions in Section 5.5.5.

Chapter 6

Large-scale experiments

In this chapter, we assess the performance of SRL methods on larger-scale offline tasks. Through experiments on Atari 2600 games we find that, without careful tuning, auxiliary representation objectives will not help learn better policies and can even be detrimental.

6.1 Atari 2600 offline



(a) Pong

(b) Yars Revenge

Figure 6.1: Sample frames from the two Atari games we pick. **Pong**: the agent controls a paddle by moving it vertically across the screen, simulating a table tennis game. **Yars Revenge**: for a detailed description of the gameplay, refer to online documentation. Note the 'neutral zone' across the screen that we consider to be a distraction.

Bellemare et al. (2013) introduced the Arcade Learning Environment, a platform to develop agents that play Atari 2600 games. The RL community has since used it as a challenging problem to measure the progress of RL methods in learning from pixels. With a discrete action-space, Atari games provide a suitable environment to evaluate the DQN with the different SRL methods in a more realistic offline task.

Offline Atari datasets

Recent work by Gulcehre et al. (2021) has proposed a benchmark called 'RL Unplugged' to evaluate and compare different offline RL methods. Among these benchmarks, the authors made available a collection of datasets that can be used to train offline agents on Atari games.

As with our distractor benchmark, each entry in an Atari dataset consists of SARS transitions. For the 57 Atari games included, the dataset was generated by running an online DQN agent and recording transitions from its replay buffer during training. Each game's data includes 5 such online runs with 50 million transitions each. Note that the data-collection agent uses sticky actions and that each observation consists of 4 stacked grayscale frames to ensure the Markov property is satisfied. As such, the dimension of an observation is $84 \times 84 \times 4$.

6.1.1 Game selection

Given our limited compute, we had to carefully select which experiments to run. We individually reviewed all of the 57 Atari 2600 games included in RL Unplugged and ranked them on several dimensions such as game-play type, dynamics complexity and the presence of task-irrelevant graphics. Using this classification, we selected two games:

- **Pong**. A simple, well-understood game that we used to scale the various methods to the pixel domain.
- **Yars Revenge**. A lesser-known member of the Atari 2600 suite. We selected it for its more complicated dynamics, and the presence of visual distractions: a "neutral zone" in the middle of the frame consisting of random flashing colours.

6.2 Benchmarking results

Our results from evaluating the different SRL methods on the games are shown in Figure (6.1). Details of the architecture used are in Appendix (B.2). Note that we exclude DBC, as it diverged in all experiments.

Remark. While state-of-the-art RL algorithms have been able to surpass human performance on most Atari games, they are still very expensive to train and require a lot of data. Our transition dataset for Pong and Yars Revenge was 370GB and 1.5TB respectively, an entire order of magnitude larger than ImageNet. Depending on the algorithm, it is not uncommon for an online agent to take 10 days on GPU to converge. In the offline setting, training is somewhat cheaper since we don't need to simulate the environment—however each run in Figure (6.1) still took 15-48 hours to complete on 16 cores and an Nvidia GPU.

Because of time and compute limitations, we were unable to tune any of the hyperparameters (Appendix B.2), including important parameters like the weighting factor of each auxiliary objective. We believe this explains, at least in part, that our results show no clear advantage to using SRL methods on the two games. The curves for each method were largely comparable, achieving similar returns after 1M steps.

Remark. Imitation learning outperformed the offline DQN on Pong. Since Pong is an easier game, the online data-collection agent solved it relatively fast and therefore the dataset consists of mostly high-quality actions that the BC agent could imitate.

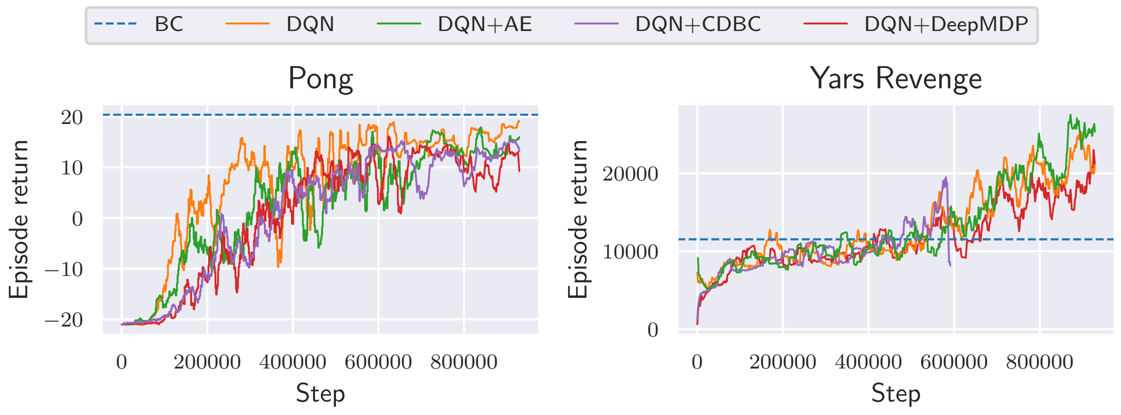


Figure 6.2: Training offline agents with different representation objectives on offline Atari data. No hyperparameters were tuned due to time/compute constraints. These results indicate no clear benefit of using representations on these tasks—which we attribute in part to the lack of tuning.

6.3 Additional evaluation

In light of these results, we decided to artificially complicate Pong—bringing it closer to a real-life distracting offline problem. The aim was to assess whether this helps explicit representation objectives stand out from the plain DQN.

6.3.1 Pong with distractions

We followed an idea by Zhang et al. (2020a) who simulate real-life distractions by incorporating natural video in the background of DeepMind Control Suite tasks. We replaced the dark background of the Pong game with a 15-minute compilation video of dashcam clips under different road and lighting conditions. The video was slightly faded to ensure the paddles and ball were discernible by the human eye. Figure (6.3) shows some samples observations with the background substitution performed.

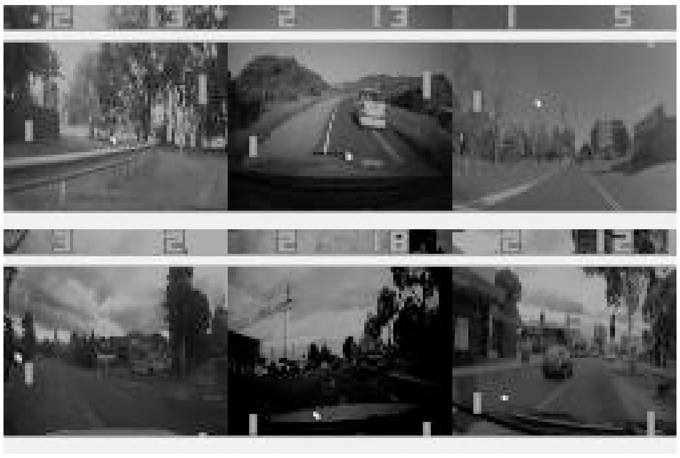


Figure 6.3: Example Atari frames with background substitution. Different backgrounds and lighting conditions can make the paddles and ball more or less visible. Note that to solve the game, an algorithm needs to locate the small ball which can be anywhere within the frame.

6.3.2 Results

After training the various methods on the modified Pong game for 600k steps, we found that only the plain DQN had made a substantial improvement in its returns (Figure 6.4). We again attribute this poor performance to a lack of tuning: SRL literature has highlighted the importance of tuning in the online setting (Gelada et al., 2019).

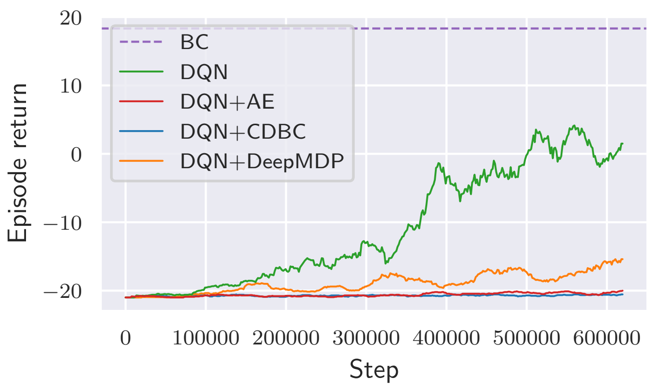


Figure 6.4: Offline training of different SRL methods on Pong with background substitution. While BC performance is unaffected by the natural video background, only the plain DQN learned a satisfactory policy after 600k steps.

To illustrate the consequences of improper tuning, we examine the reconstructed observations from the autoencoder. Figure (6.5) shows that the decoder network is unable to reconstruct the position of the Pong ball. This is not surprising since the its area relative to that of the frame is minimal: the ball has little effect on the pixel-wise reconstruction error. If we lowered the relative weighting of the L_2 loss, the

DQN objective would likely steer the encoder to a local optimum that can recover the ball.

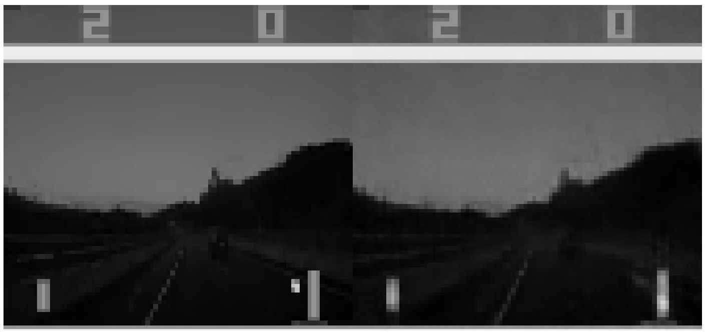


Figure 6.5: Left: a frame captured during online evaluation. **Right**: its reconstruction by the autoencoder. Despite the propagation of gradients from DQN to encoder, the jointly-trained representation is unable to capture the location of the ball.

Chapter 7

Conclusion

The primary aim of this thesis was to evaluate the effectiveness of current state representation learning techniques in offline reinforcement learning. As a first step, we developed a codebase of TensorFlow implementations for several popular SRL methods, including reconstruction-based approaches (autoencoders, VAEs) and approaches that leverage the reward signal (DBC, DeepMDP).

Second, we designed a benchmark to test the various methods' robustness to observations that contain task-irrelevant information. Through experiments on datasets generated from this benchmark, we found that auxiliary representation objectives can substantially improve the performance of offline RL algorithms. We found that on our benchmark, reward-based methods were most effective—especially in highly distracting environments. Since training with DBC was very unstable, we also proposed an alternative way to learn bisimulation metrics (Contrastive DBC) that leverages recent work in contrastive metric embeddings. In our experiments, Contrastive DBC does not diverge.

Finally, we evaluated (without tuning) the same SRL methods on larger-scale offline RL from pixels. Our initial experiments showed no performance improvement from using explicit representation. We altered the game of Pong to introduce realistic distractions and showed that meaningful representations could still not be learned. These results suggest that without extensive tuning, representation objectives can in fact be detrimental to an offline agent's performance.

Most of our findings coincide with the current understanding of state representation for online RL—and so we conjecture that there is significant overlap in the benefits of SRL for both online and offline tasks. Given that real-life scenarios often present agents with many task-irrelevant details, our results suggest that explicit representation has a role to play in offline learning but also highlight the importance of finding representations and training methods that are less sensitive to hyperparameter values. The project has opened multiple avenues for further research we would like to discuss.

7.1 Future work

The topic of representations in RL is quite vast and, naturally, our study of their use for offline tasks was limited in various aspects. Many open questions remain, forming opportunities for further research.

Different SRL objectives. While we examined a diverse set of representations, the picture we presented of the SRL landscape is far from complete. Many classes of methods that we did not explore, such as sequence models (Oh et al., 2017), or discrete representations (Oord et al., 2018), have performed very well in online RL benchmarks. We also did not consider objectives that fall outside of reward or reconstruction: downstream tasks like task transfer, exploration, or action prediction, can also lead to effective representations.

Real world. It would be interesting to extend our results to real-world domains. In robotics, for instance, online interaction with physical robots is expensive and representations can help accelerate the learning process (Arriola-Rios et al., 2020). Pre-training (robotic manipulators, for example) on offline data could enable RL to be used in a wider range of practical applications.

Hyperparameters. Some of our experiments showed representations can also have adverse effects on learning, which we attributed to a lack of tuning. It may be fruitful to explore which representations are most sensitive to the exact values of hyperparameters, and what techniques can render training more robust.

Offline RL algorithms. We only evaluated representations with an offline DQN. It may be worthwhile to validate how these results translate to other state-of-the-art offline algorithms like CQL or BRAC.
We are also curious about whether certain SRL methods could restrict latent space geometry in such a way as to limit distributional shift. This could provide an appealing alternative to policy-constraint methods.

Despite its limitations, we hope that our work will be valuable to the research community and will open up new avenues towards better offline SRL methods.

Bibliography

Abel, D.
2019. A Theory of State Abstraction for Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):9876–9877.

Agarwal, R., M. C. Machado, P. S. Castro, and M. G. Bellemare
2021. Contrastive Behavioral Similarity Embeddings for Generalization in Reinforcement Learning. *arXiv:2101.05265 [cs, stat]*.

Arriola-Rios, V. E., P. Guler, F. Ficuciello, D. Kragic, B. Siciliano, and J. L. Wyatt
2020. Modeling of Deformable Objects for Robotic Manipulation: A Tutorial and Review. *Frontiers in Robotics and AI*, 7:82.

Bansal, M., A. Krizhevsky, and A. Ogale
2018. ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst. *arXiv e-prints*, P. arXiv:1812.03079. _eprint: 1812.03079.

Barto, A. G., R. S. Sutton, and C. W. Anderson
1983. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846.

Battaglia, P. W., R. Pascanu, M. Lai, D. Rezende, and K. Kavukcuoglu
2016. Interaction Networks for Learning about Objects, Relations and Physics. *arXiv:1612.00222 [cs]*.

Bellemare, M. G., Y. Naddaf, J. Veness, and M. Bowling
2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279. arXiv: 1207.4708.

Cao, Z., E. Bıyık, W. Z. Wang, A. Raventos, A. Gaidon, G. Rosman, and D. Sadigh
2020. Reinforcement Learning based Control of Imitative Policies for Near-Accident Driving. *arXiv e-prints*, P. arXiv:2007.00178. _eprint: 2007.00178.

Chen, T., S. Kornblith, M. Norouzi, and G. Hinton
2020. A Simple Framework for Contrastive Learning of Visual Representations. *arXiv:2002.05709 [cs, stat]*. arXiv: 2002.05709.

Chicco, D.
2021. Siamese Neural Networks: An Overview. *Methods in Molecular Biology (Clifton, N.J.)*, 2190:73–94.

Du, S. S., A. Krishnamurthy, N. Jiang, A. Agarwal, M. Dudík, and J. Langford
2019. Provably efficient RL with Rich Observations via Latent State Decoding. *arXiv:1901.09018 [cs, stat]*.

Ferns, N. and D. Precup
2014. Bisimulation Metrics are Optimal Value Functions. *Uncertainty in Artificial Intelligence - Proceedings of the 30th Conference, UAI 2014*.

Finn, C., X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel
2016. Deep Spatial Autoencoders for Visuomotor Learning. *arXiv:1509.06113*.

Fujimoto, S., D. Meger, and D. Precup
2019. Off-Policy Deep Reinforcement Learning without Exploration. *arXiv:1812.02900 [cs, stat]*.

Gan, C., J. Schwartz, S. Alter, M. Schrimpf, J. Traer, J. De Freitas, J. Kubilius, A. Bhandwaldar, N. Haber, M. Sano, K. Kim, E. Wang, D. Mrowca, M. Lingelbach, A. Curtis, K. Feigelis, D. M. Bear, D. Gutfreund, D. Cox, J. J. DiCarlo, J. McDermott, J. B. Tenenbaum, and D. L. K. Yamins
2020. ThreeDWorld: A Platform for Interactive Multi-Modal Physical Simulation. *arXiv:2007.04954 [cs]*.

Gelada, C., S. Kumar, J. Buckman, O. Nachum, and M. G. Bellemare
2019. DeepMDP: Learning Continuous Latent Space Models for Representation Learning. *arXiv:1906.02736 [cs, stat]*.

Givan, R., T. Dean, and M. Greig
2003. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147(1).

Gulcehre, C., Z. Wang, A. Novikov, T. L. Paine, S. G. Colmenarejo, K. Zolna, R. Agarwal, J. Merel, D. Mankowitz, C. Paduraru, G. Dulac-Arnold, J. Li, M. Norouzi, M. Hoffman, O. Nachum, G. Tucker, N. Heess, and N. de Freitas
2021. RL Unplugged: A Suite of Benchmarks for Offline Reinforcement Learning. *arXiv:2006.13888 [cs, stat]*. arXiv: 2006.13888.

Ha, D. and J. Schmidhuber
2018. World Models. *arXiv e-prints*, P. arXiv:1803.10122. _eprint: 1803.10122.

Hasselt, H.
2010. Double Q-learning. In *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, eds., volume 23. Curran Associates, Inc.

Higgins, I., L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner
2017. beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Hinton, G. and R. Salakhutdinov
2006. Reducing the Dimensionality of Data with Neural Networks. *Science (New York, N.Y.)*, 313:504–7.

Hoffman, M., B. Shahriari, J. Aslanides, G. Barth-Maron, F. Behbahani, T. Norman, A. Abdolmaleki, A. Cassirer, F. Yang, K. Baumli, S. Henderson, A. Novikov, S. G. Colmenarejo, S. Cabi, C. Gulcehre, T. L. Paine, A. Cowie, Z. Wang, B. Piot, and N. de Freitas
2020. Acme: A Research Framework for Distributed Reinforcement Learning. *arXiv:2006.00979 [cs]*.

Hoque, R., D. Seita, A. Balakrishna, A. Ganapathi, A. K. Tanwani, N. Jamali, K. Yamane, S. Iba, and K. Goldberg
2020. VisuoSpatial Foresight for Multi-Step, Multi-Task Fabric Manipulation. *arXiv:2003.09044 [cs]*.

Jakab, T., A. Gupta, H. Bilen, and A. Vedaldi
2018. Unsupervised Learning of Object Landmarks through Conditional Image Generation. *arXiv:1806.07823 [cs]*.

Johnson, J., M. Douze, and H. Jégou
2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, P. 12.

Keskar, N. S., D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang
2017. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *arXiv:1609.04836 [cs, math]*.

Kingma, D. P. and M. Welling
2013. Auto-Encoding Variational Bayes. *arXiv e-prints*, P. arXiv:1312.6114.

Kipf, T., E. Fetaya, K.-C. Wang, M. Welling, and R. Zemel
2018. Neural Relational Inference for Interacting Systems. *arXiv:1802.04687 [cs, stat]*.

Kulkarni, T., A. Gupta, C. Ionescu, S. Borgeaud, M. Reynolds, A. Zisserman, and V. Mnih
2019. Unsupervised Learning of Object Keypoints for Perception and Control. *arXiv:1906.11883 [cs]*.

Kumar, A., A. Zhou, G. Tucker, and S. Levine
2020. Conservative Q-Learning for Offline Reinforcement Learning. *arXiv:2006.04779 [cs, stat]*. arXiv: 2006.04779.

Lample, G. and D. Chaplot
2016. Playing FPS Games with Deep Reinforcement Learning.

Lesort, T., N. Díaz-Rodríguez, J.-F. Goudou, and D. Filliat
2018. State Representation Learning for Control: An Overview. *Neural Networks*, 108:379–392.

Lesort, T., M. Seurin, X. Li, N. Díaz-Rodríguez, and D. Filliat
2017. Unsupervised state representation learning with robotic priors: a robustness benchmark. *arXiv e-prints*, P. arXiv:1709.05185.

Levine, S., A. Kumar, G. Tucker, and J. Fu
2020. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *arXiv:2005.01643 [cs, stat]*. arXiv: 2005.01643.

Li, L., T. Walsh, and M. Littman
2006. *Towards a Unified Theory of State Abstraction for MDPs.*

Li, Y., A. Torralba, A. Anandkumar, D. Fox, and A. Garg
2020. Causal Discovery in Physical Systems from Videos. *arXiv:2007.00631 [cs, stat]*.

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra
2019. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*.

Matas, J., S. James, and A. J. Davison
2018. Sim-to-Real Reinforcement Learning for Deformable Object Manipulation. *arXiv:1806.07851 [cs]*.

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller
2013. Playing Atari with Deep Reinforcement Learning. P. 9.

Monier, L., J. Kmec, A. Laterre, T. Pierrot, V. Courgeau, O. Sigaud, and K. Beguir
2020. Offline Reinforcement Learning Hands-On. *CoRR*, abs/2011.14379. _eprint: 2011.14379.

Oh, J., S. Singh, and H. Lee
2017. Value Prediction Network. *arXiv:1707.03497 [cs]*. arXiv: 1707.03497.

Oord, A. v. d., O. Vinyals, and K. Kavukcuoglu
2018. Neural Discrete Representation Learning. *arXiv:1711.00937 [cs]*. arXiv: 1711.00937.

Osband, I., Y. Doron, M. Hessel, J. Aslanides, E. Sezener, A. Saraiva, K. McKinney, T. Lattimore, C. Szepesvari, S. Singh, B. Van Roy, R. Sutton, D. Silver, and H. Van Hasselt
2020. Behaviour Suite for Reinforcement Learning. *arXiv:1908.03568 [cs, stat]*.

Pearson, K.
1901. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572. Publisher: Taylor & Francis.

Riedmiller, M.
2005. *Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method*, volume 3720.

Ross, S., G. J. Gordon, and J. A. Bagnell
2011. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. P. 9.

Seita, D., A. Ganapathi, R. Hoque, M. Hwang, E. Cen, A. K. Tanwani, A. Balakrishna, B. Thananjeyan, J. Ichnowski, N. Jamali, K. Yamane, S. Iba, J. Canny, and K. Goldberg
2020. Deep Imitation Learning of Sequential Fabric Smoothing From an Algorithmic Supervisor. *arXiv:1910.04854 [cs]*.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis
2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis
2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv:1712.01815 [cs]*. arXiv: 1712.01815.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov
2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

Suddarth, S. C. and Y. L. Kergosien
1990. Rule-injection hints as a means of improving network performance and learning time. In *Neural Networks*, L. B. Almeida and C. J. Wellekens, eds., Pp. 120–129, Berlin, Heidelberg. Springer Berlin Heidelberg.

Sutton, R. S. and A. G. Barto
1998. *Reinforcement learning: an introduction*, Adaptive computation and machine learning. Cambridge, Mass: MIT Press.

Tassa, Y., Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, T. Lillicrap, and M. Riedmiller
2018. DeepMind Control Suite. *arXiv:1801.00690 [cs]*.

Tobin, J., R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel
2017. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World. *arXiv:1703.06907 [cs]*.

Van Hasselt, H., Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil
2018. *Deep Reinforcement Learning and the Deadly Triad*.

Villani, C.
2008. Optimal transport – Old and new. volume 338, Pp. xxii+973.

Wu, Y., G. Tucker, and O. Nachum
2019. Behavior Regularized Offline Reinforcement Learning. *arXiv:1911.11361 [cs, stat]*. arXiv: 1911.11361.

Yang, M. and O. Nachum
2021. Representation Matters: Offline Pretraining for Sequential Decision Making. *arXiv:2102.05815 [cs]*. arXiv: 2102.05815.

Zhang, A., Z. C. Lipton, L. Pineda, K. Azizzadenesheli, A. Anandkumar, L. Itti, J. Pineau, and T. Furlanello
2019. Learning Causal State Representations of Partially Observable Environments. *arXiv:1906.10437 [cs, stat]*.

Zhang, A., R. McAllister, R. Calandra, Y. Gal, and S. Levine
2020a. Learning Invariant Representations for Reinforcement Learning without Reconstruction. *arXiv:2006.10742 [cs, stat]*.

Zhang, J., D. Kumor, and E. Bareinboim
2020b. Causal Imitation Learning with Unobserved Confounders. P. 26.

Zhang, Y., Y. Guo, Y. Jin, Y. Luo, Z. He, and H. Lee
2018. Unsupervised Discovery of Object Landmarks as Structural Representations. *arXiv:1804.04412 [cs]*.

Appendices

Appendix A

Manipulating deformable objects through keypoint representations

In a preliminary phase of this work, we explored the use of state representation learning methods for the interesting domain of robotic manipulation. Specifically, we looked at modelling and manipulating complex deformable objects like cloth with keypoint representations.

Keypoint representations lend themselves well to the control and prediction of deformable and articulated objects. Extracted keypoints can provide an effective "summary" of a deformable object. Using them, one can build a predictive model that efficiently approximates an object's dynamics (Kipf et al., 2018; Battaglia et al., 2016).

Completed work We initially replicated state-of-the-art results on the task of fabric-flattening following previous work by Seita et al. (2020). Then, we extended it by introducing a keypoint state-representation learned in an unsupervised fashion from pixels. Our aim was to show that keypoints can accelerate learning compared to using raw pixels.

The keypoint architecture we will be working with is based on a pixel-wise L_2 reconstruction loss. Through our experiments in learning to abstract a state by reconstruction, we encountered the problem of distracting features in the observation space. We show that even when task-irrelevant pixels appear to only mildly distract from the RL task, they can lead to a significantly less controllable representation.

Limitations After coming to terms with the limitations of reconstruction-based methods and the simulation environment (c.f. later sections), we decided not to pursue this domain further—these insights motivated us to review the literature on building state representations that are robust to distractions and helped inform the main thesis.

To facilitate future work on this interesting domain and provide a practical illustration of scenarios discussed in previous chapters, we document the steps we took.

A.1 Environments

To investigate state abstractions for deformable object manipulation, one of the first decision we needed to make was which simulation environment to use. We looked for simulation environments as 1) we need to know the ground state, 2) training Deep RL from pixel often requires a large number of samples that is prohibitively expensive to collect with a physical robot, 3) due to COVID-19, we are unable to physically access a robot in the lab.

Environments considered

One option we considered was designing our own RL environment and task. To that end, we would have needed to use a suitable simulator that supports deformable objects. Some options we evaluated include:

- **ThreeDWorld** (Gan et al., 2020) — a simulator recently released by MIT BCS (it is based on the Unity3D game engine). It supports cloth and softbody dynamics and provides an environment that is highly photo-realistic and multimodal: it synthesizes realistic collision sounds that are based on material properties of objects.
- **NVIDIA FleX** — a particle-based visual simulator that models all objects (including deformable materials) with a unified particle representation in real-time.
- **Blender** — a leading open-source 3D computer graphics software. It supports high-quality cloth simulation and rendering.

While this is a valid approach, building our own RL environment would require significant time investment—time likely better spent on the main topic of this thesis. Hence, we instead looked for environments that simulate deformable objects in the context of a manipulation task. After considering multiple alternatives, we decided to use Gym Cloth.

Gym Cloth

Gym Cloth (Seita et al., 2020) is an environment that simulates fabric for the task of sequential fabric smoothing. Given a deformable fabric lying on a flat plane of the same dimensions, the RL agent's objective is to manipulate the fabric from a start state to a state that maximally covers the plane. Gym Cloth follows the standard OpenAI Gym API convention.

At every time-step t, the agent receives an RGB image observation $o_t \in \mathcal{O}$ of the fabric, and takes action $a_t = (x_t, y_t, \Delta_{x_t}, \Delta_{y_t}) \in \mathcal{A}$ which is defined to be a 4D vector with the coordinates of a point to grasp and the pull direction. The pulling action is then simulated using a Finite Element Method (FEM) approach (Arriola-Rios et al., 2020). The performance of any learned smoothing policy will heavily depend on

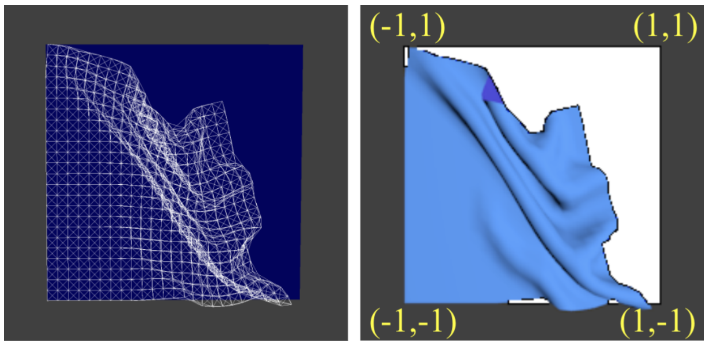


Figure A.1: Sample render of a Gym Cloth frame. Left: wireframe render using built-in rendered. Right: photorealistic render using Blender.

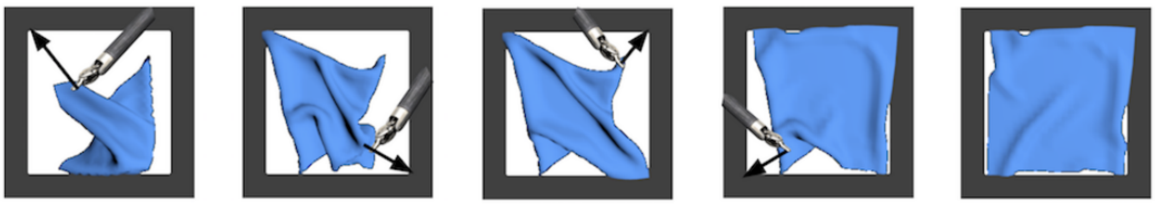


Figure A.2: A sample sequence of five actions to flatten the fabric. Taken from *"Deep Imitation Learning of Sequential Fabric Smoothing From an Algorithmic Supervisor"* (Seita et al., 2020)

the distribution of starting cloth states. Therefore Seita et al. (2020) provide three tiers of complexity for the starting state with average initial coverage of 78.3% for Tier 1, 57.6% for Tier 2 and 41.1% for Tier 3. On top of the environment, they

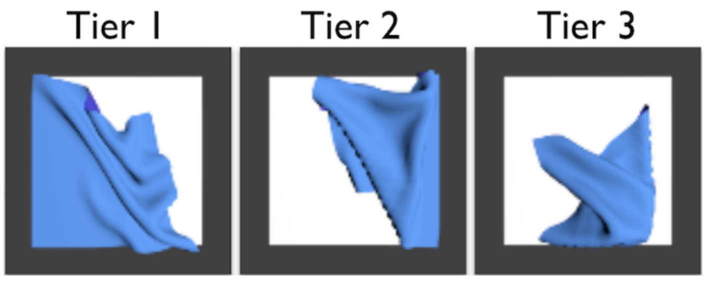


Figure A.3: Samples taken from three complexity tiers of starting distribution. Taken from *"Deep Imitation Learning of Sequential Fabric Smoothing From an Algorithmic Supervisor"* (Seita et al., 2020)

provide several algorithmic baseline policies that use the underlying state of the cloth—which the simulator provides access to—to take actions. We will use two of them:

- **Random policy**. It uniformly samples a grasp point and pulling direction.
- **Oracle policy**. It uses complete information about the fabric state to find the corner furthest from its plane target and pulls it towards the target. If that

corner is occluded, the policy will grasp the point above it—which will generally result in lower coverage. This policy achieves a high coverage (>94.5% on average) in a few actions on all Tiers.

While working with this simulation environment, we implemented several bug fixes and a modification to the Blender rendering module to leverage CUDA GPU acceleration on Colab. We also add functionality to storing frames at any frame rate instead of only keeping the last state after each action, which will be needed for the next section.

Remark. OpenAI baselines (DDPG & PPO) adapted for the Gym Cloth environment are also provided.

A.2 Learning keypoint representations

We work with the GymCloth environment to learn a keypoint representation useful to flattening a piece of cloth. To this end, we train the Transporter architecture by Kulkarni et al. (2019).

Keypoint representation is a powerful way to abstract high-dimensional visual sensor data from robots. There has been extensive work (Jakab et al., 2018; Zhang et al., 2018; Finn et al., 2016) done on methods to learn to extract keypoint from frames in an unsupervised manner. We pay special attention to the Transporter architecture, which allows for accurate tracking of objects and object parts over long time-horizons.

Transporter

The Transporter network (Fig. A.4) learns a keypoint representation in an unsupervised fashion, using a pixel reconstruction loss. This makes the model especially suitable for applications where we may have access to offline visual data generated by some unknown policy. The architecture is based on an encoder-decoder architecture with keypoint bottleneck by Jokab and Gupta et al. Jakab et al. (2018) but adds a feature transport mechanism that results in better spatially aligned keypoints (better object tracking).

For an image x, we wish to extract K keypoints (2D coordinates). During training, the network takes in pairs of images (x_s, x_t) taken from the same episode, and learns to reconstruct target image x_t from source image x_s.
The two images pass through spatial feature extractor $\Phi(x)$ and fully-differentiable Keynet $\Psi(x)$. The keypoint coordinates from Keynet are transformed into Gaussian heatmaps $\mathcal{H}_{\Psi(x)}$ with the same spatial dimensions as the extracted features. A transported feature map $\hat{\Phi}(x_s, x_t)$ is computed as:

$$\hat{\Phi}(x_s, x_t) \triangleq (1 - \mathcal{H}_{\Psi(x_s)}) \cdot (1 - \mathcal{H}_{\Psi(x_t)}) \cdot \Phi(x_s) + \mathcal{H}_{\Psi(x_t)} \cdot \Phi(x_t) \tag{A.1}$$

This transport phase,

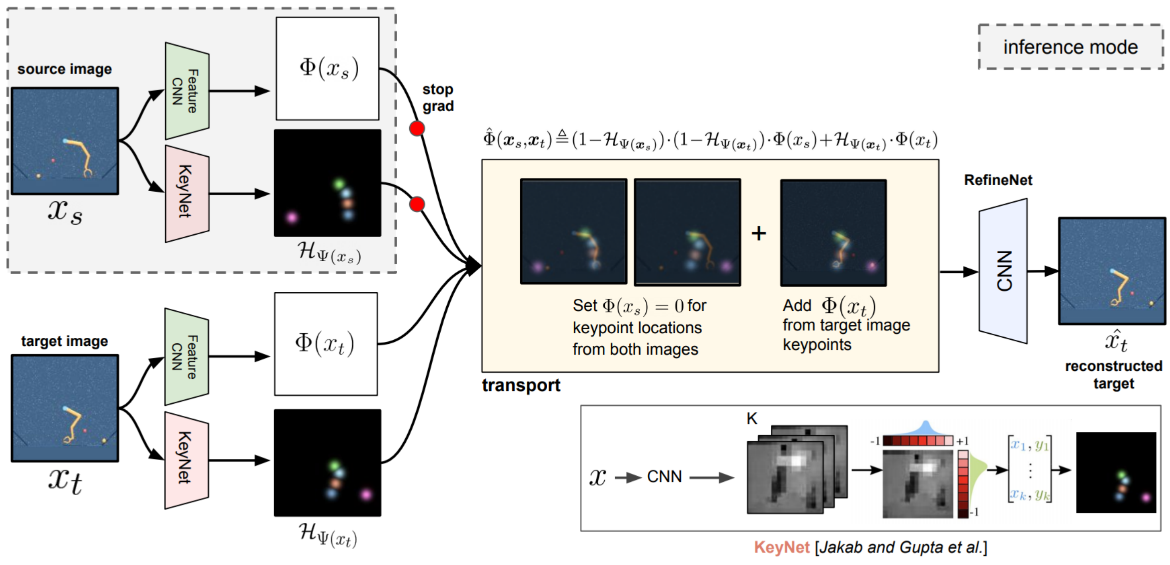

source image
Feature CNN
KeyNet
x_s
$\Phi(x_s)$
$\mathcal{H}_{\Psi(x_s)}$
stop grad
inference mode
target image
Feature CNN
KeyNet
x_t
$\Phi(x_t)$
$\mathcal{H}_{\Psi(x_t)}$
$\hat{\Phi}(x_s,x_t) \triangleq (1-\mathcal{H}_{\Psi(x_s)})\cdot(1-\mathcal{H}_{\Psi(x_t)})\cdot\Phi(x_s)+\mathcal{H}_{\Psi(x_t)}\cdot\Phi(x_t)$
Set $\Phi(x_s)=0$ for keypoint locations from both images
Add $\Phi(x_t)$ from target image keypoints
transport
RefineNet
CNN
$\hat{x}_t$
reconstructed target
x CNN
K
x_1, y_1
x_k, y_k
KeyNet [Jakab and Gupta et al.]

Figure A.4: Diagram illustrating the Transporter network architecture described above. Taken from *"Unsupervised Learning of Object Keypoints for Perception and Control"* Kulkarni et al. (2019)

1. sets the features of x_s to zeros at $\mathcal{H}_{\Psi(x_t)}$ and $\mathcal{H}_{\Psi(x_s)}$.
2. replaces the features of x_s at $\mathcal{H}_{\Psi(x_t)}$ with the features from the target image.

This explicit spatial transport enforces a stronger correlation of the keypoints coordinates with image locations and results in better long-term tracking.
Finally, the decoder CNN reconstructs the target image from the transported feature map with an L_2 reconstruction loss $||x_t - \hat{x}_t||^2$ (pixel-wise). At inference time, keypoints can be extracted for a single frame with a forward pass through Ψ (KeyNet).

A.2.1 Building a diverse dataset from GymCloth

In order to train a keypoint model on GymCloth, we need a large number of pairs of frames spanning as many possible cloth configurations as possible. For simplicity, we decide to generate data only from episodes with Tier 1 starting complexity. Furthermore, given that we will not be transferring any learnt policy to a real robotic manipulator in a Sim2Real context, we also remove any domain randomization (Tobin et al., 2017) which could affect the quality of the keypoints—although our experiments showed that similar results are obtained when cloth colour varies, and background noise is present. Frames can be generated by either following a random policy or a learned policy (e.g. the imitation learning solution from the next section). It was decided to follow a random policy since it would cover a more diverse set of states and could help achieve more robust keypoints.

Hence, the image data was generated by 4 GymCloth environments running in parallel on CPU, with each agent following a random policy (episodes terminate when the cloth goes out of bounds). We store ~ 50 frames per pulling action to disk (frame rendering in Blender is cheap compared to updating the state of the Cloth). Considering that the location of the keypoints will remain sufficiently accurate, we down-sample the frames to $56 \times 56 \times 3$. After a few hours of run-time 183k frames were collected over a large number of trajectories.

Diverse data generation The next step in the process of producing a workable dataset is to form pairs of frames that can be trained on. To ensure robust keypoint extraction, we must have a diverse range of situations covered in our selection of frame pairs. Unconditionally including pairs will often result in a high number of very similar pairs. To address this, we implement diverse data generation inspired by Kulkarni et al. (2019). Starting with the previously collected frames, we randomly populate a buffer to contain the maximum number N of pairs that we wish to select (in our case, we pick N=30k). For each of these pairs, we compute the L_2 distance to its nearest neighbour in the buffer (we use the Faiss for its efficient similarity search implementation, Johnson et al. (2019)). Then, we iteratively sample N new pairs from the trajectories and also compute their nearest-neighbour distance in the buffer. For corresponding pairs of (existing_buffer_pair, new_frame_pair) we overwrite the buffer pair with the new pair whenever the new pair has a greater distance. We continue this procedure until the average nearest neighbour distance computed for the whole buffer reduces to a desired value, or converges.

A.2.2 Training the Transporter architecture

With a dataset prepared, we can now train our model to extract keypoints. After plenty of experimentation with the open-sourced implementation, several performance issues were observed with different variations:

1. **Inaccurate tracking.** The keypoints would be well-positioned in certain configurations, but drift around and even off the cloth in other configurations (see fig A.6).

2. **Unstable tracking.** In certain conditions, the keypoints would significantly move in response to minor variations in the frame. (This effect is mainly observed when evaluating the keypoint mapping before training has fully converged).

3. **Background tracking.** In some cases, the keypoints would converge to positions in the background (as opposed to spreading over the foreground—the cloth), and yield satisfactory image reconstruction, with uncontrollable points. A possible explanation is that tracking the uniform white background is not heavily penalised—spatially transporting features corresponding to a white background spot to another background spot in the image will keep the background the same after decoding. So if the decoder can reconstruct the cloth based on where the background is missing, the keypoints may track the background. We are unsure why this issue was also encountered with smaller decoder networks, however.

Perhaps some of these problems were faced because this is a non-standard application of the Transporter. In the original paper, the demonstrations involve learning keypoints to track multiple simple objects or parts of objects. Training the model to extract meaningful keypoints over the surface of a *single* highly-deformable object makes it a more difficult problem.

Selected parameters We train the network to extract 8 keypoints, which produces a symmetrical placement around the square cloth. Following the default implementation, both encoder and decoders are standard convolutional network with ReLu activations. We train with the Adam optimizer, on a learning rate schedule. Any improvement or deterioration in performance could only be evaluated qualitatively, as a lower reconstruction does not necessarily imply better keypoint tracking. After performing hyperparameter optimization with grid search, improvements over the default configuration were only observed in increasing decoder size from 128 to 256 filters. A low batch size of 64 was used despite low GPU utilization—increasing it resulted in worse generalization (Keskar et al., 2017).

Pixel-wise loss over a mask We saw that the problems observed above were to a large extent due to the fact that we are learning keypoints on the basis of a pixel-wise reconstruction error—an error that places an equal weight to all pixels in the frame, whether they are foreground or background. To find the most meaningful and effective keypoints for control in this environment, we care about having points that 1) spread over the surface of the cloth and 2) accurately track the cloth over time. To prevent the algorithm from placing unnecessary importance on irrelevant

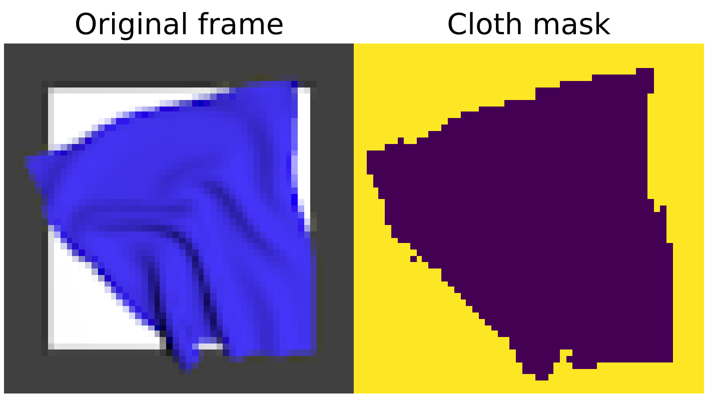


Figure A.5: We compute the pixel-wise loss over a mask that only incorporates pixels from the cloth.

aspects of the image, and ensure that the latent representation is most efficient in encoding the configuration of the cloth while ignoring other aspects of the frame, we propose to instead compute pixel-wise L_2 loss exclusively on the cloth pixels. To accomplish this, we note the frame consists of a gray background, a white square surface, and the cloth. Hence anything but the cloth is a shade of gray. In contrast, the pixel values where there is fabric are shades of blue (we are not using domain randomization Tobin et al. (2017))—except occasionally at the outer edge of the fabric, or in the deepest shadows. We leverage this observation to modify the loss function. It now computes the L_2 loss exclusively on the foreground of the image, extracted with a "not-gray" mask of similar dimensions to the frame ($56 \times 56 \times 3$).

Note in Figure (A.6) how the keypoint locations are qualitatively better at tracking the cloth. Moreover, while the mask loss does not reconstruct the background, the cloth reconstruction is more detailed. A finer reconstruction of the shadows and creases suggests that the representation better captures the ground state of the cloth.

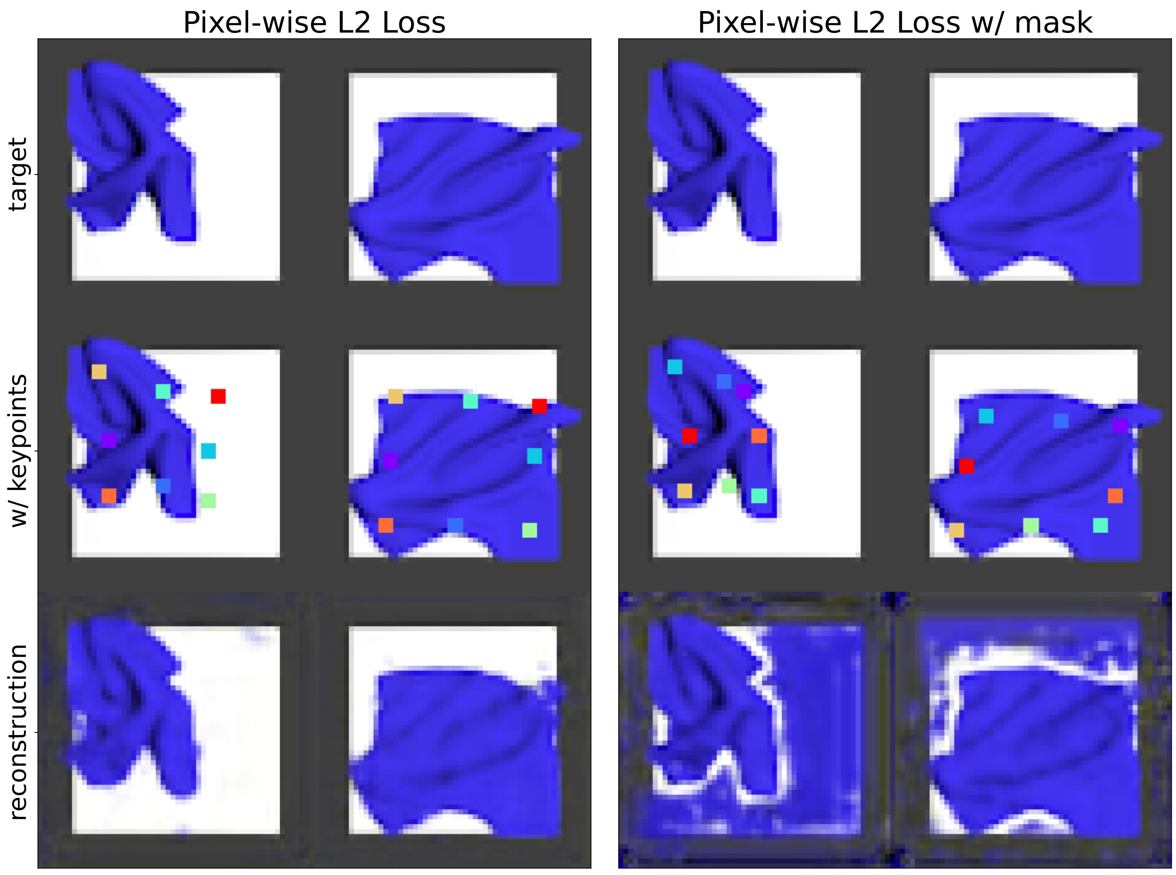


Figure A.6: Comparing keypoints produced from the original pixel-wise reconstruction error, and our modified version that uses a mask. Note how with the mask, keypoints better spread on the foreground of the cloth and can reconstruct finer cloth details.

Evaluating robustness We would like to have a qualitative estimate for how well our model generalizes. To do so, we carefully examine (fig A.9) samples from our test set where performance is very high and very low.
In the figure, we see that even in an example that ranks among the worst reconstruction losses, the model was able to roughly determine the configuration of the

cloth and only failed to reconstruct some of the finer details. In fact, the highest-magnitude pixel-wise errors (indicated in red) are concentrated on the edges and do not seem significantly worse than a sample with low error. It is common for pixel reconstruction models to find edges and higher-frequency details more challenging, so this does not come as a surprise. This suggests that our model is overall quite robust—albeit this metric can only be used as a proxy for the quality of the keypoints. It also appears that the error being higher on the left-most image is simply because the cloth is crumpled. The inaccuracies in the reconstruction are confined in a small area and this drives the mean pixel error up. In contrast, despite the errors in the low L_2 sample, the cloth being flat means that the "easy" portions bring the loss down.

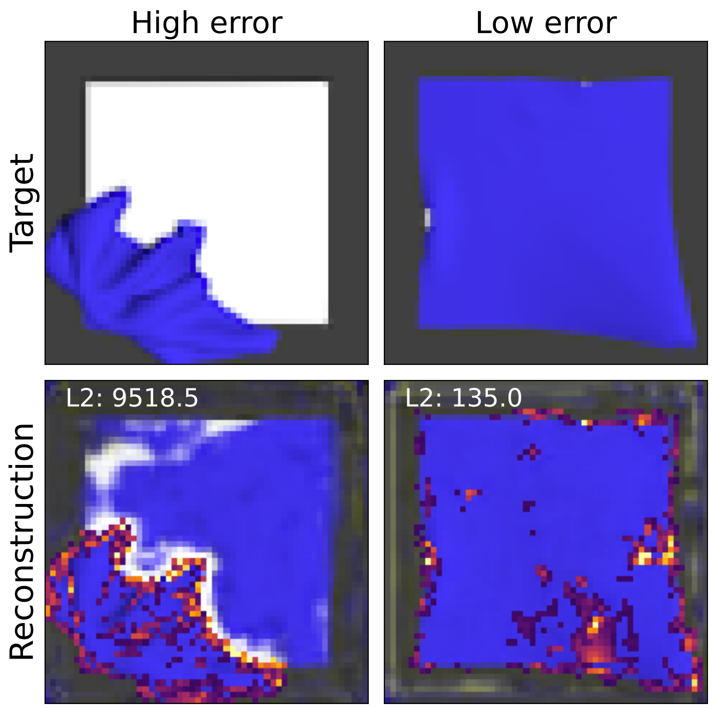


Figure A.7: Inspecting samples with the highest and lowest L_2 error.

In conclusion, while we have managed to learn an adequate mapping from pixel space to keypoint space, some questions are raised about reconstruction loss. We observed that the necessity for the representation to capture information irrelevant to our task led to worse keypoints. If the cloth colour or texture varied, the model would need to reproduce them perfectly. These high-level features are irrelevant to the task of flattening the cloth—and, in reality, all we care about is the cloth's spatial arrangement.

A.3 Manipulating fabric from keypoints

Having managed to train our keypoint extractor in an unsupervised fashion, we would now like to use this mapping as a state representation. We hope that learning to manipulate the cloth in the keypoint sample will be more data-efficient than working with raw pixel observations.

Seita et al. (2020) proposed to use imitation learning based on teacher samples provided by an algorithmic supervisor (the oracle policy described in section A.1). Specifically, they use in their paper a behaviour cloning (BC) pre-training phase together with a DAGGER phase to show that this is a valid approach and that the learned policy can be transferred to a real surgical robot. In a separate paper, Hoque et al. (2020) try to train DDPG—with several improvements—as a model-free baseline and show that the algorithm does not learn.

A.3.1 Model-free attempt

We first try to replicate the result of Hoque et al. on training DDPG and then integrate our keypoint mapping in hopes that this may enable DDPG to learn better.

DDPG without keypoints

The paper by Hoque et al. provides a DDPG baseline implementation on the Gym Cloth environment. It includes several improvements to the algorithm that were proposed in the literature (Matas et al., 2018).

The authors proposed including a teacher pre-training phase of 200 epochs to initialize the actor and critic networks. To this end, we generate 1144 action sequence episodes from the expert *Oracle* policy, in complexity Tiers 1 and 2.
An episode terminates upon achieving a coverage over 92%, if the cloth is pulled too far outside the plane boundaries, or after 10 actions. We also follow their suggestion for the following dense engineered reward (Table A.1).

reward	condition
-0.05	living penalty
-0.05	failing to grasp the cloth
δ	change in coverage from the previous state
+5	coverage over 92%
-5	cloth too far outside the plane boundaries

Table A.1: Table detailing the reward function engineering for the sequential fabric smoothing task by Hoque et al.

Upon running the algorithm and despite extensive experimentation and debugging we were unable to obtain learning—confirming the experimental results by Hoque et al. This is the case despite having restricted the task to its Tier 1 complexity, and removing domain randomization. It is possible that if we trained longer, DDPG would eventually begin to show signs of life—however, each of our experiments ran for 12-24h and our compute is limited.

DDPG with keypoints

Given these results, we now attempt to integrate the previously trained Transporter architecture. We hope that the keypoint space will provide more sample efficient learning and that the agent may learn a useful policy. We experiment with various actor-network architectures: fully-connected networks with varying number of layers, and an LSTM approach as suggested by Kulkarni et al. (2019), whereby the input is the flattened keypoint representation and the outputs are actions (we use the tanh activation function to ensure they are in the correct range $[-1,1]$). Nevertheless, we are unable to achieve any significant coverage improvements with the algorithm after 2150 episodes.

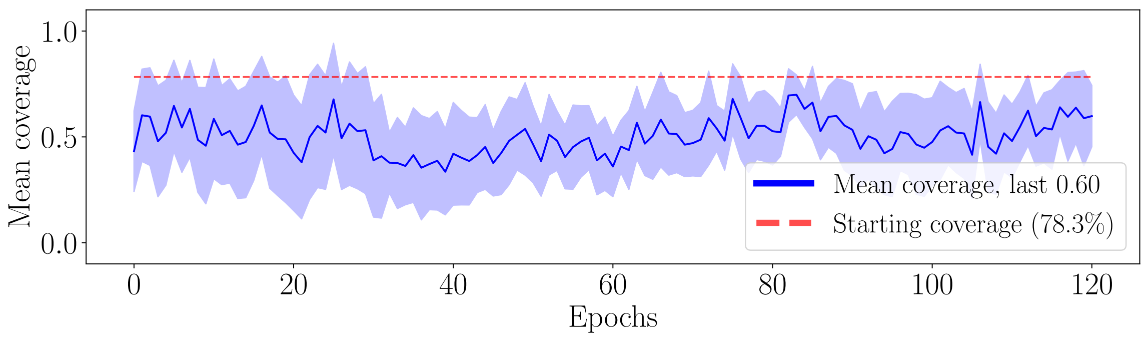


Figure A.8: Coverage vs epochs when training DDPG from keypoints with a fully-connected actor. Each epoch corresponds to 20 episodes.

A.3.2 Imitation learning

We now proceed to try to solve this fabric manipulation task through imitation learning. The problem of learning from imitation is inherently simpler, as the agent does not need to find by itself what the optimal policy is, it must only learn to replicate it. It is reasonable, therefore, to expect better results for the sequential fabric smoothing.

Seita et al. Seita et al. (2020) propose to use a behaviour-clone (BC) pre-training phase together with a DAgger phase based on teacher examples from algorithmically generated teacher examples. We will refer to the combination of these two algorithms as IMIT. The algorithmically generated teacher samples are based on access to the underlying finite element state of the fabric, as well as certain heuristics for deciding an action. We use the Oracle policy (A.1), which provides an adequate approximation of the optimal policy.

BC+DAgger without keypoints

We run the open-sourced implementation of IMIT on Tier 1 complexity with the same pre-training expert action sequences. Here we again observe a mean final coverage of 87% suggesting that the algorithm could learn to replicate the Oracle policy from pixels, in line with results from Seita et al. (we expect that longer training should be able to achieve even better performance).

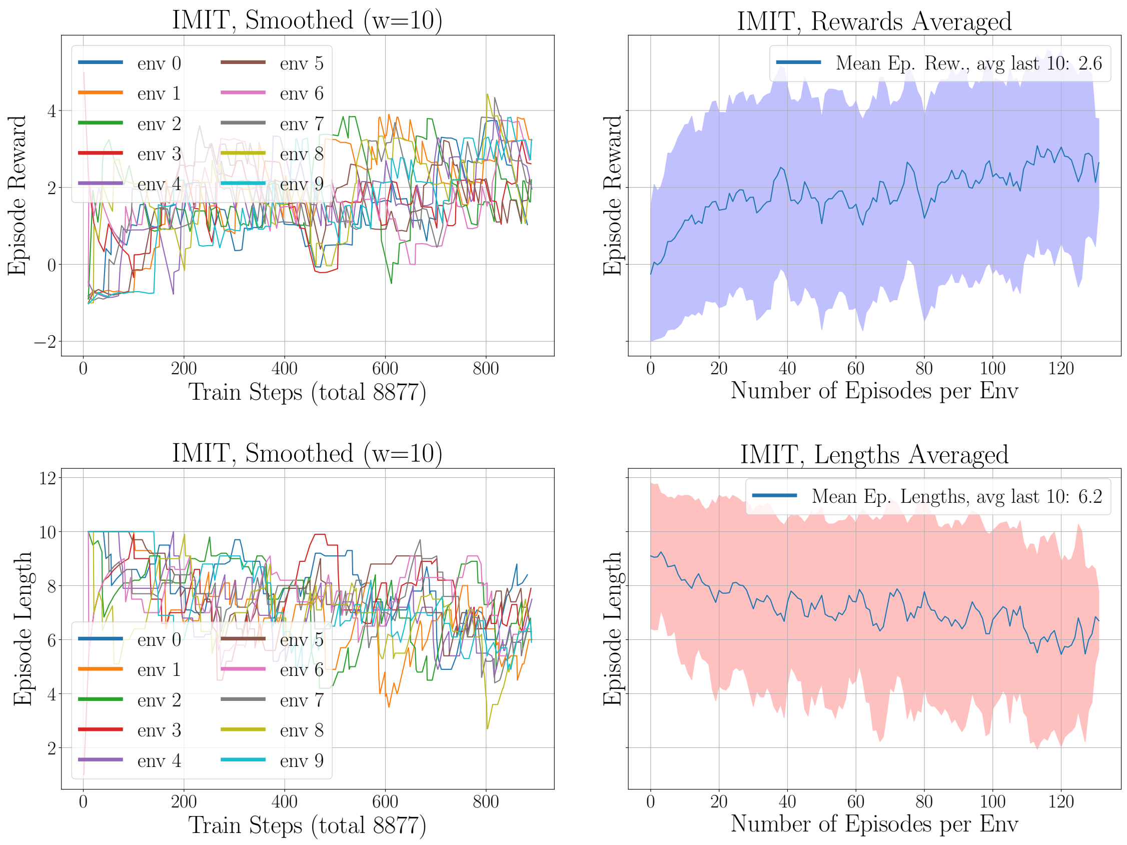


Figure A.9: Evaluating BC+DAgger from pixels, using expert examples generated by the Oracle policy. Final coverage achieved after 2150 episodes is 87%

BC+DAgger with keypoints

After successfully learning a strong policy from pixels using teacher examples, we try to learn from the same examples from keypoints. We expect that given the effective summary of the cloth's position that keypoints provide, the representation should be effective for learning this manipulation task.

We try several fully-connected architectures where the input is the flattened representation of keypoints. However, this did not succeed in learning the imitation policy. After 1k episodes, we only obtain $\approx 79\%$ final coverage on average—approximately the starting coverage for Tier 1 complexity. Occasionally the policy achieved a good configuration (over 92%), but this was often due to a relatively high starting coverage.
We also tried LSTMs, a recurrent neural network architecture. We consider the keypoints as a sequence and train an LSTM to predict actions based on this sequence.
It is currently unclear why the imitation learning algorithm that successfully learned directly from pixels is unable to achieve adequate performance from keypoints. We

believe this may be due to a strong dependency on the encoding used for the keypoint coordinates and tuning the architecture parameters. Kulkarni et al. (2019) used a thermometer encoding for the keypoints as well as a spatial aggregation of the feature map computed by the Transporter network. We will consider including the feature map in future, although our real interest lies in manipulating from extracted keypoints only.

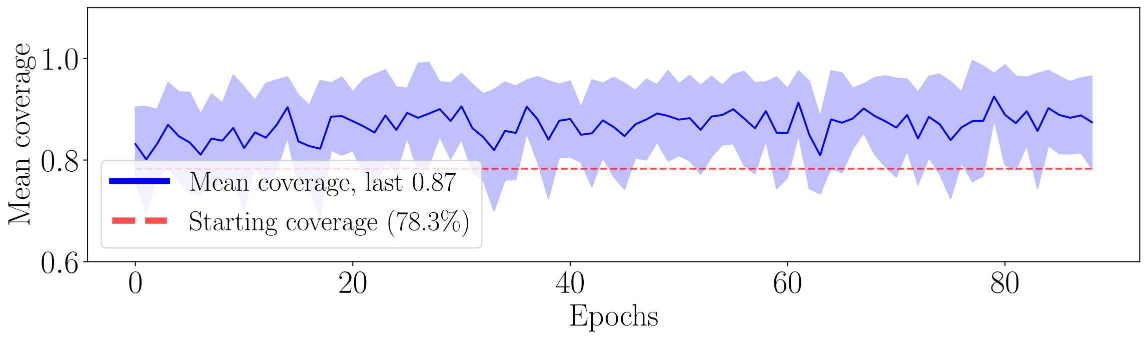


Figure A.10: Coverage vs epochs when training BC+DAgger from keypoints with a fully-connected actor. Each epoch corresponds to 20 episodes.

Appendix B

Offline SRL

B.1 Distractor benchmark

B.1.1 Cartpole data generation

When running the Online DQN agent to collect an offline dataset of transitions, we insert 5-step transitions into a replay buffer and periodically update the policy by sampling these transitions. In Table B.1, we show the hyperparameters used.

The Q-network architecture is the same as outlined in Section (5.5.1).

Table B.1: Hyperparameters used in Online DQN for dataset generation

Hyperparameter	setting
Greedy policy: ϵ	0.3
Mini-batch size	256
Target network update period	every 100 updates
Learning rate (intentionally small)	1e-4
Discount factor	0.99
Bootstrapping N	5
Number of episodes	1k per run

B.1.2 Online benchmarking

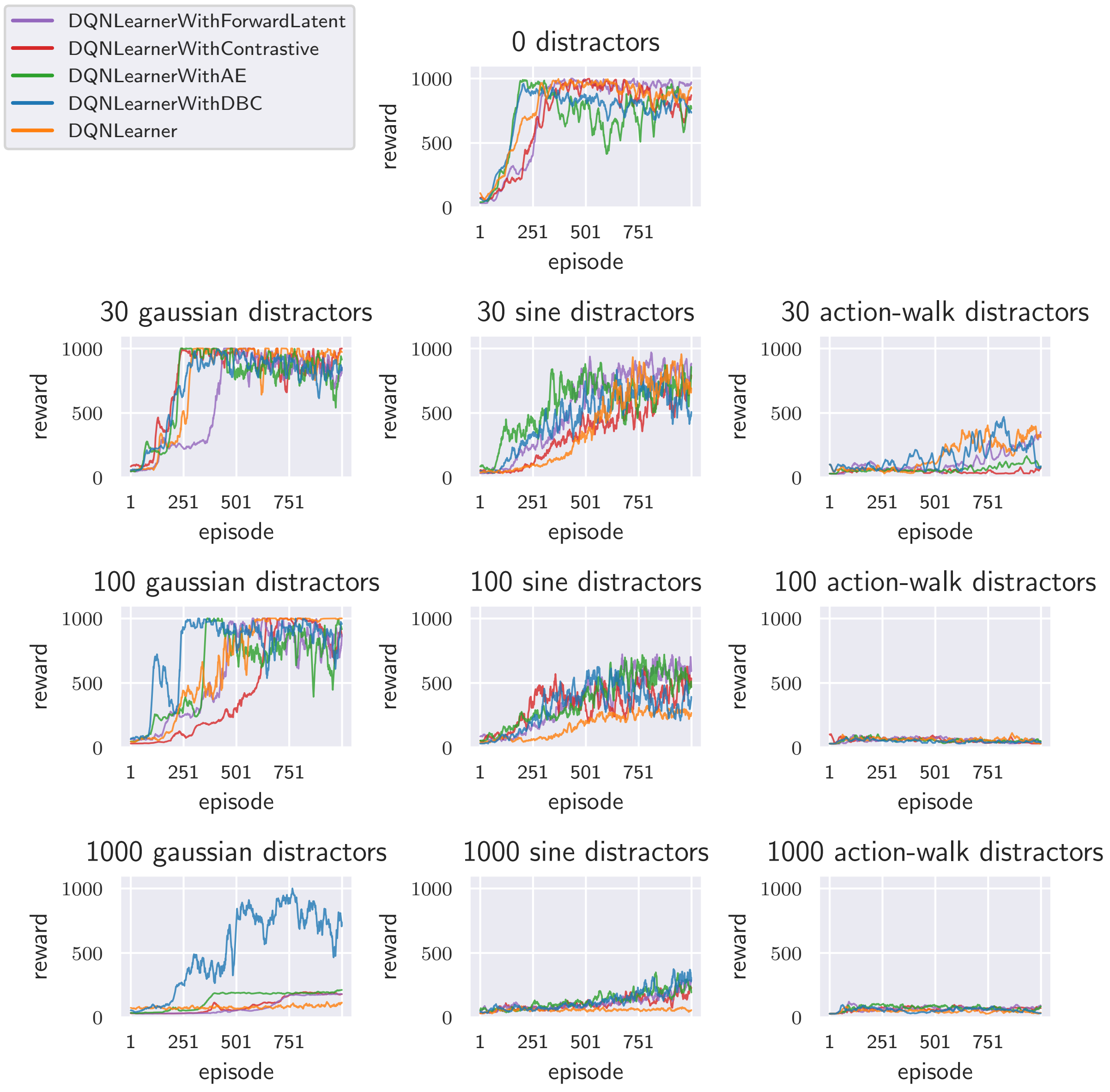


Figure B.1: Online training curves for each SRL method discussed, over different numbers and types of distractors (at least 3 seeds per curve). In these online results, we had not yet restricted the min/max values of action-walk distractors, hence the low performance on those tasks (larger magnitude distractions have a larger effect gradient updates). The representations helped the online agent learn with fewer samples and converge to better policies under high distractions.

B.1.3 BC and offline DQN parameters

Some additional details on the offline DQN parameters are shown in Table B.2. The behavioural cloning agent is trained with a learning rate of 0.001 and 0.8 dropout on every layer except the output.

Table B.2: Distractor experiments hyperparameters. The shared hyperparameters of the offline RL methods.

Hyperparameter	setting
Mini-batch size	256
Discount factor	0.99
Target network update period	every 200 updates
Encoder: hidden units	128
Encoder: output dimension	64
Q-network: hidden units	64, 32
Training Steps	70k

B.1.4 Representation hyperparameters

We tune the auxiliary loss weighting factor, as well as the joint learning rate for each method used. The parameters are summarized in Table B.3.

Table B.3: SRL method hyperparameters. Learning rates and weights were tuned through bayesian optimization.

Representation	Hyperparameter	setting
AE	Learning rate	0.0006
	Weight	0.0003
VAE	Learning rate	0.001
	Weight	0.0003
	Beta	1
DBC	Learning rate	0.0005
	Weight	1.5
CDBC	Learning rate	0.003
	Weight	3.3
	Temperature	0.5
DeepMDP	Learning rate	0.00118
	Weight	1.33
	Latent model: hidden units	64
	Reward model: hidden units	32

B.2 Atari experiments

For the representations, we use the learning rate tuned for an offline DQN on Atari (Gulcehre et al., 2021) and do not change it across SRL methods. We also uniformly use a weight of 1 for the auxiliary objectives, except for the autoencoder where we lowered it to 0.001 for the magnitude of the reconstruction and DQN loss to approximately match. For the DeepMDP, the forward-latent model and reward model use

Table B.4: Atari experiments hyperparameters. The shared hyperparameters of the offline RL methods.

Hyperparameter	setting
Discount factor	0.99
Mini-batch size	256
Target network update period	every 2500 updates
Q-network: channels	32, 64, 64
Q-network: filter size	8×8, 4×4, 3×3
Q-network: stride	4, 2, 1
Q-network: hidden units	512
Training Steps	1M learning steps
Replay Scheme	Uniform

512 hidden units and 256 hidden units, respectively.