Hierarchical Deep Reinforcement Learning For Robotics and Data Science

Sanjay Krishnan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-101
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-101.html

August 7, 2018

Hierarchical Deep Reinforcement Learning For Robotics and Data Science

by Sanjay Krishnan

A dissertation submitted in partial satisfaction of the requirements for the degree of:

Doctor of Philosophy in
Electrical Engineering and Computer Sciences
in the
Graduate Division of the
University of California at Berkeley

Committee in charge:
Kenneth Goldberg
Benjamin Recht
Joshua S. Bloom

Summer 2018

Abstract

Hierarchical Deep Reinforcement Learning for Robotics and Data Science

by Sanjay Krishnan
Doctor of Philosophy in Electrical Engineering and Computer Sciences
Ken Goldberg, Chair

This dissertation explores learning important structural features of a Markov Decision Process from offline data to significantly improve the sample-efficiency, stability, and robustness of solutions even with high dimensional action spaces and long time horizons. It presents applications to surgical robot control, data cleaning, and generating efficient execution plans for relational queries. The dissertation contributes: (1) Sequential Windowed Reinforcement Learning: a framework that approximates a long-horizon MDP with a sequence of shorter term MDPs with smooth quadratic cost functions from a small number of expert demonstrations, (2) Deep Discovery of Options: an algorithm that discovers hierarchical structure in the action space from observed demonstrations, (3) AlphaClean: a system that decomposes a data cleaning task into a set of independent search problems and uses deep q-learning to share structure across the problems, and (4) Learning Query Optimizer: a system that observes executions of a dynamic program for SQL query optimization and learns a model to predict cost-to-go values to greatly speed up future search problems.

Contents

Chapter 1

Introduction

Bellman's "Principle of Optimality" and the characterization of dynamic programming is one of the most important results in computing [11]. Its importance stems from the ubiquity of Markovian decision processes (MDPs), which formalize a wide range of problems from path planning to scheduling [49]. In the most abstract form, there is an agent who makes a sequence of decisions to effect change on a system that processes these decisions and updates its internal state possibly non-deterministically. The process is "Markovian" in the sense that the system's current state completely determines its future progression. As an example of an MDP, one might have to plan a sequence of motor commands to a robot to control it to a target position. Or, one might have to schedule a sequence of cluster computing tasks while avoiding double scheduling on a node. The solution to a MDP is a decision making policy: the optimal decision to make given any current state of the system.

Since the MDP framework is extremely general—it encompasses shortest-path graph search, supervised machine learning, and optimal control—the difficulty in solving a particular MDP relates to what assumptions can be made about the system. A setting of interest is when one only assumes black-box access to the system, where no parametric model of the system is readily available and the agent must iteratively query the system to optimize its decision policy. This setting, also called *reinforcement learning* [121], has been the subject of significant recent research interest. First, there are many dynamical problems for which a closed-form analytical description of a system's behavior is not available but one has a programatic approximation of how the system transitions (i.e., simulation). Reinforcement Learning (RL) allows for direct optimization of decision policies over such simulated systems. Next, by virtue of the minimal assumptions, RL algorithms are extremely general and widely applicable across many different problem settings. From a software engineering perspective, this unification allows the research community to develop a small number of optimized libraries for RL rather each domain designing/maintaining problem-specific algorithms. Over the last few years, the combination deep neural networks and reinforcement learning, or Deep RL, have emerged as in robotics, AI, and machine learning as an important area of research [85, 110, 115, 119]. Deep RL correlates features of states to successful decisions using neural networks.

Since RL relies on black-box queries to optimize the policy, it can be very inefficient in problems where the decision space is high-dimensional and when the decision horizon (the length of the sequence of decisions the agent needs to make) is very long. The algorithm has to simultaneously correlate features that are valuable for making a decision in a state while also searching through the space of decisions to figure out which sequence of decisions are valuable. A failure to learn means that the algorithm cannot intelligently search the space of decisions, and a failure to find promising sequences early means that the algorithm cannot learn. This creates a fundementally unstable algorithm setting, where the hope is the algorithm discovers good decision sequences by chance and can bootstrap from these relatively rare examples. The higher the dimensionality of the problem, the less likely purely random search will be successful.

Fortunately, in many domains of interest, a limited amount of expert knowledge is available, e.g., a human teleoperator can guide the motion of a robot to show an example solution of one problem instance or a slow classical algorithm can be used to generate samples for limited problem instances. Collecting such "demonstrations" to exhaustively learn a solution for possible progressions on MDP (called Imitation Learning [91]) can be expensive, but we might have sufficient data to find important structural features of a search problem including task decomposition, subspaces of actions that are irrelevant to the task, and the structure of the cost function. One such structure is hierarchy, where primitive actions are hierarchically composed into higher level behaviors.

The main contribution of this dissertation is the thesis that *learning action hierarchies from data can significantly improve the sample-efficiency, stability, and robustness of Deep RL in high-dimensional and long-horizon problems.* With this additional structure, the RL process can be restricted to those sequences that are grounded in action sequences seen in the expert data. My work over the last 6 years explores this approach in several contexts for control of imprecise cable-driven surgical robots, automatically synthesizing data-cleaning programs to meet quality specifications, and generating efficient execution plans for relational queries. I describe algorithmic contributions, theoretical analysis about the implementations themselves, the architecture of the RL systems, and data analysis from physical systems and simulations.

Contributions: This dissertation contributes:

1. Deep Discovery of Options (DDO): A new bayesian learning framework for learning parametrized control hierarchies from expert data (Chapter 2). DDO infers the parameters of an Abstract HMM model to decomposes the action space into a hierarchy of discrete skills relevant to a task. I show that the hierarchical model represents policies more efficiently (requires less data to learn) than a flat model on real and simulated robot control tasks. I also show that this hierarchy can be used to guide exploration in search problems through compositions of the discrete skills seen in the data rather than arbitrary sequences of actions. I apply this model to significantly accelerate learning in self-play of Atari games. Results suggest that DDO can take 3x fewer demonstrations to achieve the same reward compared to a baseline imitation

learning approach, and cut the sample-complexity of the RL phase by up-to an order of magnitude.

Krishnan, Sanjay, Roy Fox, Ion Stoica, and Ken Goldberg. DDDO: Discovery of Deep Continuous Options for Robot Learning from Demonstrations. Proceeding of Machine Learning Research: Conference on Robot Learning. 2017.

Code for DDO is available at: https://bitbucket.org/sjyk/segment-centroid/

2. Sequential Windowed Inverse Reinforcement Learning (SWIRL): A learning framework for approximating an MDP with a sequence of shorter horizon MDPs with quadratic cost functions (Chapter 3). SWIRL can be thought of as a special parametric case of DDO where skills terminate at goal states. I show on two surgical robot control tasks, cutting along a line and tensioning fabric, SWIRL significantly reduces the policy search time of a Q-Learning algorithm. In simulation, SWIRL achieves the maximum reward on the task with 85% fewer rollouts than Q-Learning, and 8x fewer demonstrations than behavioral cloning. In physical trials, it achieves a 36% relative improvement in reward compared to baselines.

   Krishnan, Sanjay, Animesh Garg, Richard Liaw, Brijen Thananjeyan, Lauren Miller, Florian T. Pokorny, and Ken Goldberg. SWIRL: A Sequential Windowed Inverse Reinforcement Learning Algorithm for Robot Tasks with Delayed Rewards. International Journal of Robotics Research. 2018.

3. Transition State Clustering (TSC): The underlying task decomposition algorithm in SWIRL is a Bayesian clustering framework called TSC (Chapter 3). This model correlates spatial and temporal features with changes in motion of the demonstrator. The crucial advantage of this framework is that it can exploit "third person" demonstrations data where only the states of the MDP are visible but not the decisions the expert took. The motivating application is learning from expert surgeons by analyzing data from a surgical robot. I show that TSC is more robust to spatial and temporal variation compared to other segmentation methods and can apply to both kinematic and visual demonstration data. In these settings, TSC runs 100x faster than the next most accurate alternative Auto-regressive Models, which require expensive MCMC-based inference, and has fewer parameters to tune.

   Krishnan, Sanjay, Animesh Garg, Sachin Patil, Colin Lea, Gregory Hager, Pieter Abbeel, and Ken Goldberg. Transition State Clustering: Unsupervised surgical trajectory segmentation for robot learning. International Journal of Robotics Research. 2018.

   Code for TSC and SWIRL is available at: http://berkeleyautomation.github.io/tsc-dl/

4. Alpha Clean: A system for synthesizing data cleaning programs to enforce database integrity constraints (Chapter 4). The main algorithm in the system decomposes the integrity constraints on a table into a collection of independent search problems called blocks. The algorithm starts by exhaustively searching the initial blocks and

then incrementally learns a Q-function to make the search on future increasingly precise. This process leverages the fact that data cleaning is not an arbitrary constraint satisfaction problem and most database corruption is systematic, i.e., correlated with features of the data.

Krishnan, Sanjay, Eugene Wu, Michael Franklin, and Ken Goldberg. Alpha Clean: Data Cleaning with Distributed Tree Search and Learning. 2018.

Code for AlphaClean is available at: https://github.com/sjyk/alphaclean

5. A new approximate dynamic programming framework for SQL query optimization. Classical query optimizers leverage dynamic programs for optimally nesting join queries. This process creates a table that memoizes cost-to-go estimates of intermediate subplans. By representing the memoization table with a neural network, the optimizer can estimate the cost-to-go of even previously unseen plans allowing for a vast expansion of the search space. I show that this process is a form of Deep Q-Learning where the state is a query graph and the actions are contractions on the query graph. The approach achieves plan costs within a factor of 2 of the optimal solution on all cost models and improves on the next best heuristic by up to $3\times$. Furthermore, it can execute up to 10,000$\times$ faster than exhaustive enumeration and more than $10\times$ faster than left/right-deep enumeration on the largest queries in the benchmark.

   Krishnan, Sanjay, Zongheng Yang, Ion Stoica, Joseph Hellerstein, and Ken Goldberg. Learning to Efficiently Enumerate Joins with Deep Reinforcement Learning. 2018.

   Code for this project is available at: https://github.com/sjyk/rlqopt

6. An application of deep reinforcement learning to synthesizing surgical thin tissue tensioning policies. To improve the search time, the algorithm initializes its search with an analytical approximation of the equilibrium state of a FEM simulator. The result is a search algorithm that searches for a policy over several hundred timesteps in less than a minute of latency (Chapter 6).

   Krishnan, Sanjay and Ken Goldberg. Sanjay Krishnan and Ken Goldberg. Bootstrapping Deep Reinforcement Learning of Surgical Tensioning with An Analytic Model. C4 Surgical Workshop 2017.

   Code for this project is available at: https://github.com/BerkeleyAutomation/cloth$_simulation$

Background and Related Work

A discrete-time discounted Markov Decision Process (MDP) is described by a 6-tuple $\langle \mathcal{S}, \mathcal{A}, p_0, p, R, \gamma \rangle$, where $\mathcal{S}$ denotes the state space, $\mathcal{A}$ the action space, p_0 the initial state distribution, $p(s_{t+1} \mid s_t, a_t)$ the state transition distribution, $R(s_t, a_t) \in \mathbb{R}$ is the reward

function, and $\gamma \in [0, 1)$ the discount factor. The objective of an MDP is to find a decision policy, a probability distribution over actions $\pi : \mathcal{S} \mapsto \Delta(A)$. A policy π induces the distribution over trajectories $\xi = [(s_0, a_0), (s_1, a_1), ..., (s_N, a_N)]$:

$$P_\pi(\xi) = p_0(x_0) \prod_{t=0}^{T-1} \pi(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t).$$

The *value* of a policy is its expected total discounted reward over trajectories

$$V_\pi = \mathbf{E}_{\xi \sim P_\pi} \left[\sum_{t=0}^{T-1} \gamma^t R(s_t, a_t) \right].$$

The objective is to find a policy in a class of allowed policies $\pi^* \in \Pi$ to maximize the return:

$$\pi^* = \arg\max_{\pi \in \Pi} V_\pi \tag{1}$$

The Ubiquity of Markov Decision Processes

While it is true that many systems of interest are not Markov and do not offer direct observation of their internal states, the MDP actually covers a substantial number of classical problems in Computer Science. Consider the following reductions:

Shortest Path Graph Search: A graph search problem instance is defined as follows. Let $G = (V, E)$ be a graph with vertices V and edges E, and let $q \in V$ denote a starting vertex and $t \in V$ denote a target vertex. Find a path connected by edges from q to t. In MDP notation, we can consider a hypothetical agent whose state is a pointer to a vertex, its actions are moving to an adjacent vertex, the transition executes this move, and its reward function is an indicator of whether the state is t. More formally, $\mathcal{S} = V$, $\mathcal{A} = S \times S$, $R = \mathbf{1}(s_t = t)$, and $p_0 = q$. A discount factor of $\gamma = 1$ specifies that any path is optimal, and $\gamma < 1$ specifies that shorter paths are preferred.

Supervised Learning: In empirical risk minimization for supervised learning, one is given a set of tuples $(X, Y) = (x_0, y_0), ..., (x_N, y_N)$ and the objective is to find a function that minimizes $f : X \mapsto Y$ that minimizes some point-wise measure of disagreement called a loss function $\sum_{i=0}^{N} \ell(f(x_i), y_i)$. In MDP notation, this is a "stateless" problem. The agent's state is a randomly chose example $\mathcal{S} = X$, its action space is Y, and the reward function is the loss function.

Optimal Control: Optimal control problems also constitute Markov Decision Process problems:

$$\min_{a_1, ..., a_T} \sum_{i=1}^{T} \gamma^i \cdot J(s_i, a_i)$$

$$\text{subject to: } s_{i+1} = f(s_i, a_i)$$

$$a_i \in \mathcal{A} \;\; s_i \in \mathcal{S} \;\; s_1 = \mathbf{c}$$

The problem is to select T decisions where each decision a_i resides in an action space $\mathcal{A}$. The decision making problem is stateful where the world has an initial state s_1 and this state is affected by every decision the decision-making agent selects through the transition model $f(s_i, a_i)$ which transitions the state to another state in the set $\mathcal{S}$. The objective is to optimize the cumulative cost of these decisions $J(s_i, a_i)$ potentially subject to an exponential discount γ that controls a bias towards short term or long term costs.

Reinforcement Learning (RL)

The reinforcement learning setting further assumes "black box" (also called oracular) access to the state transition distribution p and the reward function R. This means the optimization algorithm is only allowed queries of the following form:

$$q_t : s_t, a_t \rightarrow \mathsf{system()} \rightarrow s_{t+1}, r_t$$

This dissertation terms any algorithm that satisfies this query model as RL. The number of such queries issued by an RL algorithm is called its sample-complexity. This relaxes the restriction of any analytic knowledge about the structure of R or p, and only requires a system model that can be queried (e.g., written in code, implemented as physical system). The focus of this dissertation is on cases where *ab initio* RL has a prohibitive sample complexity due to high-dimensional action spaces or long time horizons. With no prior knowledge of which actions lead to rewards any RL algorithm has to essentially start with random decisions, and even when the algorithm observes a positive signal it has no notion of directionality.

Imitation Learning

A contrasting approach to RL is imitation learning [91], where one assume access to an supervisor who samples from an unknown policy $\hat{\pi} \approx \pi^*$ the optimal policy; these samples are called demonstrations. Rather than querying the system to optimize the policy, the problem is to imitate the supervisor as best as possible. Consider a worker in a factory moving a robot with a joystick. Here the objective of the worker is unknown but simply a trajectory of states and action. Similarly, in programming-by-examples, one only observes input and output data and not a complete specification of the program. In the most basic form, such a problem reduces to Maximum Likelihood Estimation.

A policy $\pi_\theta(a_t|s_t)$ defines the distribution over controls given the state, parametrized by $\theta \in \Theta$. In Behavior Cloning (BC), one trains the parameter θ so that the policy fits the dataset of observed demonstrations and imitates the supervisor. For example, we can maximize the log-likelihood $L[\theta; \xi]$ that the stochastic process induced by the policy π_θ

assigns to each demonstration trajectory ξ:

$$L[\theta; \xi] = \log p_0(s_0) + \sum_{t=0}^{T-1} \log(\pi_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t)).$$

When $\log \pi_\theta$ is parametrized by a deep neural network, we can perform stochastic gradient descent by sampling a batch of transitions, e.g. one complete trajectory, and computing the gradient

$$\nabla_\theta L[\theta; \xi] = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t).$$

Note that this method can be applied model-free, without any knowledge of the system dynamics p.

Another approach to the imitation setting is Inverse Reinforcement Learning [3, 88, 137]. This approach infers a reward function from the observed data (and possibly the system dynamics)–thus, reducing the problem to the original RL problem setting. [2] argue that the reward function is often a more concise representation of task than a policy. As such, a concise reward function is more likely to be robust to small perturbations in the task description. The downside is that the reward function is not useful on its own, and ultimately a policy must be retrieved. In the most general case, an RL algorithm must be used to optimize for that reward function [2].

Reinforcement Learning with Demonstrations

However, imitation learning places a significant burden on a supervisor to exhaustively cover the scenarios the robot may encounter during execution [74]. To address the limitations on either extreme of imitation and reinforcement, this dissertation proposes a hybrid of the exploration and demonstration learning paradigms.

Problem 1 (Reinforcement Learning with Demonstrations) *Given an MDP and a set of demonstration trajectories $D = \{\xi_1, ..., \xi_N\}$ from a supervisor, return a policy π^* that maximizes the cumulative reward of the MDP with a reinforcement learning algorithm.*

This is a problem setting that has been studied by a few recent works. In Deeply Aggravated [118], the expert must provide a value function in addition to actions, which ultimately creates an algorithm similar to Reinforcement Learning. This basic setting is also similar to the problem setting consider in [96]. [117] consider a model where the random search policy of the algorithm is guided by expert demonstrations. Rather than manipulating the search strategy, [19] modify the "shape" the reward function to match trajectories seen in demonstration. This is an idea that has gotten recent traction in the robot learning community [31, 48, 51].

This recent work raises a central question: what should be learned from demonstrations to effectively bootstrap reinforcement learning? One approach is using imitation learning to learn a policy which initializes the search in RL. While this can be a very effective strategy, it neglects other structure that is latent in the demonstrations like are there common sequences of actions that tend to occur or do parts of the task decompose into simpler units. In the high-dimensional setting, it is crucial to exploit such structure and this dissertation shows that different aspects of the structure such as task decomposition, action hierarchy, and action relevancy can be cast as Bayesian latent variable problems. I explore learning this structure in detail and the impact of learning this structure on several RL problems.

Chapter 2

Deep Discovery of Deep Options: Learning Hierarchies From Data

In high-dimensional or combinatorial action spaces, the vast majority of action sequences are irrelevant to the task at hand. Therefore, purely random exploration is likely to be prohibitively wasteful. What makes many sequential problems particularly challenging for is that all of these random action sequences are, in a sense, equally irrelevant. This means that there might not be any signal in the sampled data that the Q-learning algorithm can exploit. Until the agent serendipitously discovers such a sequence, no learning can occur.

In the first section of this dissertation, I explore this problem in the context of robotics, self-play in atari games, and program imitation. The basic insight is that while the space of all action sequences $\mathcal{A}^T$ is very large, there is often a much smaller subset of those that are potentially relevant to typical problem instances $\mathcal{A}_{relevant} \subset \mathcal{A}^T$. Given a small amount of expert information, I describe techniques for learning $\mathcal{A}_{relevant}$ from data.

How do we parametrize $\mathcal{A}_{relevant}$? One approach is to describe the agent's actions as a composition higher-level behaviors called [122], each consisting of a control policy for one region of the state space, and a termination condition recognizing leaving that region. This augmentation naturally defines a hierarchical structure of high-level *meta-control* policies that invoke lower-level options to solve sub-tasks. This leads to a "divide and conquer" relationship between the levels, where each option can specialize in short-term planning over local state features, and the meta-control policy can specialize in long-term planning over slowly changing state features. This means that any search can be restricted to the action sequences that can be formed as a composition of skills.

Abstractions for decomposing an MDP into subtasks have been studied in the area of hierarchical reinforcement learning (HRL) [9, 95, 122]. Early work in hierarchical control demonstrated the advantages of hierarchical structures by handcrafting hierarchical policies [18] and by learning them given various manual specifications: state abstractions [28, 47, 60, 61], a set of waypoints [53], low-level skills [6, 50, 78], a set of finite-state meta-controllers [94], or a set of subgoals [29, 122]. The key abstraction in HRL is the "op-

tions framework" [122], which defines a hierarchy of increasingly complex meta-actions. An option represent a lower-level control primitive that can be invoked by the *meta-control* policy at a higher-level of the hierarchy, in order to perform a certain subroutine (a useful sequence of actions). The meta-actions invoke specialized policies rather than just taking primitive actions.

Formally, an option h is described by a triplet

$$\langle \mathcal{I}_h, \pi_h, \psi_h \rangle,$$

where $\mathcal{I}_h \subset \mathcal{X}$ denotes the initiation set, $\pi_h(u_t \mid x_t)$ the control policy, and $\psi_h(s_t) \in [0, 1]$ the termination policy. When the process reaches a state $s \in \mathcal{I}_h$, the option h can be invoked to run the policy π_h. After each action is taken and the next state s' is reached, the option h terminates with probability $\psi_h(s')$ and returns control up the hierarchy to its invoking level. The options framework enables multi-level hierarchies to be formed by allowing options to invoke other options. A higher-level meta-control policy is defined by augmenting its action space $\mathcal{A}$ with the set $\mathcal{H}$ of all lower-level options.

The options framework has been applied in robotics [63, 65, 106] and in the analysis of biological systems [15, 16, 113, 131, 135].Since then, the focus of research has shifted towards discovery of the hierarchical structure itself, by: trading off value with description length [125], identifying transitional states [73, 80, 82, 112, 116], inference from demonstrations [20, 27, 65, 66], iteratively expanding the set of solvable initial states [62, 63], policy gradient [77], trading off value with informational constraints [34, 36, 41, 52], active learning [44], or recently value-function approximation [5, 45, 107].

2.1 Overview

First, I describe a new learning framework for discovering a parametrized option structure from a small amount of expert demonstrations. I assume that these demonstrations are *state-action* demonstrations. Despite recent results in option discovery, some proposed techniques do not generalize well to multi-level hierarchies [5, 45, 72], while others are inefficient for learning expressive representations (e.g., options parametrized by neural networks) [20, 27, 44, 73].

I introduce the *Discovery of Deep Options (DDO)*, an algorithm for efficiently discovering deep hierarchies of deep options. DDO is a policy-gradient algorithm that discovers parametrized options from a set of demonstration trajectories (sequences of states and actions) provided either by a supervisor or by roll-outs of previously learned policies. These demonstrations need not be given by an optimal agent, but it is assumed that they are informative of the preferred actions to take in each visited state, and are not just random walks. DDO is an inference algorithm that applies to the **supervised demonstration setting**.

Given a set of trajectories, the algorithm discovers a fixed, predetermined number of options that are most likely to generate the observed trajectories. Since an option is represented by both a control policy and a termination condition, my algorithm simultaneously

(1) infers option boundaries in demonstrations which segment trajectories into different control regimes, (2) infers the meta-control policy for selecting options as a mapping of segments to the option that likely generated them, and (3) learns a control policy for each option, which can be interpreted as a soft clustering where the centroids correspond to prototypical behaviors of the agent.

2.2 Generative Model

First, we describe a generative model for expert demonstrations. This can be thought of as a generalization of standard imitation learning to hierarchical control. In this generative model, the meta-control signals that form the hierarchy are unobservable, latent variables of the generative model, that must be inferred.

Consider a trajectory $\xi = (s_0, a_0, s_1, \ldots, s_T)$ that is generated by a two-level hierarchy. The low level implements a set $\mathcal{H}$ of options $\langle \pi_h, \psi_h \rangle_{h \in \mathcal{H}}$. The high level implements a meta-control policy $\eta(h_t|s_t)$ that repeatedly chooses an option $h_t \sim \eta(\cdot|s_t)$ given the current state, and runs it until termination. Our hierarchical generative model is:

Initialize $t \leftarrow 0$, $s_0 \sim p_0$, $b_0 \leftarrow 1$
for $t \leftarrow 0, \ldots, T-1$ **do**
  if $b_t = 1$ **then**
    Draw $h_t \sim \eta(\cdot|s_t)$
  else
    Set $h_t \leftarrow h_{t-1}$
  end if
  Draw $a_t \sim \pi_{h_t}(\cdot|s_t)$
  Draw $s_{t+1} \sim p(\cdot|s_t, a_t)$
  Draw $b_{t+1} \sim \mathrm{Ber}(\psi_{h_t}(s_{t+1}))$
end for

2.3 Expectation-Gradient Inference Algorithm

We denote by θ the vector of parameters for π_h, ψ_h and η. For example, θ can be the weights and biases of a feed-forward network that computes these probabilities. This generic notation allows us the flexibility of a completely separate network for the meta-control policy and for each option, $\theta = (\theta_\eta, (\theta_h)_{h \in \mathcal{H}})$, or the efficiency of sharing some of the parameters between options, similarly to a Universal Value Function Approximator [101].

We want to find the $\theta \in \Theta$ that maximizes the log-likelihood assigned to a given dataset of trajectories. The likelihood of a trajectory depends on the latent sequence $\zeta = (b_0, h_0, b_1, h_1, \ldots, h_{T-1})$ of meta-actions and termination indicators, and in order to use a

gradient-based optimization method we rewrite the gradient using the following *EG-trick*:

$$\begin{aligned}\nabla_\theta L[\theta;\xi] = \nabla_\theta \log \mathbb{P}_\theta(\xi) &= \frac{1}{\mathbb{P}_\theta(\xi)} \sum_{\zeta \in (\{0,1\}\times\mathcal{H})^T} \nabla_\theta \, \mathbb{P}_\theta(\zeta,\xi) \\ &= \sum_\zeta \frac{\mathbb{P}_\theta(\zeta,\xi)}{\mathbb{P}_\theta(\xi)} \nabla_\theta \log \mathbb{P}_\theta(\zeta,\xi) = \mathbb{E}_{\zeta|\xi;\theta}[\nabla_\theta \log \mathbb{P}_\theta(\zeta,\xi)],\end{aligned}$$

which is the so-called *Expectation-Gradient* method [81, 100]. θ^- denotes the current parameter taken as fixed outside the gradient.

The generative model in the previous section implies the likelihood

$$\mathbb{P}_\theta(\zeta,\xi) = p_0(s_0)\delta_{b_0=1}\eta(h_0|s_0) \prod_{t=1}^{T-1} \mathbb{P}_\theta(b_t,h_t|h_{t-1},s_t) \prod_{t=0}^{T-1} \pi_{h_t}(a_t|s_t) p(s_{t+1}|s_t,a_t),$$

with

$$\begin{aligned}\mathbb{P}_\theta(b_t{=}1,h_t|h_{t-1},s_t) &= \psi_{h_{t-1}}(s_t)\eta(h_t|s_t) \\ \mathbb{P}_\theta(b_t{=}0,h_t|h_{t-1},s_t) &= (1-\psi_{h_{t-1}}(s_t))\delta_{h_t=h_{t-1}}.\end{aligned}$$

$\delta_{h_t=h_{t+1}}$ denotes the indicator that $h_t = h_{t+1}$.

Applying the EG-trick and ignoring the terms that do not depend on θ, we can simplify the gradient to:

$$\nabla_\theta L[\theta;\xi] = \mathbb{E}_{\zeta|\xi;\theta}\left[\nabla_\theta \log \eta(h_0|s_0) + \sum_{t=1}^{T-1} \nabla_\theta \log \mathbb{P}_\theta(b_t,h_t|h_{t-1},s_t) + \sum_{t=0}^{T-1} \nabla_\theta \log \pi_{h_t}(a_t|s_t)\right].$$

The log-likelihood gradient can therefore be computed as the sum of the log-probability gradients of the various parameterized networks, weighed by the marginal posteriors

$$\begin{aligned}u_t(h) &= \mathbb{P}_\theta(h_t{=}h|\xi) \\ v_t(h) &= \mathbb{P}_\theta(b_t{=}1,h_t{=}h|\xi) \\ w_t(h) &= \mathbb{P}_\theta(h_t{=}h,b_{t+1}{=}0|\xi).\end{aligned}$$

In the Expectation-Gradient algorithm, the E-step computes u, v and w, and the G-step updates the parameter with a gradient step, namely

$$\begin{aligned}\nabla_\theta L[\theta;\xi] = \sum_{h\in\mathcal{H}} \Bigg(&\sum_{t=0}^{T-1} \Big(v_t(h)\nabla_\theta \log \eta(h|s_t) + u_t(h)\nabla_\theta \log \pi_h(a_t|s_t) \Big) \\ &+ \sum_{t=0}^{T-2} \Big((u_t(h)-w_t(h))\nabla_\theta \log \psi_h(s_{t+1}) + w_t(h)\nabla_\theta \log(1-\psi_h(s_{t+1})) \Big) \Bigg).\end{aligned}$$

These equations lead to the natural iterative algorithm. In each iteration, the marginal posteriors u, v and w can be computed with a forward-backward message-passing algorithm similar to Baum-Welch, with time complexity $O(|\mathcal{H}|^2 T)$. Importantly, this algorithm can be performed without any knowledge of the state dynamics. Then, the computed posteriors can be used in a gradient descent algorithm to update the parameters:

$$\theta \leftarrow \theta + \alpha \sum_i \nabla_\theta L[\theta; \xi_i].$$

This update can be made stochastic using a single trajectory, uniformly chosen from the demonstration dataset, to perform each update.

Intuitively, the algorithm attempts to jointly optimize three objectives:

- Infer the option boundaries in which $b = 1$ appears likely relative to $b = 0$, as given by $(u - w)$ and w respectively — this segments the trajectory into regimes where we expect h to persist and employ the same control law; in the G-step we reduce the cross-entropy loss between the unnormalized distribution $(w, u - w)$ and the termination indicator ψ_h;
- Infer the option selection after a switch, given by v; in the G-step we reduce the cross-entropy loss between that distribution, weighted by the probability of a switch, and the meta-control policy η; and
- Reduce the cross-entropy loss between the empirical action distribution, weighted by the probability for h, and the control policy π_h.

This can be interpreted as a form of soft clustering. The data points are one-hot representations of each a_t in the space of distributions over actions. Each time-step t is assigned to option h with probability $u_t(h)$, forming a soft clustering of data points. The G-step directly minimizes the KL-divergence of the control policy π_h from the weighted centroid of the corresponding cluster.

Let $\delta_{a_t}(a|s_t) = \delta_{a=a_t}$ be the degenerate "empirical" action distribution of step t. The KL divergence of π_h from the weighted centroid of the cluster corresponding to option h.

Forward-Backward Algorithm

Despite the exponential domain size of the latent variable ζ, Expectation-Gradient for trajectories allows us to decompose the posterior $\mathbb{P}_\theta(\zeta|\xi)$ and only concern ourselves with each marginal posterior separately. These marginal posteriors can be computed by a forward-backward dynamic programming algorithm, similar to Baum-Welch [**?**].

Omitting the current parameter θ and trajectory ξ from out notation, we start by computing the likelihood of a trajectory prefix

$$\phi_t(h) = \mathbb{P}(s_0, a_0, \ldots, s_t, h_t = h),$$

using the forward recursion

$$\begin{aligned}\phi_0(h) &= p_0(s_0)\eta(h|s_0)\\ \phi_{t+1}(h') &= \sum_{h\in\mathcal{H}} \nu_t(h)\pi_h(a_t|s_t)p(s_{t+1}|s_t,a_t)\,\mathbb{P}(h'|h,s_{t+1}),\end{aligned}$$

with

$$\mathbb{P}(h'|h,s_{t+1}) = \psi_h(s_{t+1})\eta(h'|s_{t+1}) + (1-\psi_h(s_{t+1}))\delta_{h,h'}.$$

We similarly compute the likelihood of a trajectory suffix

$$\omega_t(h) = \mathbb{P}(a_t, s_{t+1}, \ldots, s_T|s_t, h_t = h),$$

using the backward recursion

$$\begin{aligned}\omega_T(h) &= 1\\ \omega_t(h) &= \pi_h(a_t|s_t)p(s_{t+1}|s_t,a_t)\sum_{h'\in\mathcal{H}} \mathbb{P}(h'|h,s_{t+1})\omega_{t+1}(h').\end{aligned}$$

We can now compute our target likelihood using any $0 \leqslant t \leqslant T$

$$\mathbb{P}(\xi) = \sum_{h\in\mathcal{H}} \mathbb{P}(\xi, h_t = h) = \sum_{h\in\mathcal{H}} \phi_t(h)\omega_t(h).$$

The marginal posteriors are

$$\begin{aligned}u_t(h) &= \frac{\phi_t(h)\omega_t(h)}{\mathbb{P}(\xi)}\\ v_t(h) &= \frac{\phi_t(h)\pi_h(a_t|s_t)p(s_{t+1}|s_t,a_t)\psi_h(s_{t+1})\sum_{h'\in\mathcal{H}}\eta(h'|s_{t+1})\omega_{t+1}(h')}{\mathbb{P}(\xi)}\\ w_t(h') &= \frac{\sum_{h\in\mathcal{H}}\phi_t(h)\pi_h(a_t|s_t)p(s_{t+1}|s_t,a_t)\psi_h(s_{t+1})\eta(h'|s_{t+1})\omega_{t+1}(h')}{\mathbb{P}(\xi)}.\end{aligned}$$

Note that the constant $p_0(s_0)\prod_{t=0}^{T-1} p(s_{t+1}|s_t,a_t)$ is cancelled out in these normalizations. This allows us to omit these terms during the forward-backward algorithm, which can thus be applied without any knowledge of the dynamics.

Stochastic Variant

We may collect a large number of trajectories making it difficult to scale the EG algorithm. The expensive step in this algorithm is usually the forward-backward calculation, which is an $O(h^2T)$ operation. To address this problem, we can apply a stochastic variant of the EG algorithm, which optimizes a single trajectory for each iterate:

- **E-Step:** Draw a trajectory i at random, calculate $q_i(t,h)$, and $b_i(t,h)$ with the forward-backward algorithm.
- **G-Step:** Update the parameters of the policies and the termination conditions:

$$\theta_h^{(j+1)} \leftarrow \theta_i^{(j)} - \alpha \sum_{t=1}^{T} w_i(h,t) \nabla_\theta \log \pi_\theta(a_t|s_t).$$

$$\mu_h^{(j+1)} \leftarrow \mu_i^{(j)} - \alpha \sum_{t=1}^{T} q_i(h,t) \nabla_\mu \log \rho_\mu(\perp |s_t).$$

Deeper Hierarchies

Our ultimate goal is to use the algorithm presented here to discover a multi-level hierarchical structure — the key insight being that the problem is recursive in nature. A D-level hierarchy can be viewed as a 2-level hierarchy, in which the "high level" has a $(D-1)$-level hierarchical structure. The challenge is the coupling between the levels; namely, the value of a set of options is determined by its usefulness for meta-control [36], while the value of a meta-control policy depends on which options are available. This potentially leads to an exponential growth in the size of the latent variables required for inference. The available data may be insufficient to learn a policy so expressive.

We can avoid this problem by using a simplified parametrization for the intermediate meta-control policy η_d used when discovering level-d options. In the extreme, we can fix a uniform meta-control policy that chooses each option with probability $1/|\mathcal{H}_d|$. Discovery of the entire hierarchy can now proceed recursively from the lowest level upward: level-d options can invoke already-discovered lower-level options; and are discovered in the context of a simplified level-d meta-control policy, decoupled from higher-level complexity. One of the contributions of this work is to demonstrate that, perhaps counter-intuitively, this assumption does not sacrifice too much during option discovery. An informative meta-control policy would serve as a prior on the assignment of demonstration segments to the options that generated them, but with sufficient data this assignment can also be inferred from the low-level model, purely based on the likelihood of each segment to be generated by each option.

We use the following algorithm to iteratively discover a hierarchy of D levels, each level d consisting of k_d options:

for $d = 1, \ldots, D-1$ **do**
  Initialize a set of options $\mathcal{H}_d = \{h_{d,1}, \ldots, h_{d,k_d}\}$
  DDO: train options $\langle \pi_h, \psi_h \rangle_{h \in \mathcal{H}_d}$ with η_d fixed
  Augment action space $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{H}_d$
end for
Use RL algorithm to train high-level policy

First, we approximate the high-level policy ψ, which selects policies based on the current state, with a i.i.d selection of policies based on only the previous primitive:

$$\psi \sim \mathbf{P}(h'|h)$$

This approximation is so that we do not have to simultaneously learn parameters for the high-level policy, while trying to optimize for the parameters of the policies and termination conditions. Next, if we collect trajectories from multiple high-level policies, there may not exist a single high-level policy that can capture all of the trajectories. The approximation allows us to make the fewest assumptions about the structure of this policy.

Using an approximation that ψ^* is a state-independent uniform distribution and sampled i.i.d, I will show the we can apply an algorithm similar to typical imitation learning approaches that recovers the most likely parameters to estimate $\pi_1^*, ..., \pi_k^*$ and $\{\rho_1^*, ..., \rho_k^*\}$. The basic idea is to define a probability that the current (s_t, a_t) tuple is generated by the particular primitive h:

$$w(h, t) = \mathbf{P}[h|(s_t, a_t), \tau_{0<t}],$$

and given that the current selected primitive is h probability that the current time-step is termination:

$$q(h, t) = \mathbf{P}[\perp |(s_t, a_t), \tau_{0<t}, h].$$

Given these probabilities, we can define an Expectation-Gradient descent over the parameter vector θ

$$\theta \leftarrow \theta + \alpha \nabla_\theta L[\theta; \xi].$$

Cross-Validation For Parameter Tuning

The number of options k is a crucial hyper-parameter of the algorithm. In simulation experiments, one can roll out the learned hierarchy online and tune the hierarchy based on task success. Such tuning is infeasible on a real robot as it would require many executions of the learned policy. We explored whether it is possible to tune the number of options offline. DDO is based on a maximum-likelihood formulation, which describes the likelihood that the observed demonstrations are generated by a hierarchy parametrized by θ. However, the model expressiveness is strictly increasing in k, causing the optimal training likelihood to increase even beyond the point where the model overfits to the demonstrations and fails to generalize to unseen states.

This likelihood is a proxy for task success. Therefore, we tune k in a way that maximizes the likelihood. However, we sometimes encounter a problem similar to over-fitting. Increasing k actually changes expressiveness the hierarchy, as with more options it can fit to more complicated behaviors. This means that the tuned parameters may not generalize.

We therefore adopt a cross-validation technique that holds out 10% of the demonstration trajectories for each of 10 folds, trains on the remaining data, and validates the trained model on the held out data. We select the value of k that achieves the highest average log-likelihood over the 10 folds, suggesting that training such a hierarchical model generalizes

well. We train the final policy over the entire data. This means the tuned parameter must work well on a hold out set. We find empirically that the cross-validated log-likelihood serves as a good proxy to actual task performance.

Vector Quantization For Initialization

One challenge with DDO is initialization. When real perceptual data is used, if all of the low-level policies initialize randomly the forward-backward estimates needed for the Expectation-Gradient will be poorly conditioned where there is an extremely low likelihood assigned to any particular observation. The EG algorithm relies on a segment-cluster-imitate loop, where initial policy guesses are used to segment the data based on which policy best explains the given time-step, then the segments are clustered, and the policies are updated. In a continuous control space, a randomly initialized policy may not explain any of the observed data well. This means the small differences in initialization can lead to large changes in the learned hierarchy.

We found that a necessary pre-processing step was a variant of vector quantization, originally proposed for problems in speech recognition. We first cluster the state observations using a k-means clustering and train k behavioral cloning policies for each of the clusters. We use these k policies as the initialization for the EG iterations. Unlike the random initialization, this means that the initial low level policies will demonstrate some preference for actions in different parts of the state-space. We set k to be the same as the k set for the number of options, and use the same optimization parameters.

2.4 Experiments: Imitation

In the first set of experiments, we evaluate DDO in its ability to represent hierarchical control policies.

Box2D Simulation: 2D Surface Pushing with Friction and Gravity

In the first experiment, we simulate a 3-link robot arm in Box2D (Figure 2.1). This arm consists of three links of lengths 5 units, 5 units, and 3 units, connected by ideal revolute joints. The arm is controlled by setting the values of the joint angular velocities $\dot{\phi}_1, \dot{\phi}_2, \dot{\phi}_3$. In the environment, there is a box that lies on a flat surface with uniform friction. The objective is to push this box without toppling it until it rests in a randomly chosen goal position. After this goal state is reached, the goal position is regenerated randomly. The task is for the robot to push the box to as many goals as possible in 2000 time-steps. Our algorithmic supervisor runs the RRT Connect motion planner of the Open Motion Planning Library, ompl, at each time-step planning to reach the goal. Due to the geometry of the configuration space and the task, there are two classes of trajectories that are generated, when the goal is to the left or right of the arm.

We designed this experiment to illustrate how options can lead to more concise representations since they can specialize in different regions of the state-spaces. Due to the geometry of the configuration space and the task, there are two classes of trajectories that are generated, when the goal is to the left or right of the arm. The 50 sampled trajectories plotted in joint angle space in Figure 2.1 are clearly separated into two distinct skills, backhand and forehand. Furthermore, these skills can be implemented by a locally affine policies in the joint angle space.

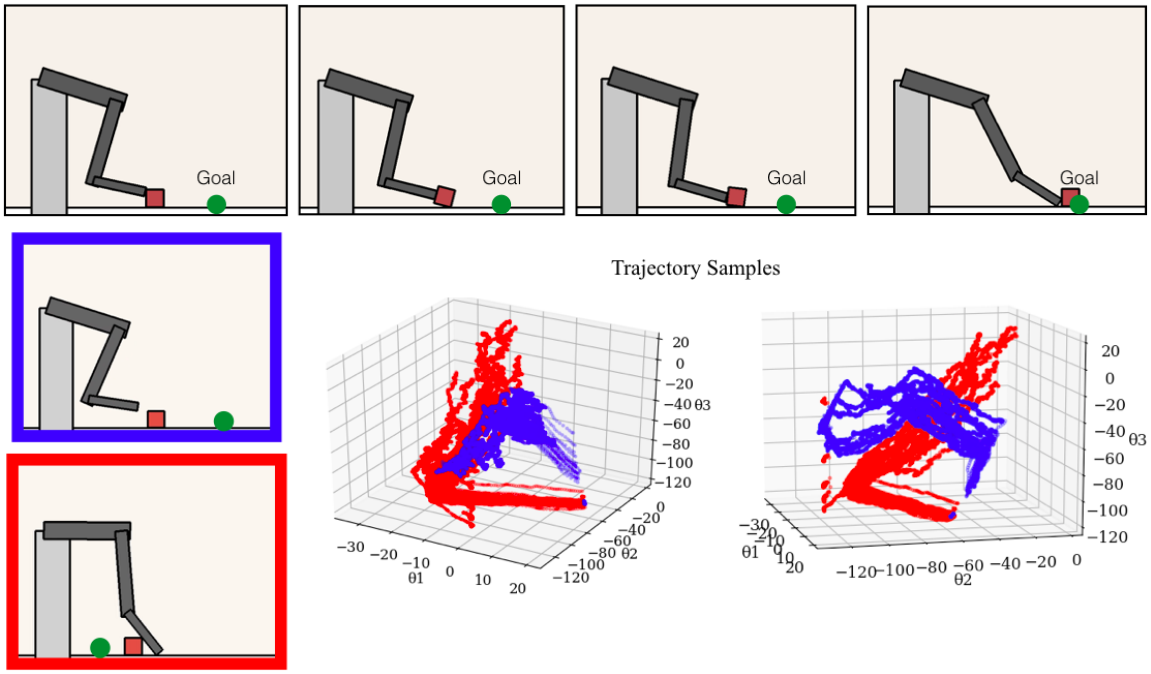


Figure 2.1: A 3-link robot arm has to push a box along a surface with friction to a randomly chosen goal state to the box's left or right without toppling it. Due to the collision and contact constraints of the 2D Surface Pushing task, it leads to two geometrically distinct pushing skills, backhand and forehand, for goals on the left and on the right. Right: 50 sampled trajectories from a motion-planning supervisor.

Our algorithmic supervisor runs an RRT Connect motion planner (from the Open Motion Planning Library ompl) at each time-step planning till the goal. The motion planning algorithm contains a single important hyper-parameter which is the maximum length of branches to add to the tree. We set this at 0.1 units (for a 20 x 20 unit space). First, we consider the case where we observe the full low-dimensional state of the system: three joint angles, the box's position and orientation, and the position of the goal state. We compare our hierarchical policies with two flat, non-hierarchical policies. One baseline policy we consider is an under-parametrized linear policy which is not expressive enough to approximate the supervisor policy well. The other baseline policy is a multi-layer perceptron (MLP) policy. We train each of these policies via Behavior Cloning (BC), i.e. by maximiz-

ing the likelihood each gives to the set of demonstrations. As expected, the flat linear policy is unsuccessful at the task for any number of observed demonstrations (Figure 2.2 Top-A). The MLP policy, on the other hand, can achieve the maximum reward when trained on 60 demonstrations or more.

We apply DDO and learn two 2-level hierarchical policies, one with linear low-level options, and the other with MLP low-level options of the same architecture used for the flat policy.

Multi-Layer Perceptron Flat Policy: One of the baseline policies is a multi-layer perceptron (MLP) policy which has a single Rectified Linear Unit (ReLU) hidden layer of 64 nodes. This policy is implemented in Tensorflow and is trained with a stochastic gradient descent optimizer with learning rate 10^{-5}.

DDO Policy 1: In the first policy trained by DDO, we have a logistic regression meta-policy that selects from one of k linear sub-policies. The linear sub-policies execute until a termination condition determined again by a logistic regression. This policy is implemented in Tensorflow and is trained with an ADAM optimizer with learning rate 10^{-5}. For the linear hierarchy, we set DDO to discover 5 options, which is tuned using the cross-validation method described before.

DDO Policy 2: In the second policy trained by DDO, we have a logistic regression meta-policy that selects from one of k multi-layer perceptron sub-policies. As with the flat policy, it has a single ReLU hidden layer of 64 nodes. The MLP sub-policies execute until a termination condition determined again by a logistic regression. This policy is implemented in Tensorflow and is trained with an ADAM optimizer with learning rate 10^{-5}. For the MLP hierarchy, we set DDO to discover 2 options, which is tuned using the cross-validation method described before.

In both cases, the termination conditions are parametrized by a logistic regression from the state to the termination probability, and the high-level policy is a logistic regression from the state to an option selection. The MLP hierarchical policy can achieve the maximum reward with 30 demonstrations, and is therefore 2x more sample-efficient than its flat counterpart (Figure 2.2 Top A). We also vary the number of options discovered by DDO, and plot the reward obtained by the resulting policy (Figure 2.2 Top-B). While the performance is certainly sensitive to the number of options, we find that the benefit of having sufficiently many options is only diminished gradually with each additional option beyond the optimum. Importantly, the peak in the cross-validated log-likelihood corresponds to the number of options that achieves the maximum reward (Figure 2.2 Top-C). This allows us to use cross-validation to select the number of options without having to evaluate the policy by rolling it out in the environment.

Observing Images: Next, we consider the case where the sensor inputs are 640×480 images of the scene. The low-dimensional state is still fully observable in these images, however these features are not observed explicitly, and must be extracted by the control policy. We consider two neural network architectures to represent the policy: a convolutional layer followed by either a fully connected layer or an LSTM, respectively forming

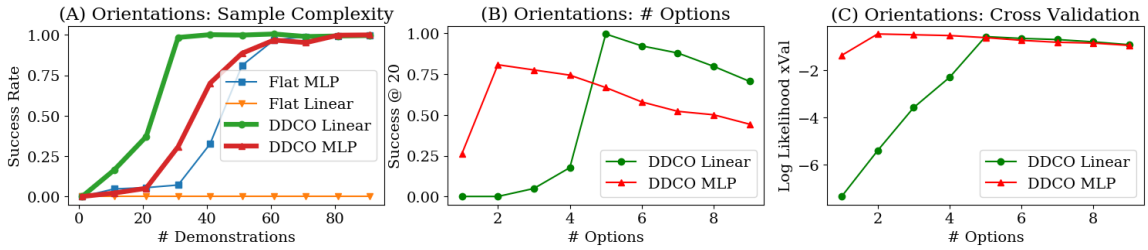


Figure 2.2: 2D Surface Pushing from low-dimensional observations and image observations. (A) The sample complexity of different policy representations, (B) The success rate for different numbers of discovered options, (C) The peak in the reward correlates over the number of options with the 10-fold cross-validated log-likelihood.

a feed-forward (FF) network and a recurrent network (RNN). We use these architectures both for flat policies and for low-level options in hierarchical policies. For the hierarchical policies, as in the previous experiment, we consider a high-level policy that only selects options. For the FF policy we discover 4 options, and for the RNN policy we discover 2 options.

DDO FF Policy: First, we consider a neural network with a convolutional layer with 64 5×5 filters followed by a fully connected layer forming a feed-forward (FF) network. The high-level model only selects options and is parametrized by the same general architecture with an additional softmax layer after the fully connected layer. This means that the meta-control policy is a two-layer convolutional network whose output is a softmax categorical distribution over options. This policy is implemented in Tensorflow and is trained with an ADAM optimizer with learning rate 10^{-5}. We used $k = 2$ options for the FF policy.

DDO RNN Policy: Next, we consider a neural network with a convolutional layer with 64 5×5 filters followed by an LSTM layer forming a recurrent network (RNN). The high-level model only selects options and is parametrized by the same general architecture with an additional softmax layer after the LSTM. This policy is implemented in Tensorflow and is trained with a Momentum optimizer with learning rate 10^{-4} and momentum 10^{-3}. We used $k = 4$ options for the RNN policy.

The hierarchical policies require 30% fewer demonstrations than the flat policies to achieve the maximum reward (Figure 2.2 Bottom-A). Figure 2.2 Bottom-B and C show how the success rate and cross-validated log-likelihood vary with the number of options. As for the low-dimensional inputs, the success rate curve is correlated with the cross-validated log-likelihood. We can rely on this to select the number of options offline without rolling out the learned hierarchical policy.

Control Space Augmentation (Kinematic): We test two different architectures for the output layer of the high-level policy: either a softmax categorical distribution selecting an option, or the hybrid categorial–continuous distribution output described in. The low-level policies are linear.

Figure 2.4 describes the empirical estimation, through policy rollouts, of the success

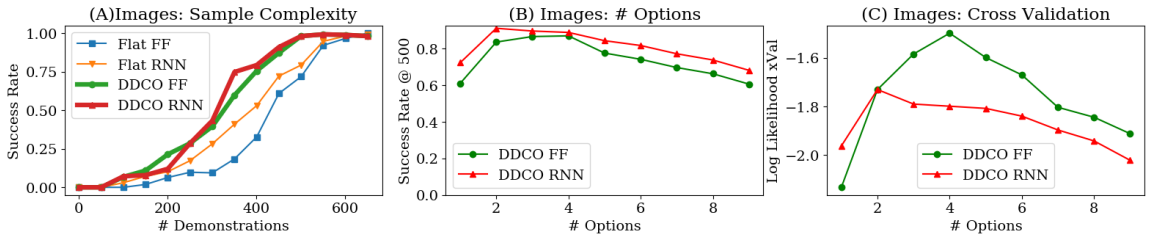


Figure 2.3: 2D Surface Pushing from image observations and image observations. (A) The sample complexity of different policy representations, (B) The success rate for different numbers of discovered options, (C) The peak in the reward correlates over the number of options with the 10-fold cross-validated log-likelihood.

rate as a function of the number of options, and of the fraction of time that the high-level policy applies physical control. When the options are too few to provide skills that are useful throughout the state space, physical controls can be selected instead to compensate in states where no option would perform well. This is indicated by physical control being selected with greater frequency. As more options allow a better coverage of the state space, the high-level policy selects physical controls less often, allowing it to generalize better by focusing on correct option selection. With too many discovered options, each option is trained from less data on average, making some options overfit and become less useful to the high-level policy.

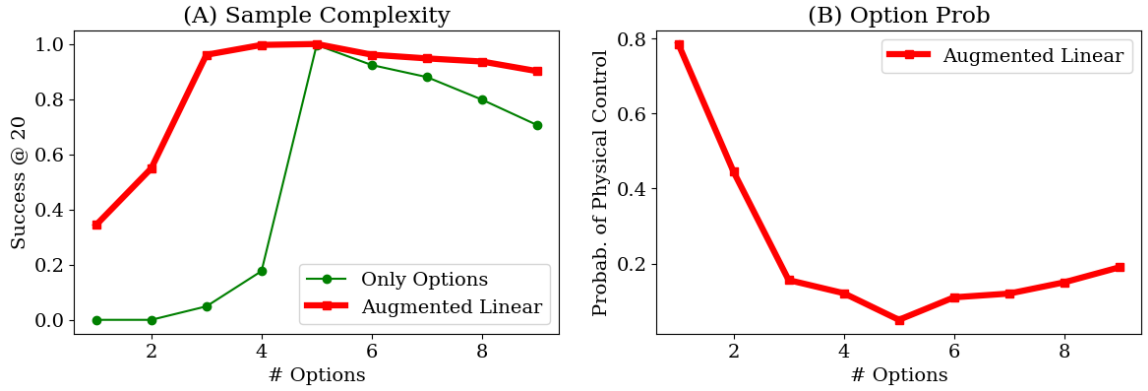


Figure 2.4: (A) The sample complexity of augmented v.s. option-only high-level control spaces. (B) The fraction of high-level selections that are of physical control. The frequency of selecting a physical control first decreases with the number of options, then increases as options overfit.

Control Space Augmentation (Images): We ran the same experiment but on the image state-space instead of the low dimensional state. The sensor inputs are 640×480 images of the scene and the task is the Box2D pushing task. Figure 2.5 describes the empirical estimation, through policy rollouts, of the success rate as a function of the number of op-

tions, and of the fraction of time that the high-level policy applies physical control. When the options are too few to provide skills that are useful throughout the state space, physical controls can be selected instead to compensate in states where no option would perform well.

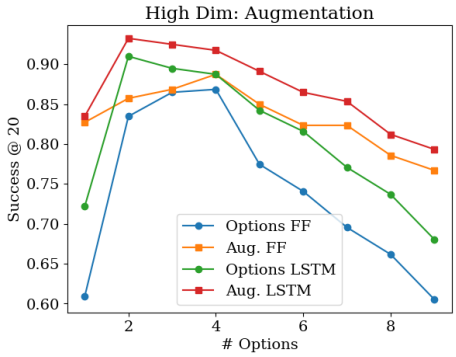


Figure 2.5: Augmenting the action space instead of selecting only options with the meta-policy leads attenuates the losses when selecting too few or too many options.

Box2D Simulation: Discussion

These results should be counterintuitive. While it is easy to reason about the low-dimensional kinematic case (where motions are essentially piecewise affine in the state), the results for image observations are less clear–why does hierarchy appear to help? To understand why, we ran a a follow up experiment. Figure 3.20 illustrates two plots color coded by kinematic class (forehand and backhand pushes). The flat plot takes a TSNE visualizatio of the output first convolutional layer of the "flat" network and illustrates how the policy is structured. While there is clear structure, the *same neuron are active for forehand motions and backhands*. On the other hand, when we visualize how the hierarchical policy partitions the trajectory space. It gives a cleaner seperation between the forehand and the backhand motions. We speculate that while the action space is continuous there is a discontinuity in the set of actions relevant to the task due to the geometry and kinematic constraints. The inherently discrete hierarchical representation captures this dicontinuity more accurately without a lot of data.

Physical Experiment 1: Needle Insertion

We constructed a physical environment to replicate the simulation results on a real robotic platform. We consider a needle orientation and insertion task on the da Vinci Research Kit (DVRK) surgical robot (Figure 2.7). In this task, the robot must grasp a surgical needle, reorient it in parallel to a tissue phantom, and insert the needle into the tissue phantom. The task is successful if the needle is inserted into a 1cm diameter target region on the

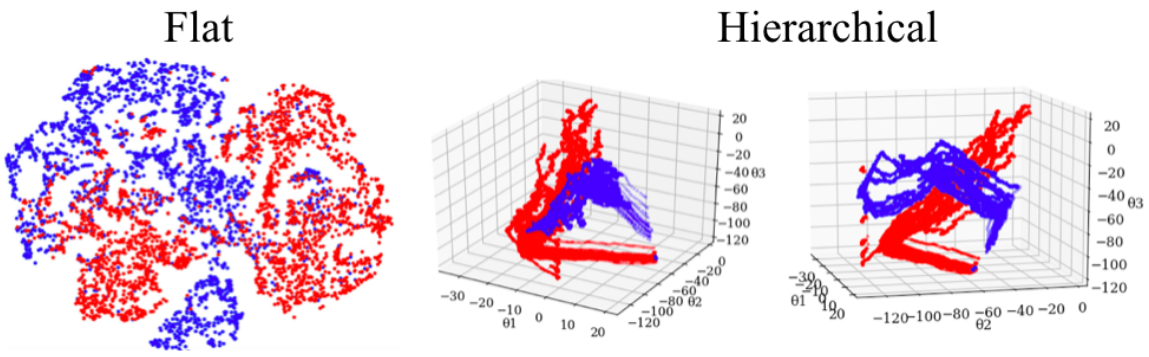


Figure 2.6: A visualization of how the flat and hierarchical policy seperate the space of observations. The flat policy is visualized with a tsne plot of its first

phantom. Small changes in the needle's initial orientation can lead to large changes in the in-gripper pose of the needle due to deflection. The state is the current 6-DoF pose of the robot gripper, and algorithmically extracted visual features that describe the estimated pose of the needle. These features are derived from an image segmentation that masks the needle from the background and fits an ellipsoid to the resulting pixels. The principal axis of this 2D ellipsoid is a proxy for the pose of the needle. The task runs for a fixed 15 time-steps, and the policy must set the joint angles of the robot at each time-step.

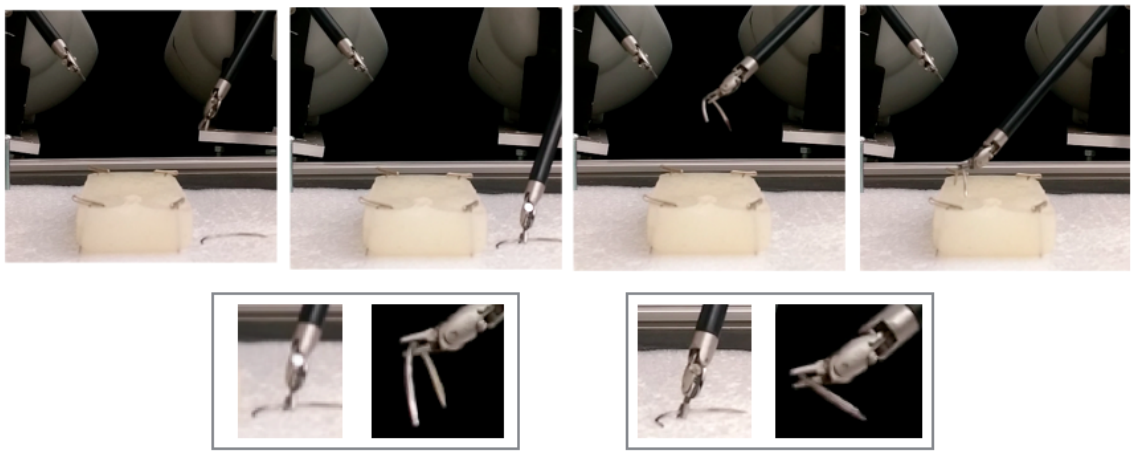


Figure 2.7: A needle orienting and insertion task. The robot must grasp a surgical needle, reorient it in parallel to a tissue phantom, and insert the needle.

The needle's deflection coupled with the inaccurate kinematics of the DVRK make it challenging to plan trajectories to insert the needle properly. A visual servoing policy needs to be trained that can both grasp the needle in the correct position, as well as reorient the gripper in the correct direction after grasping. To collect demonstrations, we programmed an initial open-loop control policy, interrupted the robot via keyboard input when adjust-

ment was needed, and kinesthetically adjusted the pose of the gripper. We collected 100 such demonstrations.

We evaluated the following alternatives: (1) a single flat MLP policy with continuous output, (2) a flat policy consisting of 15 distinct MLP networks, one for each time-step, (3) a hierarchical policy with 5 options trained with DDO. We considered a hierarchy where the high-level policy is a MLP with a softmax output that selects the appropriate option, and each option is parametrized by a distinct MLP with continuous outputs. DDO learns 5 options, two of which roughly correspond to the visual servoing for grasping and lifting the needle, and the other three handle three different types of reorientation. For 100 demonstrations, the hierarchical policy learned with DDO has a 45% higher log-likelihood measured in cross-validation than the flat policy, and a 24% higher log-likelihood than the per-timestep policy.

Task Description: The robot must grasp a 1mm diameter surgical needle, re-orient it parallel to a tissue phantom, and insert the needle into a tissue phantom. The task is successful if the needle is inserted into a 1 cm diameter target region on the phantom. In this task, the state-space is the current 6-DoF pose of the robot gripper and visual features that describe the estimated pose of the needle. These features are derived from an image segmentation that masks the needle from the background and fits an ellipsoid to the resulting pixels. The principal axis of this 2D ellipsoid is a proxy for the pose of the needle. The task runs for a fixed 15 time-steps and the policy must set the joint angles of the robot at each time-step.

Robot Parameters: The challenge is that the curved needle is sensitive to the way that it is grasped. Small changes in the needle's initial orientation can lead to large changes to the in-gripper pose of the needle due to deflection. This deflection coupled with the inaccurate kinematics of the DVRK leads to very different trajectories to insert the needle properly.

The robotic setup includes a stereo endoscope camera located 650 mm above the 10cm x 10 cm workspace. After registration, the dvrk has an RMSE kinematic error of 3.3 mm, and for reference, a gripper width of 1 cm. In some regions of the state-space this error is even higher, with a 75% percentile error of 4.7 mm. The learning in this task couples a visual servoing policy to grasp the needle with the decision of which direction to orient the gripper after grasping.

Demonstration Protocol: To collect demonstrations, we programmed an initial open-loop control policy. This policy traced out the basic desired robot motion avoiding collisions and respecting joint limits, and grasping at where it believed the needle was and an open-loop strategy to pin the needle in the phantom. This was implemented by 15 joint angle way points which were interpolated by a motion planner. We observed the policy execute and interrupted the robot via keyboard input when adjustment was needed. This interruption triggered a clutching mechanism and we could kinesthetically adjusted the joints of the robot and pose of the gripper (but not the open-close state). The adjustment was recorded as a delta in joint angle space which was propagated through the rest of the trajectory. We collected 100 such demonstrations and images of these adjustments are visualized in image (Figure 2.8).

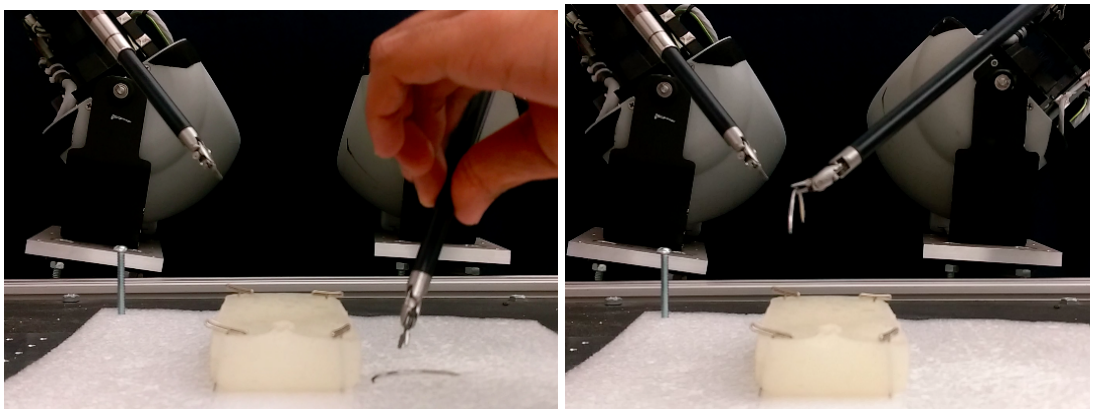


Figure 2.8: To collect demonstrations, we programmed an initial open-loop control policy. We observed the policy execute and interrupted the robot via keyboard input when adjustment was needed.

Learning the Parameters: Figure 2.9A plots the cross-validation log-likelihood as a function of the number of demonstrations. We find that the hierarchical model has a higher likelihood than the alternatives—meaning that it more accurately explains the observed data and generalizes better to held out data. At some points, the relative difference is over 30%. It, additionally, provides some interpretability to the learned policy. Figure 2.9B visualizes two representative trajectories. We color code the trajectory based on the option active at each state (estimated by DDO). The algorithm separates each trajectory into 3 segments: needle grasping, needle lifting, and needle orienting. The two trajectories have the same first two options but differ in the orientation step. One of the trajectories has to rotate in a different direction to orient the needle before insertion.

Flat Policy: One of the baseline policies is a multi-layer perceptron (MLP) policy which has a single ReLU hidden layer of 64 nodes. This policy is implemented in Tensorflow and is trained with an ADAM optimizer with learning rate 10^{-5}.

Per-Timestep Policy: Next, we consider a degenerate case of options where each policy executes for a single-timestep. We train 15 distinct multi-layer perceptron (MLP) policies each of which has a single ReLU hidden layer of 64 nodes. Thes policies are implemented in Tensorflow and are trained with an ADAM optimizer with learning rate 10^{-5}.

DDO Policy: DDO trains a hierarchical policy with 5 options. We considered a hierarchy where the meta policy is a multilayer perceptron with a softmax output that selects the appropriate option, and the options are parametrized by another multilayer perceptron with continuous outputs. Each of the MLP policies has a single ReLU hidden layer of 64 nodes. Thes policies are implemented in Tensorflow and are trained with an ADAM optimizer with learning rate 10^{-5}.

Execution: For each of the methods, we execute ten trials and report the success rate (successfully grasped and inserted the needle in the target region), and the accuracy. The

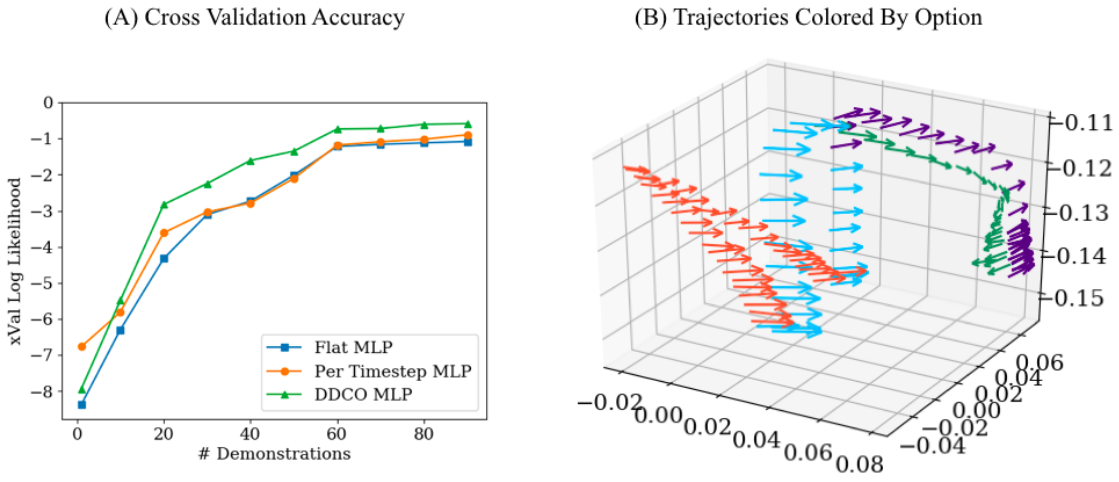


Figure 2.9: (A) We plot the cross validation likelihood of the different methods as a function of the number of demonstrations. (B) We visualize two representative trajectories (position and gripper orientation) color coded by the most likely option applied at that timestep. We find that the two trajectories have the same first two options but then differ in the final step due to the re-orientation of the gripper before insertion.

results are described in aggregate in the table below:

	Overall Success	Grasp Success	Insertion Success	Insertion Accuracy
Open Loop	2/10	2/10	0/0	7 ± 1 mm
Behavioral Cloning	3/10	6/10	3/6	6 ± 2 mm
Per Timestep	6/10	7/10	6/7	5 ± 1 mm
DDO	8/10	10/10	8/10	5 ± 2 mm

We ran preliminary trials to confirm that the trained options can be executed on the robot. For each of the methods, we report the success rate in 10 trials, i.e. the fraction of trials in which the needle was successfully grasped and inserted in the target region. All of the techniques had comparable accuracy in trials where they successfully grasped and inserted the needle into the 1cm diameter target region. The algorithmic open-loop policy only succeeded 2/10 times. Surprisingly, Behavior Cloning (BC) did not do much better than the open-loop policy, succeeding only 3/10 times. Per-timestep BC was far more successful (6/10). Finally, the hierarchical policy learned with DDO succeeded 8/10 times. On 10 trials it was successful 5 times more than the direct BC approach and 2 times more than the per-timestep BC approach. While not statistically significant, our preliminary results suggest that hierarchical imitation learning is also beneficial in terms of task success, in addition to improving model generalization and interpretability.

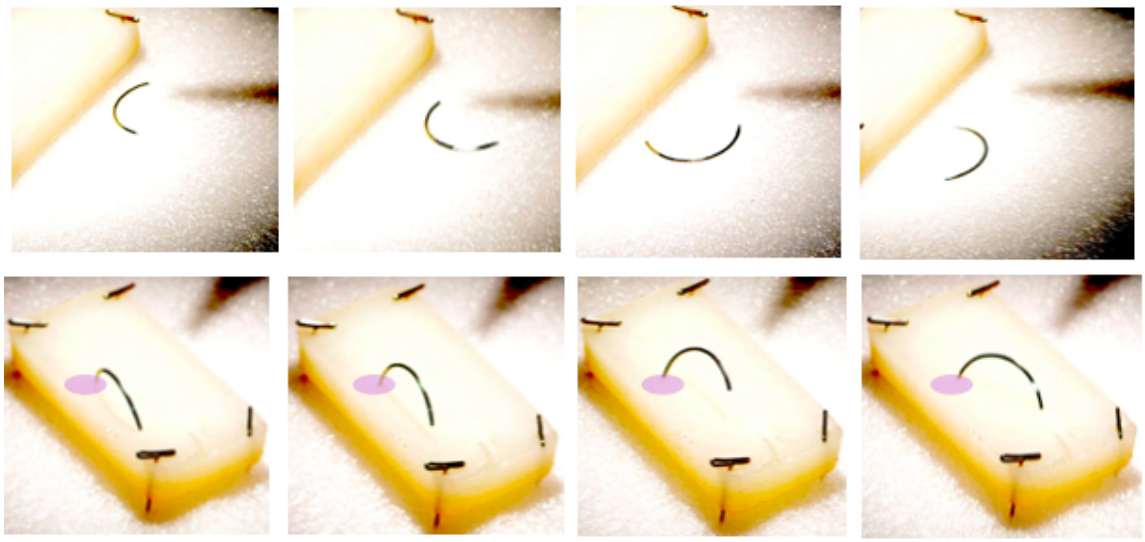


Figure 2.10: Illustration of the needle orientation and insertion task. Above are images illustrating the variance in the initial state, below are corresponding final states after executing DDO.

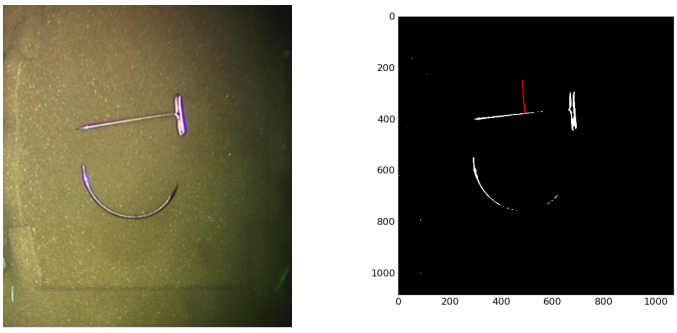


Figure 2.11: The endoscope image and a corresponding binary mask with a selected grasp. The arrow corresponds to the orientation of the gripper along the grasp axis.

Physical Experiment 2: Surgical Bin Picking

In this task, the robot is given a foam bin with a pile of 5–8 needles of three different types, each 1–3mm in diameter. The robot must extract needles of a specified type and place them in an "accept" cup, while placing all other needles in a "reject" cup. The task is successful if the entire foam bin is cleared into the correct cups.

In initial trials, the kinematics of the DVRK were not precise enough for grasping needles. We then realized that visual servoing is needed, which requires learning. However, even with visual servoing, failures are common, and we would like to also learn automatic recovery behaviors. To define the state space for this task, we first generate binary images

from overhead stereo images, and apply a color-based segmentation to identify the needles (the image input). Then, we use a classifier trained in advance on 40 hand-labeled images to identify and provide a candidate grasp point, specified by position and direction in image space (the grasp input). Additionally, the 6 DoF robot gripper pose and the open-closed state of the gripper are observed (the kin input). The state space of the robot is (image, grasp, kin), and the control space is the 6 joint angles and the gripper angle.

Each sequence of grasp, lift, move, and drop operations is implemented in 10 control steps of joint angle positions. As in the previous task, we programmed an initial open-loop control policy, interrupted the robot via keyboard input when adjustment was needed, and kinesthetically adjusted the pose of the gripper. We collected 60 such demonstrations, in each fully clearing a pile of 3–8 needles from the bin, for a total of 450 individual grasps.

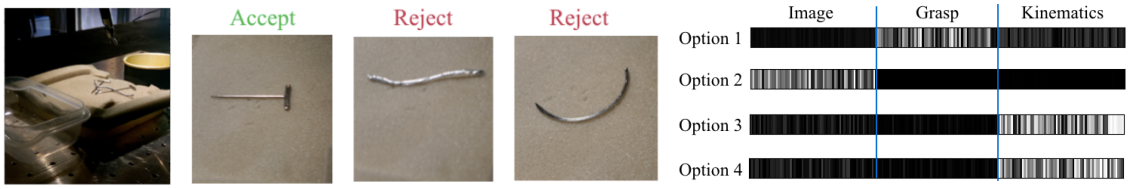


Figure 2.12: (Left) Surgical Bin Picking: the robot is given a foam bin with a pile of 5–8 needles of three different types, each 1–3mm in diameter. The robot must extract needles of a specified type and place them in an "accept" cup, while placing all other needles in a "reject" cup. (Right) For each of the 4 options, we plot how heavily the different inputs are weighted (image, grasp, or kin) in computing the option's action. Nonzero values of the ReLU units are marked in white and indicate input relevance.

We apply DDO to the collected demonstrations with the policy architecture described in **SM 1.4**. In this network, there are three inputs: a binary image, a candidate grasp, and kinematics (kin). These inputs are processed by distinct branches of the network before being aggregated into a single feature layer. Using the cross-validation technique described before, we selected the number of options to be $k = 4$.

We plot the average activations of the feature layer for each low-level option (Figure 2.12). While this is only a coarse analysis, it gives some indication of which inputs (image, grasp, or kin) are relevant to the policy and termination. We see that the options are clearly specialized. The first option has a strong dependence only on the grasp candidate, the second option attends almost exclusively to the image, while the last two options rely mostly on kin. The experimental details are described below:

Task Description: We consider a task with a more complicated high-level structure. In this task, the robot is given a foam bin with 3 different types of needles (1mm-3mm in diameter) lying in a pile (5-8 needles in experiments). The robot must extract a particular type of needle and place it in the accept cup and place all others in a reject cup. The task is successful if the entire foam bin is cleared.

Figure 2.13 shows representive objects for this task. We consider three types of "nee-

dles": dissection pins, suturing needles, and wires. Dissection pins are placed in the accept cup and the other two are placed in the reject cup.

Robot and State-Space Parameters: As in the previous task, the task requires learning because the kinematics of the dvrk are such that the precision needed for grasping needles requires visual servoing. However, even with visual servoing, failures are common due to the pile (grasps of 2, 3, 4 objects). We would like to automatically learn recovery behaviors. In our robotic setup, there is an overhead endoscopic stereo camera, and it is located 650mm above the workspace.

To define the state-space for this task, we first generate binary images from the stereo images and apply a color-based segmentation to identify the needles (we call this feature image). Then, we use a classifier derived from 40 hand-labeled images to identify possible grasp points to sample a candidate grasp (left pixel value, right pixel value, and direction) (we call this feature grasp). These features are visualized in Figure 2.11. Additionally, there is the 6 DoF robot gripper pose and the open-close state of the gripper (we call this feature kin). The state-space of the robot is (kin, image, grasp), and the action space for the robot is 6 joint angles and the gripper angle. Each grasp, lift, move, and drop operation consists of 10 time steps of joint angle positions. The motion between the joint angles is performed using a SLURP-based motion planner.

Figure 2.13: There are two bins, one accept and one reject bin. In the accept bin, we place dissection pins and place the suturing needles and the wires in the other.

Demonstration Protocol: As in the previous task, to collect demonstrations, we start with a hard-coded open-loop policy. We roll this policy out and interrupt the policy when we anticipate a failure. Then, we kinesthetically adjust the pose of the dvrk and it continues. We collected 60 such demonstrations of fully clearing the bin filled with 3 to 8 needles each– corresponding to 450 individual grasps. We also introduced a key that allows the robot to stop in place and drop it current grasped needle. Recovery behaviors were triggered when the robot grasps no objects or more than one object. Due to the kinesthetic corrections, a

very high percentage of the attempted grasps (94%) grasped at least one object. Of the successful grasps, when 5 objects are in the pile 32% grasps picked up 2 objects, 14% picked up 3 objects, and 0.5% picked up 4. In recovery, the gripper is opened and the epsiode ends leaving the arm in place. The next grasping trial starts from this point.

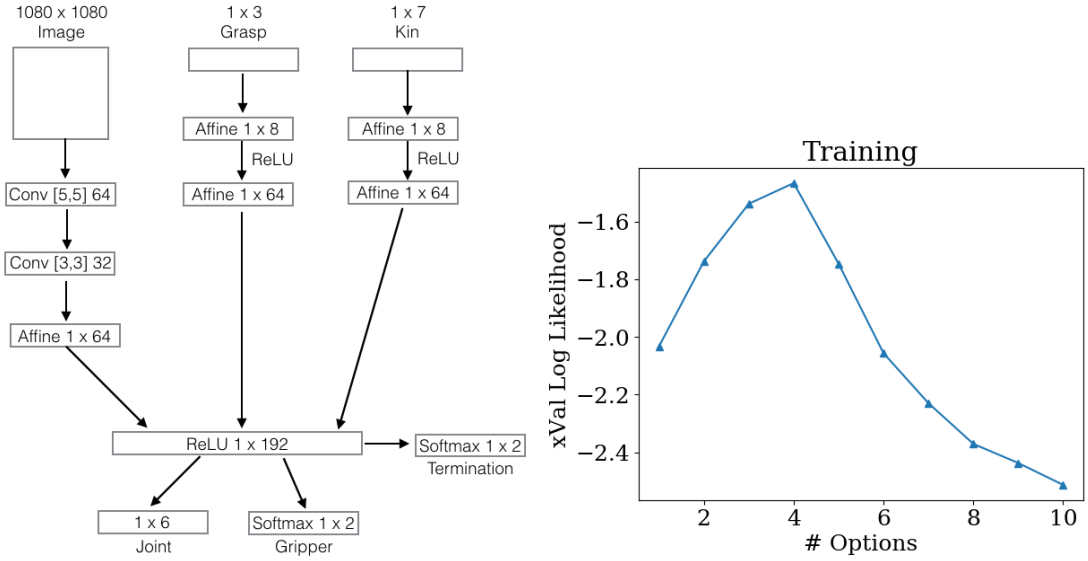


Figure 2.14: We use the above neural network to parametrize each of the four options learned by DDO. In this network, there are three inputs the a binary image, a candidate grasp, and kinematics. These inputs are processed through individual neural net branches and aggregated into a single output layer. This output layer sets the joint angles of the robot and the gripper state (open or closed). Option termination is also determined from this output.

Policy Parametrization: We apply DDO to the collected demonstrations with the policy parametrization described in Figure 2.14. In this network, there are three inputs the a binary image, a candidate grasp, and kinematics. These inputs are processed through individual neural net branches and aggregated into a single output layer. This output layer sets the joint angles of the robot and the gripper state (open or closed). Option termination is also determined from this output. Using the cross-validation technique described in before, we identify 4 options (Figure 2.14). The high-level policy has the same parametrization but after the output layer there is a softmax operation that selects a lower-level option.

Coordinate Systems: We also experimented with different action space representations to see if that had an effect on single object grasps and categorizations. We trained alternative policies with the collected dataset where instead of predicting joint angles, we alternatively predicted 6 DoF end-effector poses with a binary gripper open-close state , and end-effector poses represented a rotation/translation matrix with a binary gripper open-close state. We found that the joint angle representation was the most effective. In particular, we found that

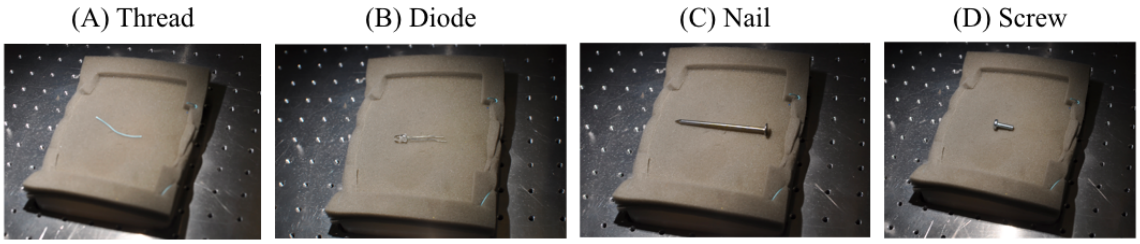


Figure 2.15: We evaluated the generalization of the learned policy on a small set of unseen objects. This was to understand what features of the object binary mask is used to determine behaviors.

for the grasping part of the task, a policy that controlled the robot in terms of tooltip poses was unreliable.

	Items	Successful Grasp	Successful Recovery	Successful Categorizations
Joint Angle	8	7/8	2/2	7/7
Tooltip Pose	8	3/8	5/5	3/3
Rotation	8	2/8	0/8	0

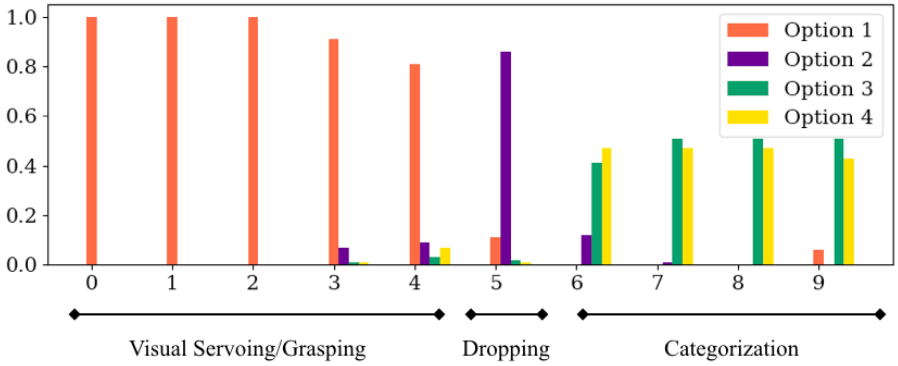


Figure 2.16: We plot the time distribution of the options selected by the high-level policy. The x-axis represents time, and the bars represent the probability mass assigned to each option. We find that the structure of the option aligns with key phases in the task such as servoing to the needle, grasping it, and categorizing.

Interpreting Learned Options: We additionally analyzed the learned options to see if there was an interpretable structure. We examined the collected demonstrations and looked at the segmentation structure. We average over the 60 trials the probability for the high-level policy to choose each option in each of first 10 time-steps during training (Figure 2.16). We find that the options indeed cluster visited states and segment them in alignment with key phases in the task, such as servoing to the needle, grasping it, dropping it if necessary, and categorizing it into the accept cup.

Full Break Down of Experimental Results: In 10 trials, 7 out of 10 were successful (Table 2.4). The main failure mode was unsuccessful grasping (defined as either no needles or multiple needles). As the piles were cleared and became sparser, the robot's grasping policy became somewhat brittle. The grasp success rate was 66% on 99 attempted grasps. In contrast, we rarely observed failures at the other aspects of the task. Of the grasps that failed, nearly 34% were due to grasping multiple objects.

Trial #	# of Needles	# Needles Cleared	Grasping	Recovery	Categorization
1	6	**6**	6/9	3/3	6/6
2	8	6	7/12	5/6	6/7
3	7	**7**	7/8	1/1	7/7
4	6	**6**	6/10	4/4	6/6
5	7	6	6/11	5/5	6/6
6	8	**8**	8/13	5/5	8/8
7	6	5	5/9	4/4	5/5
8	7	**7**	7/10	3/3	7/7
9	8	**8**	8/10	2/2	8/8
10	6	**6**	6/7	1/1	6/6
Totals	69	Success: 70%	Success: 66%	Success: 97%	Success: 98%

Generalization to unseen objects: We also evaluated the learned policy on a few unseen objects (but similarly sized and colored) to show that there is some level of generalization in the learning. We tried out four novel objects and evaluated what the learned policy did for each (Figure 2.15). For each of the novel objects we tried out 10 grasps in random locations and orientations in the bin (without any others). We evaluated the grasp success and whether it was categorized consistently (i.e., does the learned policy consistently think it is a pin or a needle).

We found that the diode was consistently grasped and categorized as a dissection pin. We conjecture this is because of its head and thin metallic wires. On the other hand, the screw and the thread were categorized in the reject cup. For 8/10 of the successful grasps, the nail was categorized as a failure mode. We conjucuture that since it is the large object it looks similar to the two object grasps seen in the demonstrations.

	Grasps	Successful	Drop	Categorize Accept	Categorize Reject
Thread	10	10	1	0	9
Diode	10	10	0	10	0
Nail	10	8	8	0	0
Screw	10	4	0	0	4

Finally, we evaluate the success of the learned hierarchical policy in 10 full trials, according to 4 different success criteria. First, the overall task success is measured by the success rate of fully clearing the bin without failing 4 consecutive grasping attempts. Second, we measure the success rate of picking exactly one needle in individual grasp attempts. Third, we measure the success rate of appropriately reacting to grasps, by dropping the load and retrying unsuccessful grasps, and not dropping successful grasps. Fourth, we measured the success rate of needle categorization.

In 10 trials, 7/10 were successful. The main failure mode was unsuccessful grasping due to picking either no needles or multiple needles. As the piles were cleared and became sparser, the robot's grasping policy became somewhat brittle. The grasp success rate was 66% on 99 attempted grasps. In contrast, we rarely observed failures at the other aspects of the task, reaching 97% successful recovery on 34 failed grasps.

Vector Quantization For Initialization

One challenge with DDO is initialization. When real perceptual data is used, if all of the low-level policies initialize randomly the forward-backward estimates needed for the Expectation-Gradient will be poorly conditioned where there is an extremely low likelihood assigned to any particular observation. The EG algorithm relies on a segment-cluster-imitate loop, where initial policy guesses are used to segment the data based on which policy best explains the given time-step, then the segments are clustered, and the policies are updated. In a continuous control space, a randomly initialized policy may not explain any of the observed data well. This means the small differences in initialization can lead to large changes in the learned hierarchy.

We found that a necessary pre-processing step was a variant of vector quantization, originally proposed for problems in speech recognition. We first cluster the state observations using a k-means clustering and train k behavioral cloning policies for each of the clusters. We use these k policies as the initialization for the EG iterations. Unlike the random initialization, this means that the initial low level policies will demonstrate some preference for actions in different parts of the state-space. We set k to be the same as the k set for the number of options, and use the same optimization parameters.

Layer-wise Hierarchy Training

We also found that layer-wise training of the hierarchy greatly reduced the likelihood of a degenerate solution. While, at least in principle, one can train When the meta-policy is very expressive and the options are initialized poorly, sometimes the learned solution can degenerate to excessively using the meta-policy (high-fitting). We can avoid this problem by using a simplified parametrization for the meta-control policy η_d used when discovering the low-level options. For example, we can fix a uniform meta-control policy that chooses each option with probability $1/k$. Now, once these low-level options are discovered, then we can augment the action space with the options and train the meta-policy.

This same algorithm can recursively proceed to deep hierarchies from the lowest level upward: level-d options can invoke already-discovered lower-level options; and are discovered in the context of a simplified level-d meta-control policy, decoupled from higher-level complexity. Perhaps counter-intuitively, this layer-wise training does not sacrifice too much during option discovery, and in fact, initial results seem to indicate that it improves the stability of the algorithm. An informative meta-control policy would serve as a prior on the assignment of demonstration segments to the options that generated them, but with suffi-

cient data this assignment can also be inferred from the low-level model, purely based on the likelihood of each segment to be generated by each option.

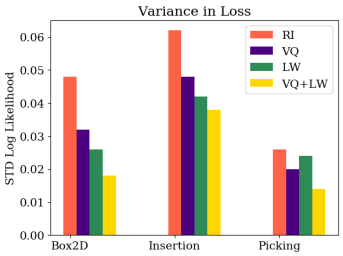


Figure 2.17: This experiment measures the variance in log likelihood over 10 different runs of DDO with different training and initialization strategies. Vector Quantization (VQ) and Layer-Wise training (LW) help stabilize the solution.

Results: Stability of DDO

In the first experiment, we measure the variance in log likelihood over 10 different runs of DDO with different training and initialization strategies (Figure 2.17). We use the entire datasets presented before and use the FF architecture for the Box2D experiments. Vector Quantization (VQ) and Layer-Wise training (LW) help stabilize the solution and greatly reduce the variance of the algorithm. This reduction in variance is over 50% in the Box2D experiments, and a little more modest for the real data.

In the next experiment, we measure the consistency of the solutions found by DDO (Figure 2.18). For each of the demonstration trajectories, we annotate the time-step by the most likely option. One can view this annotation as a hard clustering of the state-action tuples. Then, we measure the average normalized mutual information (NMI) between all pairs of the 10 different runs of DDO. NMI is a measure of how aligned two clusterings are between 0 and 1, where 1 indicates perfect alignment. As with the likelihood, Vector Quantization (VQ) and Layer-Wise training (LW) significantly improve the consistency of the algorithm.

In the last experiment, we measure the symptoms of the "high-fitting" problem (Figure 2.19). We plot the probability that the high-level policy selects an option. In some sense, this measures how much control the high-level policy delegates to the options. Surprisingly, VQ and LW have an impact on this. Hierarchies trained with VQ and LW have a greater reliance on the options.

The Effects of Dropout

For the real datasets, we leverage a technique called dropout, which has been widely used in neural network training to prevent overfitting. Dropout is a technique that randomly removes a unit from the network along with all its connections. We found that this technique

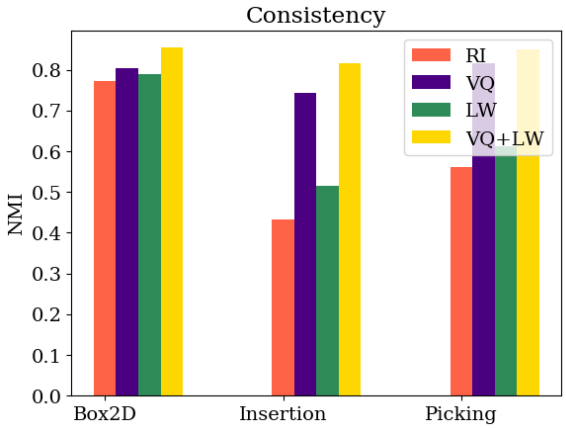


Figure 2.18: We measure the average normalized mutual information between multiple runs of DDO. This measures the consistency of the solutions across intializations.

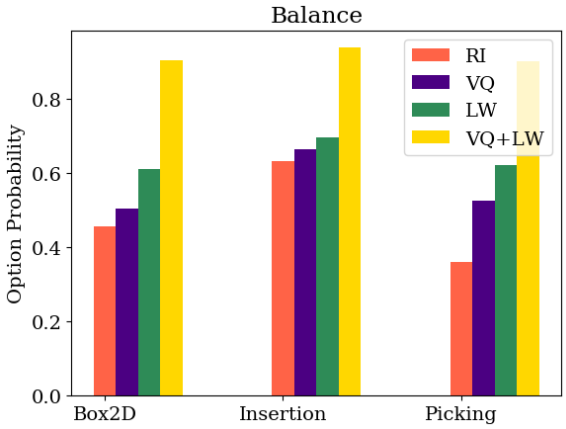


Figure 2.19: We measure probability that an option is selected by the high-level policy.

improved performance when the datasets were small. We set the dropout parameter to 0.5 and measured the performance on a hold out set. We ran an experiment on the needle insertion task dataset, where we measured the cross-validation accuracy on held out data with and without dropout (Figure 2.20). Dropout seems to have a substantial effect on improving cross-validation accuracy on a hold out set.

2.5 Segmentation of Robotic-Assisted Surgery

In this section, we illustrate the wide applicability of the DDO framework by applying it to human demonstrations in a robotic domain. We apply DDO to long robotic trajectories (e.g. 3 minutes) demonstrating an intricate task, and discover options for useful subtasks, as well as segmentation of the demonstrations into semantic units. The JIGSAWS dataset consists

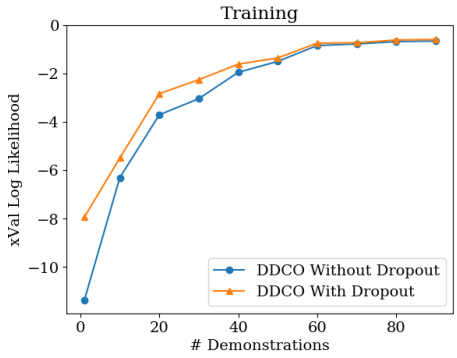


Figure 2.20: Dropout seems to have a substantial effect on improving cross-validation accuracy on a hold out set.

of surgical robot trajectories of human surgeons performing training procedures [39]. The dataset was captured using the da Vinci Surgical System from eight surgeons with different skill levels, performing five repetitions each of *needle passing*, *suturing*, and *knot tying*. This dataset consists of videos and kinematic data of the robot arms, and is annotated by experts identifying the activity occurring in each frame.

Each policy network takes as input a three-channel RGB 200×200 image, downscaled from 640×480 in the dataset, applies three convolutional layers with ReLU activations followed by two fully-connected dense layers reducing to 64 and then eight real-valued components. An action is represented by 3D translations and the opening angles of the left and right arm grippers.

We investigate how well the segmentation provided by DDO corresponds to expert annotations, when applied to demonstrations of the three surgical tasks. Figure 2.21 shows a representative sample of 10 trajectories from each task, with each time step colored by the most likely option to be active at that time. Human boundary annotations are marked in $\times$. We quantify the match between the manual and automatic annotation by the fraction of option boundaries that have exactly one human annotation in a 300 ms window around them. By this metric, DDO obtains 72% accuracy, while random guessing gets only 14%. These results suggest that DDO succeeds in learning some latent structure of the task.

2.6 Experiments: Reinforcement

We present an empirical study of DDO in the Reinforcement Learning setting. Our results suggest that DDO can discover options that accelerates reinforcement learning.

Four Rooms GridWorld

We study a simple four-room domain (Figure 2.22). On a 15×11 grid, the agent can move in four directions; moving into a wall has no effect. To simulate environment noise, we

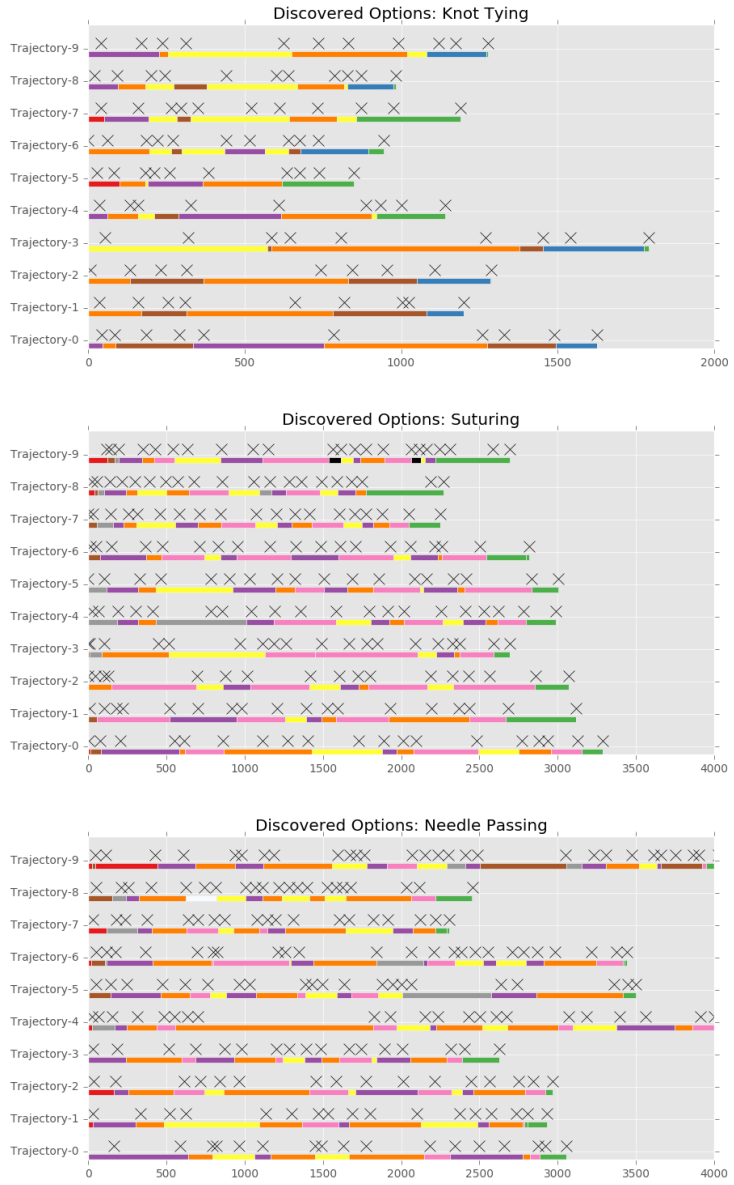


Figure 2.21: Segmentation of human demonstrations in three surgical tasks. Each line represents a trajectory, with segment color indicating the option inferred by DDO as most likely to be active at that time. Human annotations are marked as $\times$ above the segments. Automated segmentation achieves a good alignment with human annotation — 72% of option boundaries have exactly one annotation in a 300 ms window around them.

replace the agent's action with a random one with probability 0.3. An observable apple is spawned in a random location in one of the rooms. Upon taking the apple, the agent gets a unit reward and the apple is re-spawned.

We use the following notation to describe the different ways we can parametrize option discovery: A a baseline of only using atomic actions; H1u discovering a single level of options where the higher-level is parametrized by a uniform distribution; H1s discovering a single level of options where the higher-level is parametrized by an multi-layer perceptron (MLP); H2u and H2s are the two-level counterparts of H1u and H1s, respectively. All of these discovered options are used in an RL phase to augment the action space of a high-level global policy.

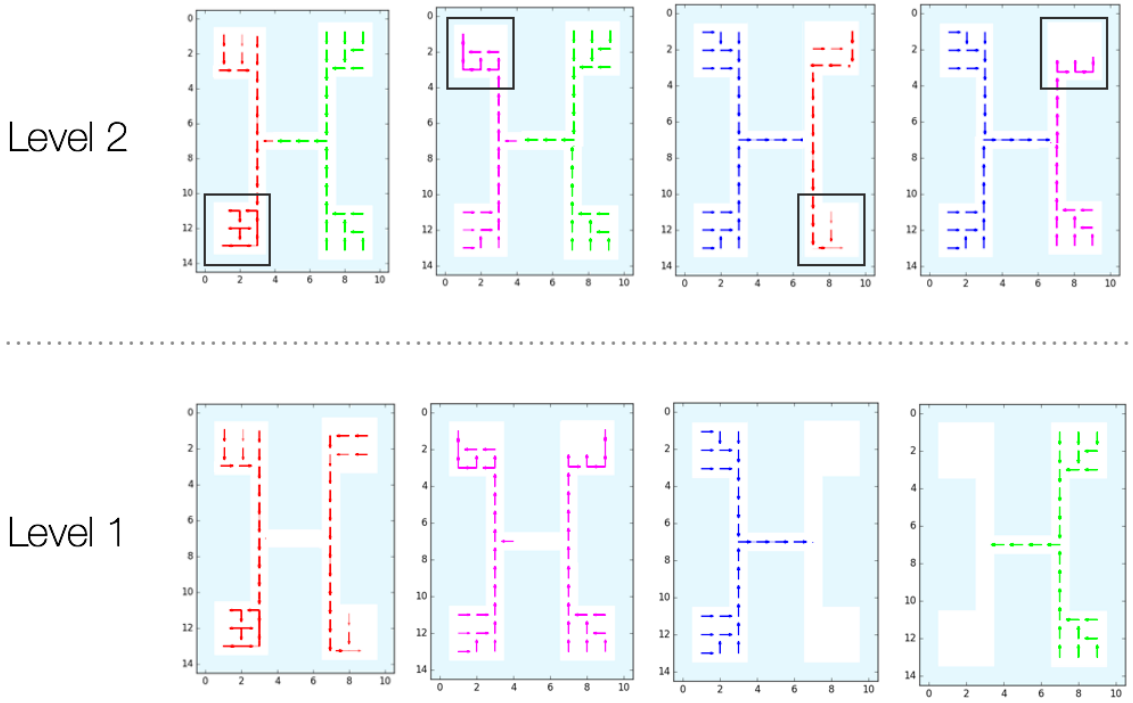


Figure 2.22: Trajectories generated by options in a hierarchy discovered by DDO. Each level has 4 options, where level 2 builds on the level 1, which in turn uses atomic actions. In the lower level, two of the options move the agent between the upper and lower rooms, while the other two options move it from one side to the other. In the higher level, each option takes the agent to a specific room. The lower-level options are color-coded to show how they are composed into the higher-level options.

Baseline. We use Value Iteration to compute the optimal policy, and then use this policy as a supervisor to generate 50 trajectories of length 1000. All policies, whether for control, meta-control or termination, are parametrized by a MLP, with a single two-node hidden layer, and $\tanh$ activation functions. The MLP's input consists of the full state (agent and apple locations), and the output is computed by a $\mathrm{softmax}$ function over the MLP output vector, which has length $|\mathcal{A}|$ for control policies and two for termination.

The options corresponding to H2u are visualized in Figure 2.22 by trajectories generated using each option from a random initial state until termination. At the first level,

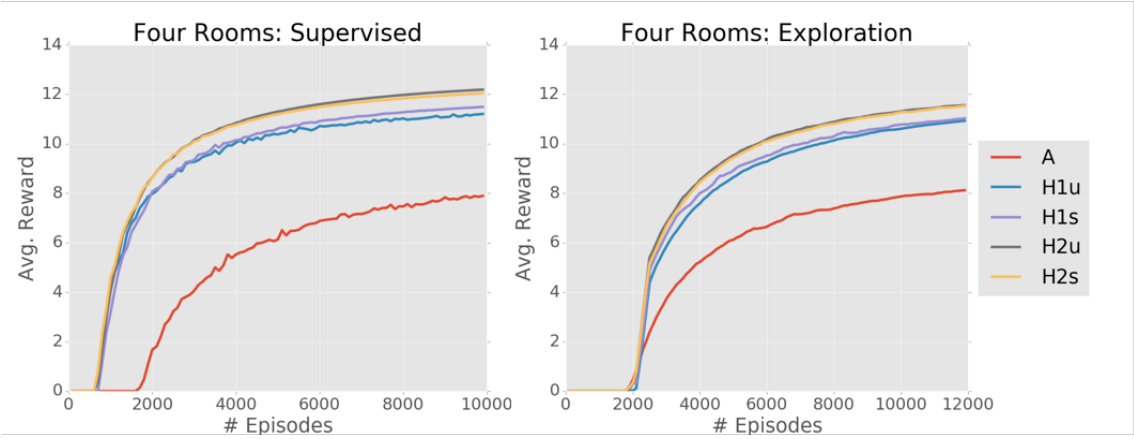


Figure 2.23: 15-trial mean reward for the Supervised and Exploration problem settings when running DQN with no options (A), low-level options (H1u) and lower- and higher-level options (H2u) augmenting the action space. The options discovered by DDO can accelerate learning, since they benefit from not being interrupted by random exploration.

two of the discovered options move the agent between the upper and lower rooms, and two move it from one side to the other. At the second level, the discovered options aggregate these primitives into higher-level behaviors that move the agent from any initial location to a specific room.

Impact of options and hierarchy depth. To evaluate the quality of the discovered options, we train a Deep Q-Network (DQN) with the same MLP architecture, and action space augmented by the options. The exploration is ϵ-greedy with $\epsilon = 0.2$ and the discount factor is $\gamma = 0.9$. Figure 2.23 shows the average reward in 15 of the algorithm's runs in the Supervised and Exploration experimental settings. The results illustrate that augmenting the action space with options can significantly accelerate learning. Note that the options learned with a two-level hierarchy (H2u) provide significant benefits over the options learned only with a single-level hierarchy H1u. The hierarchical approaches achieve roughly the same average reward after 1000 episodes as A does after 5000 episodes.

Impact of policy parametrization. To evaluate the effect of meta-control policy parametrization, we also compare the rewards during DQN reinforcement learning with options discovered with MLP meta-control policies (H1s and H2s). Our empirical results suggest that less expressive parametrization of the meta-control policy does not significantly hurt the performance, and in some cases can even provide a benefit (Figure 2.23). This is highly important, because the high sample complexity of jointly training all levels of a deep hierarchy necessitates simplifying the meta-control policy — which would otherwise be represented by a one level shallower hierarchy. We conjecture that the reason for the improved performance of the less expressive model is that more complex parametrization of the meta-control policy increases the prevalence of local optima in the inference problem, which may lead to worse options.

Exploration Setting. Finally, we demonstrate that options can also be useful when discovered from self-demonstrations by a partially trained agent, rather than by an expert supervisor. We run the same DQN as above, with only atomic actions, for 2000 episodes. We then use the greedy policy for the learned Q-function to generate 100 trajectories. We reset the Q-function (except for the baseline A), and run DQN again with the augmented action space. Figure 2.23 illustrates that even when these options are not discovered from demonstrations of optimal behavior, they are useful in accelerating reinforcement learning. The reason is that options are policy fragments that have been discovered to lead to interesting states, and therefore benefit from not being interrupted by random exploration.

Atari RAM Games

The RAM variant of the popular Atari Deep Reinforcement Learning domains considers a game-playing agent which is given not the screen, but rather the RAM state of the Atari machine. This RAM state is a 128-byte vector that completely determines the state of the game, and can be encoded in one-hot representation as $s \in \mathbb{R}^{128\times256}$. The RAM state-space illustrates the power of an automated option discovery framework, as it would be infeasible to manually code options without carefully understanding the game's memory structure. With a discovery algorithm, we have a general-purpose approach to learn in this environment.

All policies are parametrized with a deep network. There are three dense layers, each with $\tanh$ activations, and the output distribution is a $\mathrm{softmax}$ of the last layer, which has length $|\mathcal{A}|$ for control policies and two for termination. We use a single level of options, with the number of options tuned to optimize performance, and given in Figure 2.25.

For each game, we first run the DQN for 1000 episodes, and then generate 100 trajectories from the greedy policy, and use them to discover options with DDO. The DQN has the same architecture, using ϵ-greedy exploration for 1000 episodes with $\epsilon = 0.05$ and discount factor $\gamma = 0.85$ (similar to the parameters used in sygnowski2016learning). Finally, we augment the action space with the discovered options and rerun DQN for 4000 episodes. We compare this to the baseline of running DQN for 5000 episodes with actions only.

Figure 2.25 plots the estimate value, averaged over 50 trials, of the learned policies for five Atari games: Atlantis, Pooyan, Gopher, Space Invaders, and Sea Quest. In four out of five games, we see a significant acceleration learning. The relative improvements are the largest for the three hardest domains: Gopher, Sea Quest, and Space Invaders. It is promising that DDO offers such an advantage where other methods struggle. Figure 2.24 shows four frames from one of the options discovered by DDO for the Atlantis game. The option appears to identify an incoming alien and determine when to fire the gun, terminating when the alien is destroyed. As in the GridWorld experiments, the options are policy fragments that have been discovered to lead to high-value states, and therefore benefit from not being interrupted by random exploration.

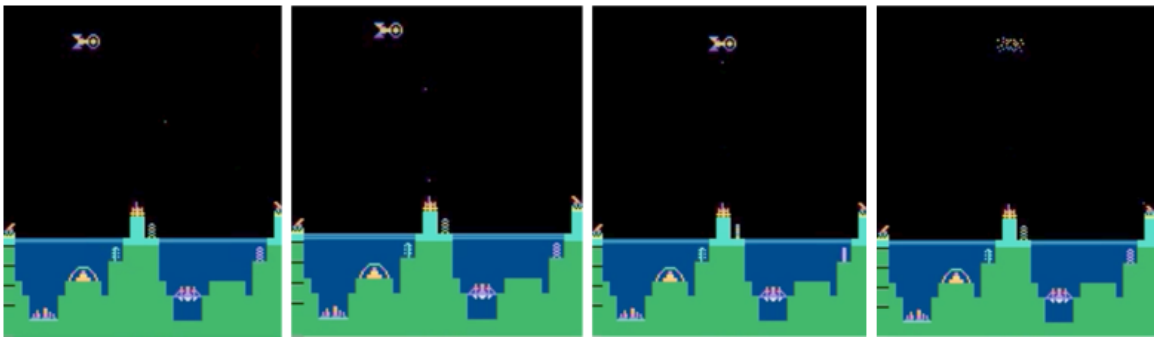


Figure 2.24: Frames sampled from a demonstration trajectory assigned to one of the primitives learned from DDO.

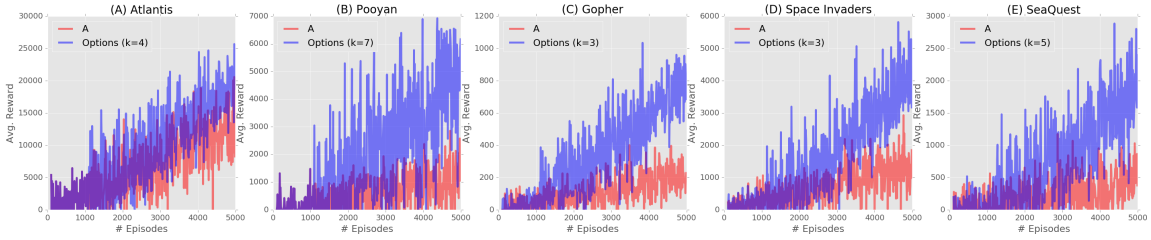


Figure 2.25: Atari RAM Games: Average reward computed from 50 rollouts when running DQN with atomic actions for 1000 episodes, then generating 100 trajectories from greedy policy, from which DDO discovers options in a 2-level hierarchy. DQN is restarted with action space augmented by these options, which accelerates learning in comparison to running DQN with atomic actions for 5000 episodes. Results suggest significant improvements in 4 out of 5 domains.

Chapter 3

Sequential Windowed Inverse Reinforcement Learning: Learning Reward Decomposition From Data

In the previous chapter, I considered a model where sequences of state-action tuples from an expert demonstrator were available. Next, we consider a setting where only the state sequence is observed. This can happen when the demonstration modality differs from the execution setting, e.g., learning from third person videos, motion capture of a human doing a task, or kinesthetic demonstrations. Thus, the data cannot be directly used to derive a controller since the actions are not visible. So, we approach this problem with a simple premise; *change in reward function implies change in behavior*. Like DDO, we apply unsupervised learning to a small number of initial expert demonstrations to learn a latent reward structure. This requires two pieces: robustly detecting consistent changes in state trajectories and constructing reward functions around those change points.

As a motivating example from the problem setting consider the widespread adoption of robot-assisted minimally invasive surgery (RMIS), which is generating datasets of kinematic and video recordings of surgical procedures [39]. Explicitly modeling the system dynamics can require learning a large number of parameters. This makes a direct system identification approach somewhat sensitive to any noise in the dataset, especially when the datasets are small. Furthermore, the states are often high-dimensional with combinations of kinematic and visual features.

Suppose, one could decompose a surgical task into a consistent sequence of sub-tasks. While each demonstration motion may vary and be noisy, each demonstration also contains roughly the same order of true segments. This consistent, repeated structure can be exploited to infer global segmentation criteria. By assuming known sequential segment-to-

segment ordering, the problem reduces to identifying a common set of segment-to-segment transition events–not corresponding entire the trajectory segments across the whole dataset. This allows us to apply coarser, imperfect motion-based segmentation algorithms first that create a large set of candidate transitions. Then, we can filter this set by identifying transition events that occurred at similar times and states. Our experiments suggest that this approach has improved robustness and sample-efficiency, while approximating the behavior of more complicated dynamical systems-based approaches in many real problems.

I formalize this intuition into a new hierarchical clustering algorithm for unsupervised segmentation called Transition State Clustering. The proposed approach is also relevant to problems in other domains, but I will focus on results from surgical applications. TSC first applies a motion-based segmentation model over the noisy trajectories and identifies a set of candidate segment transitions in each. TSC then clusters the transition states (states at times transitions occur) in terms of kinematic, sensory, and temporal similarity. The clustering process is hierarchical where the transition states are first assigned to Gaussian Mixture clusters according to kinematic state, then these clusters are sub-divided using the sensory features, and finally by time. We present experiments where these sensory features are constructed from video. The learned clustering model can be applied to segment previously unseen trajectories by the same global criteria. To avoid setting the number of clusters at each level of the hierarchy in advance, the number of regions are determined by a Dirichlet Process prior. A series of merging and pruning steps remove sparse clusters and repetitive loops.

3.1 Transition State Clustering

This section describes the problem setting, assumptions, and notation. Let $D = \{d_i\}$ be a set of demonstrations of a robotic task. Each demonstration of a task d is a discrete-time sequence of T state vectors in a feature-space $\mathcal{X}$. The feature space is a concatenation of kinematic features X (e.g., robot position) and sensory features V (e.g., visual features from the environment).

Definition 1 (Segmentation) *A segmentation of a task is defined as a function* $\mathbf{S}$ *that assigns each state in every demonstration trajectory to an integer* $1, 2, ..., k$*:*

$$\mathcal{S} : d \mapsto (a_n)_{1,...,|d|}, a_n \in 1, ..., k.$$

and $\mathcal{S}$ *is a non-decreasing function in time (no repetitions).*

Suppose we are given a function that just identifies candidate segment endpoints based on the kinematic features. Such a function is weaker than a segmentation function since it does not globally label the detected segments. This leads to the following definition:

Definition 2 (Transition Indicator Function) *A transition indicator function* $\mathbf{T}$ *is a function that maps each kinematic state in a demonstration* d *to* $\{0, 1\}$*:*

$$\mathbf{T} : d \mapsto (a_n)_{1,...,|d|}, a_n \in 0, 1.$$

The above definition naturally leads to a notion of transition states, the states and times at which transitions occur.

Definition 3 (Transition States) *For a demonstration d_i, let $o_{i,t}$ denote the kinematic state, visual state, and time (x, v, t) at time t. Transition States are the set of state-time tuples where the indicator is 1:*

$$\Gamma = \bigcup_{i}^{N} \{o_{i,t} \in d_i \ : \mathbf{T}(d_i)_t = 1\}.$$

The goal of TSC is to take the transition states Γ and recover a segmentation function $\mathbf{S}$. This segmentation function is stronger than the provided $\mathbf{T}$ since it not only indicates that a transition has occurred but labels the segment transition consistently across demonstrations.

Assumptions

We assume that all possible true segments are represented in each demonstration by at least one transition (some might be false positives). Given the segmentation function $\mathcal{S}(d_i)$, one can define a set of *true* transition states:

$$\Gamma^* = \{o_{i,t} \in d_i : \ \mathcal{S}(d_i)_{t-1} \neq \mathcal{S}(d_i)_t, \ t > 0\}.$$

These satisfy the following property:

$$\Gamma^* \subseteq \Gamma.$$

In other words, we assume that a subset of transition states discovered by the indicator function correspond with the true segment transitions. There can be false positives but no false negatives (a demonstration where a segment transition is missed by the transition indicator function). Since the segmentation function is sequential and in a fixed order, this leads to a model where we are trying to find the $k - 1$ true segment-segment transition points in Γ.

Problem Statement and Method Overview

These definitions allow us to formalize the transition state clustering problem.

Problem 2 (Transition State Clustering) *Given a set of regular demonstrations D and transition identification function $\mathbf{T}$, find a segmentation $\mathbf{S}$.*

Candidate Transitions: To implement $\mathbf{T}$, TSC fits a Gaussian mixture model to sliding windows over each of the demonstration trajectories and identifies consecutive times with different most-likely mixture components.

Algorithm 1 Transition Identification

1: Input: D demonstrations, ℓ a window size, and α a Dirichlet Process prior.
2: For each demonstration, generate a set of sliding windows of $\mathbf{w}_t^{(\ell)} = [\mathbf{x}_{t-\ell}, ..., \mathbf{x}_t]^\intercal$. Let W be the set of all sliding windows across all demonstrations.
3: Fit a mixture model to W assigning each state to its most likely component.
4: Identify times t in each demonstration when $\mathbf{w}_t$ has a different most likely mixture component than $\mathbf{w}_{t+1}$, start and finish times ($t = 0, t = T_i$) are automatically transitions.
5: Return: A set of transition states Γ, the (x, v, t) tuples at which transitions occur.

Transition State Clusters: The states at which those transitions occur are called transition states. TSC uses a GMM to cluster the transition states in terms of spatial and temporal similarity to find $\mathbf{S}$.

Optimizations: To avoid setting the number of clusters at each level of the hierarchy in advance, the number of regions are determined by a Dirichlet Process prior. A series of merging and pruning steps remove sparse clusters and repetitive loops.

Gaussian Mixture Transition Identification

While we can use any transition identification function to get Γ (as long as it satisfies the assumptions), we present one implementation based of Gaussian Mixtures that we used in a number of our experiments. We found that this GMM approach was scalable (in terms of data and dimensionality) and had fewer hyper-parameters to tune than more complex models. Combined with the subsequent hierarchical clustering, this approach proved to be robust in all of our experiments.

Each demonstration trajectory d_i is a trajectory of T_i state-vectors $[x_1, ..., x_{T_i}]$. For a given time t, we can define a window of length ℓ as:

$$\mathbf{w}_t^{(\ell)} = [s_{t-\ell}, ..., s_t]^\intercal$$

We can further normalize this window relative to its first state:

$$\mathbf{n}_t^{(\ell)} = [s_{t-\ell} - s_{t-\ell}, ..., s_t - s_{t-\ell}]^\intercal$$

This represents the "delta" in movement over the time span of a window. Then, for each demonstration trajectory we can also generate a trajectory of $T - \ell$ windowed states:

$$\mathbf{d}^{(\ell)} = [\mathbf{n}_\ell^{(\ell)}, ..., \mathbf{n}_T^{(\ell)}]$$

Over the entire set of windowed demonstrations, we collect a dataset of all of the $\mathbf{n}_t^{(\ell)}$ vectors. We fit a GMM model to these vectors. The GMM model defines m multivariate Gaussian distributions and a probability that each observation $\mathbf{n}_t^{(\ell)}$ is sampled from each

Algorithm 2 Transition State Clustering

Input: Γ Transition States, ρ pruning parameter
Fit a mixture model to the set of transition states Γ in the kinematic states.
Fit a mixture model to the sensory features for transitions within every kinematic cluster i.
Fit a mixture model to the times from every kinematic and sensory cluster pair (i, j).
Remove clusters that contain fewer than transition states from fewer than $\rho \cdot N$ distinct demonstrations.
Output: A set of transitions, which are regions of the state-space and temporal intervals defined by Gaussian confidence intervals.

of the m distributions. We annotate each observation with the most likely mixture component. Times such that $\mathbf{n}_t^{(\ell)}$ and $\mathbf{n}_{t+1}^{(\ell)}$ have different most likely components are marked as transitions. This model captures some dynamical behaviors while not requiring explicit modeling of the state-to-state transition function.

Sometimes the MDP's states are more abstract and do not map to space where the normalized windows make sense. We can still apply the same method when we only have a positive definite kernel function over all pairs of states $\kappa(s_i, s_j)$. We can construct this kernel function for all of the states observed in the demonstrations and apply Kernelized PCA to the features before learning the transitions–a technique used in Computer Vision [84]. The top p' eigenvalues define a new embedded feature vector for each ω in $\mathbb{R}^{p'}$. We can now apply the algorithm above in this embedded feature space.

TSC Inference Algorithm

We present the clustering algorithm which is summarized in Algorithm 2. In a first pass, the transition states are clustered with respect to the kinematic states, then sub-clustered with respect to the sensory states, and then, we temporally sub-cluster. The sub-clusters can be used to formulate the segmentation criteria.

Kinematic Step: We want our model to capture that transitions that occur in similar positions in the state-space across all demonstrations are actual transitions, and we would like to aggregate these transitions into logical events. Hypothetically, if we had infinite demonstrations Γ would define a density of transition events throughout the state-space. The modes of the density, which intuitively represent a propensity of a state x to trigger a segment change, are of key interest to us.

We can think of the set of identified transition states Γ as a sample of this density. We fit a DP-GMM to kinematic features of the transition states. Each transition state will have an assignment probability to one of the mixture components. We convert this to a hard assignment by assigning the transition state to the most likely component.

Sensory Step: Then, we apply the second level of DP-GMM fitting over the sensory features (if available). Within each kinematic cluster, we fit a DP-GMM to find sub-clusters in

the sensory features. Note that the transitions were only identified with kinematic features. This step grounds the detected transitions in sensory clusters.

Temporal Step: Finally, we apply the last level of DP-GMM fitting over the time axis. Without temporal localization, the transitions may be ambiguous. For example, in a figure 8 motion, the robot may pass over a point twice in the same task. Conditioned on the particular state-space cluster assignment, we can fit a DP-GMM each to each subset of times. The final result contains sub-clusters that are indexed both in the state-space and in time.

Enforcing Consistency: The learned clusters will vary in size as some may consist of transitions that appear only in a few demonstrations. The goal of TSC is to identify those clusters that correspond to state and time transition conditions common to all demonstrations of a task. We frame this as a pruning problem, where we want to enforce that all learned clusters contain transitions from a fraction of ρ distinct demonstrations. Clusters whose constituent transition states come from fewer than a fraction ρ demonstrations are *pruned*. ρ should be set based on the expected rarity of outliers. For example, if ρ is 100% then the only mixture components that are found are those with at least one transition state from every demonstration (i.e., the regularity assumption). If ρ is less than 100%, then it means that every mixture component must cover some subset of the demonstrations. In our experiments, we set the parameter ρ to 80% and show the results with and without this step.

Segmentation Criteria: Finally, if there are k remaining clusters $\{C_1, ..., C_k\}$, we can use these clusters to form a criteria for segmentation. Each cluster is formed using a GMM triplet in the kinematic state, visual state, and time. The quantiles of the three GMMs will define an ordered sequence of regions $[\rho_1, ..., \rho_k]$ over the state-space and each of these regions has an associated time interval defined by the Gaussian confidence interval for some confidence level z_α.

TSC Simulated Experimental Evaluation

We evaluate TSC's robustness in the following way:

1. *Precision.* Results suggest TSC significantly reduces the number of false positive segments in simulated examples with noise.

2. *Recall.* Among algorithms that use piecewise linear segment models, results suggest TSC recovers segments of a generated piecewise linear trajectory more consistently in the presence of process and observation noise.

3. *Applicability to Real-World Data.* Result suggest that TSC recovers qualitatively relevant segments in real surgical trajectory data.

Precision in Synthetic Examples

Our first experiment evaluates the following hypothesis: TSC significantly reduces the number of false positive segments in a simple simulated example with noise. These experiments evaluate TSC against algorithms with a single level of clustering.

Comparison of 7 alternative segmentation criteria:

1. *Zero-Velocity Crossing (VEL):* This algorithm detects a change in the sign of the velocity.

2. *Smoothed Zero-Velocity Crossing (VELS):* This algorithm applies a low-pass filter (exponentially weighted moving average) to the trajectory, and then detects a change in the sign of the velocity.

3. *Acceleration (ACC):* This algorithm detects any change in the velocity by looking for non-zero acceleration.

4. *Gaussian Mixture Model (GMM):* This algorithm applies a GMM model to the observed states and detects changes in most likely assignment. The number of clusters were set to 2.

5. *Windowed Gaussian Mixture Model (GMMW):* This algorithm is the first phase of TSC. It applies a GMM to windows of size 2, and detects changes in most likely assignment. The number of clusters was set to 2, unlike in TSC where we use the DP to set the number of clusters.

6. *Auto-Regressive Mixture (AR):* This model fits a piecewise linear transition law to the observed data.

7. *Coresets (CORE):* We evaluate against a standard coreset model [120, 130], and the particular variant is implemented with weighted k-means. We applied this to the same augmented state-vector as in the previously mentioned GMM.

8. *TSC:* Our proposed approach with a pruning threshold of 0.8 and no loop compaction.

Bouncing Ball: We first revisit the example in the introduction of the bouncing ball, which can be modeled as the following 1D double-integrator system:

$$\ddot{x} = [-9.8]\prime\prime^2$$

This system is observed with additive Gaussian white noise with std. 10 (Moderate Noise):

$$y = x + N(0, 10)$$

and std. 20 (High Noise):

$$y = x + N(0, 20)$$

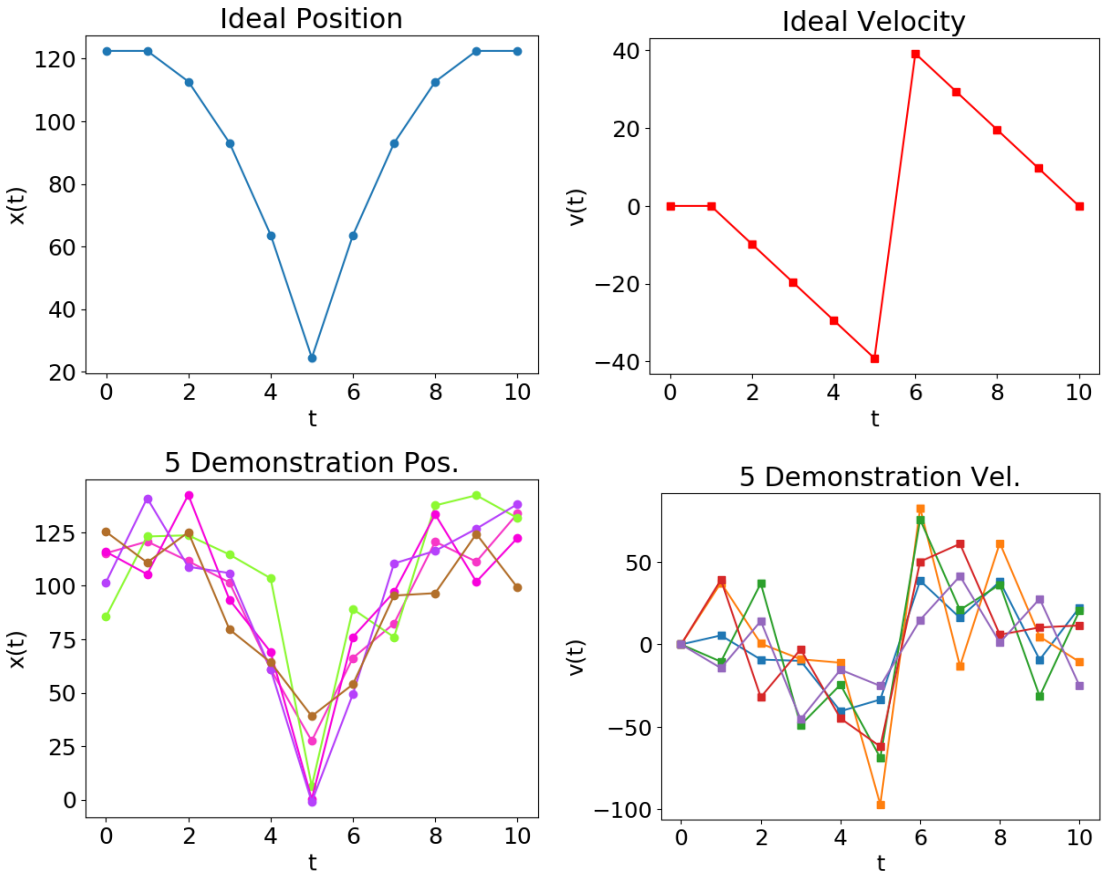


Figure 3.1: (Above) The position and velocity of the bouncing ball without noise. (Below) 5 trajectories of the ball with different realizations of the noise.

The system is initialized at $x_0 = 122.5$ and bounces when $x = 20$, at which point the velocity is negated. Figure 3.1 illustrates the ideal trajectory and noisy realizations of these trajectories.

We apply the segmentation algorithms to the trajectories and plot the results in Figure 3.2. When there is no noise, all of the algorithms are equally precise, and there is no trouble corresponding segments across demonstrations. All of the "rate-of-change" methods (VEL, VELS, ACC) reliably identify the point where the ball bounces. The GMM and the Coreset methods do not segment the trajectory at the bounce point. On the other hand, the windowed GMM takes two consecutive positions and velocities into account during the clustering. Similarly, the autoregressive model can accurately identify the bounce point. With no noise, TSC has little difference with the windowed GMM.

Differences arise when we observe the trajectory with additive Gaussian noise. The "rate-of-change" methods have some spurious segmentation points due to noise. The GMM based methods are more robust to this noise, and they retain similar precision. This motivates our choice of the first phase of the TSC algorithm using a windowed GMM approach.

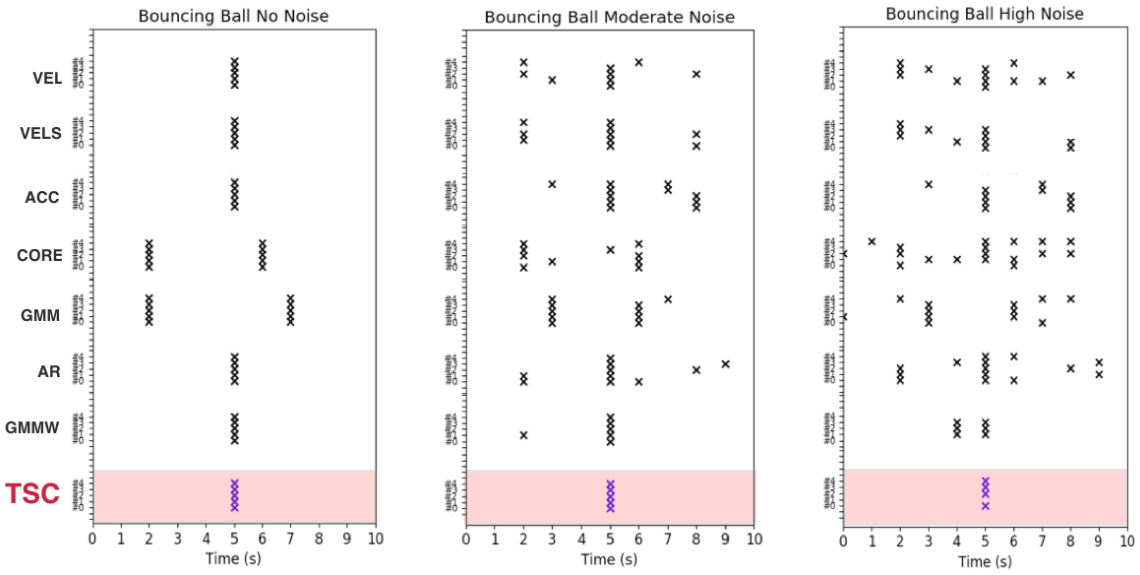


Figure 3.2: Plots the identified transitions with each segmentation algorithm with and without noise. While all techniques are precise when there is no noise, TSC is the most robust in the presence of noise.

However, the GMM approaches still have some spurious transitions. With these spurious points, it becomes challenging to reliably correspond trajectories across segments. So, TSC applies a second phase of clustering to correspond the transitions and prune the sparse clusters. This results in accurate segmentation even in the presence of noise.

As the noise increases, TSC is still able to find accurate segments. In the high noise case, the bounce point is still identified in 4 out of 5 trajectories. It is important to note that we do not claim that one segmentation algorithm is more accurate than another, or that TSC more accurately reflects "real" segments. These results only suggest that TSC is more precise than alternatives; that is, given the assumptions in TSC it consistently recovers segments according to those assumptions. The next experiments will study the recall characteristics.

Bouncing Ball with Air Resistance: In the first set of experiments, we illustrate TSC's robustness to variance in the state-space. Next, we illustrate how TSC can still correspond segments with temporal variation. Consider the dynamics of the bouncing ball with an term to account for air resistance:

$$\ddot{x} = [-9.8]\prime\prime^2 + K_v\dot{x}$$

We draw the air-resistance constant K_v uniformly from $K_v \sim U[1, 5]$. The consequence is that that the ball will bounce at different times in different trajectories.

Figure 3.3 illustrates the results. In the 5 trajectories, the ball bounces between time-step 5 and 7. With no noise VEL, VELS, ACC, GMMW, and TSC can identify the bounce

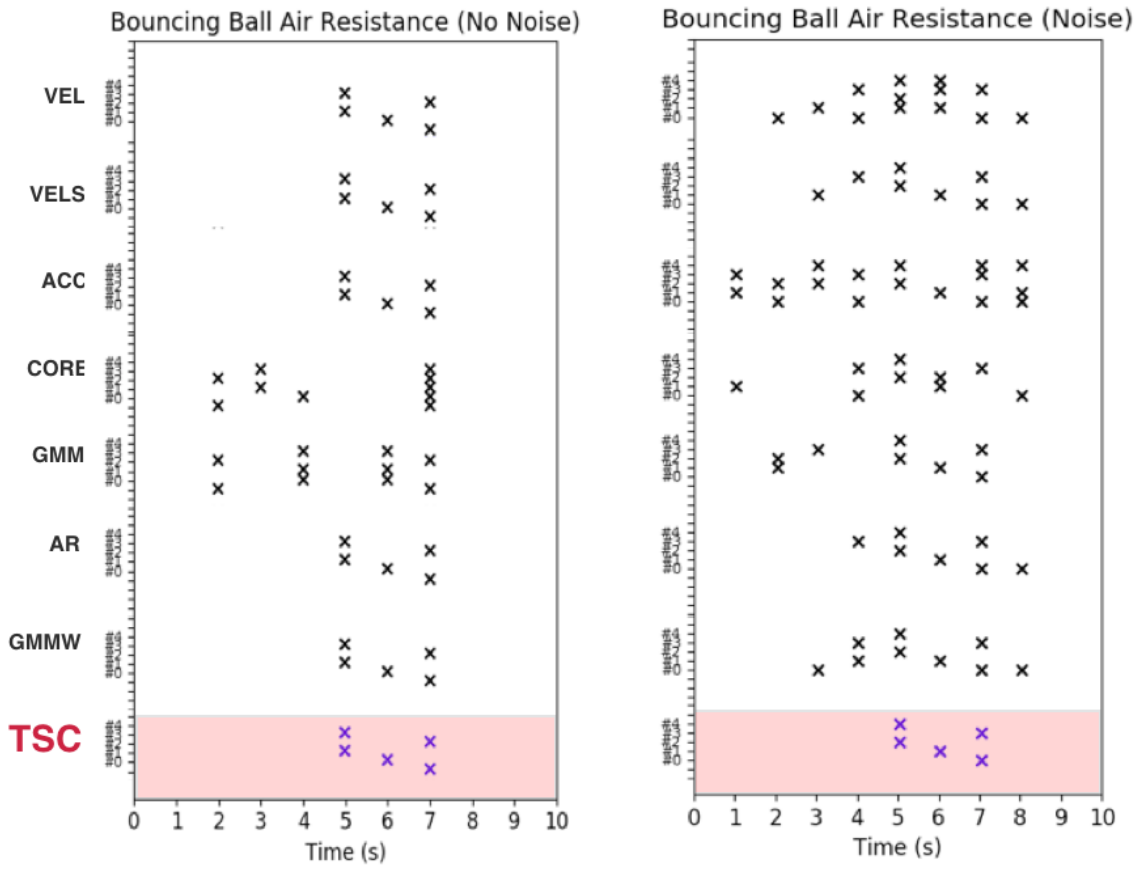


Figure 3.3: Plots the identified transitions with each segmentation algorithm with and without noise. In this example, temporal variation is added by incorporating a random "air resistance" factor. TSC is consistent even in the presence of this temporal variation.

point. Then, the system is observed with additive Gaussian white noise with std. 10:

$$y = x + N(0, 10)$$

We find that TSC recovers a consistent set of segments even with the temporal variation.

Hybrid Approaches: In the previous experiments, we presented TSC using a windowed GMM approach to identify transitions. Next, we consider TSC with alternative transition identification functions. Consider a "Figure 8" trajectory defined parametrically as:

$$x = \mathbf{cos}(t)$$

$$y = 0.5\mathbf{sin}(2t)$$

The trajectory is visualized in Figure 3.4. The trajectory starts at the far right and progresses until it returns to the same spot. Velocity based segmentation finds one transition point where there is a change in direction (far left of the trajectory) (Figure 3.5). A windowed

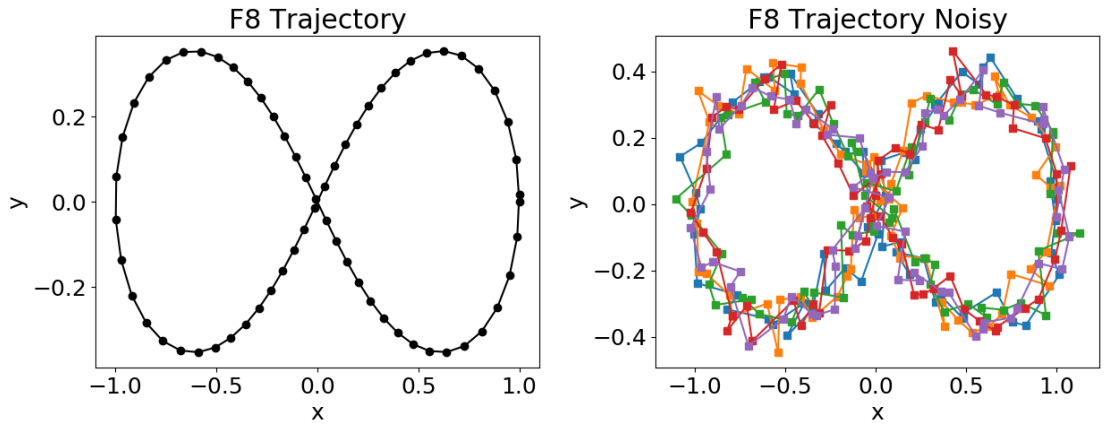


Figure 3.4: A "figure 8" trajectory in the plane, and 5 noisy demonstrations. The trajectory starts at the far right and progresses until it returns to the same spot.

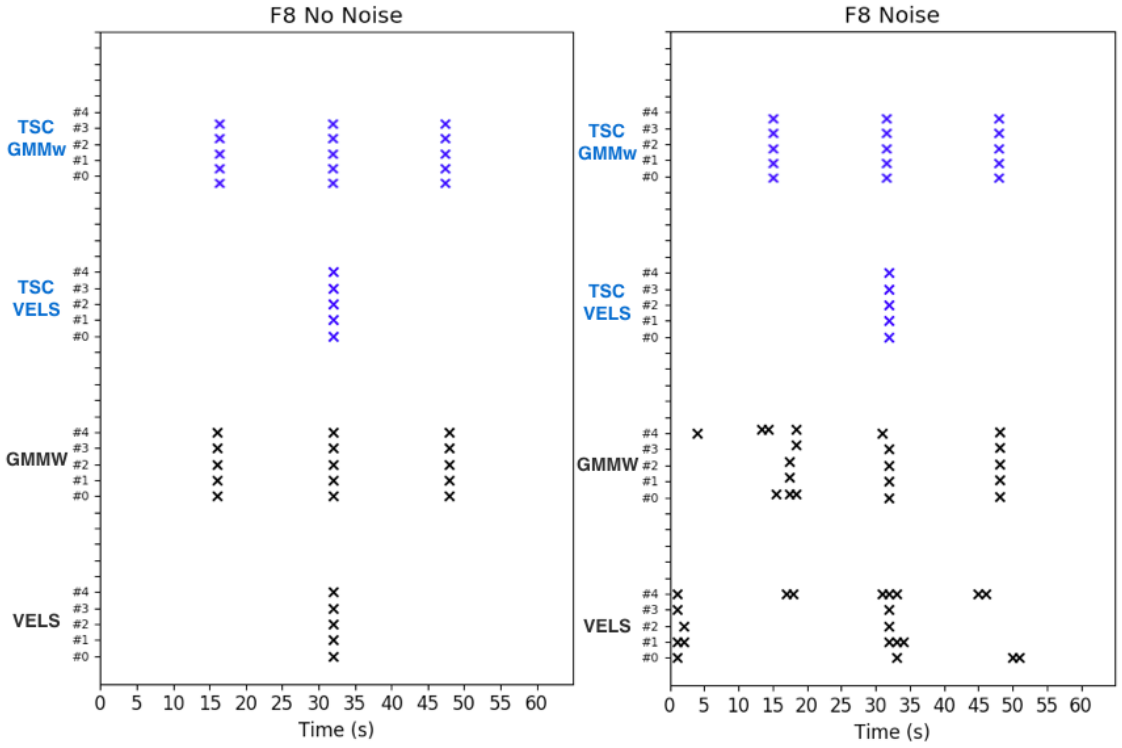


Figure 3.5: Plots the identified transitions with each segmentation algorithm with and without noise. Velocity based segmentation finds one transition point where there is a change in direction. A windowed GMM where the number of clusters is set by a DP finds three transition points. TSC can improve the precision of both techniques.

GMM where the number of clusters is set by a DP finds three transition points. These three points correspond to the far left point as well as the crossing point in the figure 8 (happens twice). These are two different segmentation criteria, and both are reasonable with respect to their respective assumptions.

Next, this parametric trajectory is observed with additive Gaussian noise of std. 0.1 (Figure 3.4). We see that both the GMM approach and the velocity approach have several spurious transitions (Figure 3.5). TSC can improve the precision of both techniques by adding a layer of clustering.

Rotations

Handling orientations is a challenging problem due to the topology of $SO(3)$ [126]. As an example of what can go wrong consider a 2D square rotating in the plane. We construct a 1x1 meter 2D square and track a point on the corner of the 2D square. The 2D square rotates clockwise in $\frac{\pi}{10}$ radian/s for 10 time-steps, then switches, and rotates the other direction at the same angular speed. The state of the system is the (x, y) position of the corner. We add .1 meter standard deviation Gaussian observation noise to the observed trajectories.

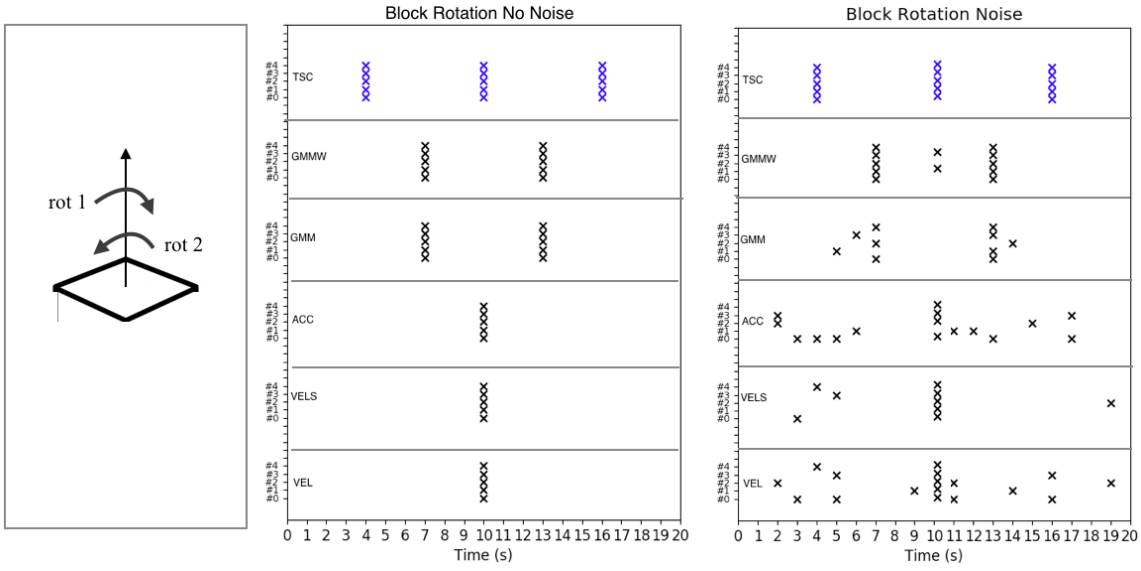


Figure 3.6: Plots the identified transitions with each segmentation algorithm with and without noise. While all techniques are precise when there is no noise, TSC is the most robust in the presence of noise but finds additional segments.

We apply the segmentation algorithms to 5 trajectories and plot the results in Figure 3.6. As before, with no noise, all of the techniques are equally precise. In this example, there is a difference between how the different techniques segment the trajectories. The rate-of-change methods segment the trajectory at the point when the block changes rotation direction. The GMM and the windowed GMM approaches cuts the trajectory into 3 even

segments–missing the direction change. TSC cuts the trajectory into 4 segments including the direction change. TSC differs from the windowed GMM because it sets the number of clusters using the Dirichlet Process prior. With noise, the rate-of-change techniques have a number of spurious segments. The GMM-based approaches are more robust and TSC improves the windowed GMM even further by clustering the detected transitions. However, if the initial transitions were found in angular space, then TSC would have found one segment. In this sense, the definition of the state-space changes the segments found. We hope to explore these issues in more detail in future work.

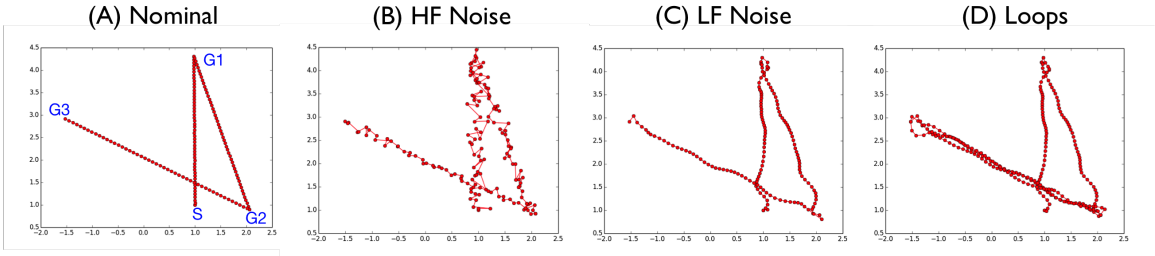


Figure 3.7: One of 20 instances with random goal points $G1$, $G2$, $G3$. (a) Observations from a simulated demonstration with three regimes, (b) Observations corrupted with Gaussian white sensor noise, (c) Observations corrupted with low frequency process noise, and (d) Observations corrupted with an inserted loop. See Figure 3.11 for evaluation on loops.

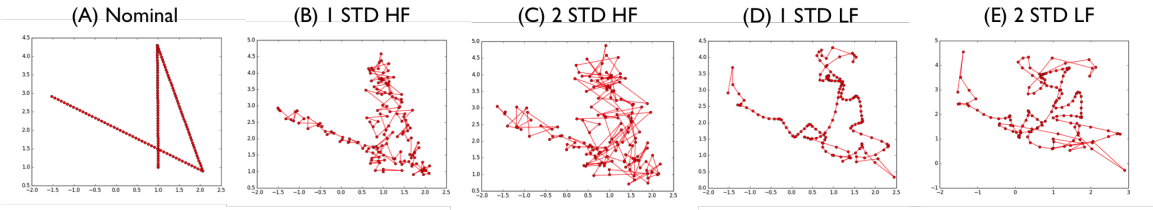


Figure 3.8: (a) Nominal trajectory, (b) 1 std. of high frequency observation noise, (c) 2 std. of high frequency observation noise, (d) 1 std. of low frequency process noise, and (e) 2 std. of low frequency process noise.

Recall in Synthetic Examples

Comparing different segmentation models can be challenging due to differing segmentation criteria. However, we identified some algorithms that identify locally linear or near linear segments. We developed a synthetic dataset generator to generate piecewise linear segments and compared the algorithms on the generated dataset. Note, we do not intend this to be a comprehensive evaluation of the accuracy of the different techniques, but more a characterization of the approaches on a locally linear example to study the key tradeoffs.

We model the trajectory of a point robot with two-dimensional position state (x, y) between k goal points $\{g_1, ..., g_k\}$. We apply position control to guide the robot to the targets and without disturbance, this motion is linear (Figure 3.7a). We add various types of disturbances (and in varying amounts) including Gaussian observation noise, low-frequency process noise, and repetitive loops (Figure 3.7b-d). We report noise values in terms of standard deviations. Figure 3.8 illustrates the relative magnitudes. A demonstration d_i is a sample from the following system.

Task: Every segmentation algorithm will be evaluated in its ability to identify the $k-1$ segments (i.e., the paths between the goal points). Furthermore, we evaluate algorithms on random instances of this task. In the beginning, we select 3 random goal points. From a fixed initial position, we control the simulated point robot to the goal points with position control. Without any disturbance, this follows a linear motion. For a given noise setting, we sample demonstrations from this system and apply/evaluate each algorithm. We present results aggregated over 20 such random instances. This is important since many of the segmentation algorithms proposed in literature have some crucial hyper-parameters, and we present results with a *single* choice of parameters averaged over multiple tasks. This way, the hyper-parameter tuning cannot overfit to any given instance of the problem and has to be valid for the entire class of tasks. We believe that this is important since tuning these hyper-parameters in practice (i.e., not in simulation) is challenging since there is no ground truth. The experimental code is available at: http://berkeleyautomation.github.io/tsc/.

5 Algorithms: We compare TSC against alternatives where the authors explicitly find (or approximately find) locally linear segments. It is important to reiterate that different segmentation techniques optimize different objectives, and this benchmark is meant to characterize the performance on a common task. All of the techniques are based on Gaussian Distributions or Linear auto-regressive models.

1. *(GMM)* (Same as previous experiment). In this experiment, we set the parameter to the optimal choice of 3 without automatic tuning.

2. *(GMM+HMM)* A natural extension to this model is to enforce a transition structure on the regimes with a latent Markov Chain [4, 21, 71, 127]. We use the same state vector as above, without time augmentation as this is handled by the HMM. We fit the model using the forward-backward algorithm.

3. *Coresets* (Same as previous experiment).

4. *HSMM* We evaluated a Gaussian Hidden Semi-Markov Model. We directly applied this model to the demonstrations with no augmentation or normalization of features. This was implemented with the package pyhsmm. We directly applied this model to the demonstrations with no augmentation as in the GMM approaches. We ran our MCMC sampler for 10000 iterations, discarding the first 2500 as burn-in and thinning the chain by 15.

5. *AR-HMM* We evaluated a Bayesian Autoregressive HMM model as used in [89]. This was implemented with the packages pybasicbayes and pyhsmm-ar. The autoregressive order was 10 and we ran our MCMC sampler for 10000 iterations, discarding the first 2500 as burn-in and thinning the chain by 15.

Evaluation Metric: There is considerable debate on metrics to evaluate the accuracy of unsupervised segmentation and activity recognition techniques, e.g. frame accuracy wu2015watch, hamming distance fox2009sharing. Typically, these metrics have two steps: (1) segments to ground truth correspondence, and (2) then measuring the similarity between corresponded segments. We have made this feature extensible and evaluated some different accuracy metrics (Jaccard Similarity, Frame Accuracy, Segment Accuracy, Intersection over Union). We found that the following procedure led to the most insightful results–differentiating the different techniques.

In the first phase, we match segments in our predicted sequence to those in the ground truth. We do this with a procedure identical to the one proposed in [132]. We define a bi-partite graph of predicted segments to ground truth segments, and add weighted edges where weights represent the overlap between a predicted segment and a ground-truth segment (i.e., the recall over time-steps). Each predicted segment is matched to its highest weighted ground truth segment. Each predicted segment is assigned to exactly one ground-truth segment, while a ground-truth segment may have none, one, or more corresponding predictions.

After establishing the correspondence between predictions and ground truth, we consider a true positive (a ground-truth segment is correctly identified) if the overlap (intersection-over-union) between the ground-truth segment and its corresponding predicted segments is more than a default threshold 60%. Then, we compute Segment Accuracy as the ratio of the ground-truth segments that are correctly detected. In [132], the authors use a 40% threshold but apply the metric to real data. Since this is a synthetic example, we increase this threshold to 60%, which we empirically found accounted for boundary effects especially in the Bayesian approaches (i.e., repeated transitions around segment endpoints).

Accuracy vs. Noise

In our first experiment, we measured the segment accuracy for each of the algorithms for 50 demonstrations. We also varied the amount of process and observation noise in the system. As Figure 3.8 illustrates, this is a very significant amount of noise in the data, and successful techniques must exploit the structure in multiple demonstrations. Figure 3.9a illustrates the performance of each of the techniques as a function of high-frequency observation noise. Results suggest that TSC is more robust to noise than the alternatives (nearly 20% more accurate for 2.5 std of noise). The Bayesian ARHMM approach is nearly identical to TSC when the noise is low but quickly loses accuracy as more noise is added. We attribute this robustness to the TSC's pruning step which ensures that only transition state clusters with sufficient coverage across all demonstrations are kept. These results

are even more pronounced for low-frequency process noise (Figure 3.9b). TSC is 49% more accurate than all competitors for 2.5 std of noise added. We find that the Bayesian approaches are particularly susceptible to such noise. Furthermore, Figure 3.9c shows TSC requires no more data than the alternatives to achieve such robustness. Another point to note is that TSC is solved much more efficiently than ARHMM or HSMM which require expensive MCMC samples. While parameter inference on these models can be solved more efficiently (but approximately) with Mean-Field Stochastic Variational Inference, we found that the results were not as accurate. TSC is about 6x slower than using Coresets or the direct GMM approach, but it is over 100x faster than the MCMC for the ARHMM model. Figure 3.10 compares the runtime of each of the algorithms as a function of the number of demonstrations.

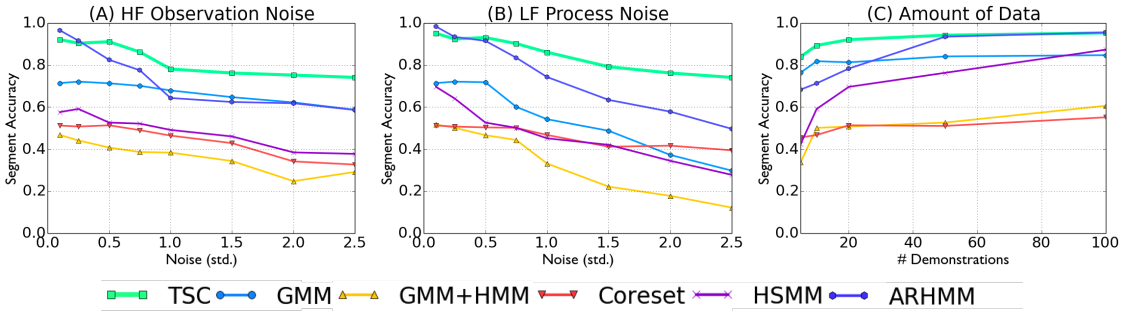


Figure 3.9: Each data point represents 20 random instances of a 3-segment problem with varying levels of high-frequency noise, low-frequency noise, and demonstrations. We measure the segmentation accuracy for the compared approaches. (A) TSC finds a more a accurate segmentation than all of the alternatives even under significant high-frequency observation noise, (B) TSC is more robust to low-frequency process noise than the alternatives, (C) the Bayesian techniques solved with MCMC (ARHMM, HSMM) are more sensitive to the number of demonstrations provided than the others.

TSC Hyper-Parameters: Next, we explored the dependence of the performance on the hyper-parameters for TSC. We focus on the window size and the pruning parameter. Figure 3.11a shows how varying the window size affects the performance curves. Larger window sizes can reject more low-frequency process noise. However, larger windows are also less efficient when the noise is low. Similarly, Figure 3.11b shows how increasing the pruning parameter affects the robustness to high-frequency observation noise. However, a larger pruning parameter is less efficient at low noise levels. Based on these curves, we selected $(w = 3, \rho = 0.3)$ in our synthetic experiments.

Loops: Finally, we evaluated 4 algorithms on how well they can detect and adjust for loops. TSC compacts adjacent motions that are overly similar, while HMM-based approaches correspond similar looking motions. An HMM grammar over segments is clearly more expressive than TSC's, and we explore whether it is necessary to learn a full transition structure to compensate for loops. We compare the accuracy of the different segmentation techniques in detecting that a loop is present (Figure 3.12). Figure 3.12a shows that TSC

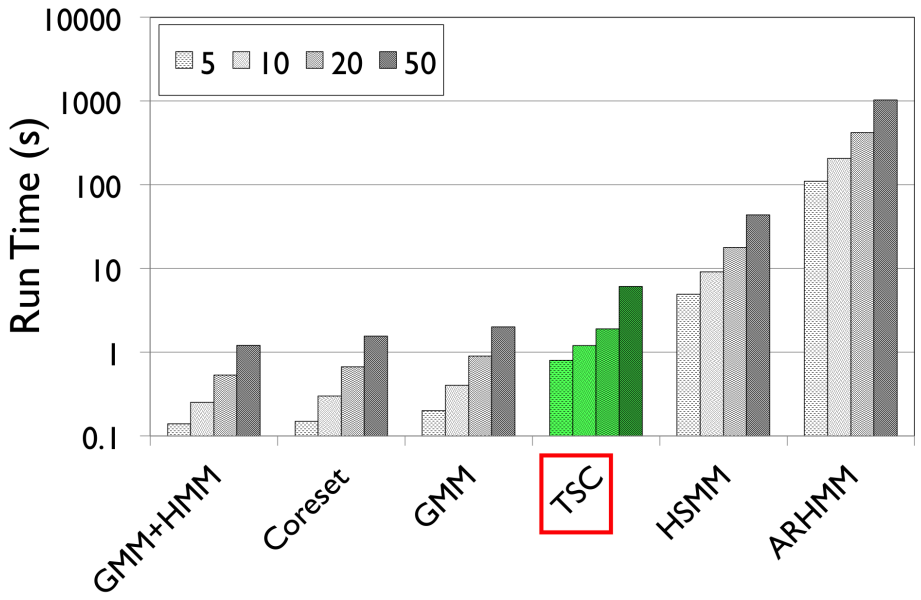


Figure 3.10: TSC is about 6x slower than using Coresets or the direct GMM approach, but it is over 100x faster than the MCMC for the ARHMM model.

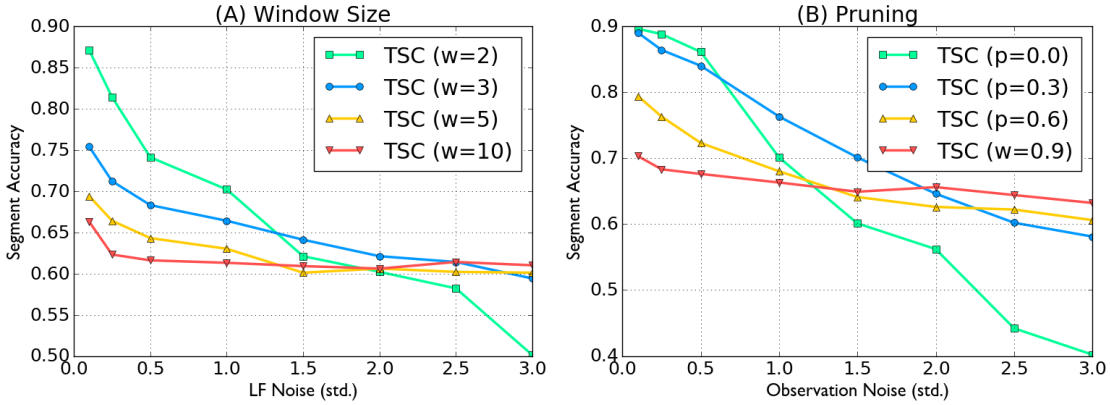


Figure 3.11: (A) shows the performance curves of different choices of windows as a function of the process noise. Larger windows can reject higher amounts of process noise but are less efficient at low noise levels. (B) the performance curves of different choices of the pruning threshold. Larger pruning thresholds are more robust to high amounts of observation noise but less accurate in the low noise setting. We selected $(w = 3, \rho = 0.3)$ in our synthetic experiments.

is competitive with the HMM approaches as we vary the observation noise; however, the results suggest that ARHMM provides the most accurate loop detection. On the other hand, Figure 3.12b suggests that process noise has a very different effect.

TSC is actually more accurate than the HMM approaches when the process noise is

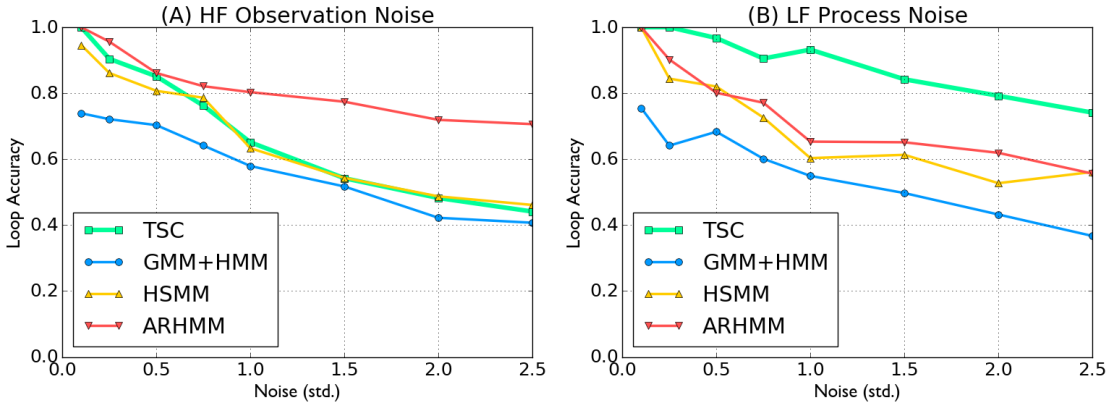


Figure 3.12: (A) illustrates the accuracy of TSC's compaction step as a function of observation noise. TSC is competitive with the HMM-based approaches without having to model the full transition matrix. (B) TSC is actually more robust to low-frequency process noise in the loops than the HMM-based approaches.

high–even without learning a transition structure.

Scaling with Dimensionality: We investigate how the accuracy of TSC scales with the dimensionality of the state-space. As in the previous experiments, we measured the segment accuracy for each of the algorithms for 50 demonstrations. This time we generated the line segments in increasingly higher dimensional spaces (from 2-D to 35-D). The noise added to the trajectories has a std of 0.1. Figure 3.13a plots the segment accuracy as a function of the dimensionality of the state-space. While the accuracy of TSC does decreases as the dimensionality increases it is more robust than some of the alternatives: ARHMM and HSMM. One possible explanation is that both of those techniques rely on Gibbs Sampling for inference, which is a little more sensitive to dimensionality than the expectation-maximization inference procedure used in GMM and GMM+HMM. Figure 3.13b shows one aspect of TSC that is more sensitive to the dimensionality. The loop compaction step requires a dynamic time-warping and then a comparison to fuse repeated segments together. This step is not as robust in higher dimensional state-spaces. This is possibly due to the use of the L_2 distance metric to compare partial trajectories to compact. TSC runs in 4 seconds on the 2-D case, 16 seconds on the 10-D case, and in 59 seconds on the 35-D case.

Surgical Data Experiments

We describe the three tasks used in our evaluation and the corresponding manual segmentation (Figure 3.14). This will serve as ground truth when qualitatively evaluating our segmentation on real data. This set of experiments primarily evaluates the utility of segments learned by TSC. Data was collected before hand as a part of prior work. Our hypothesis is that even though TSC is unsupervised, it identifies segments that often align with man-

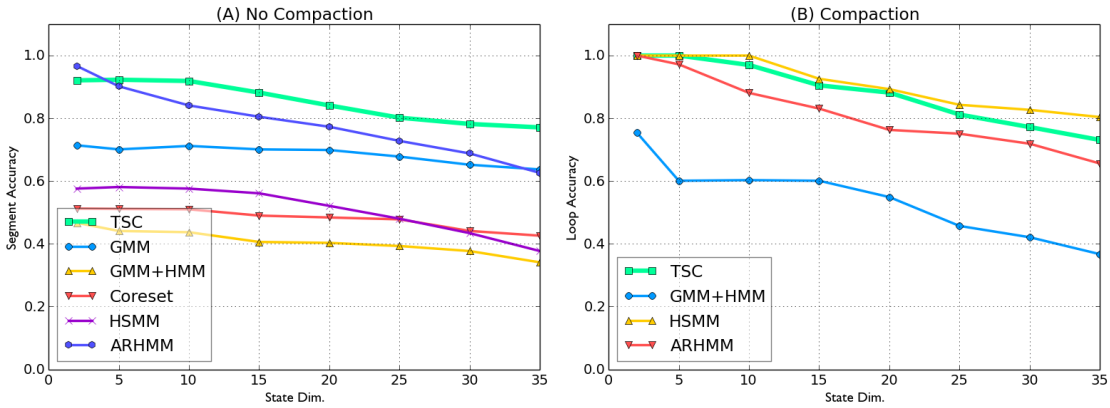


Figure 3.13: We investigate how the accuracy of TSC scales with the dimensionality of the state-space. In (A) we consider the problem with no loops or compaction, and in (B) we measure the accuracy of the compaction step as a function of dimensionality.

ual annotations. In all of our experiments, the pruning parameter ρ is set to 80% and the compaction heuristic δ is to 1cm.

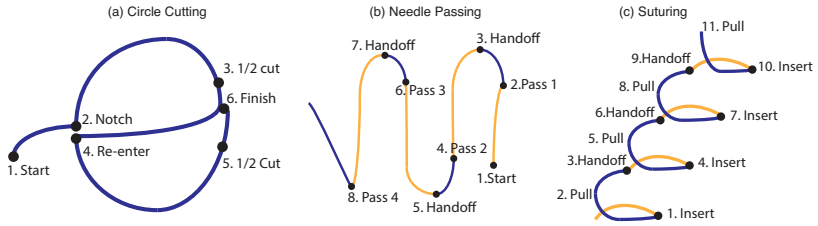


Figure 3.14: Hand annotations of the three tasks: (a) circle cutting, (b) needle passing, and (c) suturing. Right arm actions are listed in dark blue and left arm actions are listed in yellow.

The state-space is the 6D end-effector position. In some experiments, we augment this state-space with the following visual features:

1. *Grasp*. 0 if empty, 1 otherwise.

2. *Needle Penetration*. We use an estimate of the penetration depth based on the robot kinematics to encode this feature. If there is no penetration (as detected by video), the value is 0, otherwise the value of penetration is the robot's z position.

Our goal with these features was to illustrate that TSC applies to general state-spaces as well as spatial ones, and not to address the perception problem. These features were constructed via manual annotation, where the Grasp and Needle Penetration were identified by reviewing the videos and marking the frames at which they occurred.

Circle Cutting: A 5 cm diameter circle drawn on a piece of gauze. The first step is to cut a notch into the circle. The second step is to cut clockwise half-way around the circle. Next, the robot transitions to the other side cutting counter clockwise. Finally, the robot finishes the cut at the meeting point of the two cuts. As the left arm's only action is to maintain the gauze in tension, we exclude it from the analysis. In Figure 3.14a, we mark 6 manually identified transitions points for this task from [87]: (1) start, (2) notch, (3) finish 1st cut, (4) cross-over, (5) finish 2nd cut, and (6) connect the two cuts. For the circle cutting task, we collected 10 demonstrations by researchers who were not surgeons but familiar with operating the da Vinci Research Kit (dVRK).

We also perform experiments using the JIGSAWS dataset [39] consisting of surgical activity for human motion modeling. The dataset was captured using the da Vinci Surgical System from eight surgeons with different levels of skill performing five repetitions each of Needle Passing and Suturing.

Needle Passing: We applied TSC to 28 demonstrations of the needle passing task. The robot passes a needle through a hoop using its right arm, then its left arm to pull the needle through the hoop. Then, the robot hands the needle off from the left arm to the right arm. This procedure is repeated four times as illustrated with a manual segmentation in Figure 3.14b.

Suturing: Next, we explored 39 examples of a 4 throw suturing task (Figure 3.14c). Using the right arm, the first step is to penetrate one of the points on right side. The next step is to force the needle through the phantom to the other side. Using the left arm, the robot pulls the needle out of the phantom and then the robot hands it off to the right arm for the next point.

Results

Circle Cutting: Figure 3.15a shows the transition states obtained from our algorithm. And Figure 3.15b shows the TSC clusters learned (numbered by time interval midpoint). The algorithm found 8 clusters, one of which was pruned using our $\rho = 80\%$ threshold rule.

The remaining 7 clusters correspond well to the manually identified transition points. It is worth noting that there is one extra cluster (marked $2'$), that does not correspond to a transition in the manual segmentation. At $2'$, the operator finishes a notch and begins to cut. While at a logical level notching and cutting are both penetration actions, they correspond to two different linear transition regimes due to the positioning of the end-effector. Thus, TSC separates them into different clusters even though the human annotators did not. This illustrates why supervised segmentation is challenging. Human annotators segment trajectories on boundaries that are hard to characterize mathematically, e.g., is frame 34 or frame 37 the segment boundary. Supervisors may miss crucial motions that are useful for automation or learning.

Needle Passing: In Figure 3.16a, we plot the transition states in (x, y, z) end-effector space for both arms. We find that these transition states correspond well to the logical

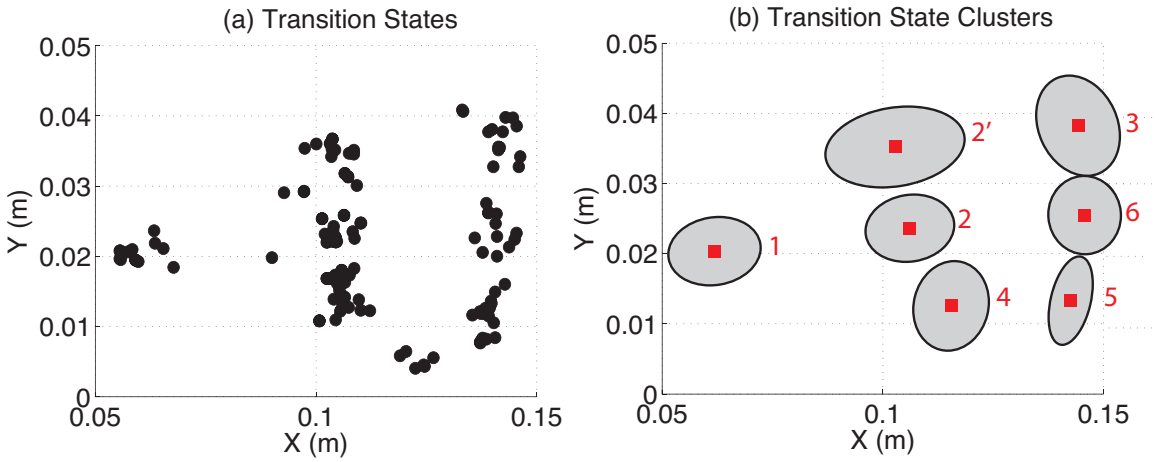


Figure 3.15: (a) The transition states for the circle cutting task are marked in black. (b) The TSC clusters, which are clusters of the transition states, are illustrated with their 75% confidence ellipsoid.

segments of the task (Figure 3.14b). These demonstrations are noisier than the circle cutting demonstrations, and there are more outliers. The subsequent clustering finds 9 clusters (2 pruned). Next, Figures 3.16b-c illustrate the TSC clusters. We find that again TSC learns a small parametrization for the task structure with the clusters corresponding well to the manual segments. However, in this case, the noise does lead to a spurious cluster (4 marked in green). One possible explanation is that the demonstrations contain many adjustments to avoid colliding with the needle hoop and the other arm while passing the needle through leading to numerous transition states in that location.

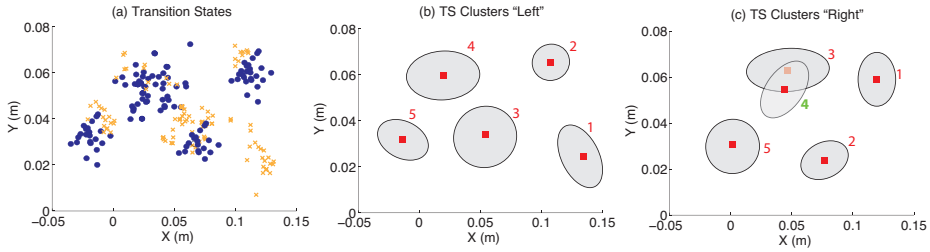


Figure 3.16: (a) The transition states for the task are marked in orange (left arm) and blue (right arm). (b-c) The TSC clusters, which are clusters of the transition states, are illustrated with their 75% confidence ellipsoid for both arms

Suturing: In Figure 3.17, we show the transition states and clusters for the suturing task. As before, we mark the left arm in orange and the right arm in blue. This task was far more challenging than the previous tasks as the demonstrations were inconsistent. These inconsistencies were in the way the suture is pulled after insertion (some pull to the left, some to the right, etc.), leading to transition states all over the state space. Furthermore, there were numerous demonstrations with looping behaviors for the left arm. In fact, the DP-GMM

Table 3.1: This table compares transitions learned by TSC and transitions identified by manual annotators in the JIGSAWS dataset. We found that the transitions mostly aligned. 83% and 73% of transition clusters for needle passing and suturing respectively contained exactly one surgeme transition when both kinematics and vision were used. Results suggest that the hierarchical clustering is more suited for mixed video and kinematic feature spaces.

	No. of Surgeme Segments	No. of Clusters	seg-surgeme	surgeme-seg
Needle Passing TSC(Kin+Video)	14.4 ± 2.57	11	83%	74%
Needle Passing TSC(Video)	14.4 ± 2.57	7	62%	69%
Needle Passing TSC(Kin)	14.4 ± 2.57	16	87%	62%
Needle Passing TSC(VELS)	14.4 ± 2.57	13	71%	70%
Needle Passing TSC(No-H)	14.4 ± 2.57	5	28%	34%
Suturing TSC(Kin+Video)	15.9 ± 3.11	13	73%	66%
Suturing TSC(Video)	15.9 ± 3.11	4	21%	39%
Suturing TSC(Kin)	15.9 ± 3.11	13	68%	61%
Suturing TSC(VELS)	15.9 ± 3.11	17	48%	57%
Suturing TSC(No-H)	15.9 ± 3.11	9	51%	52%

method gives us 23 clusters, 11 of which represent less than 80% of the demonstrations and thus are pruned (we illustrate the effect of the pruning in the next section). In the early stages of the task, the clusters clearly correspond to the manually segmented transitions. As the task progresses, we see that some of the later clusters do not.

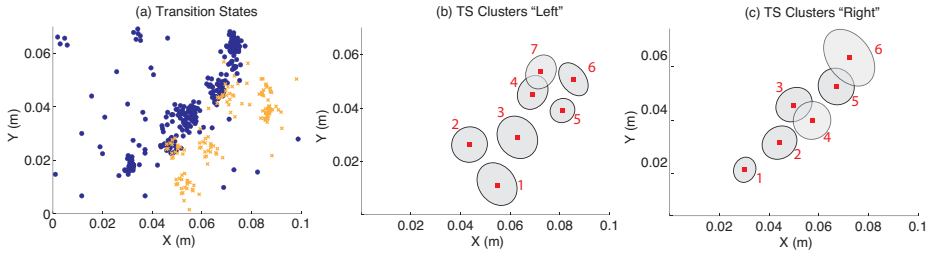


Figure 3.17: (a) The transition states for the task are marked in orange (left arm) and blue (right arm). (b-c) The clusters, which are clusters of the transition states, are illustrated with their 75% confidence ellipsoid for both arms

Comparison to Surgemes

Surgical demonstrations have an established set of primitives called surgemes, and we evaluate if segments discovered by our approach correspond to surgemes. In Table 3.1, we compare the number of TSC segments for needle passing and suturing to the number of annotated surgeme segments. We apply different variants of the TSC algorithm and evaluate its ability to recover segments similar to surgemes. We consider: (Kin+Video) which is the full TSC algorithm, (Kin) which only uses kinematics, (Video) which only uses the visual annotations, (VELS) which uses the zero-crossing velocity heuristic to get the initial transitions, and (NO-H) which treats all of the variables as one big feature space and does not hierarchically cluster. A key difference between our segmentation and number of annotated surgemes is our compaction and pruning steps. To account for this, we first select

a set of surgemes that are expressed in most demonstrations (i.e., simulating pruning), and we also apply a compaction step to the surgeme segments. When surgemes appear consecutively, we only keep the one instance of each. We explore two metrics: **seg-surgeme** the fraction of TSC clusters with only one surgeme switch (averaged over all demonstrations), and **surgeme-seg** the fraction of surgeme switches that fall inside exactly one TSC cluster.

We found that the transitions learned by TSC with both the kinematic and video features were the most aligned with the surgemes. 83% and 73% of transition clusters for needle passing and suturing respectively contained exactly one surgeme transition when both were used. For the needle passing task, we found that the video features alone could give a reasonably accurate segmentation. However, this did not hold for the suturing dataset. The manual video features are low dimensional and tend to under-segment. For the suturing dataset, a combination of the visual and kinematic features was most aligned with the surgemes. Similarly, this scaling problem affects the variant that does not hierarchically cluster–leading to a small number of clusters–and inaccuracy.

Pruning and Compaction

In Figure 3.18, we highlight the benefit of pruning and compaction using the Suturing task as exemplar. First, we show the transition states without applying the compaction step to remove looping transition states (Figure 3.18a). We find that there are many more transition states at the “insert" step of the task. Compaction removes the segments that correspond to a loop of the insertions. Next, we show all of the clusters found by the first step of segmentation. The centroids of these clusters are marked in Figure 3.18b. In all, 11 clusters are pruned by this rule.

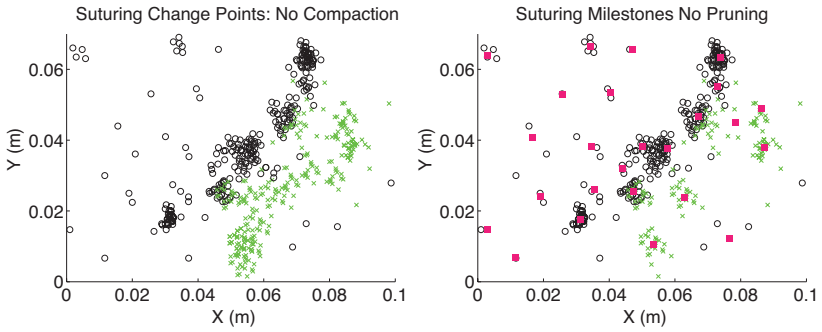


Figure 3.18: We first show the transition states without compaction (in black and green), and then show the clusters without pruning (in red). Compaction sparsifies the transition states and pruning significantly reduces the number of clusters.

TSC with Deep Features

Inspired by the recent success of deep neural networks in reinforcement learning [75, 76], we then explored how visual features extracted from Convolutional Neural Networks can

be used for transition state identification.

We use layers from a pre-trained Convolutional Neural Network (CNNs) to derive the features frame-by-frame. In particular, we explore two architectures designed for image classification task on natural images: (a) **AlexNet:** Krizhevsky et al. proposed multilayer (5 in all) a CNN architecture [70], and (b) **VGG:** Simoyan et al. proposed an alternative architecture termed VGG (acronym for Visual Geometry Group) which increased the number of convolutional layers significantly (16 in all) [111]. In our experiments, we explore the level of generality of features required for segmentation. We also compare these features to other visual featurization techniques such as SIFT for the purpose of task segmentation using TSC.

Evaluation of Visual Featurization

In our first experiment, we explore different visual featurization, encoding, and dimensionality reduction techniques. We applied TSC to our suturing experimental dataset and measured the silhouette score of the resulting transition state clusters. On this dataset, our results suggest that features extracted from the pre-trained CNNs resulted in tighter transition state clusters compared to SIFT features with a 3% lower ss than the worst CNN result. We found that features extracted with the VGG architecture resulted in the highest ss with a 3% higher ss than the best AlexNet result. We also found that PCA for dimensionality reduction achieved a ss performance of 7% higher than the best GRP result and 10% higher than best CCA result. Because CCA finds projections of high correlation between the kinematics and video, we believe that CCA discards informative features resulting in reduced clustering performance. We note that neither of the encoding schemes, VLAD or LCDLCD$_{VLAD}$ significantly improves the ss.

There are two hyper-parameters for TSC which we set empirically: sliding window size (T = 3), and the number of PCA dimensions (k = 100). In Figure 3.19, we show a sensitivity plot with the ss as a function of the parameter. We calculated the ss using the same subset of the suturing dataset as above and with the VGG conv5_3 CNN. We found that T = 3 gave the best performance. We also found that PCA with k = 1000 dimensions was only marginally better than k = 100 yet required $>$30 mins to run. For computational reasons, we selected k = 100.

t-SNE visualization of visual features

One of the main insights of this study is that features from pre-trained CNNs exhibit locally-linear behavior which allows application of a switching linear dynamical system model. We experimentally tested this by applying dimensionality reduction to trajectories of features from different video featurization techniques. Figure 3.20 shows t-SNE embeddings of visual features extracted for a single demonstration of suturing. The deep features display clear locally-linear properties and can be more easily clustered than SIFT features extracted for the corresponding frames. We speculate that SIFT breaks up trajectory structure due to its natural scale and location invariance properties. We also compared to using the raw

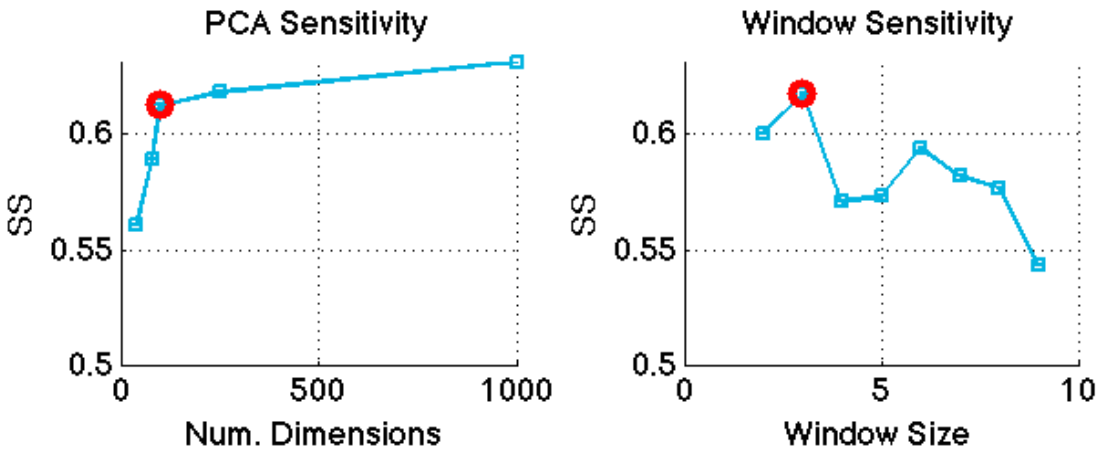


Figure 3.19: We evaluate the sensitivity of two hyperparameters set in advance: number of PCA dimensions and sliding window size. The selected value is shown in red double circles.

RGB image pixel values and discovered that the deep features result in more well-formed locally linear trajectories. However, it is important to note that unlike spatial trajectories there are discrete jumps in the convolutional trajectories. We hope to explore this problem in more detail in future work.

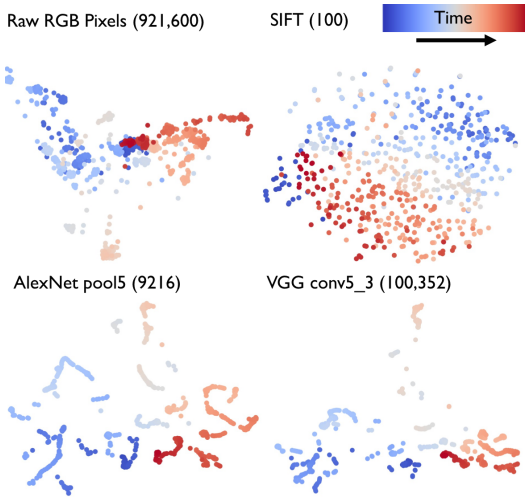


Figure 3.20: Each data point in the figure corresponds to a t-SNE visualization of features of a single frame in the video. (a) RGB pixel values of original image (b) shallow SIFT features (c) CNN features from AlexNet pool5 (d) CNN features from VGG Conv5_3.

End-to-End Evaluation

For all subsequent experiments on real data, we used a pre-trained VGG CNN conv5_3 and encoded with PCA with 100 dimensions.

1. Suturing: We apply our method to a subset of the JIGSAWS dataset [39] consisting of surgical task demonstrations under teleoperation using the da Vinci surgical system. The dataset was captured from eight surgeons with different levels of skill, performing five repetitions each of suturing and needle passing. jigsaws lists quantitative results for both needle passing and suturing with both ss and NMI agreement with the human labels. Demonstrations from the JIGSAWS dataset were annotated with the skill-level of the demonstrators (Expert (E), Intermediate (I), and Novice (I)). For the suturing dataset, we find that using both kinematics and video gives up-to 30.1% improvement in ss and 52.3% improvement in NMI over using kinematics alone. Not surprisingly, we also find that the expert demonstrations, which are usually smoother and faster, lead to improved segmentation performance when using only the kinematic data. However, when we incorporate the visual data, the trend is not as clear. We speculate this has to do with the tradeoff between collecting more data (denser clusters and more accurate modeling) versus inconsistencies due to novice errors, and this tradeoff is evident in higher dimensional data.

We visualize the results of the segmentation on one representative trajectory (Figure 3.21). With combined kinematics and vision, TSC learns many of the important segments identified by annotation in [39]. Upon further investigation of the false positives, we found that they corresponded to meaningful actions missed by human annotators. TSC discovers that a repositioning step where many demonstrators penetrate and push-through the needle in two different motions. While this is largely anecdotal evidence, we were able to find some explanations for some of the false positives found by TSC.

2. Needle Passing: Next, we applied TSC to 28 demonstrations of the needle passing task. These demonstrations were annotated in [39]. In this task, the robot passes a needle through a loop using its right arm, then its left arm to pull the needle through the loop. Then, the robot hands the needle off from the left arm to the right arm. This is repeated four times. Similar to the suturing dataset, we find that the combination of the features gives the best results. For the needle passing dataset, we find that using both kinematics and video gives up to 22.2% improvement in ss and 49.7% improvement in NMI over using the best of either kinematics or vision alone.

We found that the learned segments for the needle passing task were less accurate than those learned for the suturing task. We speculate that this is due to the multilateral nature of this task. This task uses both arms more than the suturing task, and as a result, there are many visual occlusions for a fixed camera. Important features such as the needle pose and the thread may be obscured at different points during the task. Furthermore, we constructed the state-space using the states of both arms. For such a task, it may be better to segment each of the arms independently.

3. PR2: Legos and Toy Plane Assembly: In our next experiment, we explore segmenting a multi-step assembly task using (1) large *Lego* blocks and (2) toy *Plane* from the YCB

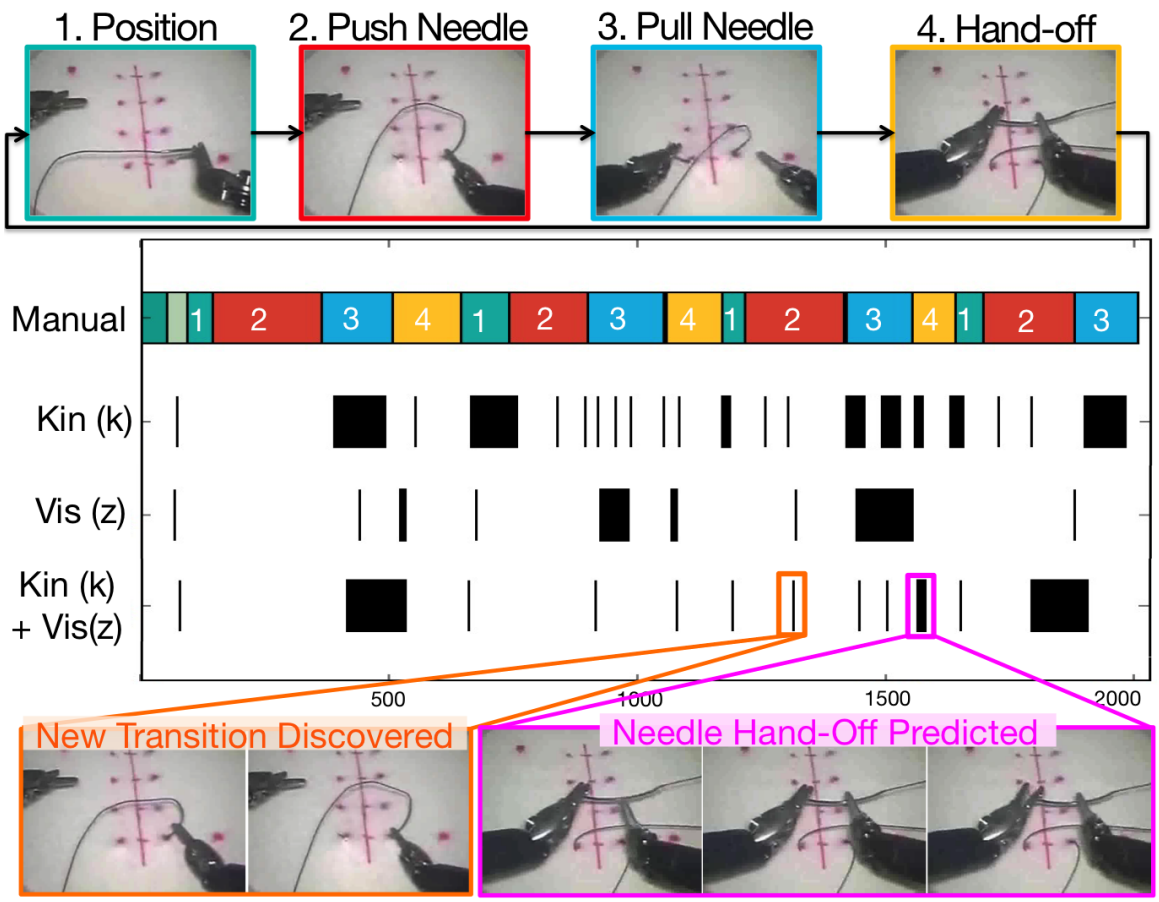


Figure 3.21: The first row shows a manual segmentation of the suturing task in 4 steps: (1) Needle Positioning, (2) Needle Pushing, (3) Pulling Needle, (4) Hand-off. TSC extracts many of the important transitions without labels and also discovers un-labled transition events.

dataset [22]. We demonstrate that TSC applies generally outside of surgical robotics. We collect eight kinesthetic demonstrations for each task through kinesthetic demonstrations of the task on the PR2 robot. Figure 3.23 illustrates the segmentation for the plane assembly task. We find the plane assembly task using kinematics or vision alone results in a large number of segments. The combination can help remove spurious segments restricting our segments to those transitions that occur in most of the demonstrations–agreeing in similarity both kinematically and visually.

4. Human Demonstration of Toy Plane Assembly: We extend the toy plane assembly experiment to collect eight demonstrations each from two human users. These examples only have videos and no kinematic information. We note that there was a difference between users in the grasping location of fuselage. We also to learn segmentation for human demonstrations. The results of TSC performance are summarized in Table 3.2. We omit

Figure 3.22: Comparison of TSC performance on Suturing and Needle Passing Tasks. We compare the prediction performance by incrementally adding demonstrations from Experts (E), Intermediates (I), and Novices (N) respectively to the dataset.

		Kin	Vid	Kin+Vid
Silhouette Score – Intrinsic Evaluation				
Suturing	E+I+N	0.518±0.008	0.576±0.018	0.733±0.056
	E+I	0.550±0.014	0.548±0.015	0.716±0.046
	E	0.630±0.014	0.515±0.021	0.654±0.065
Needle Passing	E+I+N	0.513±0.007	0.552±0.011	0.557±0.010
	E+I	0.521±0.006	0.536±0.013	0.666±0.067
	E	0.524±0.004	0.609±0.010	0.716±0.097
NMI Score – Extrinsic evaluation against manual labels				
Suturing	E+I+N	0.307 ± 0.045	0.157 ± 0.022	0.625 ± 0.034
	E+I	0.427 ± 0.053	0.166 ± 0.057	0.646 ± 0.039
	E	0.516 ± 0.026	0.266 ± 0.025	0.597 ± 0.096
Needle Passing	E+I+N	0.272 ± 0.035	0.186 ± 0.034	0.385 ± 0.092
	E+I	0.285 ± 0.051	0.150 ± 0.048	0.471 ± 0.023
	E	0.287 ± 0.043	0.222 ± 0.029	0.565 ± 0.037

	K	Z	K+Z
Silhouette Score – Intrinsic Evaluation			
Lego (Robot)	0.653±0.003	0.644±0.026	0.662±0.053
Plane (Robot)	0.741±0.011	0.649±0.007	0.771±0.067
Plane (Human 1)	–	0.601 ±0.010	–
Plane (Human 2)	–	0.628 ±0.015	–
NMI Score – Extrinsic evaluation against manual labels			
Lego (Robot)	0.542 ± 0.058	0.712 ± 0.041	0.688 ± 0.037
Plane (Robot)	0.768 ± 0.015	0.726 ± 0.040	0.747 ± 0.016
Plane (Human 1)	–	0.726 ± 0.071	–
Plane (Human 2)	–	0.806 ± 0.034	–

Table 3.2: Plane and Lego Assembly Tasks. Both tasks show improvements in clustering and prediction accuracy using multi-modal data as compared to either modality. Further, only vision (Z) is available for human demos of the plane assembly task. Comparable segmentation results are obtained using only video input for human demos. *Higher* Silhoutte Scores and NMI scores are better, respectively.

a visualization of the results for the Lego assembly. However, we summarize the results quantitatively in the table.

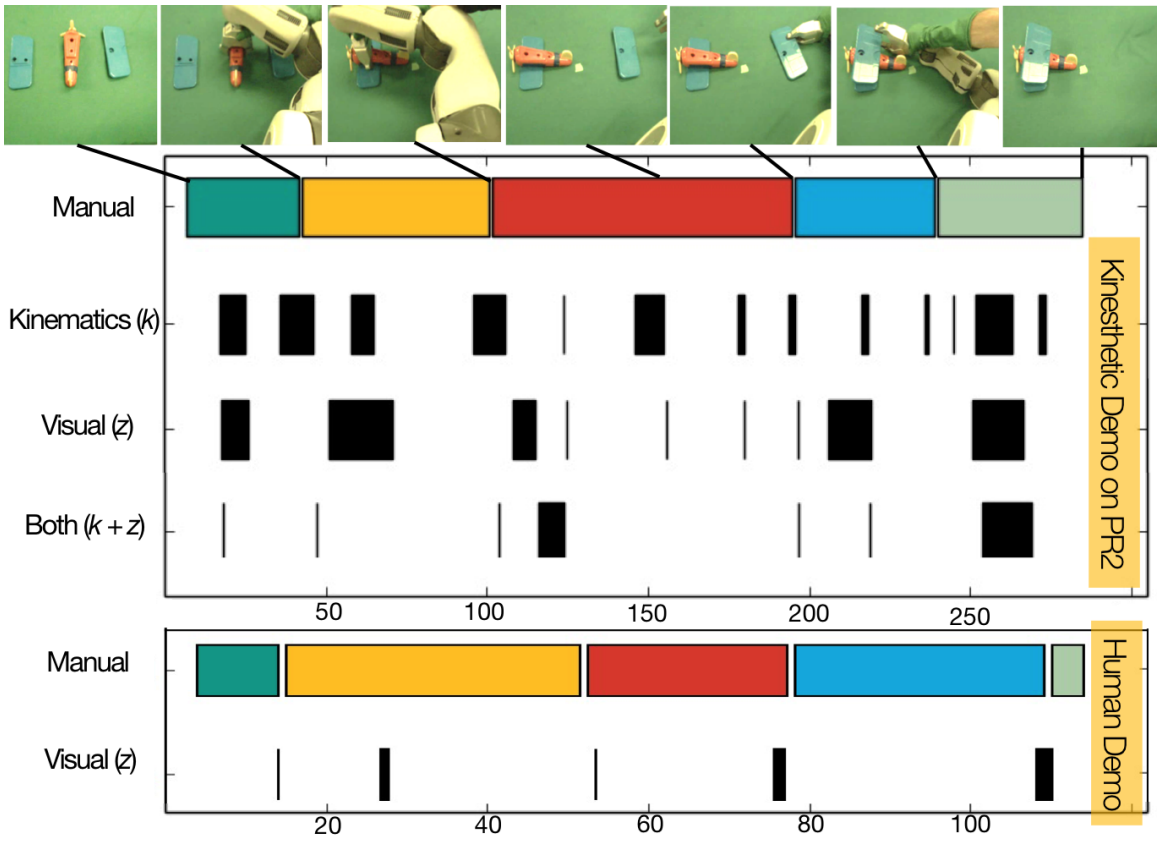


Figure 3.23: We compare TSC on 12 kinesthetic demonstrations (top) and 8 human demonstrations (bottom). No kinematics were available for the human demonstrations. We illustrate the segmentation for an example demonstration of each. Our manual annotation of the task has five steps and TSC recovers this structure separately for both Kinesthetic demos on PR2 and Human demos with the same visual features.

3.2 SWIRL: Learning With Transition States

Next, we build on TSC to construct a planner that leverages the structure learned by TSC. Real-world tasks often naturally decompose into a sequence of simpler, locally-solvable sub-tasks. For example, an assembly task might decompose into completing the part's constituent sub-assemblies or a surgical task might decompose into a sequence of movement primitives. Such structure imposes a strong prior on the class of successful policies and can focus exploration in reinforcement learning. It reduces the effective time horizon of learning to the start of the next subtask rather than until task completion. We apply a clustering algorithm to identify a latent set of state-space subgoals that sequentially compose to form the global task. This leads to a novel policy search algorithm, called Sequential Windowed Inverse Reinforcement Learning (SWIRL), where the demonstrations can bootstrap a self-supervised Q-learning algorithm.

Transitions are defined as significant changes in the state trajectory. These transitions can be spatially and temporally clustered to identify if there are common conditions that trigger a change in motion across demonstrations. SWIRL extends this basic model with an Inverse Reinforcement Learning step that extracts subgoals and computes local cost functions from the learned clusters. Learning a policy over the segmented task is nontrivial because solving k independent problems neglects any shared structure in the value function during the policy learning phase (e.g., a common failure state). Jointly learning over all segments introduces a dependence on history, namely, any policy must complete step i before step $i+1$. Learning a memory-dependent policy could lead to an exponential overhead of additional states. We show that the problem can be posed as a proper MDP in a lifted state-space that includes an indicator variable of the highest-index $\{1, ..., k\}$ transition region that has been reached so far if there are Markovian regularity assumptions on the clustering algorithm.

Model and Notation

Directly optimizing the reward function $\mathcal{R}$ in the MDP from the previous section might be very challenging. We propose to approximate $\mathcal{R}$ with a sequence of smoother reward functions.

Definition 4 (Proxy Task) *A proxy task is a set of k MDPs with the same state set, action set, and dynamics. Associated with each MDP i is a reward function $R_i : S \times A \mapsto \mathbb{R}$. Additionally, associated with each R_i is a transition region $\rho_i \subseteq S$, which is a subset of the state-space. A robot accumulates a reward R_i until it reaches the transition ρ_i, then the robot switches to the next reward and transition pair. This process continues until ρ_k is reached.*

A robot is deemed *successful* when all of the ρ_i are reached in sequence within a global time-horizon T. SWIRL uses a set of initial supervisor demonstrations to construct a proxy task that approximates the original MDP. To make this problem computationally tractable, we make some modeling assumptions.

Modeling Assumption 1. Successful Demonstrations: We need conditions on the demonstrations to be able to infer the sequential structure. We assume that all demonstrations are successful, that is, they visit each ρ_i in the same sequence.

Modeling Assumption 2. Quadratic Rewards: We assume that each reward function R_i can be expressed as a quadratic of the form $-(s - s_0)^T \Psi (s - s_0)$ for some positive semi-definite Ψ and a center point s_0 with $s_0^T \Psi s_0 = 0$.

Modeling Assumption 3. Ellipsoidal Approximation: Finally, we assume that the transition regions in ρ_i can be approximated by a set of disjoint ellipsoids.

Algorithm Overview

SWIRL can be described in terms of three sub-algorithms:

Inputs: Demonstrations D

1. **Sequence Learning:** Given D, SWIRL applies TSC to partition the task into k sub-tasks whose start and end are defined by arrival at a sequence of transitions $G = [\rho_1, ..., \rho_k]$.

2. **Reward Learning:** Given G and D, SWIRL associates a local reward function with each segment resulting in a sequence of rewards $\mathbf{R}_{seq} = [R_1, ..., R_k]$.

3. **Policy Learning:** Given $\mathbf{R}_{seq}$ and G, SWIRL applies reinforcement learning to optimize a policy for the task π.

Outputs: Policy π

In principle, one could couple steps 1 and 2 similar to the results in [97]. We separate these steps since that allows us to use a different set of features for segmentation than used for reward learning. Perceptual features can provide a valuable signal for segmentation but quadratic reward functions may not be meaningful in all perceptual feature spaces.

Sequence Learning Algorithm

First, SWIRL applies a clustering algorithm to the initial demonstrations to learn the transition regions. The clustering model is based on our prior work on Transition State Clustering (TSC) [67, 86]. Transitions are defined as significant changes in the state trajectory. These transitions can be spatially and temporally clustered to identify if there are common conditions that trigger a change in motion across demonstrations.

Reward Learning Algorithm

After the sequence learning phase, each demonstration is partitioned into k segments. The reward learning phase uses the learned $[\rho_1, ..., \rho_k]$ to construct the local rewards $[R_1, ..., R_k]$ for the task. Each R_i is a quadratic cost parametrized by a positive semi-definite matrix Ψ.

The role of the reward function is to guide the robot to the next transition region ρ_i. A first approach is for each segment i, we can define a reward function as follows:

$$R_i(s) = -\|s - \mu_i\|_2^2,$$

which is just the Euclidean distance to the centroid.

A problem with using Euclidean distance directly is that it uniformly penalizes disagreement with μ in all dimensions. During different stages of a task, some directions will likely naturally vary more than others. To account for this, we can derive :

$$\Psi[j, l] = \Sigma^{-1},$$

Algorithm 3 Reward Inference

Input: Demonstration $\mathcal{D}$ and sub-goals $[\rho_1, ..., \rho_k]$
Segment each demonstration d_i into k sub-sequences where the j^{th} is denoted by $d_i[j]$.
Apply Equation 2 to each set of sub-sequences $1...k$.
Return: $\mathbf{R}_{seq}$

which is the inverse of the covariance matrix of all of the state vectors in the segment:

$$\Psi = (\sum_{t=start}^{end} ss^T)^{-1}, \tag{2}$$

which is a $p \times p$ matrix defined as the covariance of all of the states in the segment $i-1$ to i. Intuitively, if a feature has low variance during this segment, deviation in that feature from the desired target it gets penalized. This is exactly the Mahalonabis distance to the next transition.

For example, suppose one of the features j measures the distance to a reference trajectory u_t. Further, suppose in step one of the task the demonstrator's actions are perfectly correlated with the trajectory ($\Psi_i[j,j]$ is low where variance is in the distance) and in step two the actions are uncorrelated with the reference trajectory ($\Psi_i[j,j]$ is high). Thus, Ψ will respectively penalize deviation from $\mu_i[j]$ more in step one than in step two.

Policy Learning

policy-learning SWIRL uses the learned transitions $[\rho_1, ..., \rho_k]$ and $\mathbf{R}_{seq}$ to construct a proxy task to solve via Reinforcement Learning. In this section, we describe learning a policy π given rewards $\mathbf{R}_{seq}$ and an ordered sequence of transitions G. However, this problem is not trivial since solving k independent problems neglects potential shared value structure between the local problems (e.g., a common failure state). Furthermore, simply taking the aggregate of the rewards can lead to inconsistencies since there is nothing enforcing the order of operations. We show that a single policy can be learned jointly over all segments over a modified problem where the state-space with additional variables that keep track of the previously achieved segments. We present a Q-Learning algorithm [85, 121] that captures the coupling noted above between task segments. In principle, similar techniques can be used for any other policy search method.

Jointly Learning Over All Segments

In our sequential task definition, we cannot transition to reward R_{i+1} unless all previous transition regions $\rho_1, ...\rho_i$ are reached in sequence. We can leverage the definition of the Markov Segmentation function formalized earlier to jointly learn across all segments, while leveraging the segmented structure. We know that the reward transitions (R_i to R_{i+1}) only depend on an arrival at the transition state ρ_i and not any other aspect of the history. Therefore, we can store an index v, that indicates whether a transition state $i \in 0, ..., k$ has been

reached. This index can be efficiently incremented when the current state $s \in \rho_{i+1}$. The result is an augmented state-space $\binom{s}{v}$ to account for previous progress. In this lifted space, the problem is a fully observed MDP. Then, the additional complexity of representing the reward with history over $S \times [k]$ is only $\mathcal{O}(k)$ instead of exponential in the time horizon.

Segmented Q-Learning

At a high-level, the objective of standard Q-Learning is to learn the function $Q(s, a)$ of the optimal policy, which is the expected reward the agent will receive taking action a in state s, assuming future behavior is optimal. Q-Learning works by first initializing a random Q function. Then, it samples rollouts from an exploration policy collecting (s, a, r, s') tuples. From these tuples, one can calculate the following value:

$$y_i = R(s, a) + \arg\max_a Q(s', a)$$

Each of the y_i can be used to define a loss function since if Q were the true Q function, then the following recurrence would hold:

$$Q(s, a) = R(s, a) + \arg\max_a Q(s', a)$$

So, Q-Learning defines a loss:

$$L(Q) = \sum_i \|y_i - Q(s, a)\|_2^2$$

This loss can be optimized with gradient descent. When the state and action space is discrete, the representation of the Q function is a table, and we get the familiar Q-Learning algorithm [121]–where each gradient step updates the table with the appropriate value. When Q function needs to be approximated, then we get the Deep Q Network algorithm [85].

SWIRL applies a variant of Q-Learning to optimize the policy over the sequential rewards. The basic change to the algorithm is to augment the state-space with indicator vector that indicates the transition regions that have been reached. So each of the rollouts, now records a tuple $(s, \mathbf{v}, a, r, s', \mathbf{v'})$ that additionally stores this information. The Q function is now defined over states, actions, and segment index–which also selects the appropriate local reward function:

$$Q(s, a, v) = R_v(s, a) + \arg\max_a Q(s', a, v')$$

We also need to define an exploration policy, i.e., a stochastic policy with which we will collect rollouts. To initialize the Q-Learning, we apply Behavioral Cloning locally for each of the segments to get a policy π_i. We apply an ϵ-greedy version of these policies to collect rollouts.

Algorithm 3: Policy Learning

Data: Transition States G, Reward Sequence $\mathbf{R}_{seq}$, exploration policy π

Initialize $Q(\binom{s}{v}, a)$ randomly
foreach $iter \in 0, ..., I$ **do**
    Draw s_0 from initial conditions
    Initialize v to be $[0, ..., 0]$
    Initialize j to be 1
    foreach $t \in 0, ..., T$ **do**
        Choose best action a based on π.
        Observe Reward R_j
        Update state to s' and Q via Q-Learning update
        If s' is $\in \rho_j$ update $v[j] = 1$ and $j = j + 1$

Result: Policy π

Simulated Experiments

We constructed a parallel parking scenario for a robot car with non-holonomic dynamics and two obstacles (Figure 3.24a), and we also experimented on the standard acrobot domain (Figure 3.24b).

Experimental Methodology

First, we overview our basic experimental methodology. We evaluate SWIRL on two simulated RL benchmarks and two deformable manipulation tasks on the da Vinci surgical robot [57]. In all of these tasks, there is a single MDP of interest where the reward is sparse (for a substantial amount of the state-space the reward is zero). The goal of SWIRL is to improve convergence on this task.

Supervision

In both of the RL benchmarks, the reward function defines a target configuration of the robot. We generated the initial demonstrations using an RRT* motion planner (assuming deterministic dynamics) [55]. As both the RL benchmarks are stochastic in nature, we used an MPC-style re-planning approach to control the robot to the target region. On the physical experiments, we provided the robot with demonstrations collected in tele-operation. We collected fully observed trajectories with both states and actions.

Basic Baselines

Pure Exploration: The first set of baselines study a pure exploration approach to learning a policy. The baseline approach applies RL to the global task. In all of our experiments, we use variants of Q-Learning with different function approximators. The Q function is randomly initialized and is updated with data collected from episodic rollouts. The hyper-parameters for each experiment are listed in the appendix. We use approximate Q-Learning [13, 124] in the simulated benchmarks and Deep Q-Networks in the physical experiments [85].

Pure Demonstration: This baseline directly learns a policy from an initial set of demonstrations using supervised learning. This approach is called Behavioral Cloning (see survey of imitation learning in [91]), and in each of our experiments, we describe the policy models used. It is important to note that this approach requires fully observed demonstrations.

Warm Start Exploration: Next, we consider approaches that leverage both demonstrations and exploration. One approach is to use demonstrations to initialize the Q-function for RL and then perform rollouts. This approach requires fully observed demonstrations as well.

Inverse Reinforcement Learning: Alternatively, we can also use the demonstrations to infer a new reward function. We use IRL to infer a smoother quadratic reward function that explains the demonstrator's behavior. We infer this quadratic reward function using MaxEnt-IRL. We consider both using estimated dynamics and ground truth dynamics for this baseline. When the dynamics are estimated this approach requires fully observed demonstrations. After inferring the reward, the task is solved with RL w.r.t the quadratic reward function.

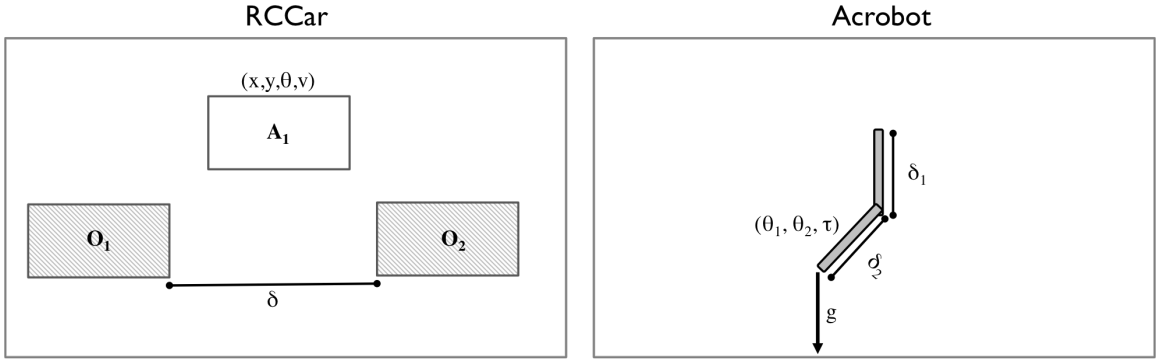


Figure 3.24: (A) Simulated control task with a car with noisy non-holonomic dynamics. The car (A_1) is controlled by accelerating and turning in discrete increments. The task is to park the car between two obstacles.

Tasks

For the car domain, the car can accelerate or decelerate in discrete ± 0.1 meters per second squared increments (the car can reverse), and change its heading by 5° degree increments. The car's velocity and heading θ are inputs to a bicycle steering model which computes the next state. The car observes its x position, y position, orientation, and speed in a global coordinate frame. The car's dynamics are noisy and with probability 0.1 will randomly add or subtract 2.5° degrees to the steering angle. If the car parks between the obstacles, i.e., 0 speed within a 15° tolerance and a positional tolerance of 1 meter, the task is a success and the car receives a reward of 1. The obstacles are 5 meters apart (2.5 car lengths). If the car collides with one of the obstacles or does not park in 200 timesteps, the episode ends with a reward of 0.

The acrobot domain consists of an under-actuated two-link pendulum with gravity and with torque controls on the joint. There are 4 discrete actions that correspond to clockwise and counter-clockwise torques on each of the links. The robot observes the angle θ_1, θ_2 and angular velocity ω_1, ω_2 at each of the links. The dynamics are noisy where a small amount of random noise is added to each torque applied to the pendulum. The robot has 1000 timesteps to raise the arm above horizontal ($y = 1$ in the image). If the task is successful and the robot receives a reward of 1. The expected reward is equivalent to the probability that the current policy will successfully raise the arm above horizontal.

Pure Exploration vs. SWIRL

In the first set of experiments, we compare the learning efficiency of pure exploration to SWIRL (Figure 3.25). The baseline line Q-Learning approach (QL) is very slow because it relies on random exploration to achieve the goal at least once before it can start estimating the value of states and actions. We fix the number of initial demonstrations provided to SWIRL. We apply the segmentation and reward learning algorithms and construct a proxy task. In both domains, SWIRL significantly accelerates learning and converges to a successful policy with significantly fewer demonstrations. We find that in the parallel parking domain this improvement is more substantial. This is likely because the task more naturally partitions into discrete subtasks. In the appendix, we visualize the segments discovered by the algorithm.

Pure Demonstration vs. SWIRL

Next, we evaluate SWIRL against a behavioral cloning approach (Figure 3.26). We collect the initial set of demonstrations and directly learn a policy with an SVM. For the parallel parking task, we use a linear SVM. For the acrobot task, we use a kernel SVM with an RBF kernel. We fix the number of autonomous rollouts that SWIRL can observe (500 for the parallel parking task and 3000 for the acrobot task). Note that the SVM technique requires observing the actions in the demonstration trajectories, which may not be possible in all applications. The SVM approach does have the advantage that it doesn't require

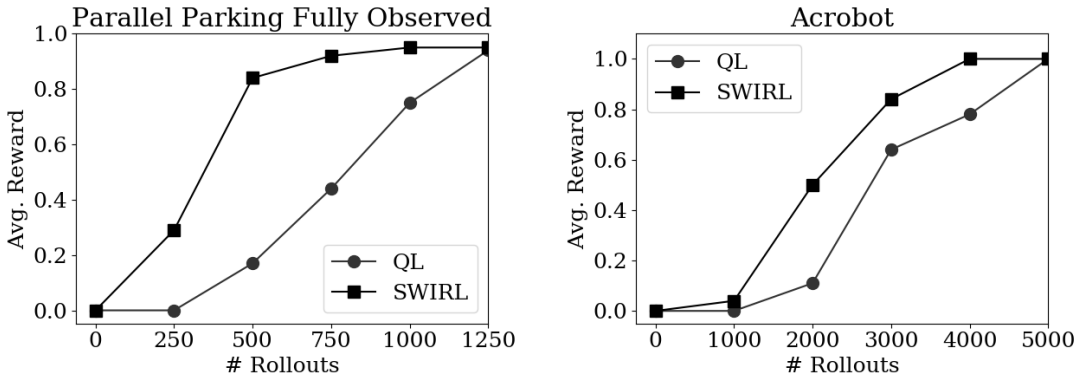


Figure 3.25: Learning curves for Q Learning (QL) and SWIRL on both simulated tasks. (Parallel Parking) For a fixed number of demonstrations 5, we vary the number of rollouts and measure the average reward at each rollout. (Acrobot) For a fixed number of demonstrations 15, we vary the number of rollouts and measure the average reward at each rollout.

any further exploration. However, SWIRL and the pure demonstration approach are not mutually exclusive. As we show in our physical experiments, we can initialize Q-learning with a behavioral cloning policy. The combination of the two approaches allows us to take advantage of a small number of demonstrations and learn to refine the initial policy through exploration.

The SVM approach requires more than 10x the demonstrations to be competitive. In particular, there is an issue with exhaustively demonstrating all the scenarios a robot may encounter. Learning from autonomous trials in addition to the initial demonstrations can augment the data without the burdening the supervisor. Perhaps surprisingly, the initial dataset of demonstrations can be quite small. On both tasks, with only five demonstrations, SWIRL is within 20% of its maximum reward. Representing a policy is often more complex than a reward function that guides the agent to valuable states. Learning this structure requires less data than learning a full policy. This suggest that SWIRL can exploit a very small number of expert demonstrations to dramatically reduce the number of rollouts needed to learn a successful policy.

SWIRL vs. Other Hybrid Approaches

Finally, we compare SWIRL to two other hybrid demonstration-exploration approaches (Figure 3.27). The goal of these experiment is to show that the sequential structure learned in SWIRL is a strong prior. As in the previous experiment, it is important to note that SWIRL only requires a state-trajectory as a demonstration and does not need to explicitly observe the actions taken by the expert demonstrator.

Initializing the Q-Function with the demonstrations, did not yield a significant improvement over random initialization. This is because in expert demonstration one rarely ob-

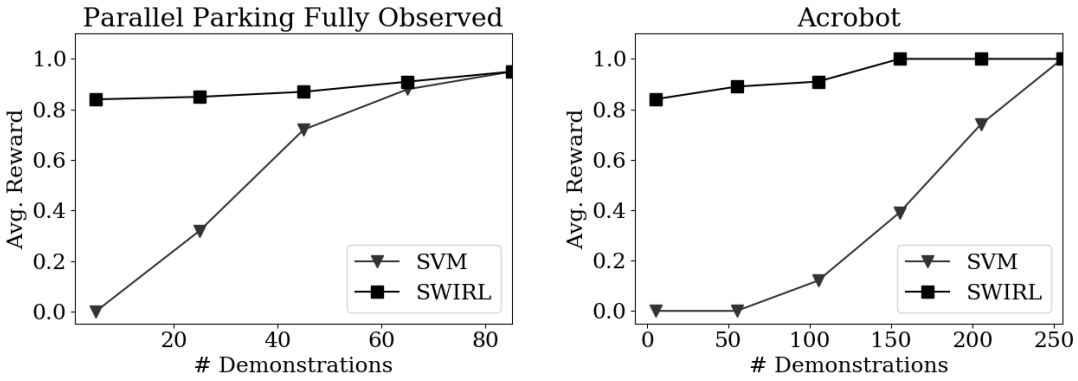


Figure 3.26: Demonstration curves for imitation learning (SVM) and SWIRL on both simulated tasks. (Parallel Parking) We fix the number of rollouts to 500 and vary the number of demonstration trajectories each approach observes. (Acrobot) For a number of rollouts 3000, we vary the number of demonstration trajectories given to each technique.

serves failures, and if the q-learning algorithm does not observe poor decisions it will not be able to avoid them in the future. We also applied an IRL approach using estimated dynamics. The IRL approach is substantially better than the basic Q-Learning algorithm in the parallel parking domain. This is likely because it smooths out the sparse reward to fit a quadratic function. This does serve to guide the robot to the goal states to some extent. Finally, SWIRL is the most sample-efficient algorithm. This is because the sequential quadratic rewards learned better align with the true value functions in both tasks. This structure can be learned from a small number of demonstrations.

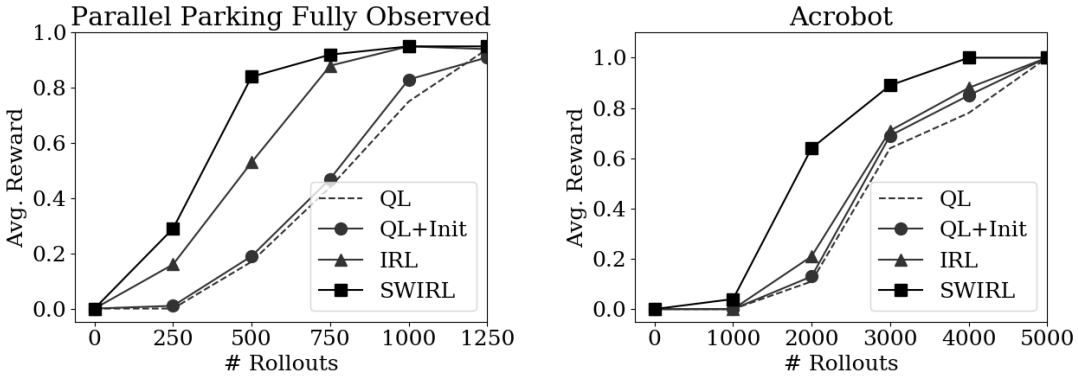


Figure 3.27: Comparison of hybrid approaches. (Parallel Parking) For a fixed number of demonstrations 5, we vary the number of rollouts and measure the average reward at each rollout. (Acrobot) For a fixed number of demonstrations 15, we vary the number of rollouts and measure the average reward at each rollout.

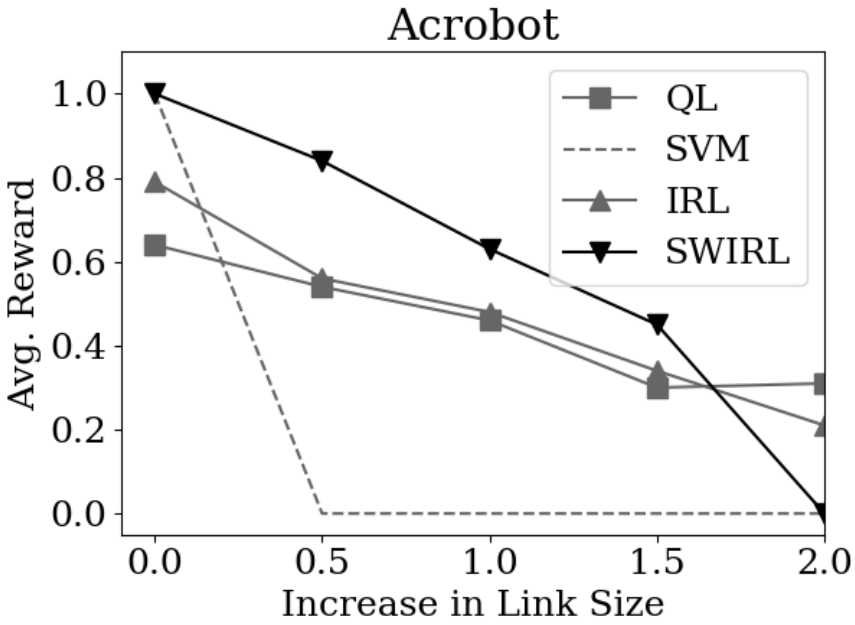


Figure 3.28: For a number of rollouts 3000 and 250 demonstrations, we measure the transfer as a function of varying the link size. (QL) denotes Q-learning, (SVM) denotes a baseline of behavioral cloning with a Kernel SVM policy representation, (IRL) denotes MaxEnt-IRL using estimated dynamics learned from the demonstrations, (SWIRL) denotes SWIRL. The SVM policy fails as soon the link size is changed. SWIRL is robust until the change becomes very large.

The Benefit Of Models

Next, we consider the benefits of using Inverse Reinforcement Learning with the ground truth dynamics models compared to the ones estimated from data (Figure 3.29). One scenario where this problem setting is useful is when the demonstration dynamics (are known) but differ from the execution dynamics. Most common IRL frameworks, like Maximum Entropy IRL, assume access to the dynamics model. In the previous experiments, these models were estimated from data, and here we show the benefit of providing the true models to the algorithms. Both IRL and SWIRL improve their sample efficiency significantly when ground truth models are given. This experiment illustrates that the principles behind SWIRL are compatible with model-based methodologies.

Different Segmentation Methods

In our general framework, SWIRL is compatible with any heuristic to segment the initial demonstration trajectories. This heuristic serves to oversegment and the unsupervised learning model builds a model for sequential rewards from this heuristic. The previous

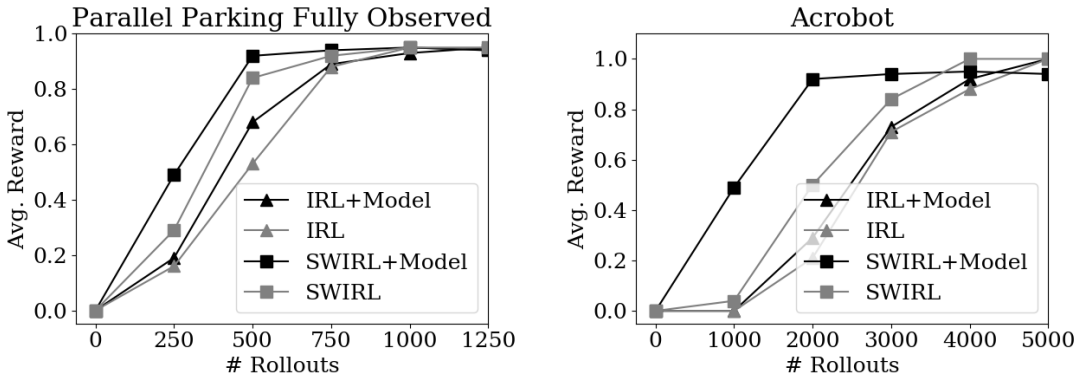


Figure 3.29: (Parallel Parking) For a fixed number of demonstrations 5, we vary the number of rollouts and measure the average reward at each rollout. (Acrobot) For a fixed number of demonstrations 15, we vary the number of rollouts and measure the average reward at each rollout.

experiments use a GMM-based approach as a segmentation heuristic, this experiment evaluates the same domains with other heuristics. In particular, we consider two other models: segmentation based on changes in direction of velocity and segmentation based on linear dynamical regimes. Figure 3.30 illustrates the results. While there are differences between the performance of different heuristics, we found that the GMM-based approach was the most reliable across domains.

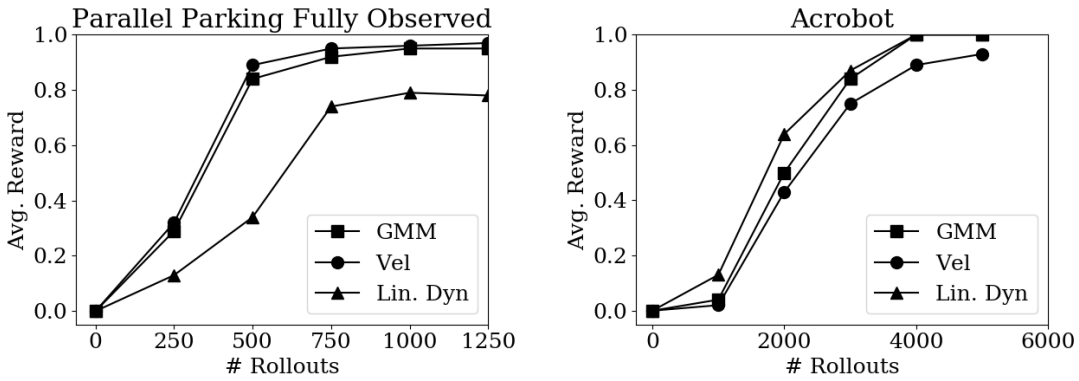


Figure 3.30: We compare to different transition indicator heuristics with SWIRL. (Parallel Parking) For a fixed number of demonstrations 5, we vary the number of rollouts and measure the average reward at each rollout. (Acrobot) For a fixed number of demonstrations 15, we vary the number of rollouts and measure the average reward at each rollout.

Transfer

We constructed two transfer scenarios to evaluate situations whether the structure learned overfits to the initial demonstrations. In essense, this is an evaluation of how well the approaches handle transfer if the dynamics change between demonstration and execution. We collect demonstrations $N = 100$ on the original task, and then used the learned rewards or policies on a perturbed task. For the parallel parking task, we modified the execution environment such that the dynamics are coupled in a way that turning right causes the car to accelerate forward by 0.05 meters per second. In the perturbed task, the car must learn to adjust to this acceleration during the reversing phase. In the new domain, each approach is allowed 500 rollouts. We report the results (Figure 3.31).

The success rate of the policy learned with Q-Learning is more or less constant between the two domains. This is because Q-learning does not use any information from the original domain. The SVM behavioral cloning policy has a drastic change. On the original domain it achieves a 95% success rate (with 100 demonstrations), however, on the perturbed domain, it is never successful. This is because the SVM learned a policy that causes it to crash into one of the obstacles in the perturbed environment.

The IRL techniques are more robust during the transfer. This is because the rewards learned are quadratic functions of the state and do not encode anything specific about the dynamics. Similarly, in SWIRL, the rewards and transition regions are invariant to the dynamics in this transfer problem. For SWIRL-E and SWIRL-G, there is only a drop of 5% in the success rate. On the other hand, the model-free version of SWIRL reports a larger drop of 16%. This is because the model-free version is not a true IRL algorithm and may enocde some aspects of the dynamics in the learned reward function.

Coincidentally, this experiment also shows us how to construct a failure mode for SWIRL. If the perturbation in the task is such that it "invalidates" a transition region, e.g., a new obstacle, then SWIRL may not be able to learn to complete the task. However, the transition regions give us a formalism to detecting such problems during learning as we can keep track of which regions are possible to reach.

As in the parallel parking scenario, we evaluate how the different approaches handle transfer if the dynamics change between demonstration and execution. With $N = 250$ demonstrations, we learn the rewards, policies, and segments on the standard pendulum, and then during learning, we vary the size of the second link in the pendulum. We plot the success rate (after a fixed 3000 rollouts) as a function of the increasing link size (Figure 3.28).

As the link size increases the even the baseline Q-learning becomes less successful. This is because the system becomes more unstable and it is harder to learn a policy. The behavioral cloning SVM policy immediately fails as the link size is increased. IRL is more robust but does not offer much of an advantage in this problem. SWIRL is robust until the change in the link size becomes large. This is because for the larger link size, SWIRL might require different segments (or one of the learned segments in unreachable).

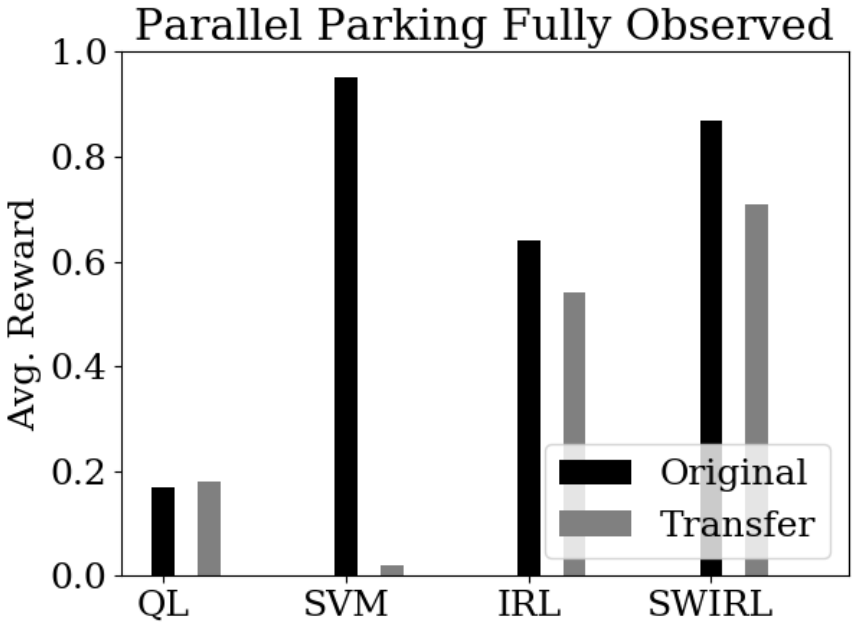


Figure 3.31: For 500 rollouts and 100 demonstrations, we measure the robustness of the approaches to changes in the execution dynamics. (QL) denotes Q-learning, (SVM) denotes a baseline of behavioral cloning with a SVM policy representation, (IRL) denotes MaxEnt-IRL with estimated dynamics, and (SWIRL) denotes the model-free version of SWIRL. While the SVM is 95% successful on the original domain, its success does not transfer to the perturbed setting. On the other hand, SWIRL learns rewards and segments that transfer to the new dynamics since they are state-space goals.

Sensitivity

Next, we evaluated the sensitivity of SWIRL to different initial demonstration sets (Figure 3.32). We sampled random initial demonstration sets and re-ran the algorithm on each of the two domains 100 times. Figure 3.32 plots the mean reward as a function of the number of rollouts and 2 standard deviations over all of the trials. We find that SWIRL is not very sensitive to the particular initial demonstration dataset. In fact, the 2 standard deviation error bar is less than the improvement in convergence in previous experiments.

Segmentation and Partial Observation

Next, we made the Parallel Parking domain more difficult to illustrate the connection between segmentation and memory in RL. We hid the velocity state from the robot, so the car only sees (x, y, θ). As before, if the car collides with one of the obstacle or does not park in 200 timesteps the episode ends. We call this domain Parallel Parking with Partial Observation (PP-PO).

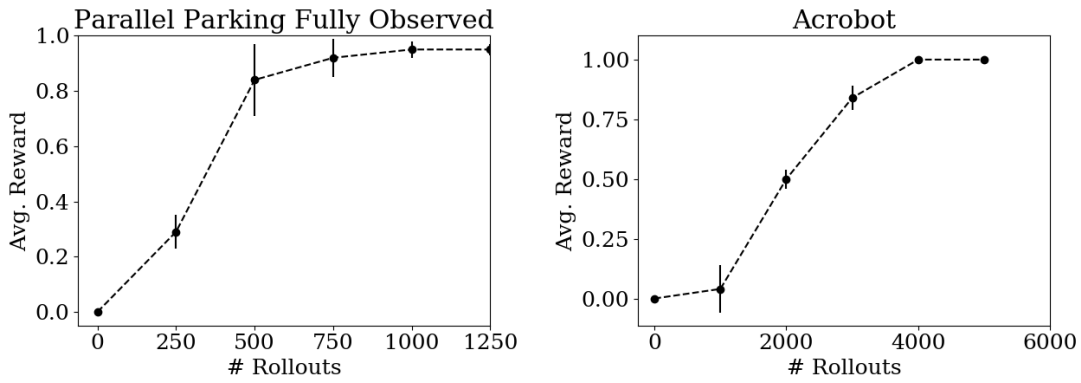


Figure 3.32: Sensitivity of SWIRL. (Parallel Parking) We generate a random set of 5 demonstration, we vary the number of rollouts and measure the average reward at each rollout. We plot the mean and standard deviation over 100 trials. (Acrobot) We generate a random set of 15 demonstration, we vary the number of rollouts and measure the average reward at each rollout. We plot the mean and 2 standard deviations over 100 trials

This form of partial observation creates an interesting challenge. There is no longer a stationary policy that can achieve the reward. During the reversing phase of parallel parking, the car does not know that it is currently reversing. So there is ambiguity in that state whether to pull up or reverse. We will see that segmentation can help disambiguate the action in this state.

As before, we generated 5 demonstrations using an RRT* motion planner (assuming deterministic dynamics) and applied each of the approaches. The techniques that model this problem with a single MDP all fail to converge. The Q-Learning approach achieves some non-zero rewards by chance. The learned segments in SWIRL help disambiguate dependence on history, since the segment indicator tells the car which stage of the task is currently active (pulling up or reversing) After 250000 time-steps, the policy learned with model-based SWIRL has a 95% success rate in comparison to a <10% success rate for the baseline RL, 0% for MaxEnt-IRL, and 0% for the SVM.

Physical Experiments with the da Vinci Surgical Robot

In the next set of experiments, we evaluate SWIRL on two tasks on the da Vinci Surgical Robot. The da Vinci Research Kit is a surgical robot originally designed for tele-operation, and we consider autonomous execution of surgical subtasks. Based on a chessboard calibration, we found that the robot has an RMSE kinematic error of 3.5 mm, and thus, requires feedback from vision for accurate manipulation. In our robotic setup, there is an overhead endoscopic stereo camera that can be used to find visual features for learning, and it is located 650mm above the workspace. This camera is registered to the workspace with a RSME calibration error of 2.2 mm.

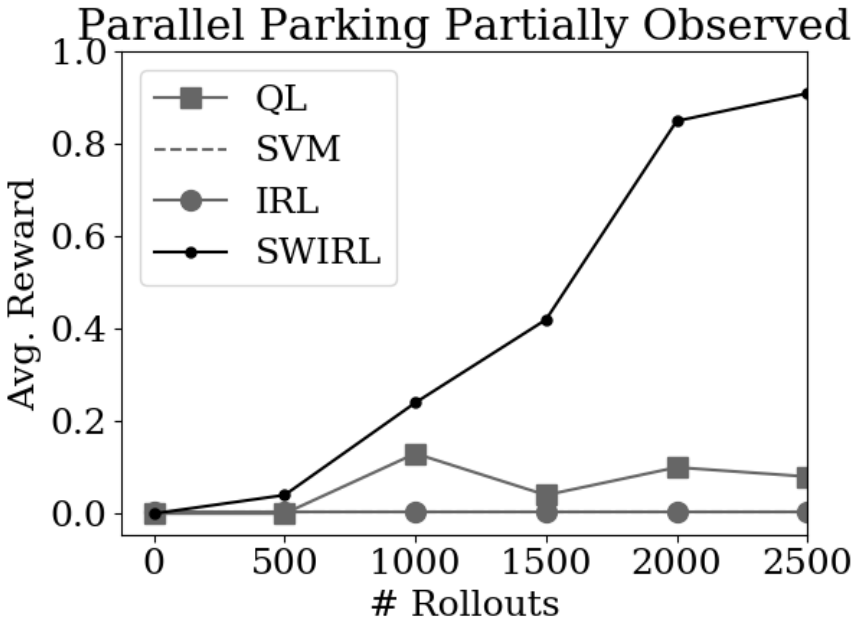


Figure 3.33: We hid the velocity state from the robot, so the robot only sees (x, y, θ). For a fixed number of demonstrations 5, we vary the number of rollouts and measure the average reward at each rollout. (QL) denotes Q-learning, (SVM) denotes a baseline of behavioral cloning with a SVM policy representation, (IRL) denotes MaxEnt-IRL with estimated dynamics, and (SWIRL) denotes SWIRL. SWIRL converges will the other approaches fail to reach a reliable success rate

Deformable Sheet Tensioning

In the first experiment, we consider the task of deformable sheet tensioning. A sheet of surgical gauze is fixtured at the two far corners using a pair of clips. The unclipped part of the gauze is allowed to rest on soft silicone padding. The robot's task is to reach for the unclipped part, grasp it, lift the gauze, and tension the sheet to be as planar as possible. An open-loop policy typically fails on this task because it requires some feedback of whether gauze is properly grasped, how the gauze has deformed after grasping, and visual feedback of whether the gauze is planar. The task is sequential as some grasps pick up more or less of the material and the flattening procedure has to be accordingly modified.

The state-space is the 6 DoF end-effector position of the robot, the current load on the wrist of the robot, and a visual feature measuring the flatness of the gauze. This is done by a set of fiducial markers on the gauze which are segmented by color using the stereo camera. Then, we correspond the segmented contours and estimate a z position for each marker (relative to the horizontal plane). The variance in the z position is a proxy for flatness and we include this as a feature for learning (we call this disparity). The action

space is discretized into an 8 dimensional vector ($\pm x$, $\pm y$, $\pm z$, open/close gripper) where the robot moves in 2mm increments.

We provided 15 demonstrations through a keyboard-based tele-operation interface. The average length of the demonstrations was 48.4 actions (although we sampled observations at a higher frequency about 10 observations for every action). From these 15 demonstrations, SWIRL identifies four segments. One of the segments corresponds to moving to the correct grasping position, one corresponds to making the grasp, one lifting the gauze up again, and one corresponds to straightening the gauze. One of the interesting aspects of this task is that the segmentation requires multiple features.

Then, we tried to learn a policy from the rewards constructed by SWIRL. In this experiment, we initialized the policy learning phase of SWIRL with the Behavioral Cloning policy. We define a Q-Network with a single-layer Multi-Layer Perceptron with 32 hidden units and sigmoid activation. For each of the segments, we apply Behavioral Cloning locally with the same architecture as the Q-network (with an additional softmax over the output layer) to get an initial policy. We rollout 100 trials with an $\epsilon = 0.1$ greedy version of these segmented policies.

The learning results of this experiment are summarized below with different baselines. The value of the policy is a measure of average disparity over the gauze accumulated over the task (if the gauze is flatter longer, then the value is greater). As a baseline, we applied RL for 100 rollouts with no other information. RL did not successfully grasp the gauze even once. Next, we applied behavioral cloning (BC) directly. BC was able to reach the gauze and but not successfully grasping it. Then, we applied the segmentation from SWIRL and applied BC directly to each local segment (without further refinement). This was able to complete the full task with a cumulative disparity score of -3516. Finally, we applied all of SWIRL and found the highest-value results (-3110). For comparison, we applied SWIRL without the BC initialization and found that it was only successful at the first two steps. This indicates that in real tasks the initialization is crucial.

Table 3.3: Results from the deformable sheet tensioning experiment

Technique	# Demonstrations	# Rollouts	Value
Pure Exploration (RL)	-	100	-8210
Pure Demonstration (BC)	15	-	-7591
Segmented Demos.	15	-	-3516
SWIRL	15	100	**-3110**

Surgical Line Cutting

In the next experiment, we evaluate generalization to different task instances. We apply SWIRL to learn to cut along a marked line in gauze similar to [87]. This is a multi-step problem where the robot starts from a random initial state, has to move to a position that allows it to start the cut, and then cut along the marked line. We provide the robot 5 kinesthetic demonstrations by positioning the end-effector and then following various marked straight lines. The state-space of the robot included the end-effector position (x, y)

as well as a visual feature indicating its pixel distance to the marked line (pix). This visual feature is constructed using OpenCV thresholding for the black line. Since the gauze is planar, the robot's actions are unit steps in the $\pm x, \pm y$ axes. Figure 3.34 illustrates the training and test scenarios.

SWIRL identifies two segments corresponding to the positioning step and the termination. The learned reward function for the position step minimizes x, y, pix distance to the starting point and for the cutting step the reward function is more heavily weighted to minimize the pix distance. We defined task success as positioning within 1 cm of the starting position of the line and during the following stage, missing the line by no more than 1 cm (estimated from pixel distance). We evaluated the model-free version of SWIRL, Q-Learning, and Behavioral Cloning with an SVM. SWIRL was the only technique able to achieve the combined task.

We evaluated the learned tracking policy to cut gauze. We ran trials on different sequences of curves and straight lines. Out of the 15 trials, 11 were successful. 2 failed due to SWIRL errors (tracking or position was imprecise) and 2 failed due to cutting errors (gauze deformed causing the task to fail). 1 of the failures was on the 4.5 cm curvature line and 3 were one the 3.5 cm curvature line.

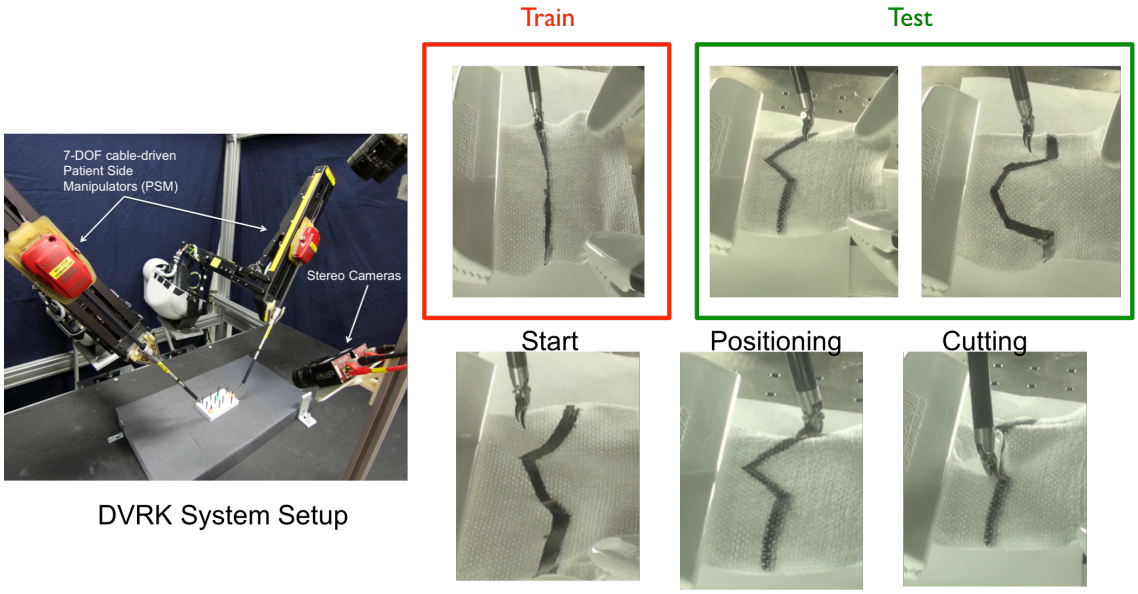


Figure 3.34: We collected demonstrations on the da Vinci surgical robot kinesthetically. The task was to cut a marked line on gauze. We demonstrated the location of the line without actually cutting it. The goal is to infer that the demonstrator's reward function has two steps: position at a start position before the line, and then following the line. We applied this same reward to curved lines that started in different positions.

Next, we characterized the repeatability of the learned policy. We applied SWIRL to lines of various curvature spanning from straight lines to a curvature radius of 1.5 cm. Table

Table 3.4: With 5 kinesthetic demonstrations of following marked straight lines on gauze, we applied SWIRL to learn to follow lines of various curvature. After 25 episodes of exploration, we evaluated the policies on the ability to position in the correct cutting location and track the line. We compare to SVM on each individual segment. SVM is comparably accurate on the straight line (training set) but does not generalize well to the curved lines.

Curvature Radius (cm)	SVM Pos. Error (cm)	SVM Tracking Error (cm)	SWIRL Pos. Error (cm)	SWIRL Tracking Error (cm)
straight	0.46	0.23	0.42	0.21
4.0	0.43	0.59	0.45	0.33
3.5	0.51	1.21	0.56	0.38
3.0	0.86	3.03	0.66	0.57
2.5	1.43	-	0.74	0.87
2.0	-	-	0.87	1.45
1.5	-	-	1.12	2.44

3.4 summarizes the results on lines of various curvature. While the SVM approach did not work on the combined task, we evaluated its accuracy on each individual step to illustrate the benefits of SWIRL. On following straight lines, SVM was comparable to SWIRL in terms of accuracy. However, as the lines become increasingly curved, SWIRL generalizes more robustly than the SVM. A single SVM has to learn both the positioning and cutting policies. The combined policy is much more complicated than the individual policies, e.g., go to a goal and follow a line.

Chapter 4

Alpha Clean: Reinforcement Learning For Data Cleaning

It is widely known that data cleaning is one of the most time-consuming steps of the data analysis process, and designing algorithms and systems to automate or partially automate data cleaning continues to be an active area of research [24]. Automation in data cleaning is challenging because real-world data is highly variable. A single data set can have many different types of data corruption such as statistical outliers, constraint violations, and duplicates. Once an error is detected, there is a further question of how to repair this error, which often depends on how the data will be used in the future.

This variability creates a tension between generality and efficiency. While one might want a single cleaning framework that addresses all types of errors and repair operations, it is far more efficient to consider more specialized frameworks. Data cleaning tools are often highly optimized for particular problems (e.g., see statistical outliers [46], enforcing logical constraints [24], entity resolution [64]). Consequently, a recent survey of industry suggests that data cleaning pipelines are often a patchwork of custom scripts and multiple specialized systems [68]. The overhead to setup, learn, and manage multiple cleaning systems can easily outweigh the benefits. Some data scientists eschew automated tools altogether and simply write data cleaning programs from scratch. The customized programming approach quickly becomes difficult to mantain and interpret; especially for users without data engineering expertise [104].

A single cleaning system that exposes a sufficiently flexible interface is clearly desirable; as long as the runtime and accuracy can be made comparable to existing alternatives. To achieve this, we turn to the contemporary AI literature, which has produced exciting results such as AlphaGo [109] that were previously considered not possible. This work shows that a combination of machine learning and massively parallelized search can effectively optimize highly complex objective functions. Our main observation is that data cleaning is very similar to the search problems considered in AI [99]. The classic application is a computer chess program that must plan a sequence of chess moves that change the state of

the board in order to maximize the likelihood of a checkmate. Likewise, in data cleaning, one is given a dirty relation and a way to measure data quality (e.g., number of integrity constraints violated), and the data cleaning problem is to find a sequence of modifications to the relation that maximize the data quality metric.

Most non-trivial problems are very hard without good search heuristics. Advances in machine learning have made it possible to replace hand-crafted heuristics with automatically learned pruning functions that estimate the expected quality of candidate plans. AlphaGo learned pruning heuristics using a neural network trained on data generated through self-play [109]. This insight is highly relevant to data cleaning, where datasets tend to have structure amenable to learning heuristics adaptively. Data errors are often systematic where they are correlated with with certain attributes and values in the dataset [69, 98]. Consequently, as more data is cleaned, we can better identify common patterns to prioritize the search on future data. We may not necessarily need a neural network for data cleaning, but the overall algorithmic structure of AlphaGo, tree-search combined with a learned heuristic, can be very useful.

AlphaClean is a new data cleaning system that is designed around a search-based model. It takes as input a *quality function* that models the data quality of a relation as a real-value between $[0, 1]$ and a *language* of parameterized data cleaning operators, and outputs a sequence of data cleaning transformations (a cleaning program) from the language that seeks to maximize the quality function. This API imposes minimal restrictions on the quality function, giving it tremendous flexibility in terms of the data errors that it can express. As a comparison, a recent system called Holoclean [98] is similar in that it provides a rich interface to specify denial constraints and lookup tables, and integrates them within a probabilistic model to more accurately resolve constraint violations. In contrast, AlphaClean's quality function is a user-defined function that can express *arbitrary combinations* of denial constraints *as well as* statistical, quantitative, text formatting, and other classes of data errors. For instance, Section 4.4 shows how AlphaClean can perform cleaning for machine learning applications by embedding machine learning training and evaluation within the quality function, while Section 4.4 combines entity resolution, outlier cleaning, and functional dependency violations within the same function.

AlphaClean uses a best-first search that greedily appends data transformations to the set of best candidate programs seen so far, and adopts parallelization and pruning ideas from the search-based planning literature. In contrast to traditional search problems, where the search state (e.g., chess board) is compact and largely trivial to parallelize in a distributed setting, the data cleaning search state is the size of the input dataset and introduces a trade-off between communication costs to share intermediate state and the degree of parallelism possible. To further accelerate its runtime, AlphaClean can also encode problem-specific optimizations as search pruning rules (e.g., disallowed transformation sequences) or modifications to the data representation (e.g., clustering similar records). We find that many optimizations in existing cleaning systems may be cast as search pruning rules.

AlphaClean can adaptively learn pruning rules to avoid unprofitable search branches during the search process. While AlphaGo used a deep neural network and massive amounts

of training data to model the heurisitic, we found that a simple logistic regression classifier and training data gathered during the search process was sufficient to reduce the runtime by multiple orders of magnitude. It is important to acknowledge that AlphaClean loses many of the provable guarantees provided by specialized systems based on logical constraints, and is in some sense, best-effort. Across 8 real-world datasets used in prior data cleaning literature, we show that AlphaClean matches or exceeds the cleaning accuracy and exhibits competitive run-times to state-of-the-art approaches that are specialized to a specific error domain (constraint, statistical, or quantitative errors).

- **Optimization:** Existing cleaning pipelines must combine disparate cleaning solutions and manage glue code to transfer data between systems. However, data transfer can be a non-trivial portion of the total cleaning cost and recent work [93] has shown that a common runtime can avoid data movement and improve runtimes by up to $30\times$.

- **Generalization and Robustness:** In contrast to existing cleaning systems, that output a cleaning relation, AlphaClean outputs a composable cleaning program that is both simpler to understand because it groups common fixes together, and can be applied to new data. In addition, the cleaning program allows us to analyze cleaning systems for overfitting. We indeed find that the complexity of the quality function and cleaning language can lead to overfitting, and believe this is the case for any data cleaning system. We also show that simple changes to the quality function can act as regularization terms to control the degree of overfitting.

- **Software Engineering:** Users do not need to write and manage glue code; do not need to manage ETL between systems; and do not need to learn system specific abstractions, languages, and assumptions. All of these are arguably intangible, but critical friction points in the highly iterative cleaning process.

- **New Cleaning Applications:** We envision that a single API makes it possible to support an ecosystem of domain-specific cleaning specification libraries. For instance, interactive visualization interfaces akin to [1, 54, 133, 134] can directly translate user interactions into a wide range of cleaning constraints.

4.1 Problem Setup

First, we overview the basic formalism of AlphaClean and present its relationship to related work. We focus on data transformations that concern a single relational table. Let R be a relation over a set of attributes A, $\mathcal{R}$ denote the set of all possible relations over A, and $r.a$ be the attribute value of $a \in A$ for row $r \in R$. A data transformation $T(R) : \mathcal{R} \mapsto \mathcal{R}$ maps an input relation instance $R \in \mathcal{R}$ to a new (possibly cleaner) instance $R' \in \mathcal{R}$ that is union compatible with R. For instance, "replace all `city` attribute values equal to *San Francisco*

with *SF*" may be one data transformation, while "delete the 10^{th} record" may be another. Aside from union compatibility, transformations are simply UDFs. Although it is desirable for the transformation to be deterministic and idempotent, if they are not, it simply makes the cleaning problem harder.

Data transformations can be composed using the binary operator $\circ$ as follows: $(T_i \circ T_j)(R) = T_i(T_j(R))$. The composition of one or more data transformations is called a *cleaning program* p. If $p = p' \circ T$, then p' is the parent of p; the parent of a single data transformation is a NOOP. In practice, users will specify *transformation templates* $\mathbb{T}(\theta_1, \cdots, \theta_k)$, and every assignment of the parameters represents one possible transformation. Although $\mathbb{T}$ can in theory be an arbitrary deterministic template function, our current implementation makes several simplifying assumptions to bound the number of data transformations that it can output. We assume that a parameter θ_i is typed as an attribute or a value. The former means that θ_i's domain is the set of attribute names in the relation schema; the latter means that θ_i's domain is the set of cell values found in the relation, or otherwise provided by the user.

Example 1 *The following* `City` *relation contains two attributes* city_name *and* city_code*. Suppose there is a one-to-one relationship between the two attributes. In this case, the relation is inconsistent with respect to the relationship and contains errors highlighted in red.*

	city_name	city_code
1	San Francisco	SF
2	**New York**	NY
3	New York City	**NYC**
4	**San Francisc**	SF
5	San Jose	SJ
6	San Mateo	SM
7	New York City	NY

The following transformation template uses three parameters: `attr` *specifies an attribute,* `srcstr` *specifies a source string, and* `targetstr` *specifies a target string.*

$$T = \textit{find\_replace}(srcstr, targetstr, attr)$$

The output of the above is a transformation T that finds all `attr` *values equal to* `srcstr` *and replaces those cells with* `targetstr`*. For instance,* `find_replace("NYC", "NY", "city_code")(City)` *returns a data transformation that finds records in* `City` *whose city_code is "NYC" and replaces their value with "NY".*

Let Σ be a set of distinct data transformations $\{T_1, \cdots, T_N\}$, and Σ^* be the set of all finite compositions of Σ, i.e., $T_i \circ T_j$. A formal language L over Σ is a subset of Σ^*. A program p is valid if it is an element of L.

Example 2 *Continuing Example 1, Σ is defined as all possible parameterizations of* `find_replace`*. Since many possible possible parameterizations are non-sensical (e.g., the source string does not exist in the relation), we may bound Σ to only source and target strings present in each attribute's instance domain (a standard assumption in other work as well [33]). In this case, there are 61 possible data transformations, and Σ^* defines any finite composition of these 61 transformations. The language L can be further restricted to compositions of up to k data transformations.*

Finally, let $Q(R) : \mathcal{R} \mapsto [0, 1]$ be a quality function where 1 implies that the instance R is clean, and a lower value correspond to a dirtier table. Since running a program $p \in \mathcal{L}$ on the initial dirty table R_{dirty} returns another table, $Q(p(R_{dirty}))$ returns a quality score for each program in the language. Q is a UDF and we do not impose any restrictions on it. In fact, one experiment embeds training and evaluating a machine learning model *within* the quality function (Section 4.4). Another experiment shows that AlphaClean can be robust to random noise injected in the function (Section 4.4).

Even so, we call out two special cases that provide optimization opportunities. We define two sub-classes of quality functions: row-separable and cell-separable quality functions. The former expresses the overall quality based on row-wise quality function $q(r) : R \mapsto [0, 1]$ where 0 implies that the record is clean:

$$Q(R) \propto \sum_{r \in R} q(r)$$

Similarly, a cell-separable quality function means that there exists a cell-wise quality function $q(r, a) : (R \times A) \mapsto [0, 1]$, such that the quality function is the sum of each cell's quality: $Q(R) \propto \sum_{r \in R} \sum_{a \in A} q(r, a)$.

These special cases are important because they can define hints on what types of transformations are irrelevant. For example, if the quality function is cell-separable, and we have identified that a set of cells C are dirty (e.g., they violate constraints), then we can ignore transformations that do not modify cells in C. This restricts the size the language and makes the problem much easier to solve.

We are now ready to present data cleaning as the following optimization problem:

Problem 3 (clean(Q, R_{dirty}, L)) *Given quality function Q, relation R_{dirty}, and language L, find valid program $p^* \in L$ that optimizes Q:*

$$p^* = \min_{p \in L} Q(p(R_{dirty})).$$

$p^*(R_{dirty})$ returns the cleaned table, and p^* can be applied to any table that is union compatible with R_{dirty}. A desirable property of this problem formulation is that it directly trades off runtime with cleaning accuracy and can be stopped at any time (though the cleaning program may be suboptimal). At the limit, AlphaClean simply explores L and identifies the optimal program.

Example 3 *Continuing Example 1, let us assume the following functional dependencies over the example relation:* city_name → city_code *and* city_code → city_name*. We can efficiently identify inconsistencies by finding the cities that map to* > 1 *city code, and vice versa. Let such city names and codes be denoted* D_{city_name} *and* D_{city_code}*, respectively.* $Q(R)$ *is a cell-separable quality function where the cell-wise quality function is defined as* $q(r, a) = 1-(r.a \in D_a)$*, such that* $r.a$ *is* 1 *if the attribute value does not violate a functional dependency, and* 0 *otherwise.*

By searching through all possible programs up to length 3 in L*, we can find a cleaning program based on* `find_replace` *that resolves all inconsistencies:*

```
find_replace(New York, New York City, city_name)
find_replace(San Francisc, San Francisco, city_name)
find_replace(NYC, NY, city_code)
```

Approach Overview and Challenges

Our problem formulation is a direct instance of *planning* in AI [99], where an agent identifies a sequence of actions to achieve a goal. In our setting, the agent (AlphaClean) explores a state space ($\mathcal{R}$) from an initial state (the input relation) by following transitions (applying $T_i \in \Sigma$) such that the sequence of actions is valid (within Σ^*) and the quality of the final state ($Q(R_{final})$) is maximized.

For readers familiar with stochastic processes, this search problem is equivalent to a deterministic Markov Decision Process (MDP), where the states are $\mathcal{R}$, the actions Σ, the transition function updates the instance with the transformation, the initial state is the dirty instance R_{dirty}, and the reward function is Q.

One may be hesitant in adopting our problem formulation because, although it is sufficiently general to model many existing data cleaning problems, such generality often comes at the expense of runtime performance. The planning problem is APX-Hard, meaning there does not exist a polynomial time approximation unless P=NP. Let R be a single-attribute relation of Booleans. Let L be the set of all assignments to a single value. Given a list of N Boolean clauses over all the boolean variables, let Q assign to each record one minus the fraction of clauses that evaluate to true. This formulation is equivalent to MAX-SAT and solution to the optimization problem.

For this reason, *the key technical challenge is to show that AlphaClean can solve data cleaning problems with comparable run-time as existing specialized systems, and can be easily extended to support new optimizations.* Despite the problem's worst-case complexity, recent successes in similar planning problems—ranging from AlphaGo [109] to automatically playing Atari video games [85] have shown that a prudent combination Machine Learning and distributed search can find practical solutions by leveraging the structure of the problem. Not every problem instance is as pathological as the worst case complexity suggests, and there are many reasonable local optima.

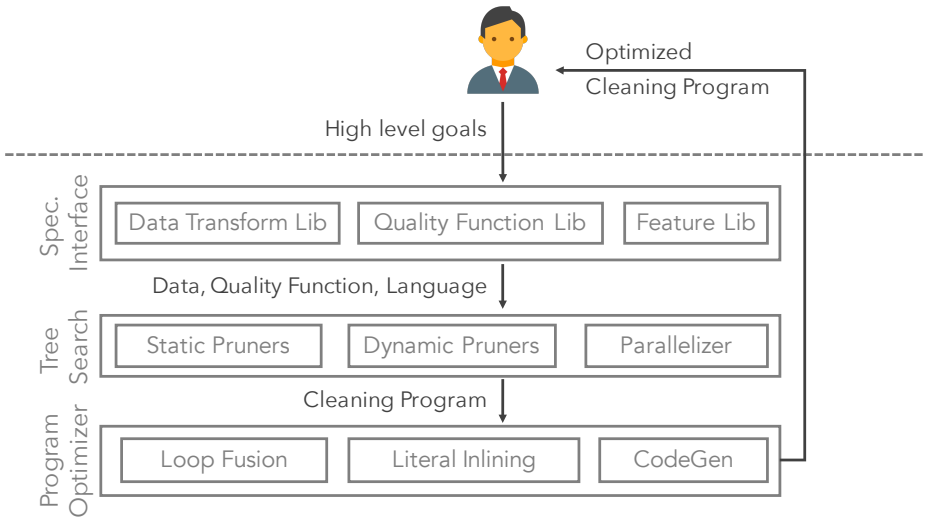


Figure 4.1: AlphaClean transforms high level cleaning goals (e.g., integrity constraints, statistical models, cleaning operations) into a quality measure and cleaning language, and uses an optimized tree search to find and optimize a sequence of transformations (cleaning program) to maximize the quality measure.

4.2 Architecture and API

This section describes the three major components of the AlphaClean system architecture (Figure 4.1), split between the user interface to specify the cleaning problem, the core search algorithm, and optimization of the resulting cleaning program. We detail the interface and program optimization in this section, and focus on the search algorithm and optimizations in the rest of the section. Designing data cleaning interfaces [43] and query optimization for data cleaning [38, 58] are interesting problems in their own right, and we believe AlphaClean can enable simpler and more powerful approaches. However, these were not a focus of the current study.

The *Specification Interface* contains extensible libraries of domain-specific quality functions, cleaning data transformations, and pruning feature hints that the user can use to define a high level cleaning goal. Our current implementation can translate a wide range of domain-specific goals—including type constraints, functional dependencies, denial constraints, lookup tables, parametric and non-parametric outlier models, and correlations between numeric attributes—into a single quality function. Users can also constrain the cleaning language, and we currently support any type of conditional cell and record transformation that applies a black-box transformation to all cells or records satisfying a predicate; both the transformation parameters and the predicate are learned by AlphaClean. Finally, users can optionally provide features that AlphaClean uses to dynamically learn pruning rules.

The *Search* component takes as input the quality function Q and language L, and per-

forms a greedy search heuristic to find a program $p^* \in L$ that maximizes Q. Users can supply hints that exploit the problem structure to reduce the runtime by pruning the search space and parallelizing the search. To bound the search space, users specify a maximum program length k. AlphaClean also supports three classes optimizations. *Static pruning* invalidates candidate programs based on the program structure (sequence of actions). For instance, composing the same idempotent transformation (e.g., `find_replace(SFO, SF, city_name)`) in succession is unnecessary. *Dynamic pruning* can access the result of a candidate program (search state) when making pruning decisions, and we propose a novel approach to learn automatic dynamic pruning rules that can reduce end-to-end runtimes by up to 75% at the expense of slightly lower recall. Finally, AlphaClean *parallelizes* the search in both shared memory and distributed settings. We describe the pruning rules and parallelization optimization in detail in the following text.

Finally, once search component outputs the cleaning program p^*, the *Program Optimizer* performs query compilation optimizations and provides an API to add new optimizations. AlphaClean currently replaces variables with literal values whenever possible, inlines data transformations into loops that scan over the input relation, and uses loop fusion [25, 93] to avoid unnecessary scans over the input and intermediate relations for each transformation in p^*. Consider the program in Example 3. Since the `find_replace` operations do not conflict, it is inefficient to loop through over the relation instance three separate times. Since they do not conflict, it would be inefficient to execute them sequentially and iterate over the data three separate times. Instead, they can be fused:

```
for r in rows:
 if r[city_name] == 'New York':
   r[city_name] = 'New York City'
 elif r[city_name] == 'San Francisc':
   r[city_name] = 'San Francisco'
 if r[city_code] == 'NYC'
   r[city_code] = 'NY'
```

These simple optimizations improve the final program runtime by up-to 20x, and we leave further improvements to future work (e.g., could be optimized with a framework like Weld [93]).

4.3 Search Algorithm

We now provide an overview of AlphaClean's search algorithm and optimizations.

Naive Search Procedures

In principle, any tree search algorithm over L would be correct. However, the traversal order and expansion policy is important in this search problem. We describe the algorithmic

Algorithm 1: Greedy Best-First Tree Search

Data: Q, R, Σ, L, (k, γ)

// Initialize priority queue of candidate programs
$P = \{NOOP\}$
while $|\{p \in P \mid p.len < k\}| > 0$ **do**
  for $p \in P : \|p\| < k$ **do**
    Pop p from the queue.
    for $T \in \Sigma$ **do**
      $p' = p \circ T$
      if $p' \in L$ **then**
        $P.push(p', Q(p'(R)))$
  $\bar{p} = \arg\max_{p \in P} Q(p(R))$
  $P = \{p \in P \mid p < \gamma \times Q(\bar{p}(R))\}$
return Highest priority item on the queue

Figure 4.2: Main Algorithmic Loop

and practical reasons why two naive procedures—breadth-first search (BFS) and depth-first search (DFS)—exhibit poor search runtimes.

BFS: This approach extends each program in the search frontier with every possible data transformation in Σ. To extend a candidate program l_c with $T \in \Sigma$, it evaluates $Q((T \circ l_c)(R))$. Unfortunately, the frontier grows exponentially with each iteration. Additionally, evaluating every new candidate program $T \circ l_c$ can be expensive if the input relation is large. Although the cost can be reduced by materializing $l_c(R)$, it is not possible to materialize all candidate programs in the frontier for all but the most trivial cleaning problems. It is desirable to use asearch procedure that bounds the size of the frontier and the materialization costs.

The first problem with this algorithm is that since each node in this tree o represents a sequence of transformations. Evaluating the value of o can be very expensive since it would have to evaluate the entire path to the root. o is a composition of many transformations and may require a number of passes over the dataset. This can be avoided if we can materialize (either to disk or memory) the frontier,that is, for each node in the priority queue $o \in O$, we have a cached result of $o(R)$. However, with BFS, the frontier is exponential in the support of the language and the system would quickly run out of memory.

DFS: Depth-first search only needs to materialize the intermediate results for a single program at a time, however it is highly inefficient since the vast majority of programs that it explores will have low quality scores.

Search Algorithm and Optimizations

Best-first search expands the most promising nodes chosen according to a specified cost function. We consider a greedy version of this algorithm, which removes nodes on the frontier that are more than γ times worse than the current best solution. Making γ smaller makes the algorithm asympotically consistent but uses more memory to store the frontier, whereas $\gamma = 1$ is a pure greedy search with minimal memory requirements.

The frontier is modeled as a priority queue P where the priority is the quality of the candidate program, and is initialized with a NOOP program with quality $Q(R)$. The algorithm iteratively extends all programs in the queue with less than k transformations; a program p is extended by composing it with a transformation T in Σ. If the resulting program p' is in the language L, then we add it to the queue. Finally, let $\bar{p}$ be the highest quality program in the queue. The algorithm removes all programs whose quality is $< \gamma \times Q(\bar{p}(R))$ from the frontier. This process repeats until the candidate programs cannot be improved, or all programs have k transformations.

In a naive and slow implementation, the above algorithm computes p's quality by fully running p on the input relation before running Q on the result, explores all possible data transformation sequences, and runs sequentially. One of the benefits of its simple structure is that it is amenable to a rich set of optimizations to prune the search space, incrementally compute quality functions, and parallelize the search. In fact, we find that many optimizations in existing specialized cleaning systems can be cast in terms of the following classes.

We can materialize (either to disk or memory) the frontier,that is, for each node in the priority queue $p \in P$, we have a cached result of $p(R)$. Then, when we expand the nodes to $p' = p \circ t$, we only have to incrementally evaluate $t(R)$. After the node is expanded, the result is added to the cache if it within γ of the best solution. The basic algorithm described above is well-suited for this problem. Without the greediness, the frontier might be exponentially large leading to an impractical amount of materialization. By tuning γ, the user can essentially set how much memory is used for materialization.

Static Pruning Rules are boolean functions that take a candidate program p as input and decides whether it should be pruned. AlphaClean currently models static rules as regular expressions over Σ. Static rules are can be viewed as filters over L.

$$\mathsf{static\_rule}(p) \mapsto \{0, 1\}$$

For example, since the find-and-replace operations are idempotent, i.e., $T(T(R)) = T(R)$, we may want to only consider the set of all sequences with no neighboring repeated transformations. Similarly, we may also want to prune all search branches that make no effect (i.e., find-and-replace New York with New York). These two regular expressions alone reduce the above example's language by 48% (from 226981 to 120050). Other rules, such as avoiding changes that undo previous changes $T^{-1}(T(R)) = R$, are similarly easy to add.

Dynamic Pruning Rules also have access to the input relation and quality function, and can make instance-specific pruning decisions.

$$\mathsf{dyn\_rule}(p, Q, R) \mapsto \{0, 1\}$$

For example, suppose Q is based on functional dependencies and is cell-separable, and we want to ensure that cell-level transformations made by a candidate program p individually improve Q. In this case, we find the cells C that initially violate the functional dependencies and ensure that the cells transformed by p are all in C. Applying this optimization, in addition to the others in AlphaClean, to the example reduces the search space by $143\times$ from 226,981 candidate programs to only 1582.

Since it can be challenging to hand-write pruning rules, there is a dynamic approach that uses simple machine learning models to automatically identify the characteristics of candidate programs to decide whether a particular search brach will be promising. In essence, it generates and refines static pruning rules during the search process.

For example, we may want to ensure that all the evaluations are "correlated" with the cost function–that is it makes modifications that are likely to affect the costs. This is possible if the cost separable where we have a score for each cell. In this case, we can find all the cells in violation of the functional dependencies and make sure that the "source" field of the find-and-replace operations only match values that are in violation. These optimizations are called "dynamic' because they can be determined from the active domain (i.e., after applying a transformation, recalculate new optimization rules). Applying this optimization (in addition to the others) to the example reduces the search space to 1582 evaluations v.s. 226981 unoptimized (143x reduction).

Block-wise Cleaning: A major cost is that independent errors in the relation must be cleaned sequentially in the search algorithm. For instance, records 2, 3, and 4 in the example exhibit independent errors and a fix for a given record does not affect the other records. Thus, if each record were cleaned in isolation, the search space would be $O(|\Sigma|)$. Unfortunately, the entire relation requires a program of three transformation to fix the records, which increases the search space to $O(|\Sigma|^3)$.

The main insight in block-wise cleaning is that many errors are local to a small number of records. In these cases, it is possible to partition R into a set of blocks $B_1, \cdots, B_m$, execute the search algorithm over each block independently, and concatenate their programs to generate the final cleaning program. This gather-scatter approach can exponentially reduce the search space for each block, and reduces the cost of evaluating super-linear quality functions that require e.g., computing a pair-wise similarity scores for the input relation. For example, quality functions derived from functional dependencies can define blocks by examining the violating tuples linked through the dependency. Similarly, users can define custom partitioning functions or learn them via e.g., clustering algorithms. In our current implementation, we partition the input relation by cell or row if the quality function is cell or row separable.

Parallel Program Evaluation: It is clear that the candidate programs can be evaluated and pruning in a parallel fashion across multiple cores and multiple machines, and is one of the major innovations in modern planning systems. However, unlike classic planning problems where communications are small due to the compact size of each state, AlphaClean's state size is determined by the relation instance, and can lead to prohibitively high communication costs and memory caching requirements. We describe how we manage the

trade-offs when parallelizing AlphaClean in shared memory and distributed settings.

Materialization: Since each candidate program $p' = p \circ T$ is the composition of a previous candidate program and a transformation, an obvious optimization is to materialize the output of $p(R)$ and incrementally compute $p'(R)$ as $T(p(R))$ over the materialized intermediate relation. Although this works well in a single threaded setting, memory and communication challenges arise when combining materialization with parallelization.

4.4 Experiments

Next, we present experimental results that suggest three main conclusions: (1) as a single framework, AlphaClean can achieve parity in terms of accuracy with state-of-the-art approaches to a variety of different problems ranging from integrity constraint satisfaction, statistical data cleaning, and also data cleaning for machine learning, (2) it is possible to significantly reduce the runtime gap between AlphaClean and specialized frameworks using simple pruning and distributed parallelization techniques, and (3) AlphaClean enables automatic data cleaning for datasets containing a mixture of error types in an acceptable amount of time.

Datasets and Cleaning Tasks

We list the main characteristics of the 8 experimental datasets.

Flight: The flight dataset [30] contains arrival time, departure time, and gate information aggregated from 3 airline websites (AA, UA, Continental), 8 airport websites (e.g., SFO, DEN), and 27 third-party websites. There are 1200 flight departures and arrivals at airline hubs recorded from each source. Each flight has a unique and globally consistent ID, and the task is to reconcile data from different sources using the functional dependency `ID`→ `arrival, departure, gate information`.

FEC: The election contributions dataset has 6,410,678 records, and 18 numerical, discrete, and text attributes. This dataset records the contribution amount and contributor demographic information e.g., name, address, and occupation. The task is to enforce `city`→`zipcode`, and match occupation to a codebook on canonical occupations. The quality function is 1 if the occupation is in the codebook, and 0 otherwise; we penalize the edit distance between the original and edited occupation values.

Malasakit: This dataset contains 1493 survey disaster preparedness responses from the Philippines, with 15 numeric and discrete attributes. The task removes improper numerical values and remove dummy test records. This consists of domain integrity constraints that force the values to be within a certain dictionary.

Physician: This dataset from Medicare.gov contains 37k US physican records, 10 attributes, with spelling errors in city names, zipcodes, and other text attributes. We use the data cleaning rules described in [98], which consists of 9 functional dependencies.

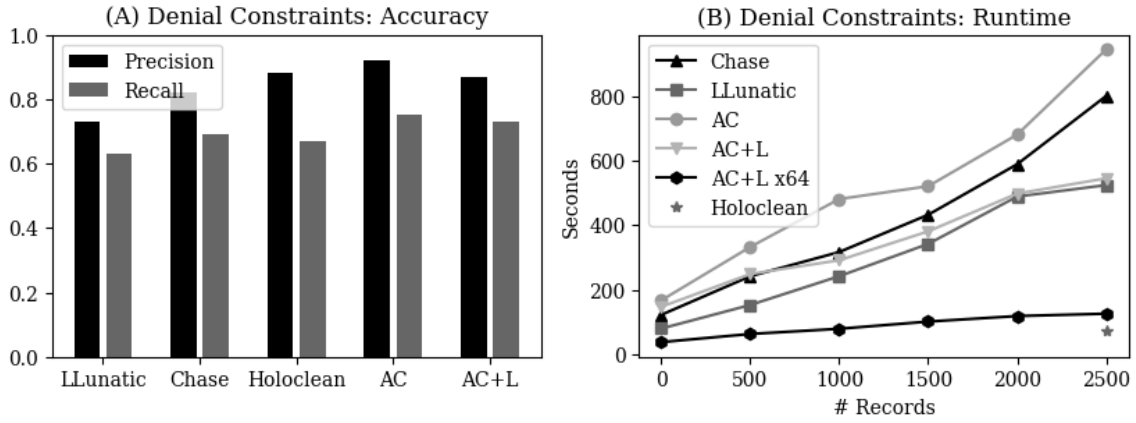


Figure 4.3: Comparison with denial constraint systems on the Flight dataset. (A) AlphaClean (AC) matches or exceeds the accuracy of the specialized systems. (B) The structure of the data errors, along the learning and parallelization (AC+L x64) lets AlphaClean scale sub-linearly and outperform all but HoloClean's reported runtimes.

Census: This US adult census data contains 32k records, and 15 numeric and discrete attributes. There are many missing values coded as 999999. The task is to clean numeric values to build a classification model that predicts whether an adult earns more than $\$50k$ annually.

EEG: The 2406 records are each a variable-length time-series of EEG readings (16 numeric attributes), and labeled as "Preictal" for pre-seizure and "Interictal" for non-seizure. The goal is to predict seizures based on 32 features computed as the mean and variance of the numeric EEG attributes. The task is to identify and remove outlier reading values.

Stock: There are 1000 ticker symbols from 55 sources for every trading day in a month [30]. The cleaning task involves (1) integrating the schemas by matching attributes across the sources (e.g., 'prev. close' vs 'Previous Close'), and then (2) reconciling daily ticker values values using primary-key-based functional dependencies akin to the flight dataset.

Terrorism: The Global Terrorism Database [35] is a dataset of around terrorist attacks scraped from news sources. Each record contains the date, location, details about the attack, and the number of fatalities and injuries. The dataset contains a mixture of data errors: (1) there are many duplicates for the same terrorist incident, (2) many missing values in the fatalities and injuries attributes are encoded as zeros, which overlaps with attacks that did not have any fatalities/injuries, and (3) the location attributes are inconsistently encoded. We used the dataset from 1970, and there are 170000 records. We downloaded this dataset and sought to understand whether terrorist attacks have become more lethal than they were in the 1970s. To do so, we hand cleaned the records to create a gold standard. It turns out that, in this dataset, attacks have become more lethal, but fewer in number than 50 years ago. This task was intentionally open-ended to represent the nature of the iterative analysis process.

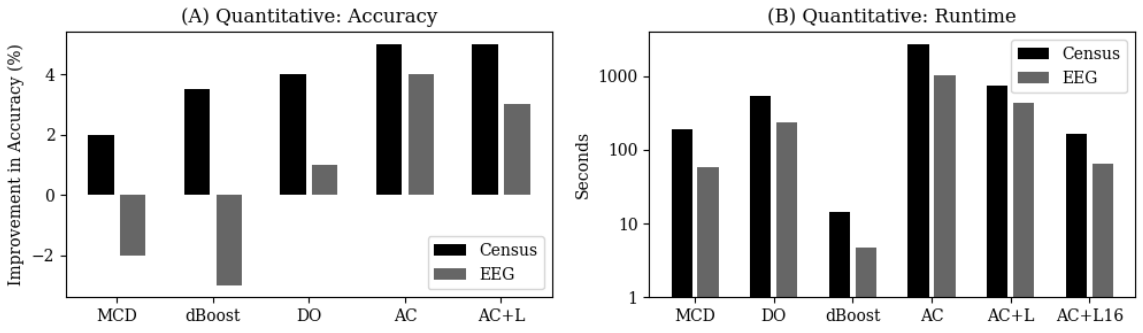


Figure 4.4: Quantitative data cleaning on census and EEG datasets for a classification application. AlphaClean tranformation can clip outliers or set them to a default value. (A) AlphaClean ha higher accuracy than outlier detection algorithms (MCD, dBoost), and AlphaClean with a single transform template (DO). (B) Optimizations improve AlphaClean runtime by over an order of magnitude.

Denial Constraints

Denial constraints express a wide range of integrity constraints and form the basis of many data cleaning systems. Although AlphaClean may not fully enforce integrity constraints, we can compose a quality function can quantifies the number of constraint violations and find transformation that reduce the number of violations. We use the Flight dataset for these experiments.

Baselines: We run against (1) Llunatic, a denial constraint-based cleaning system [26] implemented in C++ on top of PostgreSQL[1], and (2) a restricted chase algorithm [12] implemented in Python. We compare against the chase because a large portion of denial constraints are functional dependencies, and can be resolved using fixed-point iteration. We report numbers from the recent Holoclean publication [98] that used the same datasets and constraints, but did not run the experiment ourselves.

Results: Figure 4.3a shows the precision and recall of each approaches based on known ground truth. AlphaClean matches or beats the accuracy of the baselines, however its runtime (AC) without any learning scales poorly compared to alternatives (Figure 4.3b). Using learning (AC+L) shows performance on par with LLunatic, and parallelization on 64 threads is comparable to Holoclean's reported runtime. The results suggest that learning exhibits sublinear scaling due to AlphaClean learning more effective pruning rules as it sees more data. These performance gains are at the expense of slightly reduced accuracy.

We also evaluated AlphaClean (single threaded, without learning) on the FEC, Malasakit, and Physician datasets. Their precision, recall, and runtimes are as follows: FEC: 94% prec, 68% rec, 5hrs; Malasakit: 100% prec, 85% rec, 0.39hrs; Physician: 100% prec, 84%, 3.4hrs.

[1]Constraints are specified as Tuple-Generating Dependencies

Quantitative Data Cleaning

This experiment performs numerical cleaning on machine learning data. In these applications, prediction labels and test records are typically clean and available (e.g., results of a sales lead), whereas the training features are often integrated from disparate sources and exhibit considerable noise (e.g., outliers). Our quality function is simply defined as the model's accuracy on a training hold-out set, and we report the test accuracy on a separate test set.

We trained a binary classification random forest model using `sklearn` on the Census and EEG datasets. We used standard featurizers (hot-one encoding for categorical data, bag-of-words for string data, numerical data as is) similar to [42]. We split the dataset into 20% test and 80% training, and further split training into 20/80 into hold-out and training. We run the search over the training data, and evaluate the quality function using the hold-out. Final numbers are reported using the test data.

We defined the following three data transformation templates that sets numerical attribute values in R if they satsify a predicate:

- **clip_gt(attr, thresh):** $R.attr = thresh$ if $R.attr > thresh$
- **clip_lt(attr, thresh):** $R.attr = thresh$ if $R.attr < thresh$
- **default(attr, badval):** $R.attr$ set to mean val if $R.attr = badval$

Baselines: We compare with 4 baselines: *No Cleaning (NC)*, *Minimum Covariance Determinant (MCD)* is a robust outlier detection algorithm used in [7] and sets all detected outliers to the mean value, *dBoost* uses a fast partitioned histogram technique to detect outliers [79], and *Default Only (DO)* runs AlphaClean with only the default() transformation.

Results: The classifer achieves 82% and 79% accuracy on the uncleaned census and EEG data, respectively. Most outliers in the census data are far from the mean, so MCD and dBoost can effectively find. Further, setting census outliers to the mean is sensible. However, the same fix is not appropriate for the EEG data; it is better to clip the outlier values, thus MCD, dBoost, and DO have negligible or negative impact on accuracy. When we realized this from running DO, it was straightforward to add the clipping transformations to the language, and with no other changes, re-run AlphaClean with drastically improved EEG accuracies.

Vanilla AlphaClean (AC) is nearly $10\times$ slower than MCD, but the adding learning and 16-thread parallelization matches MCD's runtimes. dBoost is specialized for fast outlier detection and AlphaClean is unlikely to match its runtime.

Schema Integration

We now evaluate cleaning on the stock dataset [30] from 55 different sources that contains a mixture of schema integration and functional dependency errors.

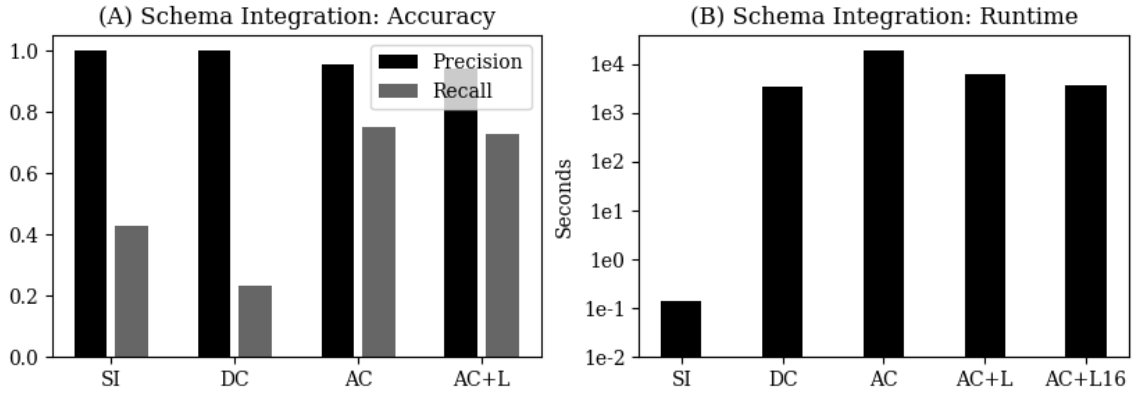


Figure 4.5: Schema integration and functional dependency errors in the stock dataset. Cleaning schema integration (SI) or functional dependencies (DC) in isolation results in high precision but poor recall. AlphaClean cleans both types of errors with higher recall, and is as fast as DC.

Baselines: We compare approaches for each error type in isolation. Schema Integration (SI) matches attributes based on attribute name string similarity, and Data Cleaning (DC) assumes schemas are consistent and enforces functions dependencies using a restricted chase [12].

Results: The specialized cleaning approaches exhibit high precision but low recall, as they miss records with multiple errors (Figure 4.5). AlphaClean can mix both tasks in the quality function and has comparable precision with much higher recall. SI is significantly faster since it only reads schema metadata, however AlphaClean with learning and 16 threads is competitive with DC.

Mixing Error Classes

This experiment, we use the multi-error Terrorism dataset [35].

Baselines: We hand-coded blocking-based entity matching (EM) and restricted chase (FD) algorithms in Python, and used dBoost for missing values. We also ran EM, dBoost, and Chase serially on the dataset (Combined).

Results: Figure 4.7a shows that combining and cleaning all three classes of errors within the same AlphaClean framework (AC) achieves higher precision and recall than all baselines. In fact, the combined baselines still does not achieve AlphaClean's level of accuracy because the cleaning operations need to be interleaved differently for different blocks. Although AlphaClean is slower than any individual baseline, parallelizing AlphaClean to 8 threads is faster than the combined baseline by 2x.

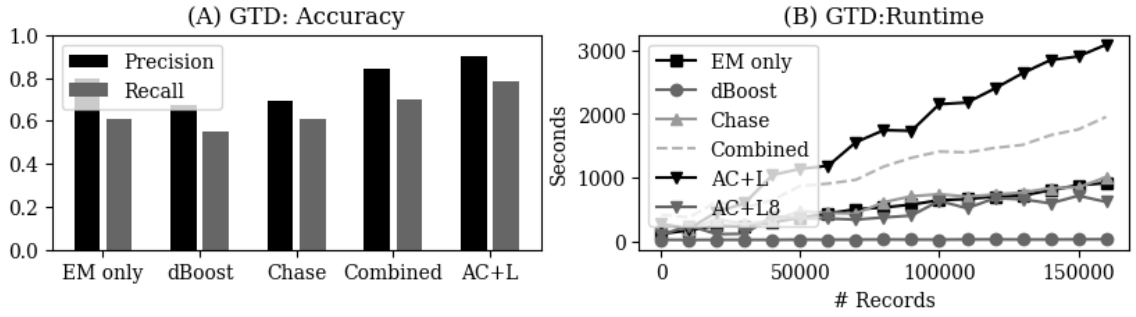


Figure 4.6: The Global Terrorism Database is a dataset of terrorist attacks scraped from news sources since 1970. (A) Shows that AlphaClean can integrate many different forms of cleaning that were previously handled by disparate systems, (B) AlphaClean achieves a competitive runtime to using all of the different system and accounting for data transfer time between them.

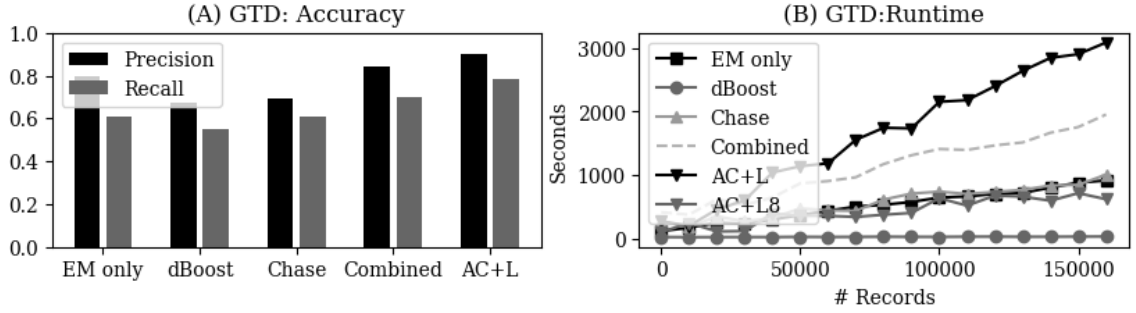


Figure 4.7: The terrorism dataset contains 3 classe of errors. (A) A unified cleaning framework outperforms individual cleaning approaches and even a serial combination. (B) AlphaClean is easily parallelized to 8 threads to outperform the combined baselines.

AlphaClean In Depth

This subsection uses the FEC setup to study the parameters that affect AlphaClean's accuracy and runtime, the robustness of its cleaning programs, and its algorithmic properties.

Algorithmic Sensitivity

Block-wise Cleaning: Partitioning the dataset into smaller blocks effectively reduces the problem complexity. This can have tremendous performance benefits when each block exhibits very few errors that are independent of the other blocks. Figure 4.8a shows the performance benefits when varying the block size; we define the blocks by partitioning on three different attributes that have different domain sizes. Reducing the block size directly improves the runtime; the search is effectively non-terminating when blocking is not used.

Language: Figure 4.8b fixes the input to a single block, and evaluates the runtime based

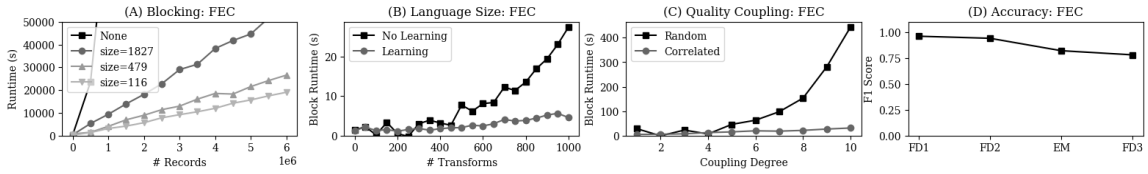


Figure 4.8: (A) The degree of block-wise partitioning directly affects search time, (B) increasing the transformation language Σ exponentially increases the search time, but learning is very effective at pruning the increased search space, (C) quality functions that couple errors between random records are significantly harder to optimize, (D) AlphaClean degrades gracefully when adding irrelevant error types to the problem.

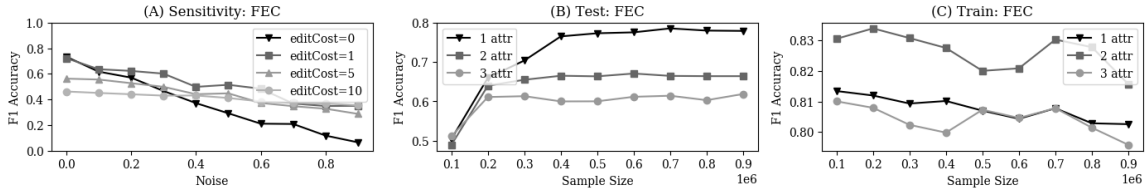


Figure 4.9: (A) Regularization by increasing an editCost penalty makes AlphaClean more robust to noisy (mis-specified) quality functions, (B-C) overly expressive transformation templates can lead to overfitting (2 attr) or an infeasibly large search space (3 attr).

on the size of the language $|\Sigma|$. Increasing the transformations increases the branching factor of the search problem. The search time is exponential in the language, however AlphaClean's learning optimization can identify a pruning model that reduces the runtime to linear.

Coupling in the Quality Function: The complexity of the quality function directly affects search time. A cell-separable quality function is the simplest to optimize because each cell in the relation can be analyzed and cleaned in isolation. In contrast, a quality function that couples multiple records together is more challenging to optimize because a longer sequence of transformation may be needed to sufficiently clean the records and improve the quality function.

We evaluate this by artificially coupling between 1-10 records together, and creating a quality function that only improves when an attribute of the coupled records all have the same value. We perform this coupling in two ways: *Random* couples randomly selected records, whereas *Correlated* first sorts the relation an attribute and couple records within a continuous range. We expect that the random coupling requires individual cleaning operations for each record based on their IDs, whereas the correlated setting both allows AlphaClean to exploit the correlated structure to learn effective pruning rules and to clean the coupled records using a single cleaning operation. Figure 4.8c shows that this is indeed the case when running AlphaClean on a single fixed-size block. *Random* slows down exponentially with increased coupling, whereas *Correlated* increases linearly.

Quality Function Complexity: Finally, we incrementally increase the quality function's

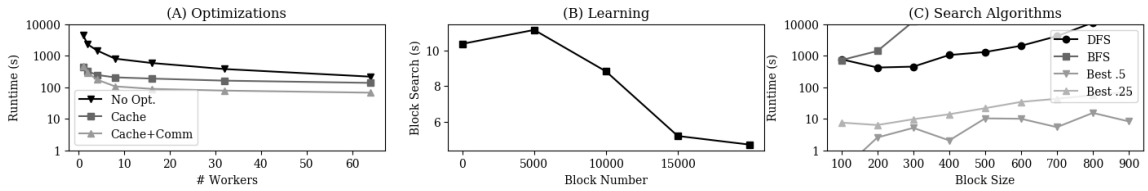


Figure 4.10: (A) Both the materialization (Cache) and distributed communication (Comm) optimizations contribute to improved scale-out runtimes. (B) The learned pruning rules improve the search costs for each subsequent block-wise partition. (C) Best-first search is better than BFS and DFS; reducing γ prunes more candidates at the expense of lower accuracy.

complexity and show haw it affects the cleaning accuracy. We add the following constraints in sequence: one functional dependency (FD), a second FD, an entity resolution similarity rule, and a third FD. We define the quality function as the sum of each constraint's quality function. Figure 4.8d shows that the F1 accuracy decreases as more constraints are added, however the F1 score is still above 75%.

Cleaning Generalization and Overfitting

An important characteristic of generating cleaning solutions as *programs* is that we can evaluate the program's robustness in terms of machine learning concepts such as overfitting and generalization. To this end, we examine two concepts in the context of data cleaning: regularizaton and overfitting. We also find that AlphaClean's high level interface is helpful for iteratively tuning the cleaning process.

Regularization: Misspecified quality functions can cause AlphaClean to output poorly performing cleaning programs. We simulate this by adding random noise to the output of the quality function. Figure 4.9A plots the F1-score of AlphaClean on the FEC experiment while varying the amount of noise. As expected, the output program's F1-score rapidly degrades as the noise increases.

Machine learning uses regularizing penalty terms to prevent overfitting. We can similarly add a penalty to the quality function to prevent too many edits. Each line line shows the edit cost penalty and shows that although the F1 is lower when there is no noise, AlphaClean is more robust to larger amounts of noise.

Overfitting: In machine learning, over-parameterized models may be susceptible to overfitting. A similar property is possible if the language Σ is overly expressive. We use a transformation template that finds records matching a parameterized predicate and sets an attribute to another value in its instance domain. We then vary the language expressiveness by increasing number of attributes in the predicate between 1 and 3. Finally, we run AlphaClean on a training sample of the dataset (x-axis), and report F1 accuracy on the training and a separate test sample (Figure 4.9B-C). Note that overfitting occurs when the training accuracy is significantly higher than test accuracy.

Indeed we find an interesting trade-off. The 1 attribute predicate performed worst on

the training sample but outperformed the alternatives on the test sample. The 2 attribute predicate was more expressive and overfit to the training data. Finally, the 3 attribute predicate is overly expressive and computationally difficult to search. Thus, it did not sufficiently explore the search space to reliably identify high quality cleaning programs.

Discussion: We have shown that data cleaning can overfit, and believe this is a potential issue in *any* cleaning procedure. These results highlight the importance of domain experts to judge and constrain the cleaning problem in ways that will likely generalize to future data. Further, it shows the value of a high-level interface that experts can use to express these contraints by iteratively tuning the quality function and cleaning language.

Scaling

Next, we present preliminary results illustrating the scaling properties of AlphaClean.

Parallelization Optimizations: The experiments run on a cluster of 4 mx.large EC2 instances, and we treat each worker in a distributed (not shared-memory) fashion. Figure 4.10a shows the benefits of the materialization and communication optimizations in. *No opt.* simply runs best-first search in parallel without any materialization; workers only synchronize at the end of an iteration by sending their top-γ candidate programs to the driver, which prunes and redistributes the candidates in the next iteration. *Cache* extends No opt by locally materializing parent programs, and *Cache+Comm* further adds the communication optimizations for distributed parallelization.

The single threaded No Opt setting runs in 4432s, and the materialization optimization reduces the runtime by $10\times$ to 432s. Scaling out improves all methods: at 64 workers, *Cache+Comm* takes 67s while *Cache* takes 137s. Surprisingly, although *No Opt* with 64 workers is slower than *Cache+Comm* by $10\times$, it scales the best because it only synchronizes at the end of an iteration and only communicates candidate programs and their quality values. In contrast, the alternative methods may communicate materialized relation instances.

Within-Block Learning: Although we have shown how learning reduce the overall search runtime, we show that learning also improves the search speed for individual blocks. We run single-threaded AlphaClean and report the time to evaluate each block. Figure 4.10b shows the i^{th} block that is processed on the x-axis, and the time to process it on the y-axis. We see that as more blocks are cleaned, the learned pruning classifier is more effective at pruning implausible candidate programs. This reduces the per-block search time by up to 75%.

Search Algorithm Choice: Figure 4.10c shows that best-first search out-performs naive depth and breadth first search. We also report AlphaClean when $\gamma = \{0.5, 0.25\}$. We see that as the block size increases, DFS and BFS quickly become infeasible, whereas AlphaClean runs orders of magnitude more quickly. In addition, reducing γ improves the runtime, however can come at the cost of reduced accuracy by pruning locally sub-optimal but globally optimal candidate programs.

Program Structure

Finally, we present results describing the structure of the data cleaning programs found with AlphaClean. It is often the case that the program found by AlphaClean is a concise description of the needed data cleaning operations, that is, the total number of cell edits is much larger than the length of the program. We consider the FEC dataset, the EEG dataset, and GTD dataset.

Sometimes, the program (FEC and GTD) encodes a significant amount of literal values. This happens in entity matching type problems. For these problems, the program length is relatively large, however, the number of cells modified is even larger (up to 10x more). For datasets like the EEG dataset, the program is very concise (26x smaller than the number of cells modified). Numerical thresholds are generalize better than categorical find-and-replace operations.

	Program Length	Cells Modified
FEC	6416	78342
EEG	6	161
GTD	1014	104992

Chapter 5

Reinforcement Learning For SQL Query Optimization

Joins are a ubiquitous primitive in data analytics. The rise of new tools, such as Tableau, that automatically generate queries has led to renewed interest in efficiently executing SQL queries with several hundred joined relations. Since most relational database management systems support only dyadic join operators (joining two tables) as primitive operations, a query optimizer must choose the "best" sequence of two-way joins to achieve the k-way join of tables requested by a query.

5.1 Problem Setup

The problem of optimally nesting joins is, of course, combinatorial and is known to be NP-Hard. To make matters worse, the problem is also known to be practically challenging since a poor solution can execute several orders of magnitude slower than an optimal one. Thus, many database systems favor "exact" solutions when the number of relations are small but switch to approximations after a certain point. For example, PostgreSQL uses dynamic programming when the query has less than 12 relations and then switches to a genetic algorithm for larger queries. Similarly, IBM DB2 uses dynamic programming and switches to a greedy strategy when the number of relations grows.

Traditionally, exact optimization is addressed with a form of sequential dynamic programming. The optimizer incrementally builds optimal joins on subsets of relations and memoizes the optimal join strategy as well as the cost-to-go based on an internal cost model. This dynamic programming algorithm still enumerates all possible join sequences but shares computation where possible. Natually, this enumeration process is sensitive when the underlying cost models in the RDBMS are inaccurate. Care has to be taken to avoid enumerating plans that are risky, or high variance, if table sizes or system parameters are not known exactly. Some work has considered incorporating feedback to the cost model after execution to improve the model for future optimization instances.

As it stands there are three challenges that modern join optimizers try to simultaneously address: (1) pruning a large space of possible plans, (2) hedging against uncertainty, and (3) feedback from query evaluation. For each of these three problems, the community has proposed numerous algorithms, approximations, and heuristics to address them. Often the dominant techniques are sensitive to both the particular workload and the data. The engineering complexity of implementing robust solutions to all of these challenges can be very significant, and commerical solutions are often incomplete—where some level of query engineering (by creating views and hints) from the developer is needed.

Recent advances in Artificial Intelligence may provide an unexpected perspective on this impasse. The emergence of deep reinforcement learning, e.g., AlphaGo, has presented a pragmatic solution to many hard Markov Decision Processes (MDPs). In an MDP, there is a decision making agent who makes a sequence of decisions to effect change on a system that processes these decisions and updates the system's internal state (potentially non-deterministically). MDPs are Markovian in the sense that the system's current state and the agent's decision completely determines any future evolution. The solution to an MDP is a decision to make at every possible state. We show that the join optimization can be posed as an MDP where the state is the join graph and decisions are edge contractions on this graph.

Given this formulation, the Deep Q Network (DQN) algorithm can be applied, and in essence, this formulation gives us an approximate dynamic programming algorithm. Instead of exactly memoizing subplans in a table with cost-to-go estimates as in dynamic programming, DQN represents this table with a neural network of a fixed size. This allows the algorithm to "estimate" the cost-to-go of subplans even if they have not been previously enumerated.

This learning based approach gives us flexibility in designing new intelligent enumeration strategies by manipulating how the neural network is trained and how it represents the subplans. For example, the neural network allows us to efficiently share query processing information across planning instances with the neural network parameters and learn from previously executed queries. The consequence is enumeration strategies tuned to the specific workload and data. We can also vary the features used to represent the subplans, to allow the network to capture different properties like interesting orders and physical operator selection. We can also train the neural network with observations of actual query execution times and make through repeated executions make it robust to uncertainty.

In short, Deep RL provides a new algorithmic framework for thinking about join enumeration. We can architect an entire query processing stack around this Deep RL framework. We explore this architecture and evaluate in what situations its behavior is more desirable (either in terms of cost or planning time) compared to classical approaches. Our system is built on Apache Calcite and integrates with Apache Spark, Postgres SQL, and an internal join planning simulator. We present results on a variety of experimental workloads and datasets.

5.2 Background

First, we will introduce the algorithmic connection between Reinforcement Learning and the join optimization problem.

Query Model

Consider the following query model. Let $\{R_1, ..., R_T\}$ define a set of relations, and let $\mathcal{R} = R_1 \times ... \times R_T$ denote the cartesian product. We define an inner join query q as a subset of $q \subseteq \mathcal{R}$ as defined by a conjunctive predicate $\rho_1 \wedge ... \wedge \rho_j$ where each expression ρ is binary boolean function of the attributes of two relations in the set $\rho = R_i.a\{=, != , >, <\}R_j.b)$. We assume that the number of conjunctive expressions is capped at some fixed size $\mathcal{N}$. We further assume that the join operation is commutative, left associative, and right associative. Extending our work to consider left and right outer joins is very straight-forward but we will defer that to that to future work.

We will use the following database of three relations denoting employee salries as a running example throughout the chapter:

$$\text{Emp}(id, name, rank)$$

$$\text{Pos}(rank, title, code)$$

$$\text{Sal}(code, amount)$$

Consider the following join query:

```
SELECT *
 FROM Emp, Pos, Sal
 WHERE Emp.rank = Pos.rank AND
 Pos.code = Sal.code
```

In this schema, $\mathcal{R}$ denotes the set of tuples $\{(e \in Emp, p \in Pos, s \in Sal)\}$. There are two predicates $\rho_1 = Emp.rank = Pos.rank$ and $\rho_2 = Pos.code = Sal.code$, combined with a conjunction. One has several possible options on how to execute that query. For example, one could execute the query as $Emp \bowtie (Sal \bowtie Pos)$. Or, one could execute the query as $Sal \bowtie (Emp \bowtie Pos)$.

Graph Model of Enumeration

Enumerating the set of all possible dyadic join plans can be expressed as operations on a graph representing the join relationships requested by a query.

Definition 5 (Query Graph) *Let G define an undirected graph called the* query graph*, where each relation R is a vertex and each ρ defines an edge between vertices. The number of connected components of G are denoted by κ_G.*

Each possible dyadic join is equivalent to a combinatorial operation called a graph contraction.

Definition 6 (Contraction) *Let $G = (V, E)$ be a query graph with V defining the set of relations and E defining the edges from the join predicates. A contraction c is a function of the graph parametrized by a tuple of vertices $c = (v_i, v_j)$. Applying c to the graph G defines a new graph with the following properties: (1) v_i and v_j are removed from V, (2) a new vertex v_{ij} is added to V, and (3) the edges of v_{ij} are the union of the edges incident to v_i and v_j.*

Each contraction reduces the number of vertices by 1. And, that every feasible dyadic join plan can be described as a sequence of such contractions $c_1 \circ c_2 ... \circ c_T$ until $|V| = \kappa_G$. Going back to our running example, suppose we start with a query graph consisting of the vertices (Emp, Pos, Sal). Let the first contraction be $c_1 = (Emp, Pos)$, this leads to a query graph where the new vertices are $(Emp + Pos, Sal)$. Applying the only remaining possible contraction, we arrive at a single remaining vertex $Sal + (Emp + Pos)$ corresponding to the join plan $Sal \bowtie (Emp \bowtie Pos)$.

The join optimization is to find the best possible one of these contraction sequences. Assume that we have access to a cost model J, which is a function that can estimate the incremental cost of a particular contraction $J(c) \mapsto \mathbb{R}_+$.

Problem 4 (Join Optimization Problem) *Let G define a query graph and J define a cost model. Find a sequence of $c_1 \circ c_2 ... \circ c_T$ terminating in $|V| = \kappa_G$ to minimize:*

$$\min_{c_1,...,c_T} \sum_{i=1}^{T} J(c_i)$$

Note how the "Principle of Optimality" arises in Problem 4. Applying each contraction generates a subproblem of a smaller size (a query graph with one less vertex). Each of these subproblems must be solved optimally in any optimal solution. Also note that this model can capture physical operator selection as well. The set of allowed contractions can be typed with an eligible join type e.g., $c = (v_i, v_j, \text{hash})$ or $c = (v_i, v_j, \text{index})$.

Greedy vs. Optimal

A *greedy* solution to this problem is to optimize each c_i independently. The algorithm proceeds as follows: (1) start with the query graph, (2) find the lowest cost contraction, (3) update the query graph and repeat until only one vertex is left. The greedy algorithm, of course, does not consider how local decision might affect future costs. Consider our running example query with the following costs (assume symmetry):

$$J(EP) = 100,\ J(SP) = 90,\ J((EP)S) = 10,\ J((SP)E) = 50$$

The greedy solution would result in a cost of 140 (because it neglects the future effects of a decision), while the optimal solution has a cost of 110. However, there is an upside, this greedy algorithm has a computational complexity of $O(|V|^3)$—despite the super-exponential search space.

The decision at each index needs to consider the long-term value of its actions where one might have to sacrifice a short term benefit for a long term payoff. Consider the following way of expressing the optimization problem in the problem statement for a particular query graph G:

$$V(G) = \min_{c_1,\ldots,c_T} \sum_{i=1}^{T} J(c_i) \tag{3}$$

We can concisely describe Equation 3 as the function $V(G)$, i.e., given the initial graph G, what is the value of acting optimally till the end of the decision horizon. This function is conveniently named the *value function*. Bellman's "Principle of Optimality" noted that optimal behavior over an entire decision horizon implies optimal behavior from any starting index $t > 1$ as well, which is the basis for the idea of dynamic programming. So, $V(G)$ can be then defined recursively for any subsequent graph G' generated by future contractions:

$$V(G) = \min_{c}\{\ J(c) + \gamma \cdot V(G')\ \} \tag{4}$$

Usually, write this value recursion in the following form:

$$Q(G, c) = J(c) + \gamma \cdot V(G')$$

Leading to the following recursive definition of the Q-Function (or cost-to-go function):

$$Q(G, c) = J(c) + \gamma \cdot \min_{c'} Q(G', c') \tag{5}$$

The Q-Function describes the long-term value of each contraction. That means at each G' local optimization of $\min_{c'} Q(G', c')$ is sufficient to derive an optimal sequence of decisions. Put another way, the Q-function is a hypothetical cost function on which greedy descent is optimal and equivalent to solving the original problem. If we revisit the greedy algorithm, and revise it as follows: (1) start with the query graph, (2) find the lowest **Q-value** contraction, (3) update the query graph and repeat. This algorithm has a computational complexity of $O(|V|^3)$ (just like the greedy algorithm) but is provably optimal.

The Q-function is implicitly stored as a table in classical dynamic programming approaches, such as the System R enumeration algorithm. In the System R algorithm, there is a hash table that maps sets of joined relations to their lowest-cost access path:

```
HashMap<Set<Relation>, (Plan, Long)> bestJoins;
```

As the algorithm enumerates more subplans, if a particular relation set exists in the table it replaces the access path if the enumerated plan has a lower cost than one in the table. We could equivalently write a different hash table that moves around the elements on the table described above. Instead of a set of relations mapping to a plan and a cost, we could consider a set of relations *and* a plan mapping to a cost:

```
HashMap<(Set<Relation>, Plan), Long> bestQJoins;
```

Learning the Q-Function

This change would be less efficient implementation in classical dynamic programming, but it crucially allows us to setup a machine learning problem. What if we could regress from features of `(Set<Relation>, Plan)` to a cost based on a small number of observations? Instead of a Q-Function as a table, it is parametrized as a model:

$$Q_\theta(f_G, f_c) \approx Q(G, c)$$

where f_G is a feature vector representing the query graph and f_c is a feature vector representing the particular contraction on the graph. θ defines the neural network parameters that represent this function.

The basic strategy to generate observational data, that is, real executions of join plans, and optimize θ to best explain the observations. Such an optimization problem is a key motivation of a general class of algorithms called Reinforcement Learning [121], where statistical machine learning techniques are used to approximate optimal behavior while observing substantially less data that full enumeration.

For those familiar with the AI literature, this problem defines a Markov Decision Process. G is exactly a representation of the **state** and c is a representation of the **action**. The utility function (or reward) of this process is the negative overall runtime. The objective of the planning problem is to find a policy, which is a map from a query graph to the best possible join c.

In the popular Q-Learning approach [121], the algorithm enumerates random samples of decision sequences containing $(G, c, runtime, G')$ tuples forming a trajectory. From these tuples, one can calculate the following value:

$$y_i = runtime + \arg\max_u Q_\theta(G', c)$$

Each of the y_i can be used to define a loss function since if Q were the true Q function, then the following recurrence would hold:

$$Q(G, c) = runtime + \arg\max_u Q_\theta(G', c)$$

So, Q-Learning defines a loss:

$$L(Q) = \sum_i \|y_i - Q_\theta(G, c)\|_2^2$$

This loss can be optimized with gradient descent. This algorithm is called the Deep Q Network algorithm [85] and was used to learn how to autonomously play Atari Games. The key implication is that the neural network allows for some ability for the optimizer to extroplate the cost-to-go even for plans that are not enumerated. This means that if the featurization f_G and f_c are designed in a sufficiently general way, then the neural network can represent cost-to-go estimates across an entire workload–not just a single planning instance.

5.3 Learning to Optimize

Next, we present the entire optimization algorithm for learning join enumeration. We will first presented the algorithm only for the enumeration problem, and then we contextualize the optimizer for full Select-Project-Join queries accounting for single relation selections, projections, and sort-orders.

Featurizing the Join Decision

We need to featurize the query graph G and a particular contraction c, which is a tuple of two vertices from the graph.

Participating Relations: The first step is to construct a set of features to represent which relations are participating the in the query and in the particular contraction. Let A be the set of all attributes in the database (e.g., $\{Emp.id, Pos.rank, ..., Sal.code, Sal.amount\}$). Each relation r (including intermediate relations that are the result of join) has a set of *visible attributes*; those attributes present in the output $A_r \subseteq A$. So every query graph G can be represented by its visible attributes A_G. Each contraction is a tuple of two relations (l, r) and we can get the visible attributes A_l and A_r for each. Each of these subsets A_G, A_l, A_r can be represented with a binary 1-hot encoding representing which subset of the attributes are present. We call the concatenation of these vectors V_{rel} and it has dimensionality of $3 \cdot |A|$.

Join Condition: The next piece is to featurize the join condition, or the predicate that defines the join. Participating relations define vertices and the predicate defines an edge. As before, let A be the set of all attributes in the database. Each expression has two attributes and an operator. As with featurizing the vertices we can 1-hot encode the attributes present. We additionally have to 1-hot encode the binary operator $\{=, != , <, >\}$. Concatenating all of these binary vectors together for each expression ρ there there is a binary feature vector f_ρ. For each of the expressions in the conjunctive predicate, we concatenate the binary feature vectors. Here is where the fixed size assumption is used. Since the maximum number of expressions in the conjunction capped at $\mathcal{N}$, we can get a fixed sized feature vector for all predicates. We call this feature vector V_{cond} and it has dimensionality of $\mathcal{N} \cdot (|A| + 4)$.

Representing the Q-Function: The Q-function is represented as a multi-layer perceptron (MLP) neural network. It takes as input two feature vectors V_{rel} and V_{cond}. Experimentally, we found that a two-layer MLP gave the best performance relative to the training time. We implmented the training algorithm in DL4J a java framework for model training with a standard DQN algorithm.

Generating the Training Data

Experimentally, we found that the process of generating the process of training data for a given workload was very important for robust learning. The basic challenge is that the Q-function must accurately be able to differentiate between good plans and bad plans. If the training data only consisted of optimal plans, then the learned Q-function may not accurately score poor plans. Likewise, if the training purely sampled random plans–it may not see very many instance of good plans. We want an efficient algorithm that avoids exhaustive enumeration to sample a variety of very good plans and poor plans.

The data generation algorithm, which we denote as $\mathsf{sample}(G, \epsilon)$, takes as input a query graph G and a randomization parameter $\epsilon \in [0, 1]$.

$\mathsf{sample}(G, \epsilon)$:

1. **if** $\kappa_G = |G|$ return.
2. For all graph contractions c_i on G:.
   (a) $G_i = c_i(G)$
   (b) $V(c_i, G_i) = \mathsf{leftDeep}(G_i)$
3. **if** rand() > ϵ: yield $c_j, G_j = \arg\max_{c_i} V(c_i, G_i)$, return $\mathsf{sample}(G_j, \epsilon)$
4. **else:** yield uniform random c_j, G_j, return $\mathsf{sample}(G_j, \epsilon)$

The algorithm essentially acts as the Q-Function algorithm described in the previous section. The algorithm proceeds as follows: (1) start with the query graph, (2) find the lowest cost contraction, (3) update the query graph and repeat. The lowest cost contraction is determined by assuming the remaining joins will be optimized with a left-deep strategy. Let leftDeep(G) be the function that calculates the final cost of the best left deep plan given the query graph G. To implement leftDeep(G), we use the System R dynamic program. This acts as an approximation to the Q function where the current step is locally optimal but future joins are efficiently approximate.

To sample some number of poor plans in the training dataset, the ϵ parameter controls the number of randomly selected decisions. This trades off reasonable optimality during training vs. covering the space of plans. In this sense, we base the data generation algorithm in a particular *workload*. The workload is distribution over join queries that we want to optimize. We have an optimizer that samples some number of initial queries from the workload and is tested on unseen data.

Execution after Training

After training, we will have a parametrized estimate of the Q-function, $Q_\theta(f_G, f_c)$. For execution, we simply go back to the standard algorithm as in the greedy method but instead of using the local costs, we use the learned Q-function: (1) start with the query graph, (2) featurize each contraction (2) find the lowest **estimated Q-value** contraction, (3) update the query graph and repeat.

5.4 Reduction factor learning

Classical approaches to reduction factor estimation include (1) histograms and heuristics-based analytical formulae, and (2) applying the predicate under estimation to sampled data, among others. In this section we explore the use of learning in reduction factor estimation. We train a neural network to learn the underlying data distributions of a pre-generated database, and evaluate the network on unseen, randomly selections that query the same database.

To gather training data, we randomly generate a database of several relations, as well as random queries each consisting of one predicate of the form "R.attr $\langle$op$\rangle$ literal". For numeric columns, the operands are $\{=, \neq, >, \geqslant, <, \leqslant\}$, whereas for string columns we restrict them to equality and inequalities. Each attribute's values are drawn from a weighted Gaussian.

To featurize each selection, we similarly use 1-hot encodings of the participant attribute and of the operand. Numeric literals are then directly included in the feature vector, whereas for strings, we embed "hash(string_literal) % B" where B is a parameter controlling the number of hash buckets. The labels are the true reduction factors by executing the queries on the generated database.

A fully-connected neural network is used[1], which is trained by stochastic gradient descent.

5.5 Optimizer Architecture

In the previous section, we described the join enumeration algorithm. Now, we contextualize the enumeration algorithm in a more fully featured optimization stack. In particular, there are aspects of the featurization and graph definitions that have to change based on the nature of the queries.

Selections and Projections

To handle single relation selections and projections in the query, we have to tweak the feature representation. This is because upstream selections and projections will change the cost properties of downstream joins. As in the classical optimizers, we eagerly apply selections and projections to each relation.

This means that each relation in the query graph potentially a different number of attributes and a different cardinality than the base relation. While the proposed featurization does capture the visible attributes, it does not capture changes in cardinality due to upstream selections.

Here, we leverage the table statistics present in most RDBMS. For each relation in the query graph, we can estimate a reduction factor δ_r, which is an estimate of the fraction of

[1] Two hidden layers, 256 units each, with ReLU activation.

tuples present after applying the selection to relation r. In the featurization described in the previous section, we have a set of binary features V_{rel} that describe the participating attributes. We multiply the reduction factors δ_r for each table with the features corresponding to attributes derived from that relation.

Physical Operator Selection

To add support for an optimizer that selects physical operators, we simply have to add "labeled" contractions, where certain physical operators are eligible. Suppose we have a set of possible physical operators, e.g., nestedLoop, sortMerge, indexLoop. We would simply add an additional feature to the Q-Function that captures which physical operator is selected.

Indexes and Sort Orders

Similarly, adding support for indexes just means adding more features. We eagerly use index scans on single relation selections, and can use them for joins if we are optimizing physical operators. We can add an addition set of binary features V_{ind} that indicate which attributes have indexes built on them. Handling, sort-orders are similar and we have to add features describing which attributes need to be finally sorted in the query.

Summary and System Architecture

Surprisingly, we found that we could build a relatively full featured query optimizer based on a Deep RL join enumeration strategy. Our system, called RL-QOPT, is built on Apache Calcite. The system connects to various database engines through a JDBC connector. It executes logically optimized queries by re-writing SQL expressions and setting hints. The system parses standard SQL and the learning steps are implemented using DL4J. The optimizer has two modes, training and execution. In the training mode, the optimizer will collect data and based on a user set parameter execute suboptimal plans. In the execution mode, the optimizer will leverage the trained model to improve its optimizer performance.

5.6 Experiments

We evaluate this framework on a standard join optimization benchmark called the Join Order Benchmark. This benchmark is derived from the Internet Movie Data Base (IMDB). It contains information about movies and related facts about actors, directors, production companies, etc. The dataset is 3.6 GB large and consists of 21 relational tables. The largest table has 36 M rows. The benchmark contains 33 queries which have between 3 and 16 joins, with an average of 8 joins per query.

Evaluation Methodology

We first evaluate RL-QOPT against 3 different cost models for the workload and data in the join order benchmark. Each of these cost models is designed to elicit a different set of optimal plans. We show that, as a learning optimizer, RL-QOPT adapts to the different search spaces efficiently.

CM1: In the first cost model, we model a main-memory database that performs two types of joins: index nested-loop joins and in-memory hash joins. Let O_l be the left operator and O_r be the right operator the costs are defined as follows:

$$\mathsf{c}_{inlj} = \mathsf{c}(O_l) + \mathsf{rf}(O_l, O_r) \cdot |O_l|$$

$$\mathsf{c}_{hj} = \mathsf{c}(O_l) + \mathsf{c}(O_r)$$

where c denotes the cost estimation function and rf denotes the estimated reduction factor of the join. As we can see, this model favors index-based joins when available. The reduction factor $\mathsf{rf}(O_l, O_r)$ is always less than 1. More importantly, this cost model is a justification for the use of left-deep plans. In c_{inlj}, the right operator does not incur a scan cost. A left deep tree where the indexed relations are on the right exploits this structure.

CM2: In the next cost model, we model a database that accounts for disk-memory relationships in the hash joins. We designate the left operator as the "build" operator and the right operator as the "probe" operator. If the previous join has already built a hash table on an attribute of interest, then the hash join does not incur another cost.

$$\mathsf{c}_{nobuild} = \mathsf{c}(O_r)$$

This model favors right-deep plans where to maximize the reuse of the built hash tables.

CM3: Finally, in the next cost model, we model temporary tables and memory capacity constraints. There is a budget of tuples that can fit in memory and an additional physical operator that allows for materialization of a join result if memory exists. Then, the downstream cost of reading from a materialized operator is 0.

$$\mathsf{c}(O) = 0 \text{ if materialized}$$

This model requires bushy plans due to the inherent non-linearity of the cost function and memory constraints. The cost model encourages plans that group tables together in ways that the join output can fit in the available memory.

Join Order Benchmark

We train RL-QUOPT on 90 Queries and hold out 23 queries from the worklaod. In the initial experiment, we assume perfect selectivity estimates of single-table predicates. We compare three optimizers as baselines: left-deep, right-deep, and bushy (exhaustive). Figure 5.1 shows the aggregate results for the entire workload over random partitions of the

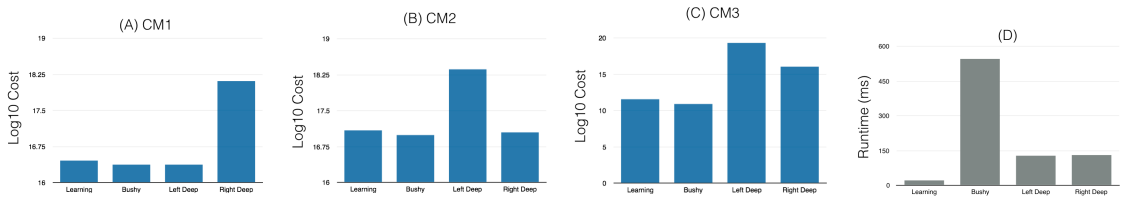


Figure 5.1: (A) The log cost of the different optimizers with CM1, (B) the log cost of the different optimizers with CM2, (C) the log cost of the different optimizers with CM3, and (D) the runtime of the different optimizers.

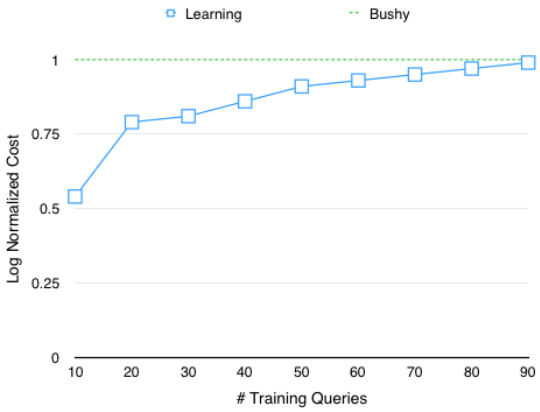


Figure 5.2: The learning curve as a function of the number of queries. The log normalized cost describes the cost difference between learning and the exhaustive bushy optimizer.

data. We present results for each of the cost models. While the bushy optimizer works well in terms of cost in all three regimes, it is very expensive to run. The left-deep and the right-deep optimizers are far quicker, but require the prior understanding of costs in the database. We see that in each of the different cost models a different optimizer (left-deep or right-deep) dominates in terms of performance. On the other hand, the learning optimizer nearly matches the bushy performance in all of the regimes while retaining a very fast run time.

For CM1, we plot the learning curve as a function of the number of training queries (Figure 5.2). We actually find that even with 50 training queries, we can consistently find plans competitive with exhaustive search. This suggests that the network is not simply memorizing and is generalizing to unseen subplans.

TPC-H

The TPC-H Benchmark is a dataset and workload of SQL queries. It consists of ad-hoc queries and concurrent data modifications. The queries and the data populating the database are inspired by those seen in industry. Unlike the JOB where there is fixed number of

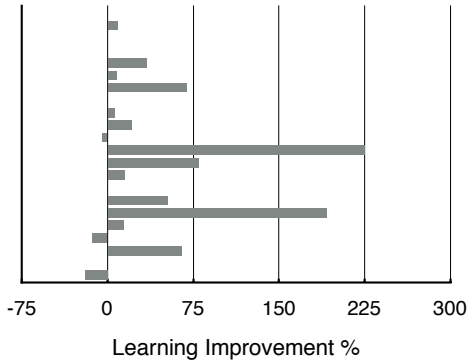


Figure 5.3: The performance improvement of learning over the postgres query optimizer for each of the TPCH template queries.

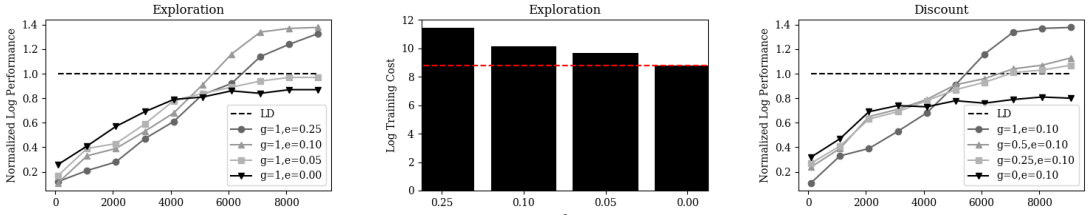


Figure 5.4: How exploration and discount parameters affect learning performance.

queries, TPCH contains a query generator that generates an arbitrary number of queries from templates. In this experiment, we learn the selectivity estimates from data as well.

Using our Apache Calcite connection, we also execute these queries on a real postgres database. Unlike the JOB, these are not idealized cost model. We train the RL algorithm with real runtimes. We first generate 10k training queries for the RL algorithm. Then, we evaluate the RL algorithm on a 100k queries from the generator. We aggregate results by query template. We find that after 10k queries, RL-QUOPT significantly improves on the postgres optimizer on several of the template (Figure 5.3).

The real execution experiment also raises questions about training data collection. During data collection the learner has to execute suboptimal query plans; this could make collecting data very expensive. We evaluate this tradeoff in Figure 5.4. As the exploration parameter ϵ increases, the query plans executed during training are increasingly suboptimal. However, the learning optimizer has to see a sufficient number of "bad plans" to learn an effective optimization policy. Similarly, the discount parameter also affected training performance. Lower discount settings lead to greedier cost attribution–and actually faster training.

Conclusion

Reinforcement learning introduces a new approximate dynamic programming framework for SQL query optimization. Classical query optimizers leverage dynamic programs for optimally nesting join queries. This process creates a table that memoizes cost-to-go estimates of intermediate subplans. By representing the memoization table with a neural network, the optimizer can estimate the cost-to-go of even previously unseen plans allowing for a vast expansion of the search space. I show that this process is a form of Deep Q-Learning where the state is a query graph and the actions are contractions on the query graph. One key result is that the same optimization algorithm can adaptively learn search strategies for both in-memory databases (where the search space is often heuristically restricted based to maximize index accesses) and disk-based databases (where the search space is often heuristically restricted to maximize re-use of intermediate results).

Chapter 6

Reinforcement Learning for Surgical Tensioning

Robotic surgical assistants (RSAs), such as Intuitive Surgical's da Vinci, facilitate precise minimally invasive surgery [129]. These robots currently operate under pure tele-operation control, but introducing assistive autonomy has potential to improve surgical training, assist surgeons, reduce fatigue, and facilitate tele-surgery. For example, when cutting thin tissue with surgical scissors, a surgeon may use a second or even a third tool to pin down and fixture the tissue. This technique, called tensioning (also called traction), adds additional constraints on the material to prevent deformations during the cutting from drastically changing the position of the desired cutting path. The optimal direction and magnitude of the tensioning force changes as the cutting progresses, and these forces must adapt to any deformations that occur. However, practically, the surgeon's tensioning policy is often sub-optimal because: (1) surgeon can automatically manipulate only two of the tools at once leaving any third arm stationary, and (2) asymmetric multilateral tasks (arms doing different procedures) are known to be challenging without significant training with the tele-operative system.

Therefore, it would be beneficial to automatically synthesize tensioning policies to best assist an open loop cutting trajectory. The first challenge is modeling the dynamics of cutting and tensioning. One way to model deformable sheets is with a 3-dimensional mass-spring-damper network. A sheet is a planar graph of point masses connected by damped springs at edges. k of the point masses are constrained to be fixed at a particular 3D location, i.e., they are tensioned, and the positions of the remaining masses is determined by the dynamics induced these constraints. At each time step, the constraint can be moved to a new location. Cutting is modeled by removing a single edge in the graph at each time-step. For k assistive arms, the optimization objective is to plan sequence of k movable boundary value constraints to maximize cutting accuracy.

This problem constitutes a highly non-convex optimal control problem, whether cutting accuracy is a non-convex objective that measures the symmetric difference between a

perfectly cut contour and the actual cut. Furthermore, the underlying deformable manipulation system is also highly non-linear. Efficient policy search techniques are required if we want to effectively assist a human surgeon seamlessly during a procedure. What is different about this problem is that we do not have access to "expert demonstrations" as in the previous chapters. Instead, we show that we can get an approximate suboptimal controller using physics-based approximations and then use Deep RL to fine-tune this controller. The archite

6.1 Motivation

Nienhuys and Van Der Stappen proposed the seminal work on modeling cutting with finite-element mesh models [90]. This work led to a bevy of follow-on work in robotics and graphics modeling the cutting problem, especially in the context of surgical simulation [14, 83, 108, 114]. In parallel, the graphics community studied fabric modeling with similar finite-element techniques [17, 37]. In our prior work, we implemented such an approach for the Pattern Cutting task in the Fundamentals of Laparoscopic Surgery (FLS) [123]. This paper is inspired by the challenges noticed in our prior work, namely, that state-estimation of a deformable object when there are occlusions is very challenging. Therefore, we investigate whether synthesizing a tensioning plan *a priori* can mitigate this problem.

Manipulation of deformable materials, particularly cutting, is a challenging area of research interest in robotic surgery [87,90] as well as in computer graphics and computational geometry [23, 136]. The use of expert demonstrations has been considered in prior work as an alternative to explicit models and simulations when studying and handling deformations in the environment. For example, Van den Berg et al. [128], Osa et al. [92], and Schulman et al. [102] all approached manipulation of suture material using Learning From Demonstrations (LfD), an approach that uses demonstration trajectories to learn how to execute specific tasks. RL has been a popular control method in robotics when dynamics are unknown or uncertain [59]. There are a few examples of RL applied to deformable object manipulation, e.g. folding [8] and making pancakes [10]. The recent marriage between RL and Neural Networks (Deep RL) opened a number of new opportunities for control in non-linear dynamical systems [76]. An obstacle to applying Deep RL to physical robots is that large amounts of data are required for training, which makes sufficient collection difficult if not impossible using physical robotic systems–leading to our study of simulation-based learning.

6.2 Physics of Multi-Dimensional Spring Systems

As an introduction, we first describe the physics of a multi-dimensional spring system. This will provide the basic intuition for the deformable sheet model in the next section. Figure 6.1 illustrates the most basic multi-dimensional spring system. A point mass m is

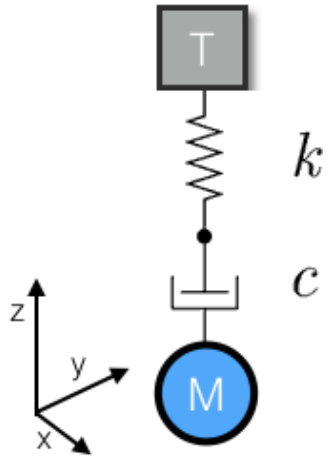


Figure 6.1: This illustration describes a basic mass-spring-damper system with spring constant k and damping constant c.

connected by a spring and damper to a infinitely strong block T. The spring constant is k (let ℓ denote the resting length) and the damping constant is c. Let denote the acceleration vector $\mathbf{a} = [\ddot{x} \;\; \ddot{y} \;\; \ddot{z}]^T$ of the point mass:

$$m\mathbf{a} = F_{spring} + F_{damper}.$$

Hooke's law states that the magnitude of the force applied by a spring is proportional to its deviation D from the rest length:

$$|F_{spring}| = kD.$$

Using unit vector notation, we can parametrize the force vector with spherical coordinates:

$$F_{spring} = \mathbf{i}\, k(D-\ell)\sin\theta\cos\psi + \mathbf{j}\, k(D-\ell)\sin\theta\sin\psi + \mathbf{k}\, k(D-\ell)\cos\theta$$

where $D = \sqrt{(x-T_x)^2 + (y-T_y)^2 + (z-T_z)^2}$, $\theta = \cos^{-1}\frac{(z-T_z)}{D}$, $\psi = \tan^{-1}\frac{(y-T_y)}{(x-T_x)}$. For the damper, let $\mathbf{v} = [\dot{x} \;\; \dot{y} \;\; \dot{z}]^T$ denote the velocity vector:

$$F_{damper} = c\mathbf{v}.$$

The resulting equations of motion are:

$$m\mathbf{a} = \mathbf{i}\,(k(D-\ell)\sin\theta\cos\psi + c\mathbf{v}_x) + \mathbf{j}\,(k(D-\ell)\sin\theta\sin\psi + c\mathbf{v}_y) + \mathbf{k}\,(k(D-\ell)\cos\theta + c\mathbf{v}_z). \quad (6)$$

Simulator

We can use this model to design a simulator for cutting deformable sheets. Let $G_{x,y,z} \subset \mathbf{R}^3$ be a three-dimensional global coordinate frame. We denote the set of points on this sheet

as Σ, and $\Sigma^{(G)}$ is the locations of these points in the global frame. The points are connected by a graph of springs and dampers. To apply the above equation of motion, we can treat each neighboring vertex as a wall. For each $p \in \Sigma$, the neighboring vertex $q \in N(p)$ applies a force F_{pq}. Cutting is modeled as removing an edge from the graph. In this paper, we assume that all of the springs have the same constant, natural length, and all the damping constants are the same.

The simulator is initialized with some initial state $\Sigma_0^{(G)} \in G_{x,y,z}$. The state is then iteratively updated based on the derived equations of motion above. For each $p \in \Sigma$, the updates have the following form:

$$m\ddot{p} = \sum_{q \in N(p)} F_{pq} + F_{external} \tag{7}$$

To update the position p, we use the implicit Adams method with a standard python toolkit [1]. Tensioning is simulated as a position constraint for a chosen pinch point $p' \in \Sigma$:

$$p' = u$$

This means that regardless of the forces applied to this point it will remain at position u.

Manipulation Actions

We are given a rectangular planar sheet and a simple algebraic desired cutting contour of bounded curvature, which can be either closed or open.

Cutting

We assume that one arm of the robot is designated as the cutting arm and the other as the tensioning arm. A cutting contour is a sequence of points C on the surface of the sheet. The cutting arm operates in an open-loop trajectory that attempts to cut along C_0, the position of the cutting contour in the global frame at time zero. Error is measured using the symmetric difference between the desired contour on the sheet and the achieved contour cut. These will be different due to deformation of the sheet during cutting. Let X be the set of points inside a closed intended trajectory and let Y be the set of points inside a closed simulated trajectory. The symmetric difference is then the exclusive-or $A \oplus B$ of the two sets. For open contours, the contours are closed by forming boundaries with the edges of the sheet.

Tensioning

Since the cutting is open-loop it cannot account for deformation, and this is why we need tensioning to apply feedback based on the state of the sheet.

[1] https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.integrate.ode.html

Definition 7 (Tensioning) *Let $s \in \Sigma$ be called a pinch point. Tensioning is defined as constraining the position of this pinch point to a specific location $u \in G_{x,y,z}$:*

$$T = \langle s, u \rangle$$

For each of the k tensioning arms of the robot, we can have one tuple T_i. We consider a single pinch point for each arm for an entire cutting trajectory. This allows us to define a tensioning policy:

Definition 8 (Tensioning Policy) *For arm $i \in \{1, .., k\}$, let $\Sigma_t^{(G)}$ be the locations of all of the points on the sheet in the global coordinate frame at time t. For a fixed pinch point s, a tensioning policy π_s is a function where $\Delta_u = u_{t+1} - u_t$:*

$$\pi_i : \Sigma^{(G)}(t) \mapsto \Delta_u$$

Problem. Tensioning Policy Search: For each arm i, find a tensioning policy that minimizes the symmetric difference of the desired vs. actual contour.

6.3 Reinforcement Learning For Policy Search

We chose an RL algorithm since the symmetric difference reward function is non-convex and the system in non-linear. We model the tensioning problem as a Markov Decision Process (MDP):

$$\langle S, A, \xi(\cdot, \cdot), R(\cdot, \cdot), T \rangle.$$

where the actions A are $1mm$ movements of the tensioning arm in the x and y directions, and the states S are described below. The action space is tuned so the policy can generate sufficient tension to manipulate the cloth significantly over a few timesteps. Reward is measured using the symmetric difference between the desired contour and the achieved contour cut. The robot receives 0 reward at all time-steps prior to the last step, and at the last time-step $T - 1$ receives the symmetric difference. We do not shape the reward, as symmetric difference is exactly the error metric used for evaluation as well.

To optimize θ, we leverage the TRPO implementation [103] in Rllab [32]. We use a neural network to parametrize the policy π_θ, which maps an observation vector to a discrete action. A two 32x32 Hidden Layer Multi-Layer Perceptron is used to represent this mapping. Since neural networks are differentiable, we can optimize the quantity $R(\theta)$. The state space is a tuple consisting of the time index of the trajectory t, the displacement vector from the original pinch point u_t, and the location $x_i \in \mathbb{R}^3$ of fiducial points chosen randomly on the surface of the sheet. In all experiments we use 12 fiducial points. This is a sample-based approximation of tracking $\Sigma_t^{(G)}$.

6.4 Approximate Solution

Directly applying the RL algorithm to the FEM simulator has a two problems: (1) large sample complexity and (2) local minima. Even in an optimized simulator, every new contour required 5 minutes of learning before a viable policy was found. The key insight was that ultimately the simulator used in RL was based on an analytic model. Thus, we explored whether we could initialize learning with a prior developed from a simplified objective.

Small Deviation Approximation

The equation of motion defines a non-linear system. We are, however, modeling point masses on a sheet that will have relatively small deformations from the resting length. We further assume that there is no damping $c = 0$ or external forces. For linearization, it is more convenient to work in Cartesian coordinates, so we can also re-parametrize the equations using Pythagorean identities:

$$ma_x = k\Delta_x - k\frac{\ell\Delta_x}{\sqrt{\Delta_x^2+\Delta_y^2+\Delta_z^2}}$$

$$ma_y = k\Delta_y - k\frac{\ell\Delta_y}{\sqrt{\Delta_x^2+\Delta_y^2+\Delta_z^2}}$$

$$ma_z = k\Delta_z - k\frac{\ell\Delta_z}{\sqrt{\Delta_x^2+\Delta_y^2+\Delta_z^2}}$$

To linearize, we can first take the gradient:

$$\nabla(ma_x) = k\begin{bmatrix} 1 - \frac{\ell(\Delta_y^2+\Delta_z^2)}{(\sqrt{\Delta_x^2+\Delta_y^2+\Delta_z^2})^{\frac{3}{2}}} \\ \frac{\ell\Delta_x\Delta_y}{(\sqrt{\Delta_x^2+\Delta_y^2+\Delta_z^2})^{\frac{3}{2}}} \\ \frac{\ell\Delta_x\Delta_z}{(\sqrt{\Delta_x^2+\Delta_y^2+\Delta_z^2})^{\frac{3}{2}}} \end{bmatrix}$$

We linearize around the operating point $\Delta_x, \Delta_y, \Delta_z = \rho = \frac{\ell}{\sqrt{3}}$, and let $\lambda = \frac{1}{3}\ell^{\frac{3}{2}}$:

$$ma_x \approx k\begin{bmatrix}1-2\lambda\\ \lambda\\ \lambda\end{bmatrix}^T \cdot \begin{bmatrix}(\Delta_x-\rho)\\(\Delta_y-\rho)\\(\Delta_z-\rho)\end{bmatrix}, ma_y \approx k\begin{bmatrix}\lambda\\ 1-2\lambda\\ \lambda\end{bmatrix}^T \cdot \begin{bmatrix}(\Delta_x-\rho)\\(\Delta_y-\rho)\\(\Delta_z-\rho)\end{bmatrix}, ma_z \approx k\begin{bmatrix}\lambda\\ \lambda\\ 1-2\lambda\end{bmatrix}^T \cdot \begin{bmatrix}(\Delta_x-\rho)\\(\Delta_y-\rho)\\(\Delta_z-\rho)\end{bmatrix}$$

We can define some notation to make the future algebra more concise:

$$\mathbf{L} = \frac{k}{m}\begin{bmatrix}1-2\lambda & \lambda & \lambda\\ \lambda & 1-2\lambda & \lambda\\ \lambda & \lambda & 1-2\lambda\end{bmatrix}$$

The resulting system can be described in the state-space model for a position of a point mass p:

$$\ddot{p} = \mathbf{L}(\sum_{q\in N(p)}(p-q) - \rho) \tag{8}$$

Tensioning Problem

The key trick to understanding tensioning based on this linearization is an efficient computation of the equilibrium state, i.e., $\ddot{p} = 0$. Let $X \in \mathbb{R}^{N\times 3}$ denote the positional state of each of the masses. For N masses and E edges, let A be an $E \times N$ matrix where every $A_{ij} = 0$ if edge i is not incident to the mass j, $A_{ij} = \pm 1$ for edges incident to the mass (pushing or pulling, can be selected arbitrarily). And, finally, let R be an $\mathbb{R}^{E\times 3}$ matrix where each component is ρ, or the natural length of the spring. It can be shown that the acceleration of all of the masses is:

$$\ddot{\mathbf{P}} = A^T AXL - A^T RL$$

making the equilibrium condition:

$$A^T AX = A^T R$$

To enforce constraints, we simply have to enforce that some subset of the components of X are fixed. This restricts the equilibrium solution to the subspace of values for which $X_{i,:} = u_i$. These are exactly the tensioning constraints, and we can solve the system of equations by partitioning the masses into free and tensioned sets.

First, we can re-write the above equation as:

$$BX = A^T R$$

which can in turn be written as:

$$B_{free}X_{free} = A^T R + B_{tens}U$$

and solving for the least squares solution

$$X_{free} = (B_{free}^T B_{free})^{-1}[B_{free}^T A^T R + B_{free}^T B_{tens}U]$$

This expression is just an affine function of the tensioning constraints:

$$X_{free} = CU + D$$

Optimization

To find tensioning directions and magnitudes, we have to pose an optimization problem to set the values of U as the grid is cut. The model and its equilibrium states give us a way to quantify deformation induced by cutting. C and D depend on the structure of the graph at time t.

We cannot directly optimize for symmetric difference so instead, we optimize for minimizing total deformation from the original state:

$$I_t = \|X_{free}[0] - X_{free}[t+1]\|_2^2$$

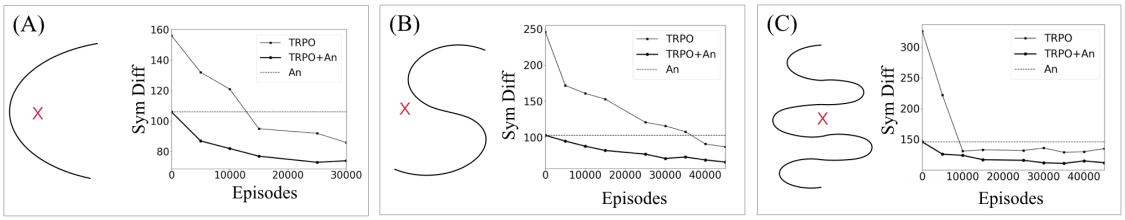


Figure 6.2: This plot shows the expected reward over 50 trials for TRPO, TRPO initialized with the analytic approximation (TRPO AN), and the analytic model on its own. Three cutting curves are visualized of increasing difficulty with a single pinch point marked in red. Results suggest that initializing with an analytic model can greatly accelerate learning.

This measures the amount of change in the position of the points after a cut and the sheet has settled into a new equilibrium state. This sets up our control objective for synthesizing a set of tensions from a fixed pinch point.

$$\min_{u_1,\ldots,u_T} \sum_{t=0}^{T-1} I_t + q\|u_{t+1} - u_t\|_2^2 \tag{9}$$

where $q\|u_{t+1} - u_t\|_2^2$ is a control penalty on changing the tensioning between time-steps for $q > 0$.

This problem is a convex program which can be solved with standard solvers. Once this open-loop strategy is learned, we can initialize TRPO by training the policy network to predict u_t from the state. Out intuition is that this analytic model gets close to the optimal policy and TRPO simply has to refine it. This mitigates the effects of bad local minima far from the optimum and slow convergence.

6.5 Experiments

Performance

We present a set of illustrative initial experiments in this paper. In the first experiment, we generated three contours of increasing difficulty and learned tensioning policies for a single pinch point $k = 1$. We compare TRPO with no initialization, TPRO+An with the analytic initialization, and An which is just the analytic model. Figure 6.2 plots the learning curves. We stopped training when the reward was no longer reliably decreasing. We find that in all three cases, the analytic initialization significantly reduces the time needed to learn a similarly successful policy. Furthermore, the analytic model is within 25% of the final reward of TRPO achieves indicating that it is a very good initialization.

The next experiment evaluates the run time of the algorithms as we increase the number of tensioning arms. Figure 6.3A measures the number of episodes needed for TRPO to crossover, i.e., match the performance of the analytic method, as a function of the number

of tensioning arms to plan for. While the analytic method is greedy, TRPO requires nearly 350000 episodes before it is at parity with this method for 4 tensioning arms. For comparison, the analytic optimization requires two orders of magnitude less time to reach the same result (Figure 6.3B). And, we explore using the combination of the two to achieve higher accuracy results with less rollouts.

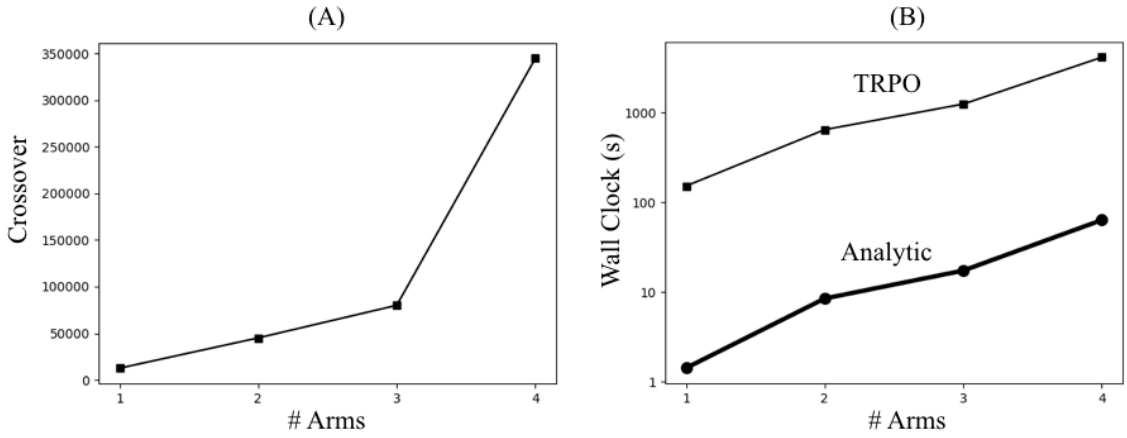


Figure 6.3: (A) We measure the number of episodes needed for TRPO to crossover, i.e., match the performance of the analytic method, as a function of the number of tensioning arms to plan for. (B) For comparison, we plot the wall clock time of TRPO and the analytic optimization on a log scale. The optimization is order of magnitudes faster, but may be suboptimal.

End-to-End

Next, we compare TPS to alternative tensioning approaches. We first evaluate the techniques in terms of performance by calculating the symmetric difference between the target pattern and actual cut. Then, we compare the techniques in terms of robustness by tuning the techniques for one simulated parameter setting and then applying them to perturbations.

Cutting Accuracy

We manually drew 17 different closed and open contour to evaluation in simulation, as illustrated in the table. For each of the contours, we evaluated four different tensioning methods:

- *Not Tensioned:* Single-arm cutting is performed with no assisted tensioning other than the stationary corner clips that fix the sheet in place.

- *Fixed Tensioning:* The material is pinched at a single point with the gripper arm (with corner points still fixed in place), but no directional tensioning is applied. We simulate area contact by pinching a circular disc.
- *Centroid Tensioning:* Tensioning is proportional to the direction and magnitude of the error in the 3D position of the cutting tool and the closest point on the desired contour. The gain was hand-tuned on randomly-chosen contours and is fixed to 0.01.
- *TPS* A separate policy is trained for each shape, this requires 20 iterations of TRPO in the simulator.

For the analytic tensioning model, the centroid of the contour is used as the pinch point. For the fixed policy, the point was chosen by random search over the feasible set of points. Performance is measured using the symmetric difference between the desired contour and the actual cut.

The averaged symmetric difference scores for each instance and tensioning method are reported in table. The success of the different tensioning algorithms are presented as the percentage of improvement in symmetric difference score over the non-tensioned baseline. The average of the scores over all 17 contours are also included in the table. For the selected set of contours, TPS achieves the best average relative improvement of 43.30%, the analytical method 9.70%, and the fixed approach 13.54%.

dVRK: Hardware and Software

We use the Intuitive Surgical da Vinci Research Kit (dVRK) surgical robot assistant, as in [40, 87, 105]. We interface with the dVRK using open-source electronics and software developed by [56]. The software system is integrated with ROS and allows direct robot pose control. We use the standard laparoscopic stereo camera for tracking fiducials on the surgical gauze. We equip the right arm with the curved scissors tooltip for cutting and the left arm with the needle driver tooltip for pinching and tensioning. The arms are located on either side of the surgical gauze to maximize the workspace for each without collision.

Physical Evaluation of TPS

We show the physical results on 4 contours using Fixed Tensioning and Deep RL using the symmetric difference as the evaluation metric in table. The tensioning policy for Deep RL was derived from simulation experiments by registering the physical gauze to a sheet in the simulator environment. In this set of experiments, we used fixed tensioning as the baseline. The no tensioning policy frequently failed in the physical experiment so we excluded that from the results table for a fair baseline comparison. We were unable to evaluate the analytic method in the physical experiments because the state-estimation needed for feedback control was not possible outside of the simulated environment.

We do not attempt analytic tensioning since it requires real-time tracking of the pattern to estimate local error. We observed that 3 out of 4 of the TPS experiments performed better than the fixed tensioning policy with respect to the symmetric difference of the cut and ideal trajectories. This is the same objective the trained policy was designed to minimize in simulation. A weakness of the policy was failure to address discrete failure modes, including discrete failure modes induced by the policy itself. Failure as a result of the scissors' entanglement in the gauze occurred in both experimental groups. The active manipulation and tensioning of TPS also caused increased deformation of the cloth which occasionally directly resulted in entanglement as well. We did not optimize our policy to minimize these discrete failures, and our simulator does not model the robot with acceptable fidelity to do so. To address this, we would require very accurate models of the robot arms and tools.

Table 6.1: **Evaluation of TPS**: For the 17 contours shown, we evaluate the three tensioning policies described in policyeval. We measure and report performance in terms of relative percentage improvement in symmetric difference over a baseline of no tensioning for the tensioning trials. The 95% confidence interval for 10 simulated trials is shown for fixed, analytic, and Deep RL tensioning, while the mean absolute symmetric difference error is reported for the no-tensioning baseline experiments. The data suggest that TPS performs significantly better in comparison to the fixed and analytic baseline. The corresponding pinch points used for fixed and TPS are indicated in red. The analytic pinch point is the centroid of the shape.

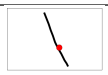
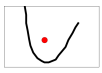
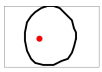
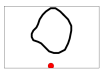
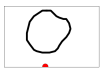
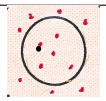
	Shape	**Tensioning Method**			
		No-Tension	**Fixed**	**Analytic**	**Deep RL**
1		17.4	-20.69±4.44	-149.43±10.24	**64.37±5.77**
2		22.5	32.44±1.74	-117.78±0.00	**55.11±7.40**
3		23.2	-21.12±5.41	18.10±1.78	**38.36±9.43**
4		102.9	7.48±0.62	30.42±1.45	**36.15±3.97**
5		41.1	0.49±7.99	9.98±0.00	**52.31±7.26**
6		42.0	**55.00±1.77**	11.43±1.52	45.95±6.60
7		40.2	22.64±1.14	3.73±1.46	**33.83±3.99**
8		40.0	-1.25±0.82	1.75±2.83	**35.50±4.67**
9		66.6	2.85±2.60	**34.68±2.25**	28.38±4.98
10		63.6	25.63±2.98	20.60±3.60	**41.35±9.90**
11		73.6	2.31±1.66	24.32±0.98	**56.11±8.99**
12		79.3	22.82±4.51	55.49±0.38	**63.56±3.61**
13		94.3	3.29±2.05	27.15±0.32	**34.04±5.87**
14		71.7	3.07±7.15	-2.51±0.61	**39.89±8.20**
15		178.7	74.71±1.22	80.75±0.88	**81.25±1.38**
16		114.6	-8.03±2.16	**31.06±0.62**	29.06±8.81
17		74.8	10.29±2.08	**28.34±2.07**	0.80±7.03
Mean (%)			13.54±9.84	9.70±21.96	**43.30±8.61**

Table 6.2: **dVRK Physical Experiments**: This table compares the relative percentage improvement in terms of symmetric difference to a baseline of fixed tensioning to TPS in experiments performed on the dVRK robot. The black dot indicates the pinch point of the gripper arm.

	1	2	3	4
Shape				
Deep RL	**118.63 %**	**36.77 %**	**75.33 %**	-44.59

Chapter 7

Conclusion

This dissertation explores reinforcement learning, a family of algorithms for finding solutions to MDP that assume query access to the underlying dynamical system:

$$q_t : s_t, a_t \rightarrow \mathsf{system}() \rightarrow s_{t+1}, r_t$$

High-dimensional and long-horizon search problems can require a prohibitive number of such queries, and exploiting prior knowledge is crucial. In several domains of interest, a limited amount of expert knowledge is available and my dissertation presents an argument *learning such structure can significantly improve the sample-efficiency, stability, and robustness of RL methods.* With this additional supervision, the search process can be restricted to those sequences that are similar to the supervision provided by the expert. The dissertation explores several Deep RL systems for control of imprecise cable-driven surgical robots, automatically synthesizing data-cleaning programs to meet quality specifications, and generating efficient execution plans for relational queries. I describe algorithmic contributions, theoretical analysis about the implementations themselves, and the architecture of the RL systems.

7.1 Challenges and Open Problems

The promise of (deep) reinforcement learning is a general algorithm that applies to a large variety of MDP settings. My dissertation focuses on the RL setting with the most minimal assumptions on the MDP. In general, there is a spectrum of assumptions one could make and interesting sub-families of algorithms arise at different points on the spectrum.

A Vision For Reinforcement Learning

I believe that role of Reinforcement Learning is analagous to the role of Stochastic Gradient Descent in supervised learning problems. While it is not practical to expect that RL

will solve every MDP in the most efficient way, we would like it to be competitive to specialized algorithms. This is similar to SGD in many supervised learning problems, while special case alterative exist that are often more performant (e.g., SDCA for SVM problems and Newton's Method for Linear Regression), SGD is a reasonable general purpose framework that works decently well across a variety of problems. This unification has allowed the community to build optimized frameworks around SGD, e.g., TensorFlow, and has ultimately, greatly accelerated the rate of applications research using supervised learning. I envision a similar future for RL, one where a small number of RL algorithms are essentially competitive with classical baselines. Better alternatives may exist in special cases but one can expect RL to work across problem domains. This requires that RL exploit much of the same problem structure that the classical algorithms exploit.

Reset Assumptions

One assumption not discussed in this work is how the system can be reset to its initial state after querying it. In many problems, one has arbitrary control over the reset process, i.e., the system can be initialized in any state. This is the principle of backtracking search in constraint satisfaction and graph search. Designing RL algorithms that intelligently select initial states to begin their search is an interesting avenue of research. For example, one could imagine a Deep Q-Learning learning algorithm which rather than sampling from the initial state distribution begins its rollouts from a random point in a trajectory in its replay buffer. This would simulate some level of backtracking behavior in the algorithm. Optimizing the initialization process over a collection of tasks would be very valuable.

Distance Assumptions

Another crucial assumption exploited in many discrete search settings is *a priori* knowledge of distance between states. For example, the A* search algorithm uses this to prioritize expansion with an admissible heuristic that lower bounds the cost-to-go. This is functionally equivalent to a lower bound on the Q-function in RL. An interesting problem is techniques to learn such lower bounds and transfer them between problems. For example, are there ways to distill a Q network into a simpler model that simply provides a lower approximation and share that with other search problem instances.

Demonstration Assumptions

State-action sequences are the most basic form of supervision from an expert. A more advanced expert may be able to provide much more information such as task segments, hierarchical information, and perhaps even hints to the learner. Thinking about how to formalize the notion of hints from the supervisor is another interesting research direction. What if the supervisor could give a partially written program to the describe the control

policy and the search algorithm could fill in the gaps? There are interesting questions of how we represent partial or incomplete knowledge in a framework like RL.

Transfer

The "holy grail" of Deep Reinforcement Learning is to demonstrate non-trivial transfer of learned policies between MDPs or perturbations of the system. Ensuring safe policy deployments involves understanding the effects of applying a policy trained in simulation or in a training environment to the non-idealities of the real world. For example, in our query optimization project, to deploy the policies trained from a cost model, we had to perform significant fine-tuning in the real world.

The challenge is that real world observations often have very different properties than the data seen in training. When we execute an SQL query, we do not observe all of the intermediate costs without significant instrumentation of the database. We may only observe a final runtime (the simulator is a proxy for this). Executing subplans can be very expensive (queries can take several minutes to run). However, we can still leverage a final runtime to adjust plans that optimize runtime, while still leveraging what we have learned from the cost-model. One can think about the fine-tuning process as first using the cost model to learn relevant features about the structure of subplans (i.e., which ones are generally beneficial). After this is learned, those features are fed into a predictor to project the effect of that decision on final runtime. Generalizing such ideas is an interesting avenue of research. Can Deep RL in a simulator be used to learn features, which allow for rapid transfer to the real world?

Bibliography

[1] Trifacta. https://www.trifacta.com/.

[2] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *ICML*, page 1. ACM, 2004.

[3] Pieter Abbeel and Andrew Y Ng. Inverse reinforcement learning. In *Encyclopedia of machine learning*, pages 554–558. Springer, 2011.

[4] Tamim Asfour, Pedram Azad, Florian Gyarfas, and Rüdiger Dillmann. Imitation learning of dual-arm manipulation tasks in humanoid robots. *I. J. Humanoid Robotics*, 5(2):183–202, 2008.

[5] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. *arXiv preprint arXiv:1609.05140*, 2016.

[6] Pierre-Luc Bacon and Doina Precup. Learning with options: Just deliberate and relax. In *NIPS Bounded Optimality and Rational Metareasoning Workshop*, 2015.

[7] Peter Bailis, Deepak Narayanan, and Samuel Madden. Macrobase: Analytic monitoring for the internet of things. In *arXiv*, 2016.

[8] Benjamin Balaguer and Stefano Carpin. Combining imitation and reinforcement learning to fold deformable planar objects. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1405–1412. IEEE, 2011.

[9] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, 2003.

[10] Michael Beetz, Ulrich Klank, Ingo Kresse, Alexis Maldonado, Lorenz Mösenlechner, Dejan Pangercic, Thomas Rühr, and Moritz Tenorth. Robotic roommates making pancakes. In *Humanoids*, pages 529–536. IEEE, 2011.

[11] Richard Bellman. *Dynamic programming*. Princeton University Press, 1957.

[12] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. Benchmarking the chase. In *PODS*, 2017.

[13] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1995.

[14] Daniel Bielser, Pascal Glardon, Matthias Teschner, and Markus Gross. A state machine for real-time cutting of tetrahedral meshes. *Graphical Models*, 66(6):398–417, 2004.

[15] Matthew M Botvinick. Hierarchical models of behavior and prefrontal function. *Trends in cognitive sciences*, 12(5):201–208, 2008.

[16] Matthew M Botvinick, Yael Niv, and Andrew C Barto. Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. *Cognition*, 113(3):262–280, 2009.

[17] Adam Brooks. A tearable cloth simulation using vertlet integration. https://github.com/Dissimulate/Tearable-Cloth, 2016.

[18] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE journal on robotics and automation*, 2(1):14–23, 1986.

[19] Tim Brys, Anna Harutyunyan, Halit Bener Suay, Sonia Chernova, Matthew E Taylor, and Ann Nowé. Reinforcement learning from demonstration through shaping.

[20] Hung Hai Bui, Svetha Venkatesh, and Geoff West. Policy recognition in the abstract hidden Markov model. *JAIR*, 17:451–499, 2002.

[21] Sylvain Calinon and Aude Billard. Stochastic gesture production and recognition model for a humanoid robot. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendai, Japan, September 28 - October 2, 2004*, pages 2769–2774, 2004.

[22] Berk Çalli, Aaron Walsman, Arjun Singh, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M. Dollar. Benchmarking in manipulation research: The YCB object and model set and benchmarking protocols. *CoRR, abs/1502.03143*, 2015.

[23] Nuttapong Chentanez, Ron Alterovitz, Daniel Ritchie, Lita Cho, Kris Hauser, Ken Goldberg, Jonathan R Shewchuk, and James F O'Brien. Interactive Simulation of Surgical Needle Insertion and Steering. *ACM Transactions on Graphics*, 28(3), 2009.

[24] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *SIGMOD*, 2016.

[25] Andrew Crotty, Alex Galakatos, and Tim Kraska. Tupleware: Distributed machine learning on small clusters. In *IEEE Data Eng. Bull.*, 2014.

[26] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. Nadeef: a commodity data cleaning system. In *SIGMOD*, 2013.

[27] Christian Daniel, Gerhard Neumann, and Jan Peters. Hierarchical relative entropy policy search. In *AISTATS*, pages 273–281, 2012.

[28] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In *NIPS*, pages 271–278, 1992.

[29] Thomas G Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR*, 13:227–303, 2000.

[30] Xin Luna Dong. Data sets for data fusion experiments. http://lunadong.com/fusionDataSets.htm.

[31] Yan Duan, Marcin Andrychowicz, Bradly Stadie, OpenAI Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. One-shot imitation learning. In *Advances in neural information processing systems*, pages 1087–1098, 2017.

[32] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continous control. In *ICML*, 2016.

[33] Wenfei Fan and Floris Geerts. *Foundations of Data Quality Management*. 2012.

[34] Carlos Florensa, Yan Duan, and Pieter Abbeel. Stochastic neural networks for hierarchical reinforcement learning. In *ICLR*, 2017.

[35] National Consortium for the Study of Terrorism and Responses to Terrorism (START). Global terrorism database [data file]. https://www.start.umd.edu/gtd/, 2016.

[36] Roy Fox, Michal Moshkovitz, and Naftali Tishby. Principled option learning in Markov decision processes. *arXiv preprint arXiv:1609.05524*, 2016.

[37] Stuart Gale and Wanda J Lewis. Patterning of tensile fabric structures with a discrete element model using dynamic relaxation. *Computers & Structures*, 2016.

[38] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *PVLDB*, 2001.

[39] Yixin et al. Gao. The JHU-ISI gesture and skill assessment dataset (jigsaws): A surgical activity working set for human motion modeling. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2014.

[40] Animesh Garg, Siddarth Sen, Rishi Kapadia, Yiming Jen, Stephen McKinley, Lauren Miller, and Ken Goldberg. Tumor localization using automated palpation with gaussian process adaptive sampling. In *CASE*, 2016.

[41] Tim Genewein, Felix Leibfried, Jordi Grau-Moya, and Daniel Alexander Braun. Bounded rationality, abstraction, and hierarchical decision-making: An information-theoretic optimality principle. *Frontiers in Robotics and AI*, 2:27, 2015.

[42] Chaitanya Gokhale, Sanjib Das, AnHai Doan, Jeffrey F Naughton, Narasimhan Rampalli, Jude Shavlik, and Xiaojin Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.

[43] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. Proactive wrangling: mixed-initiative end-user programming of data transformation scripts. In *UIST*, 2011.

[44] Mandana Hamidi, Prasad Tadepalli, Robby Goetschalckx, and Alan Fern. Active imitation learning of hierarchical policies. In *IJCAI*, pages 3554–3560, 2015.

[45] Nicolas Heess, Greg Wayne, Yuval Tassa, Timothy Lillicrap, Martin Riedmiller, and David Silver. Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*, 2016.

[46] Joseph M Hellerstein. Quantitative data cleaning for large databases. In *UNECE*, 2008.

[47] Bernhard Hengst. Discovering hierarchy in reinforcement learning with HEXQ. In *ICML*, volume 2, pages 243–250, 2002.

[48] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, et al. Deep q-learning from demonstrations. *arXiv preprint arXiv:1704.03732*, 2017.

[49] Ronald A Howard. Dynamic programming. *Management Science*, 12(5):317–348, 1966.

[50] Manfred Huber and Roderic A. Grupen. A feedback control structure for on-line learning tasks. *Robotics and Autonomous Systems*, 22(3-4):303–315, 1997.

[51] Stephen James, Andrew J Davison, and Edward Johns. Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task. *arXiv preprint arXiv:1707.02267*, 2017.

[52] Anders Jonsson and Vicenç Gómez. Hierarchical linearly-solvable Markov decision problems. *arXiv preprint arXiv:1603.03267*, 2016.

[53] Leslie Pack Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. In *ICML*, pages 167–173, 1993.

[54] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, 2011.

[55] Sertac Karaman. Incremental sampling-based algorithms for optimal motion planning. *Robotics Science and Systems VI*, 104:2.

[56] P Kazanzides, Z Chen, A Deguet, G.S. Fischer, R.H. Taylor, and S.P. DiMaio. An Open-Source Research Kit for the da Vinci Surgical System. In *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2014.

[57] Peter Kazanzides, Zihan Chen, Anton Deguet, Gregory S Fischer, Russell H Taylor, and Simon P DiMaio. An open-source research kit for the da vinci® surgical system. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 6434–6439. IEEE, 2014.

[58] Zuhair Khayyat, Ihab F Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. Bigdansing: A system for big data cleansing. In *SIGMOD*, 2015.

[59] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, page 0278364913495721, 2013.

[60] J Zico Kolter, Pieter Abbeel, and Andrew Y Ng. Hierarchical apprenticeship learning with application to quadruped locomotion. In *NIPS*, volume 20, 2007.

[61] George Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pages 895–900, 2007.

[62] George Konidaris and Andrew G Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In *NIPS*, pages 1015–1023, 2009.

[63] George Konidaris, Scott Kuindersma, Roderic A. Grupen, and Andrew G. Barto. Robot learning from demonstration by constructing skill trees. *IJRR*, 31(3):360–375, 2012.

[64] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. In *PVLDB*, 2010.

[65] Sanjay Krishnan, Animesh Garg, Richard Liaw, Brijen Thananjeyan, Lauren Miller, Florian T Pokorny, and Ken Goldberg. SWIRL: A sequential windowed inverse reinforcement learning algorithm for robot tasks with delayed rewards. In *WAFR*, 2016.

[66] Sanjay Krishnan, Animesh Garg, Sachin Patil, Colin Lea, Gregory Hager, Pieter Abbeel, and Ken Goldberg. Transition state clustering: Unsupervised surgical trajectory segmentation for robot learning. In *ISRR*, 2015.

[67] Sanjay Krishnan*, Animesh Garg*, Sachin Patil, Colin Lea, Gregory Hager, Pieter Abbeel, Ken Goldberg, and (*denotes equal contribution). Transition State Clustering: Unsupervised Surgical Trajectory Segmentation For Robot Learning. In *International Symposium of Robotics Research*. Springer STAR, 2015.

[68] Sanjay Krishnan, Daniel Haas, Michael J. Franklin, and Eugene Wu. Towards reliable interactive data cleaning: A user survey and recommendations. In *HILDA*, 2016.

[69] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. Activeclean: Interactive data cleaning for statistical modeling. In *PVLDB*, 2016.

[70] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[71] Volker Kruger, Dennis Herzog, Sanmohan Baby, Ales Ude, and Danica Kragic. Learning actions from observations. *Robotics & Automation Magazine, IEEE*, 17(2):30–43, 2010.

[72] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *NIPS*, pages 3675–3683, 2016.

[73] Aravind S Lakshminarayanan, Ramnandan Krishnamurthy, Peeyush Kumar, and Balaraman Ravindran. Option discovery in hierarchical reinforcement learning using spatio-temporal clustering. *arXiv preprint arXiv:1605.05359*, 2016.

[74] Michael Laskey, Jonathan Lee, Wesley Hsieh, Richard Liaw, Jeffrey Mahler, Roy Fox, and Ken Goldberg. Iterative noise injection for scalable imitation learning. *Conference on Robot Learning (CoRL) 2017*, 2017.

[75] Ian Lenz, Honglak Lee, and Ashutosh Saxena. Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*, 2015.

[76] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *arXiv preprint arXiv:1504.00702*, 2015.

[77] Kfir Y Levy and Nahum Shimkin. Unified inter and intra options learning using policy gradient methods. In *European Workshop on Reinforcement Learning*, pages 153–164. Springer, 2011.

[78] Richard Liaw, Sanjay Krishnan, Animesh Garg, Daniel Crankshaw, Joseph E Gonzalez, and Ken Goldberg. Composing meta-policies for autonomous driving using hierarchical deep reinforcement learning. 2017.

[79] Zelda Mariet, Rachael Harding, Sam Madden, et al. Outlier detection in heterogeneous datasets using automatic tuple expansion. 2016.

[80] Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *ICML*, pages 361–368, 2001.

[81] Geoffrey McLachlan and Thriyambakam Krishnan. *The EM algorithm and extensions*, volume 382. John Wiley & Sons, 2007.

[82] Ishai Menache, Shie Mannor, and Nahum Shimkin. Q-cut—dynamic discovery of sub-goals in reinforcement learning. In *ECML*, pages 295–306. Springer, 2002.

[83] Cesar Mendoza and Christian Laugier. Simulating soft tissue cutting using finite element models. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 1, pages 1109–1114. IEEE, 2003.

[84] Sebastian Mika, Bernhard Schölkopf, Alexander J. Smola, Klaus-Robert Müller, Matthias Scholz, and Gunnar Rätsch. Kernel PCA and de-noising in feature spaces. In *NIPS*, pages 536–542, 1998.

[85] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518, 2015.

[86] Adithyavairavan Murali, Animesh Garg, Sanjay Krishnan, Florian T. Pokorny, Pieter Abbeel, Trevor Darrell, and Ken Goldberg. Tsc-dl: Unsupervised trajectory segmentation of multi-modal surgical demonstrations with deep learning. In *ICRA Conference*, 2016.

[87] Adithyavairavan Murali, Siddarth Sen, Ben Kehoe, Animesh Garg, Seth McFarland, Sachin Patil, W. Douglas Boyd, Susan Lim, Pieter Abbeel, and Kenneth Y. Goldberg. Learning by observation for surgical subtasks: Multilateral cutting of 3d viscoelastic and 2d orthotropic tissue phantoms. In *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015*, pages 1202–1209, 2015.

[88] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, pages 663–670, 2000.

[89] Scott Niekum, Sarah Osentoski, George Konidaris, and Andrew G. Barto. Learning and generalization of complex tasks from unstructured demonstrations. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, October 7-12, 2012*, pages 5239–5246, 2012.

[90] Han-Wen Nienhuys and A Frank van der Stappen. A Surgery Simulation supporting Cuts and Finite Element Deformation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2001*, pages 145–152. Springer, 2001.

[91] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J Andrew Bagnell, Pieter Abbeel, Jan Peters, et al. An algorithmic perspective on imitation learning. *Foundations and Trends in Robotics*, 7(1-2):1–179, 2018.

[92] Takayuki Osa, Naohiko Sugita, and Mitsuishi Mamoru. Online Trajectory Planning in Dynamic Environments for Surgical Task Automation. In *Robotics: Science and Systems (RSS)*, 2014.

[93] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.

[94] Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In *NIPS*, pages 1043–1049, 1997.

[95] Ronald Edward Parr. *Hierarchical control and learning for Markov decision processes*. PhD thesis, UNIVERSITY of CALIFORNIA at BERKELEY, 1998.

[96] Bilal Piot, Matthieu Geist, and Olivier Pietquin. Boosted bellman residual minimization handling expert demonstrations. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 549–564. Springer, 2014.

[97] Pravesh Ranchod, Benjamin Rosman, and George Konidaris. Nonparametric bayesian reward segmentation for skill discovery using inverse reinforcement learning. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. IEEE, 2015.

[98] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. In *arXiv*, 2017.

[99] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. In *AI, Prentice-Hall*, 1995.

[100] Ruslan Salakhutdinov, Sam Roweis, and Zoubin Ghahramani. Optimization with EM and expectation-conjugate-gradient. In *ICML*, pages 672–679, 2003.

[101] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *ICML*, pages 1312–1320, 2015.

[102] J. Schulman, A. Gupta, S. Venkatesan, M. Tayson-Frederick, and P. Abbeel. A Case Study of Trajectory Transfer through Non-Rigid Registration for a Simplified Suturing Scenario. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 4111–4117, 2013.

[103] John Schulman, Sergey Levine, Philipp Moritz, Michael I Jordan, and Pieter Abbeel. Trust region policy optimization. *ICML*, 2015.

[104] D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. 2014.

[105] S. Sen, A. Garg, D. V. Gealy, S. McKinley, Y. Jen, and K. Goldberg. Automating Multiple-Throw Multilateral Surgical Suturing with a Mechanical Needle Guide and Sequential Convex Optimization. In *ICRA*, 2016.

[106] Pierre Sermanet, Kelvin Xu, and Sergey Levine. Unsupervised perceptual rewards for imitation learning. *arXiv preprint arXiv:1612.06699*, 2016.

[107] Sahil Sharma, Aravind S. Lakshminarayanan, and Balaraman Ravindran. Learning to repeat: Fine grained action repetition for deep reinforcement learning. In *ICLR*, 2017.

[108] Eftychios Sifakis, Kevin G Der, and Ronald Fedkiw. Arbitrary cutting of deformable tetrahedralized objects. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 73–80. Eurographics Association, 2007.

[109] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. In *Nature*. Nature Research, 2016.

[110] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[111] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

[112] Özgür Şimşek and Andrew G Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *ICML*, page 95. ACM, 2004.

[113] Alec Solway, Carlos Diuk, Natalia Córdova, Debbie Yee, Andrew G Barto, Yael Niv, and Matthew M Botvinick. Optimal behavioral hierarchy. *PLOS Comput Biol*, 10(8):e1003779, 2014.

[114] Denis Steinemann, Matthias Harders, Markus Gross, and Gabor Szekely. Hybrid cutting of deformable solids. In *Virtual Reality Conference, 2006*, pages 35–42. IEEE, 2006.

[115] Ion Stoica, Dawn Song, Raluca Ada Popa, David Patterson, Michael W Mahoney, Randy Katz, Anthony D Joseph, Michael Jordan, Joseph M Hellerstein, Joseph E Gonzalez, et al. A berkeley view of systems challenges for ai. *arXiv preprint arXiv:1712.05855*, 2017.

[116] Martin Stolle. *Automated discovery of options in reinforcement learning*. PhD thesis, McGill University, 2004.

[117] Kaushik Subramanian, Charles L Isbell Jr, and Andrea L Thomaz. Exploration from demonstration for interactive reinforcement learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 447–456. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

[118] Wen Sun, Arun Venkatraman, Geoffrey J Gordon, Byron Boots, and J Andrew Bagnell. Deeply aggrevated: Differentiable imitation learning for sequential prediction. *arXiv preprint arXiv:1703.01030*, 2017.

[119] Niko Sünderhauf, Oliver Brock, Walter Scheirer, Raia Hadsell, Dieter Fox, Jürgen Leitner, Ben Upcroft, Pieter Abbeel, Wolfram Burgard, Michael Milford, et al. The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research*, 37(4-5):405–420, 2018.

[120] Cynthia Sung, Dan Feldman, and Daniela Rus. Trajectory clustering for motion prediction. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 1547–1552. IEEE, 2012.

[121] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[122] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *AI*, 112(1-2):181–211, 1999.

[123] Brijen Thananjeyan, Animesh Garg, Sanjay Krishnan, Carolyn Chen, Lauren Miller, and Ken Goldberg. Multilateral surgical pattern cutting in 2d orthotropic gauze with deep reinforcement learning policies for tensioning.

[124] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, 1993.

[125] Sebastian Thrun and Anton Schwartz. Finding structure in reinforcement learning. In *NIPS*, pages 385–392, 1994.

[126] Aleš Ude, Bojan Nemec, Tadej Petrić, and Jun Morimoto. Orientation in cartesian space dynamic movement primitives. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 2997–3004. IEEE, 2014.

[127] Aleksandar Vakanski, Iraj Mantegh, Andrew Irish, and Farrokh Janabi-Sharifi. Trajectory learning for robot programming by demonstration using hidden markov model and dynamic time warping. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 42(4):1039–1052, 2012.

[128] J. Van Den Berg, S. Miller, D. Duckworth, H. Hu, A. Wan, X. Fu, K. Goldberg, and P. Abbeel. Superhuman Performance of Surgical Tasks by Robots using Iterative Learning from Human-Guided Demonstrations. In *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, pages 2074–2081, 2010.

[129] Ruben Veldkamp, Esther Kuhry, WC Hop, J Jeekel, G Kazemier, H Jaap Bonjer, Eva Haglind, L Pahlman, Miguel A Cuesta, Simon Msika, et al. Laparoscopic surgery versus open surgery for colon cancer: short-term outcomes of a randomised trial. *Lancet Oncol*, 6(7):477–484, 2005.

[130] Mikhail Volkov, Guy Rosman, Dan Feldman, John W Fisher, and Daniela Rus. Coresets for visual summarization with applications to loop closure. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 3638–3645. IEEE, 2015.

[131] Andrew Whiten, Emma Flynn, Katy Brown, and Tanya Lee. Imitation of hierarchical action structure by young children. *Developmental science*, 9(6):574–582, 2006.

[132] Chenxia Wu, Jiemi Zhang, Silvio Savarese, and Ashutosh Saxena. Watch-n-patch: Unsupervised understanding of actions and relations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4362–4370, 2015.

[133] Eugene Wu and Samuel Madden. Scorpion: Explaining away outliers in aggregate queries. In *VLDB*, 2013.

[134] Eugene Wu, Samuel Madden, and Michael Stonebraker. A demonstration of dbwipes: clean as you query. In *VLDB*, 2012.

[135] Jeffrey M Zacks, Christopher A Kurby, Michelle L Eisenberg, and Nayiri Haroutunian. Prediction error associated with the perceptual segmentation of naturalistic events. *Journal of Cognitive Neuroscience*, 23(12):4057–4066, 2011.

[136] Hui Zhang, Shahram Payandeh, and John Dill. On Cutting and Dissection of Virtual Deformable Objects. In *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, pages 3908–3913, 2004.

[137] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *AAAI*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.