Robot Learning: An Introduction

Joe Watson, Julen Urain, Joao Carvalho, Niklas Funk, Jan Peters

Abstract Robot learning uses advanced machine learning and statistical methods to improve the algorithms and models used in robotics, including low-level control, dynamics models and high-level planning. Having such autonomous learning for physical systems has been a long-standing vision of robotics, artificial intelligence, and cognitive sciences. In the past two decades, the Robot Learning community has developed a large repertoire of learning techniques that enable robots to acquire new skills from data. This book chapter aims to provide a straightforward and intuitive introduction to the approaches that enabled these methods, starting from optimal control theory, and incorporating ideas from machine learning that facilitate model-based and model-free reinforcement learning and imitation learning. To enable practical learning on real robotic platforms, we describe inductive biases for robotic learning, such as movement primitives; and how they can be used for sample-efficient, data-driven control.

1 Introduction

Over the past several decades, robotics research has yielded the necessary theory and understanding to design, program and deploy a vast variety of robotic platforms across a broad range of settings, from subterranean mapping to surgical procedures. More recently, advances and widespread adoption of information technology has led to increased interest in *machine learning* (ML) [11, 5], a research field devoted to the development and study of algorithms and models for prediction, statistical modelling and optimization. The intersection of robotics and machine learning is known as Robot Learning [82].

Joe Watson, Julen Urain, Joao Carvalho, Niklas Funk, Jan Peters
TU Darmstadt, Hochschulsstr. 10, 64293 Darmstadt, Germany. e-mail: firstname.lastname@tu-darmstadt.de

For robotics, machine learning provides a means of *data-driven* design – replacing human engineering with data acquisition and processing. As our robots are expected to perform increasingly more complex tasks where manual may become infeasible, this automation has the promise of accelerating the design of these systems by leveraging large datasets of experience. Breakthroughs in computer vision [48], natural language processing [106] and decision making [68, 104] have demonstrated that this paradigm enables performance that matches or even *surpasses* human capabilities. However, these breakthroughs typically occurred on either static or abstract tasks while robot learning requires considering dynamically changing tasks in real-world embedding. Thus, achieving similar results on real-world robots requires models and algorithms that respect and incorporate their physical embodiment.

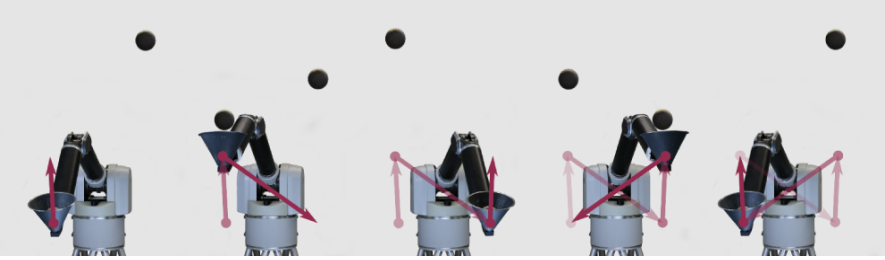
1.1 What is Robot Learning?

From the robotics perspective, Robot Learning constitutes the use of data-driven approaches to improve the effectiveness of the system. This paradigm builds on longstanding methods in robotics such as system identification [60, 111] and adaptive control [4]. From the machine learning perspective, Robot Learning considers models and algorithms that are relevant to the robotic setting, ranging from (semi-) supervised learning to make sense of visual and tactile perception [52, 56] to optimal interaction with the environment by reinforcement learning (RL) algorithms [109, 46].

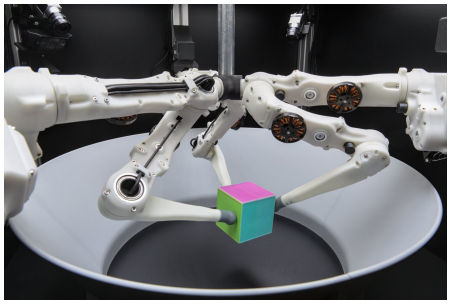
DEFINITION 1: ROBOT LEARNING

Robot learning considers the intersection of robotics and machine learning. The goal is to provide methods that allow robots to acquire novel skills, improve its performance, or adapt to dynamic environments through learning algorithms. These algorithms are starting to become applied to every aspect of the robot, including the morphology, perception, motor control and high-level reasoning. The physical embodiment of the robot provides both design constraints and inspiration for algorithm design, such as safety, latency and inductive biases such as physics and geometry.

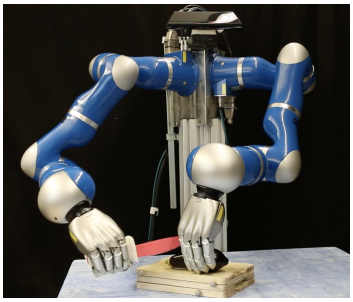
Moreover, the specific robotic setting motivates task- and system-specific inductive biases [6] is often necessary when data has to be generated in real-time on real system (which may suffer from damages as well as wear and tear) to ensure feasibility or at least to improve performance. Such inductive biases can include insights from physics-inspired models, geometry-constrained planning and approaches borrowed from control theory. Finally, the real-world embedded deployment requires additional considerations, such as computational efficiency, robustness, safety and continual learning. For a formal definition, see Definition 1. Examples are provided in Fig. 1.



(a)



(b)



(c)

Fig. 1: Diverse applications of robot learning. (a) High-acceleration robot juggling [83], (b) precise dexterous object manipulation [26], and (c) fine bi-manual manipulation [59].

Given its broad definition, this introductory book chapter focuses on a reduced subset of Robot Learning. We look specifically at learning motor actions, and chart the evolution of classical ideas from robotics, such as optimal control and system identification, to their modern interpretations as reinforcement learning and statistical machine learning. We motivate the use of purely data-driven control design methods, such as model-free policy search[1], and show how these methods can be extended to settings such as learning from demonstration (LfD; often also called imitation learning). Moreover, we discuss how robotics-inspired inductive biases can be incorporated into the statistical models, such as Bayesian linear regression with dynamic movement primitives for learning smooth robot motion from noisy demonstrations.

[1] The term "control policy" is shortened to just "policy" in the machine learning literature but typically includes both a control law and the trajectory generator from classical robotics.

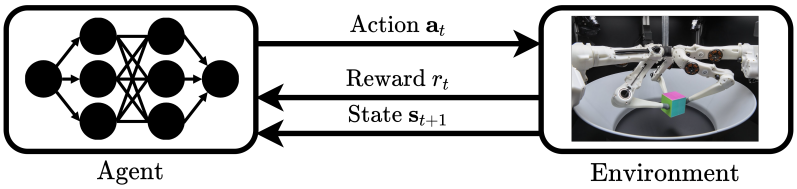


Fig. 2: Illustration of a sequential decision making process in which an agent is interacting with an environment, such as optimal control of a robot. In this example, the agent's actions are computed based on a parametric neural network policy $\boldsymbol{a}_t = \pi_\theta(\boldsymbol{s}_t)$ and applied to a robotic manipulation task which then yields the next state $\boldsymbol{s}_{t+1}$ and reward r_t. The setting can be described as Markov decision process.

1.2 Structure of the Chapter

This chapter looks specifically at Robot Learning in the context of learning motor control for robotics. Section 2 introduces optimal control and Markov decision processes (MDP) as the underlying framework. Section 3 covers the basics in statistical machine learning and deep learning for regression, which is required for learning models from data. How these models can be used for system identification and data-driven control is discussed in Section 4, and for useful policy representations in Section 5. Section 6 considers data-driven control without the notion of an explicit dynamics model, using the formalism of model-free reinforcement learning. Finally, Section 7 looks at imitation learning, which involves learning from expert demonstrations.

Notation. We adopt lowercase boldface letters $\boldsymbol{x}$ for vectors, uppercase boldface for matrices $\boldsymbol{X}$, otherwise x, X denote scalar variables. The expression $\boldsymbol{x} \sim p(\boldsymbol{x})$ means that a random variable $\boldsymbol{x}$ is distributed according to the distribution $p(\boldsymbol{x})$. The log-likelihood under this distribution is expressed as $\mathcal{L}(\boldsymbol{x}) = \log p(\boldsymbol{x})$. A multivariate Normal distribution with mean $\boldsymbol{\mu} \in \mathbb{R}^n$ and covariance $\boldsymbol{\Sigma} \in S_+^n$ is $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $+$ denotes positive (semi-) definiteness. For a random variable $\boldsymbol{x}$, $\mathbb{E}[\boldsymbol{x}]$, $\mathbb{V}[\boldsymbol{x}]$ and $\mathbb{H}[\boldsymbol{x}]$ denote the mean, covariance and entropy respectively. The Kullback-Leibler divergence between distributions $p(\boldsymbol{x})$ and $q(\boldsymbol{x})$ is denoted $D_{\mathrm{KL}}[p(\boldsymbol{x}) \,||\, q(\boldsymbol{x})]$. For scalar function $f(\boldsymbol{x}, \boldsymbol{y})$, we use the shorthand $\boldsymbol{f}_{\boldsymbol{x}} = \partial f / \partial \boldsymbol{x}$ and $\boldsymbol{F}_{\boldsymbol{xy}} = \partial^2 f / \partial \boldsymbol{x} \partial \boldsymbol{y}$ for Jacobians and Hessians respectively.

2 Optimal Control from Robot Forward Models

Classical control covers a multitude of criteria for control design, such as stability, frequency response and disturbance rejection. As machine learning methods tend to be presented as an optimization problem, optimal control is the preferred control

foundation in Robot Learning. A highly effective robot learning approach can be obtained by simply swapping the manually acquired forward model in the optimal control literature with a model identified by supervised learning. Moreover, optimal control covers not only continuous state and action spaces, but also discrete spaces, so it can be applied to both continuous control and also more abstract forms of robot decision making.

This section introduces the Markov decision process for optimal control in the discrete setting. For continuous control, we first look at the linear quadratic regulator (LQR), which has a closed-form solution for the optimal control law. Finally, we discuss approximate optimal control methods for the nonlinear dynamical systems that we typically encounter in robotics.

2.1 Markov Decision Processes

Before introducing methods for obtaining optimal controllers, we first have to define the problem. A wide range of problems in robotics ranging from path planning to low level control can be formalized as Markov decision processes (MDPs, Definition 2, Figs. 2 & 3) [8, 86].

Definition 2: Markov Decision Process

A stationary Markov decision process (MDP) describes the stochastic dynamics of a sequential decision making process that satisfies the Markov property. In particular, it is defined by state space $\boldsymbol{s} \in \mathcal{S}$, action space $\boldsymbol{a} \in \mathcal{A}$, transition dynamics $p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$, reward function $r(\boldsymbol{s}_t, \boldsymbol{a}_t)$, and initial state probabilities $\mu_0(\boldsymbol{s})$. Importantly, the transition dynamics fulfill the Markov property, i.e. the state transitions only depend on the current state and action and are thus conditionally independent of previous states and actions, $p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1}, ...) = p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$. Note that non-stationary MDPs follow the same definition but include time-varying transition dynamics, $p_t(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$ and reward functions $r_t(\boldsymbol{s}_t, \boldsymbol{a}_t)$.

MDPs do not only define the transition dynamics but also include a function that evaluates the quality of the current state and action. Hence, the goal is to obtain policies π that are reward optimal. A policy is a function mapping from states to actions which can either be deterministic, $\boldsymbol{a} = \pi(\boldsymbol{s})$, $\pi : \mathcal{S} \rightarrow \mathcal{A}$, or stochastic, i.e., defining a probability distribution over all actions given the current state, $\boldsymbol{a} \sim \pi(\cdot|\boldsymbol{s})$, $\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. Given the properties of MDPs, Richard Bellman formulated the "principle of optimality" in 1957 stating that "an optimal sequence of controls in a multistage optimization problem has the property that whatever the initial stage, state and controls are, the remaining controls must constitute an optimal sequence of decisions for the remaining problem with stage and state resulting from previous controls considered as initial conditions." In short, Bellman's principle of optimality implies that every part of the optimal solutions must itself be optimal, which directly

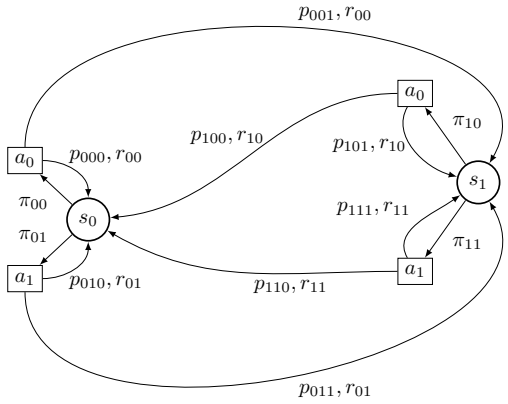


Fig. 3: A diagram of a stationary Markov decision process with two states $\{s_0, s_1\}$ and two actions $\{a_0, a_1\}$. The policy and dynamics are represented by probabilities π_{sa} and $p_{\text{sas'}}$, with the next state s' and the reward values r_{sa}.

results into a recursion often called the Bellman equation[2]

$$V_t^*(\boldsymbol{s}) = \mathbb{E}_{p,\pi^*}\left[\sum_{\tau=t}^{T} r(\boldsymbol{s}_\tau, \boldsymbol{a}_\tau)\right] = \max_{\boldsymbol{a}_t} \underbrace{r(\boldsymbol{s}_t, \boldsymbol{a}_t) + V_{t+1}^*(\boldsymbol{s}_{t+1})}_{Q_t^*(\boldsymbol{s}_t, \boldsymbol{a}_t)} = \max_{\boldsymbol{a}_t} Q_t^*(\boldsymbol{s}_t, \boldsymbol{a}_t), \tag{1}$$

with value function $V_t(\boldsymbol{s}_t)$, state-action value function $Q_t(\boldsymbol{s}_t, \boldsymbol{a}_t)$ that is sometimes also referred to as Q-function[3], optimal policy π^*, optimal value and Q-function V^*, Q^*. All methods that exploit this idea of decomposing the original problem into smaller subproblems belong to the family of dynamic programming (DP) algorithms. It is relatively straightforward to apply these ideas in tabular environments with

Algorithm 1 Value Iteration

Init: $V_T^*(\boldsymbol{s}_T) \leftarrow r(\boldsymbol{s}_t)$, $V_{t!=T}^*(\boldsymbol{s}_{t!=T}) = 0$
repeat
  /* Compute Q-function for each state action pair and point in time
  $Q_t^*(\boldsymbol{s}_t, \boldsymbol{a}_t) \leftarrow r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \sum_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t) V_{t+1}^*(\boldsymbol{s}_{t+1})$
  /* Compute V-function for each state and point in time
  $V_t^*(\boldsymbol{s}_t) \leftarrow \max_{\boldsymbol{a}_t} Q_t^*(\boldsymbol{s}_t, \boldsymbol{a}_t)$
until $V_t^*(\boldsymbol{s}_t)$ converged for all states and points in time
return Optimal policy: $\pi_t^*(\boldsymbol{s}_t) = \arg\max_{\boldsymbol{a}_t} Q_t^*(\boldsymbol{s}_t, \boldsymbol{a}_t)$

[2] The Bellman equation was named in Bellman's honor but was known to Carathéodory, von Neumann, Morgenstern, Wald and others before Bellman generalized it into the principle of optimality and the generic method of dynamic programming.

[3] Following on from value, Q stands for *quality*.

Algorithm 2 Policy Iteration

Init: $V_T^\pi(\boldsymbol{s}_T) \leftarrow r(\boldsymbol{s}_t)$, $V_{t!=T}^\pi = 0$, $\pi \leftarrow$ random
repeat
  repeat
    /* Compute Q-function for each state action pair and point in time under current policy
    $Q_t^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) \leftarrow r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \sum_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t) V_{t+1}^\pi(\boldsymbol{s}_{t+1})$
    /* Compute V-function for each state and point in time
    $V_t^\pi(\boldsymbol{s}_t) \leftarrow Q_t^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t{=}\pi_t(\boldsymbol{s}_t))$
  until $V_t^\pi(\boldsymbol{s}_t)$ converged
  /* Update Policy
  $\pi_t(\boldsymbol{s}_t) \leftarrow \arg\max_{\boldsymbol{a}_t} Q_t^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t)$
until Convergence of policy
return Optimal policy: $\pi_t^*(\boldsymbol{s}_t) \leftarrow \pi_t(\boldsymbol{s}_t)$

discrete sets of actions, states and known transition dynamics, and results in the two algorithms of value and policy iteration [8, 10, 109]. In value iteration (VI, cf. Alg. 1), one computes the optimal value function recursively starting from the final state, which directly also yields the optimal policy, i.e. the action that actually maximizes the value in each state. In policy iteration (PI, cf. Alg. 2), we are starting with a fixed policy and first compute all the Q-values $Q_t^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t)$ under the current policy, and then update the policy to greedily choose the actions that yield highest value in each state. This process is repeated until convergence and also yields the optimal policy. Policy iteration is known to be computationally more effective as it allows effects of policy changes to propagate through the value function before requiring redundant use of the max-Operator on an unchanged value function.

2.2 The Linear Quadratic Regulator

For special cases, Markov decision processes with continuous states $\boldsymbol{s} \in \mathbb{R}^{d_s}$ and actions $\boldsymbol{a} \in \mathbb{R}^{d_a}$ can be solved analogously. For example, we can perform optimal control using the value-iteration approach in a tractable fashion when the dynamics function is a time-varying linear (affine) state-space model and the reward is time-varying quadratic, where weights $\boldsymbol{Q}_t$ and $\boldsymbol{R}_t$ are positive definite matrices and $\boldsymbol{s}_g$ is the goal state. This setting is known as the linear quadratic regulation (LQR), see [13], with

$$\boldsymbol{s}_{t+1} = \boldsymbol{f}_t(\boldsymbol{s}_t, \boldsymbol{a}_t) = \boldsymbol{A}_t\boldsymbol{s}_t + \boldsymbol{B}_t\boldsymbol{a}_t, \quad r_t(\boldsymbol{s}_t, \boldsymbol{a}_t) = -(\boldsymbol{s}_g{-}\boldsymbol{s}_t)^\top \boldsymbol{Q}_t(\boldsymbol{s}_g{-}\boldsymbol{s}_t) - \boldsymbol{a}_t^\top \boldsymbol{R}_t \boldsymbol{a}_t.$$

For finite-horizon optimal control, we also can specify a terminal reward, only in the state $r_T(\boldsymbol{s}_T) = -(\boldsymbol{s}_g{-}\boldsymbol{s}_T)^\top \boldsymbol{Q}_T(\boldsymbol{s}_g{-}\boldsymbol{s}_T)$. The Bellman equation

$$V_t(\boldsymbol{s}_t) = \max_{\boldsymbol{a}_t} Q_t(\boldsymbol{s}_t, \boldsymbol{a}_t) = \max_{\boldsymbol{a}_t} r_t(\boldsymbol{s}_t, \boldsymbol{a}_t) + V_{t+1}(\boldsymbol{f}_t(\boldsymbol{s}_t, \boldsymbol{a}_t))$$

can be solved analytically as affine systems combined with quadratic rewards imply that the value function $V_t(\boldsymbol{s}_t) = \boldsymbol{s}_t^\top \boldsymbol{V}_t \boldsymbol{s}_t + \boldsymbol{v}_t^\top \boldsymbol{s}_t + v_t$ is always a quadratic form.

This Bellman equation is solved by a parametric version of the value iteration algorithm. Here, the value function is initialized by $\boldsymbol{V}_T = \boldsymbol{Q}_T$ and, subsequently, the recursive parametric update equation

$$\boldsymbol{V}_t = \boldsymbol{Q}_t + \boldsymbol{A}_t^\top \boldsymbol{V}_{t+1} \boldsymbol{A}_t - \boldsymbol{A}_t^\top \boldsymbol{V}_{t+1} \boldsymbol{B}_t (\boldsymbol{R}_t + \boldsymbol{B}_t^\top \boldsymbol{V}_{t+1} \boldsymbol{B}_t)^{-1} \boldsymbol{B}_t^\top \boldsymbol{V}_{t+1} \boldsymbol{A}_t \tag{2}$$

is computed backwards in time from $t = T-1$ down to $t = 1$. Due the specific structure of Eq. (2), it is known as a discrete algebraic Riccati equation (DARE). Solving the value function for the optimal action, we arrive at a time-varying affine controller in $\boldsymbol{s}_t$.

$$\boldsymbol{a}_t^* = \arg\max_{\boldsymbol{a}_t} Q_t(\boldsymbol{s}_t, \boldsymbol{a}_t) = -(\boldsymbol{R}_t + \boldsymbol{B}_t^\top \boldsymbol{V}_{t+1} \boldsymbol{B}_t)^{-1} (\boldsymbol{B}_t^\top \boldsymbol{V}_{t+1} \boldsymbol{A}_t \boldsymbol{s}_t - \boldsymbol{B}_t^\top \boldsymbol{v}_{t+1}) = \boldsymbol{K}_t \boldsymbol{s}_t + \boldsymbol{k}_t.$$

If the system is controllable, the control policy's parameters $\boldsymbol{K}_t, \boldsymbol{k}_t$ will converge for $t = \hat{t} \ll T$ for sufficiently large T long before $t = 1$. Thus, the stationary linear controller $\boldsymbol{a}^* = \boldsymbol{K}_{\hat{t}} \boldsymbol{s} + \boldsymbol{k}_{\hat{t}}$ c is an infinite-horizon optimal control law.

2.3 Nonlinear Approximate Optimal Control

The LQR setting was restricted to time-varying linear dynamics and quadratic rewards. For many control problems in robotics, we have nonlinear dynamics and non-quadratic objectives. In this section, we briefly describe two possible approaches to this problem that perform approximate optimal control.

Differential Dynamic Programming. Inspired by LQR, we can compute a local second-order Taylor approximation of the dynamics and reward function computed along a nominal trajectory $(\bar{\boldsymbol{s}}_t, \bar{\boldsymbol{a}}_t)$ for every time-step $t = 1 : T$ by

$$r_t(\bar{\boldsymbol{s}}_t + \delta_{\boldsymbol{s}}, \bar{\boldsymbol{a}}_t + \delta_{\boldsymbol{a}}) = r_t(\bar{\boldsymbol{s}}_t, \bar{\boldsymbol{a}}_t) + \begin{bmatrix} \boldsymbol{r}_{\boldsymbol{s}_t} & \boldsymbol{r}_{\boldsymbol{a}_t} \end{bmatrix} \begin{bmatrix} \delta_{\boldsymbol{s}} \\ \delta_{\boldsymbol{a}} \end{bmatrix} + \begin{bmatrix} \delta_{\boldsymbol{s}} \\ \delta_{\boldsymbol{a}} \end{bmatrix}^\top \begin{bmatrix} \boldsymbol{R}_{\boldsymbol{s}\boldsymbol{s}_t} & \boldsymbol{R}_{\boldsymbol{s}\boldsymbol{a}_t} \\ \boldsymbol{R}_{\boldsymbol{s}\boldsymbol{a}_t}^\top & \boldsymbol{R}_{\boldsymbol{a}\boldsymbol{a}_t} \end{bmatrix} \begin{bmatrix} \delta_{\boldsymbol{s}} \\ \delta_{\boldsymbol{a}} \end{bmatrix},$$

$$\boldsymbol{f}_t(\bar{\boldsymbol{s}}_t + \delta_{\boldsymbol{s}}, \bar{\boldsymbol{a}}_t + \delta_{\boldsymbol{a}}) = \boldsymbol{f}_t(\bar{\boldsymbol{s}}_t, \bar{\boldsymbol{a}}_t) + \begin{bmatrix} \boldsymbol{F}_{\boldsymbol{s}_t} & \boldsymbol{F}_{\boldsymbol{a}_t} \end{bmatrix} \begin{bmatrix} \delta_{\boldsymbol{s}} \\ \delta_{\boldsymbol{a}} \end{bmatrix} + \boldsymbol{f}_t'',$$

with a Jacobian matrix $\boldsymbol{F}$ and second-order term $\boldsymbol{f}_t''$ that is computed by

$$f_t^{(i)\prime\prime} = \begin{bmatrix} \delta_{\boldsymbol{s}} \\ \delta_{\boldsymbol{a}} \end{bmatrix}^\top \begin{bmatrix} \boldsymbol{F}_{\boldsymbol{s}\boldsymbol{s}_t}^{(i)} & \boldsymbol{F}_{\boldsymbol{s}\boldsymbol{a}_t}^{(i)} \\ \boldsymbol{F}_{\boldsymbol{s}\boldsymbol{a}_t}^{(i)\top} & \boldsymbol{F}_{\boldsymbol{a}\boldsymbol{a}_t}^{(i)} \end{bmatrix} \begin{bmatrix} \delta_{\boldsymbol{s}} \\ \delta_{\boldsymbol{a}} \end{bmatrix}$$

for every dimension i from 1 to d_s.

With these second-order approximations, we construct an approximate value functions using the backward recursions, and obtain local time-varying linear controllers. Due to its use of differentiation, this approach is known as differential dynamic programming (DDP) [41]. One key difference between DDP and LQR is the requirement of a trajectory about which to linearize the dynamics and cost. Therefore this algorithm is iterative, where a forward pass computes the linearization trajectory, and the backward pass performs local optimal control. The linearization trajectory in the forward pass is obtained by simulating the current controller on the dynamical system. For iteration i,

$$\boldsymbol{s}_{t+1}^i = \boldsymbol{f}_t(\boldsymbol{s}_t^i, \boldsymbol{a}_t^i), \quad \boldsymbol{a}_t^i = \boldsymbol{K}_t^{i-1}\boldsymbol{s}_t^i + \boldsymbol{k}_t^{i-1}.$$

Due to the use of the Hessian, this optimization is equivalent to a Newton method [12]. Given its local nature, line search routines are often used to ensure optimization stability and convergence. One issue with DDP in practice is that the dynamics model Hessians are high-dimensional tensors and therefore expensive to compute and slow down optimization. Ignoring these terms, and using the Jacobians to approximate the Hessians instead, is equivalent to Gauss-Newton optimization [12]. This approach also requires careful regularization and line search for convergence, due to the additional approximations. This approximate DDP method is known as sequential LQR or iterative LQR (iLQR) [57, 103].

Approximate Dynamic Programming. An alternative strategy is to use *function approximation* for the value functions and policy. Given a set of transitions $(\boldsymbol{s}_t^i, \boldsymbol{a}_t^i, r_t^i, \boldsymbol{s}_t^i),\ t{=}1{:}T,\ i{=}1{:}N$ over N rollouts, the stationary optimal value function and policy can be iteratively estimated by using Bellman's equation to compute regression targets. We will cover the details of function approximation, and how the *approximate* dynamic programming (ADP) [10] objectives can be optimized, in more detail in the next section. As ADP provides a means of performing optimal control from sampled MDP transitions, it is the foundation on which many reinforcement learning algorithms are constructed, which are discussed in more detail in Section 6.

3 Supervised Machine Learning: Acquiring Models from Data

This section will introduce some fundamental ideas from supervised machine learning for how to learn models from data. In the previous section, we assumed a fully engineered dynamics model, which we used to perform optimal control. We also already discussed the idea of function approximation to learn unknown value functions and policies of approximate optimal control. This section aims to fill the gap left for didactic reasons.

Supervised machine learning is an important tool with many applications in Robot Learning including:

- Learning models of unknown dynamical systems for control (i.e. using machine learning for structure and system identification in Section 4).
- Approximate policies and value functions from samples policies in approximate dynamic programming (Section 4) and reinforcement learning (Section 6).
- Approximate policies from experience or demonstrations (e.g., in imitation learning in Section 7).
- Learning other unknown robotics models, such as kinematic, sensor and camera models.

The simplest model we can consider first is a linear model [11], $\boldsymbol{y} = \boldsymbol{x}\boldsymbol{W}$, where $\boldsymbol{y} \in \mathbb{R}^{d_y}$, $\boldsymbol{x} \in \mathbb{R}^{d_x}$, $\boldsymbol{W} \in \mathbb{R}^{d_x \times d_y}$. Fitting n datapoints, where $\boldsymbol{X} \in \mathbb{R}^{n \times d_x}$, $\boldsymbol{Y} \in \mathbb{R}^{n \times d_y}$ and minimizing the squared error objective using the the L2 norm $|\cdot|^2$ yields

$$\boldsymbol{W}^* = \arg\min_{\boldsymbol{W}} |\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{W}|^2 = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{Y},$$

which is known as the 'normal equation'. When learning from only few datapoints, we typically require regularization of $\boldsymbol{W}$ to prevent overfitting to the observed data

$$\boldsymbol{W}^* = \arg\min_{\boldsymbol{W}} |\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{W}|^2 + \beta^2 |\boldsymbol{W}|^2 = (\boldsymbol{X}^\top \boldsymbol{X} + \beta^2 \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{Y}.$$

Adding a $|\boldsymbol{W}|^2$ term into the objective to keep the weights small is known as *ridge* or *Tikhanov* regularization, where β is a hyperparameter [11].

The simplicity of the linear model in $\boldsymbol{x}$ alone is often quite limiting. Thus, it is popular to perform linear regression in a feature space $\phi(\boldsymbol{x}) \in \mathbb{R}^{d_\phi}$ instead, which is a transformation of $\boldsymbol{x}$. Such feature transformations include adding a bias for affine models $\phi(\boldsymbol{x}) = [\boldsymbol{x}\ \ 1]^\top$ or polynomial features for quadratic model, e.g. $\phi(\boldsymbol{x}) = [x_1^2\ x_2^2\ x_1 x_2\ x_1\ x_2\ 1]^\top$. Moreover, we would also like to incorporate a notion of *uncertainty* into our learned parameters and model predictions.

3.1 Bayesian Linear Regression & Nonparametric Gaussian Processes

In the introduction of linear regression so far, we have taken a so-called "frequentist perspective" and assumed the existence of a *single*, unknown, optimal weight parameter value $\boldsymbol{W}^*$. The fallacy of this assumption was evident in the need for parameter regularization, keeping the values small, in order to avoid overfitting when presented with small sets of data. Bayesian methods take an alternative approach where many parameter values are considered and their plausibility is assessed by maintaining probability distributions over the model variables, initialized with a prior distribution. Such probabilistic treatment has two key effects: The initial probability distribution (prior) regularizes the parameter updates, and propagating the parameter uncertainty through the model provides uncertainty quantification of the

model's prediction. Such predictive uncertainty can be used for downstream tasks concerning safety, exploration and decision making – crucial in robot learning. The catch with Bayesian methods is the increased complexity in the derivations and implementations. While some Bayesian models can be learned in closed form with exact inference, many are intractable and require approximate inference methods, such as variational inference and Monte Carlo methods [5].

DEFINITION 3: BAYES' THEOREM

Bayes theorem is derived from conditional probability theory. When $\boldsymbol{\theta}$ is an unknown model parameter, it is used to construct Bayesian statistical models. Given a data likelihood $p(\mathcal{D}|\boldsymbol{\theta})$ and a prior over parameters $p(\boldsymbol{\theta})$,

$$\underbrace{p(\boldsymbol{\theta}|\mathcal{D})}_{\text{Posterior}} = \frac{\overbrace{p(\mathcal{D}|\boldsymbol{\theta})}^{\text{Likelihood}}\overbrace{p(\boldsymbol{\theta})}^{\text{Prior}}}{\underbrace{p(\mathcal{D})}_{\text{Evidence}}}$$

where the evidence, or marginal likelihood, is the probability of the data given the model $p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta}$. The probability of the data represents the normalizing constant for the probability density function in Bayes rule; it is useful for model selection, i.e., for deciding which Bayesian model is best for the data, and hyperparameter optimization.

When implementing Bayesian models, the fundamental tool is Bayes rules, which describes the updated distribution (posterior) as a combination of the prior and observed data (likelihood), as described in Definition 3. To provide a worked example of Bayesian methods, consider estimating a Euclidean position $\boldsymbol{x}$ using a noisy sensor $\boldsymbol{y} = \boldsymbol{x} + \epsilon$, corrupted by Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma^2\boldsymbol{I})$. To make an sensible estimate from limited samples, we place a Gaussian prior on x which describes our initial guess $\mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$. The sensor noise ϵ defines our Gaussian likelihood. Given a single estimate $\boldsymbol{y}_1$, the posterior distribution $\mathcal{N}(\boldsymbol{x}_1|\boldsymbol{y}_1)$ can be computed using Bayes rule in closed-form

$$\boldsymbol{\Sigma}_1 = (\boldsymbol{\Sigma}_0^{-1} + \tfrac{1}{\sigma^2}\boldsymbol{I})^{-1}, \quad \boldsymbol{\mu}_1 = \boldsymbol{\Sigma}_1(\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0 + \tfrac{1}{\sigma^2}\boldsymbol{y}_1).$$

Note this update can be used recursively for new data, by using the posterior as the subsequent prior. This aspect of Bayesian methods make them popular for localization and state estimation [96, 22].

Returning to regression, we focus on one-dimensional targets and place a multivariate Gaussian prior on $\boldsymbol{w}$, and use Bayesian inference in a similar manner to the small example above. The equations are summarized in Definition 4, and these models are also known as Gaussian processes [90]. Note that when $\boldsymbol{\mu}_0 = \mathbf{0}$ and $\boldsymbol{\Sigma}_0 = \boldsymbol{I}$, the mean updated matches the ridge regression update above where $\beta \equiv \sigma^2$.

Definition 4: Gaussian Process as Bayesian Linear Regression

For a linear model with Gaussian likelihood

$$y = \phi_{\boldsymbol{x}}^\top \boldsymbol{w} + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2),$$

where $\phi_{\boldsymbol{x}} = \phi(\boldsymbol{x})$ with Gaussian prior $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$. For a dataset $\mathcal{D}$ of n points $\{\Phi, \boldsymbol{y}\}$, $\Phi \in \mathbb{R}^{n \times d_\phi}$ and $\boldsymbol{y} \in \mathbb{R}^{n \times 1}$, the weight posterior $p(\boldsymbol{w}|\mathcal{D})$ is

$$\boldsymbol{w}_n \sim \mathcal{N}(\boldsymbol{\mu}_n, \boldsymbol{\Sigma}_n), \quad \boldsymbol{\Sigma}_n = (\sigma^2 \boldsymbol{\Sigma}_0^{-1} + \Phi^\top \Phi)^{-1}, \quad \boldsymbol{\mu}_n = \boldsymbol{\Sigma}_n(\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0 + \Phi^\top \boldsymbol{y}).$$

The closed-form Gaussian posterior predictive for query point $\boldsymbol{x}_*$ is

$$p(\boldsymbol{y}_* \mid \boldsymbol{x}_*, \mathcal{D}) = \mathcal{N}(\mu_{y_*}, \sigma^2_{y_*}), \quad \mu_{y_*} = \phi_*^\top \boldsymbol{\mu}_n, \quad \sigma^2_{y_*} = \sigma^2(1 + \phi_*^\top \boldsymbol{\Sigma}_n \phi_*).$$

Note that for Bayesian linear regression, the predictive uncertainty is now a function of the input. This uncertainty can be broken down into two components

$$\sigma^2_{y_*} = \underbrace{\sigma^2}_{\text{Aleatoric}} + \underbrace{\sigma^2 \phi_*^\top \boldsymbol{\Sigma}_n \phi_*}_{\text{Epistemic}}.$$

Aleatoric uncertain describes that statistical uncertainty, for example sensor noise, and is inherent to the problem independent from the dataset. Epistemic uncertainty describes model uncertainty and arises from uncertain model parameters given on the observed data. Once enough data is observed, Bayesian linear regression becomes more confident in $\boldsymbol{w}$ and the epistemic uncertainty disappears as $\boldsymbol{\Sigma}_n \to \mathbf{0}$. Note that using just the (regularized) posterior mean, rather than the full posterior, is called a *maximum a posteriori* (MAP) update [5].

What are the limits of feature regression? Designing features is a crucial design choice in many machine learning applications, and the more features the richer the modeling capacity. For this reason, features are often flexible and parametrized, and then optimized using the evidence. For example, radial basis function (RBF; see [11]) features $\phi_i(\boldsymbol{x}) = \exp(-(\boldsymbol{x} - \boldsymbol{c}_i)/2l_i^2)$ are defined by a centre $\boldsymbol{c}$ and length-scale l. An alternative approach for working with richer feature spaces is to use kernel methods [36]. The kernel is the inner product of a feature space at two points $k(\boldsymbol{x}_1, \boldsymbol{x}_2) = \phi(\boldsymbol{x}_1)^\top \phi(\boldsymbol{x}_2)$, $k : \mathbb{R}^{d_x} \times \mathbb{R}^{d_x} \to \mathbb{R}$. Importantly, the kernel is always a scalar function, regardless of the feature dimension d_ϕ, and, thus, we can define kernels that represent infinite feature spaces! By working purely with the kernel, rather than the features explicitly, we can enjoy the benefit of the rich representation while remaining computationally tractable. Two examples of kernels $k(\boldsymbol{x}_1, \boldsymbol{x}_2)$ are

$$k_{\text{RBF}}(\boldsymbol{x}_1, \boldsymbol{x}_2) = \exp(-|\boldsymbol{x}_1 - \boldsymbol{x}_2|^2 / 2l^2),$$

$$k_{\text{arcsine}}(\boldsymbol{x}_1, \boldsymbol{x}_2) = \frac{2}{\pi} \arcsin\left(\frac{2\bar{\boldsymbol{x}}_1 \boldsymbol{\Sigma} \bar{\boldsymbol{x}}_2}{\sqrt{(1 + 2\bar{\boldsymbol{x}}_1^\top \boldsymbol{\Sigma} \bar{\boldsymbol{x}}_1)(1 + 2\bar{\boldsymbol{x}}_2^\top \boldsymbol{\Sigma} \bar{\boldsymbol{x}}_2)}} \right), \quad \bar{\boldsymbol{x}} = [\boldsymbol{x}, \ \ 1]^\top.$$

Such RBF kernel is derived from an infinite limit of RBF features, and is *stationary* as it depends only on the distance between points. The arcsine kernel [125] is derived from the limit of a single hidden layer neural network (see next section) with erf function activations and a Gaussian distribution over the weights and bias. It is non-stationary. To use kernels for Bayesian linear regression, we rewrite the posterior update to use only the inner feature product, rather than the outer product [90]. This model is described in Definition 5. The result is that the predictive distribution is computed using the data directly, as we must use a data matrix $\boldsymbol{K} = k(\boldsymbol{X}, \boldsymbol{X})$ that computes the inner-products over the whole dataset. As the model is now explicitly represented in a flexible manner using the data, rather than finite parameters that need optimizing, we call it *non-parametric*. Non-parametric does not imply that there are no parameters but that the number of parameters of the model grows implicitly with the number of data points. Thus, the model complexity is now dependent on the dataset size. This scaling is a quality, as model complexity increases with the dataset size, but becomes an issue when working with very large datasets – both due to computation and storage reasons.

Definition 5: Non-parametric Gaussian Processes

Given a Gaussian likelihood $p(y|\boldsymbol{x}, \sigma^2)$ and kernel k, construct the data matrix $\boldsymbol{K} = k(\boldsymbol{X}, \boldsymbol{X}) \in \mathbb{R}^{n \times n}$ from n data points. Again, there is a closed-form Gaussian predictive $p(\boldsymbol{y}_* \mid \boldsymbol{x}_*, \mathcal{D}) = \mathcal{N}(\mu_{y_*}, \sigma^2_{y_*})$,

$$\mu_{y_*} = \boldsymbol{k}_*(\boldsymbol{K} + \sigma^2 \boldsymbol{I})^{-1} \boldsymbol{y}, \quad \sigma^2_{y_*} = \sigma^2 + k(\boldsymbol{x}_*, \boldsymbol{x}_*) - \boldsymbol{k}_*(\boldsymbol{K} + \sigma^2 \boldsymbol{I})^{-1} \boldsymbol{k}_*^\top,$$

where $\boldsymbol{k}_* = k(\boldsymbol{x}_*, \boldsymbol{X}) \in \mathbb{R}^{1 \times n}$.

Due to the inversion of the data matrix used in the predictive distribution (Definition 5), GP regression scales $O(n^3)$ with the dataset size. This computational complexity has motivated *sparse* approaches to GPs that improve scaling through approximation. Such approximations includes inducing points, which approximate the dataset with fewer (optimized) samples [113], and random features, which approximate the kernel in a Monte Carlo fashion with a finite feature space [87].

While the description of GPs above is focused on the weight-space perspective, GPs also permit a function-space view. The kernel, prior and likelihood can be combined into a *covariance function* $C(\boldsymbol{X}) = \sigma^2 \boldsymbol{I} + \sigma_0^2 k(\boldsymbol{X}, \boldsymbol{X})$, which in turn describes the GP prior $\mathcal{GP}(m(\boldsymbol{x}), C(\boldsymbol{x}))$. Evaluating n inputs returns an n-dimensional multivariate Gaussian. Sampling this Gaussian acts samples (evaluated) functions from the function-space specified as the prior, as shown in Figure 4. Due to this

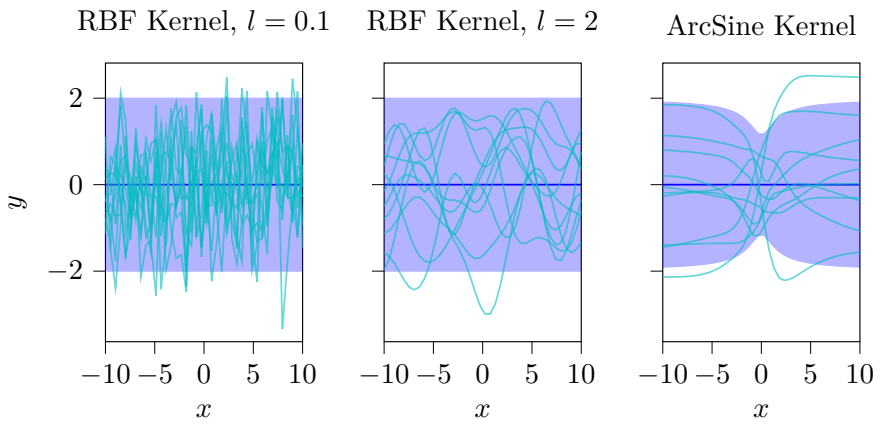


Fig. 4: Illustrating three one-dimensional, zero-mean Gaussian processes priors using different covariance functions. The plots show the GPs in blue (2 standard deviations) and function samples in cyan. Function samples are realized by sampling the $n \times n$ covariance matrix defined the the scaled kernel function, using $n = 40$ evaluation points, defined linearly between -10 and 10. Observe the RBF covariance function looks similar to IID Gaussian noise for small lengthscales, as the cross-correlation between inputs is close to zero. For larger lengthscales, the larger cross-correlations manifest through smoother function samples. As the RBF covariance function is stationary, the GP has constant variance along x. Contrast this to the arcsine covariance function, which is non-stationary so the variance varies with x. The function samples are not periodic, and rather emulate the erf activation function of the neural network architecture used to derive the kernel.

perspective, GPs are often defined through their covariance function, rather than their feature space or kernel.

3.2 Automatic Differentiation for Nonlinear Function Approximation

While non-parametric models are very expressive, their computational and memory complexity limit them for many large-scale tasks in practice. Moreover, both features and/or kernels are predefined. Such manual choices are especially problematic as the representation of our data has a significant impact on which functions the model can represent. This circumstance naturally leads to the desire to also learn and adapt the features depending on the problem that needs to be solved. For this purpose, we will now consider parametrized feature functions $\phi_\theta(\boldsymbol{x})$ with learnable parameters θ. For our feature regression model, it took the form $\hat{\boldsymbol{y}} = \boldsymbol{f}_\theta(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{\phi}_\theta(\boldsymbol{x}) = \boldsymbol{\phi}_{\tilde{\theta}}(\boldsymbol{x})$. Note that the linear weights of the model can just be viewed as additional fixed parameters of our representation that is to be learned. Describing a model in this fashion recur-

sively, we arrive at what are known as *neural networks* (NN), a biologically-inspired architecture in which nonlinear features are built by propagating the intermediate representations through linear transformations and nonlinear 'activation functions'

$$\begin{aligned}\hat{y} = \phi_\theta(\boldsymbol{x}) &= \boldsymbol{W}_n(...f_2(\boldsymbol{W}_2(f_1(\boldsymbol{W}_1\boldsymbol{x})))) = \boldsymbol{W}_n(...f_2(\boldsymbol{W}_2\phi_{\theta_1}(\boldsymbol{x})))\\ &= \boldsymbol{W}_n(...f_2(\boldsymbol{W}_2\boldsymbol{l}_1)) = \boldsymbol{W}_n(...\phi_{\theta_1,\theta_2}(\boldsymbol{x})) = \boldsymbol{W}_n(...\boldsymbol{l}_2)\end{aligned} \tag{3}$$

where each layer essentially consists of first applying a linear transformation $\boldsymbol{W}$ to the layer input $\boldsymbol{l}_i$, followed by a nonlinear activation function $\boldsymbol{f}(\cdot)$ to yield the output, which will be essentially the input to the next layer, i.e. $\boldsymbol{l}_{i+1}(\boldsymbol{x}) = \boldsymbol{f}_{i+1}(\boldsymbol{W}_{i+1}\boldsymbol{l}_i(\boldsymbol{x}))$. The weights of the linear transformation contain the parameters that are to be learned, the activation functions remain a choice to be made beforehand, and the layer input depends on the output of the previous layer. Examples for activation functions are the sigmoid, hyperbolic tangent or the rectified linear activation function (ReLU). When combined, these layers produce feature that are optimized for fitting data. The activation functions influence the nature of the features. For example, ReLU activations produce piecewise-linear features, while tanh activations produce features that transitional smoothly between two saturation points. With this model structure, a NN is defined by the width of each layer and the number of layers. An additional benefit of this structure is that the model can perform *representation learning* by learning a lower-dimensional intermediate representation of a high-dimensional input, such as RGB images. These learned intermediate features indicate increased generalization performance through the ability to compress the representation.

However, the question remains how to actually obtain the parameters, given the complexity of the multi-layered representation and the nonlinear activation functions. We will make use of gradient-based optimization and motivate automatic differentiation. Consider the loss that is obtained by making a prediction with a two layered neural network, i.e., $L(\boldsymbol{y}, \hat{\boldsymbol{y}})$ with prediction $\hat{\boldsymbol{y}}(\boldsymbol{x}) = \boldsymbol{f}_2(\boldsymbol{W}_2\boldsymbol{l}_1) = \boldsymbol{f}_2(\boldsymbol{W}_2\boldsymbol{f}_1(\boldsymbol{W}_1\boldsymbol{x}))$. As we want to minimize the loss, we have to adapt every parameter in the opposite direction of the gradient of the loss function with respect to the parameter. Assume that we calculated the prediction $\hat{\boldsymbol{y}}$ incrementally, in the forward path

$$\boldsymbol{l}_1 = \boldsymbol{f}_1(\boldsymbol{W}_1\boldsymbol{x}) \quad \boldsymbol{l}_2 = \boldsymbol{f}_2(\boldsymbol{W}_2\boldsymbol{l}_1) \quad L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = L(\boldsymbol{y}, \boldsymbol{l}_2).$$

We now determine each parameter's influence on the loss by taking the appropriate gradient in the backwards pass

$$\frac{\partial L}{\partial \hat{\boldsymbol{y}}} = \frac{\partial L}{\partial \boldsymbol{l}_2} \quad \frac{\partial L}{\partial \boldsymbol{W}_2} = \frac{\partial L}{\partial \boldsymbol{l}_2}\frac{\partial \boldsymbol{l}_2}{\partial \boldsymbol{W}_2} \quad \frac{\partial L}{\partial \boldsymbol{W}_1} = \frac{\partial L}{\partial \boldsymbol{l}_2}\frac{\partial \boldsymbol{l}_2}{\partial \boldsymbol{l}_1}\frac{\partial \boldsymbol{l}_1}{\partial \boldsymbol{W}_1}.$$

As the individual gradients that emerge from the use of the chain rule can be calculated straightforwardly, and are actually reused for calculating the gradients for the layers closer to the input, this combined procedure of forward and backward pass is tractable and does scale to very deep networks. This combination of dynamic programming and automatic differentiation is known as the backpropagation algorithm.

Algorithm 3 Expectation Maximization for Model Learning

Initialization: Initial model parameters $\boldsymbol{\theta}_0$, measurements $\boldsymbol{Y}$, initial state distribution $p(\boldsymbol{x}_1)$, dynamics likelihood $p(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t, \boldsymbol{\theta})$ and measurement model $p(\boldsymbol{y}_t|\boldsymbol{x}_t, \boldsymbol{\theta})$
repeat
  /* E-step: Bayesian filtering and smoothing of latent state trajectory given model and data
  Trajectory posterior $p(\boldsymbol{X}|\boldsymbol{Y}, \boldsymbol{\theta}_i)$ by calculating $p(\boldsymbol{x}_t|\boldsymbol{y}_{1:T}, \boldsymbol{\theta}_i)$ for t=1:T
  /* M-step: Optimize $\boldsymbol{\theta}$ for expected log-likelihood objective given latent state trajectory
  $\boldsymbol{\theta}_{i+1} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{X}\sim p(\cdot|\boldsymbol{Y},\boldsymbol{\theta}_i)}[\mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\theta})]$
until converged
return Learned model $\boldsymbol{\theta}$

Finally, all the parameters are updated according to $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha\nabla_{\boldsymbol{\theta}} L$ with learning rate α.

It has been shown that feedforward neural networks are extremely powerful representations and there exist theoretical proofs that one single layered neural network is sufficient to represent any function [38, 18]. Yet, deeper networks can be seen as performing a hierarchical decision process which in turn often simplifies the representation of complex functions and results in requiring less parameters. While the gradient based optimization scheme is very efficient it also introduces several difficulties. Just to name two examples: the repetitive multiplication of gradients causes very small gradients for the initial layers of deep networks, which slows optimization. This phenomenon is known as 'vanishing gradients'. Secondly, vast amount of parameters and their ambiguous representation results in many local optima. To counteract these issues, architectures with skip connections and the addition of momentum terms in the parameter update have been proposed. For more background information and details, we refer the interested reader to [2, 29, 51, 98].

The combination of neural network features and Bayesian linear regression is known as the neural linear model (NLM) [50, 77]. While such methods provide NNs with uncertainty quantification, they are weaker at expressing uncertainty compared to nonparametric GPs for fixed features. This weakness results from optimization for the data when acquiring the learned features. Standard training does not enforce feature diversity, therefore the model can be relatively overconfident outside of the data distribution and especially between data points. As a result, such models need to be trained in a way that specifically encourages feature diversity [50, 123, 122]. More generally, Bayesian neural networks (BNN) have a weight distribution over all parameters [72]. The nonlinear nature of neural networks means that approximate inference is required for inference over all parameters, and poses a significant challenge for achieving accurate inference for large models and datasets [40].

4 Model Learning for Control

After covering optimal control in Section 2, a logical approach for data-driven control is to first learn an approximate dynamics model from data, and then apply optimal control algorithms using that model. This approach is ubiquitous in practical robotics, and is commonly known under the moniker of 'model-based reinforcement learning' (MBRL) or 'optimal control with learned forward models' in the context of Robot Learning [21]. The idea of identifying model parameters for dynamical systems from data is well-established in robotics as *system identification* [3, 111] while the model structure is typically computed from physical insight. In Robot Learning, 'Model Learning' typically refers to the combination of system parameter and structure identification using machine learning. This section will discuss model learning from the perspective of Robot Learning, and follow-up with an overview of how optimal control can be effectively applied to approximate dynamics models.

4.1 Model Learning: System Identification meets Machine Learning

System identification typically requires specifying a parametrized dynamics model derived from physics, and then proposes a means of learning the model parameters from data. The key consideration is learning model parameters that are *physically plausible*, for example positive masses and lengths, which may require constrained optimization or reparameterization to preserve the constraint [27].

In Section 3, we introduced feature regression and neural network function approximation, which fits data using more general, expressive models that take no inspiration from physics. The use of such flexible models reduces the necessity to carefully design and implement the model, but means more data coverage over the domain of interest is required to fit these models sufficiently as generalization is no longer guaranteed. In other words, physical plausibility is now enforced through data coverage and not the inherent structure, which is problematic if the model is used outside of the data distribution (OOD) or in increasingly high-dimensional spaces. The benefit of these unstructured models is that they make no assumptions about the data, which make them more widely applicable and simpler to implement. Crucially, they are also able model phenomena that my not be captured by the physics model, such as friction effects and other disturbances.

Viewing physics-derived models as 'white-box' models, and unstructured function approximators as 'black-box' models, we can motivated an intermediate approach of 'grey-box' modelling. Such 'grey-box' modelling could involve augmenting the function approximators with inductive biases such as energy conservation [61], or combining physics-based models with black-box components, capturing the residual error [75] or complex friction phenomena in the actuators [62]. The challenge of grey-box modelling is capturing the necessary trade-off in model structure, and training the model such that each component learns its intended phenomena

adequately. While conceptually the white- and black-box components are distinct, during learning they are inherently coupled.

Another important objective across the whole spectrum of models is the training objective and algorithm. Dynamics models can be trained using forward or inverse dynamics [76]. Forward dynamics can be trained for single-step prediction, or could be trained to minimize multi-step prediction error to improve long horizon performance [76]. Gradient-based optimization can also be used, either using analytical gradients or automatic differentiation. However, care must often be taken to ensure physically plausible parameters using constraints or virtual parameters [116, 105, 62]. Taking the inference perspective, the expectation-maximization (EM) algorithm can be used [23], which naturally incorporates model-based data smoothing into the system identification procedure (Algorithm 3). While EM can be performed exactly for linear Gaussian dynamical systems [102], it required approximate inference for nonlinear dynamical systems [28, 100].

4.2 Optimal Control for Approximate Dynamics Models

Combining the ideas from Section 2 and 4.1, a viable approach to data-driven control is combining optimal control algorithms with learned dynamics models. Compared to model-free methods (see Section 6), that perform more of a trial-and-error approach for learning, combining finite-horizon optimization with approximate dynamics incorporates an inductive bias that improved the sample efficiency of learning, reaching better performance with less experience. For example, assuming local extrapolation of the approximate dynamics, the planning of optimal control should direct data-collection towards regions where either the model uncertain or constitute approximately optimal trajectories. Model-free exploration lacks such an informed search strategy. A key pitfall of model-based, data-driven control is designing optimal control solvers to operate effectively under model error. For example, optimal controls solvers are encouraged to exploit model errors that improve control performance, that could lead to catastrophic controls that damage the robot, as they have no indication of the model accuracy. For example, consider a navigation task of a maze where the robot assumes that walls do not exist. Greedily optimizing for navigation would lead to repeated crashing rather than a safer exploration strategy. One means to alleviate this greedy phenomena is to use Bayesian models for the dynamics, and use the epistemic uncertainty as a measure for possible prediction error [99]. Maintaining a distribution over the future state trajectory, the expected future reward evaluation is regularized by the uncertainty. To appreciate why, consider a quadratic state reward with a Gaussian belief in the state. Due to the quadratic law, the expected reward incorporates the state variance as well as the mean

$$r(x) = -x^2, \quad x \sim \mathcal{N}(\mu, \sigma^2), \quad r(\mathbb{E}[x]) = -\mu^2, \quad \mathbb{E}[r(x)] = -\mu^2 - \sigma^2.$$

As $\mathbb{E}[r(x)] \leq r(\mathbb{E}[x])$, incorporating the uncertainty in our state predictions into reward evaluation will regularize the expected reward as long as uncertainty exists. However, computing an accurate trajectory distribution can pose a challenge in the nonlinear case due to the necessity of approximate inference methods.

Another issue with model-based reinforcement learning is long-horizon prediction. Compounding prediction errors render long-horizon planning inaccurate, especially if the model is trained using one-step prediction error. To avoid these prediction errors, model predictive control (MPC) [66] is often used for MBRL. MPC uses replanning each time step over a shorter time horizon, in order to reach to prediction errors or disturbances. The consequence of this adaptivity is greater computational cost due to the repeated replanning computation. Another benefit of MPC is the amortization of the optimal control problem, which can avoid convergence to local optima for complex tasks such as locomotion [112].

Finally, optimal control methods are designed for true dynamics functions, which are generally continuous and smooth by definition or for numerical reasons, e.g. smooth contact models. Learned dynamics models, especially neural networks with discontinuous activations, do not behave in this smooth fashion and may have a 'rough' quality. Due to this aspect, gradient-based optimal control solvers are less suitable for optimization, as they will likely suffer numerical issues or get stuck in local optima. Some approximate model classes, such as kernel-based and time-varying linear models, are well-suited to gradient-based optimization [20, 55]. As a result of this phenomena, sample-based black-box optimization methods, such as the cross-entropy methods (CEM) [94], have become a popular MPC solver for deep MBRL [16, 126].

5 Policy Representations for Robotics

A policy $\pi(\boldsymbol{a}|\boldsymbol{s})$ is a function that maps a state $\boldsymbol{s} \in \mathcal{S}$ to an action $\boldsymbol{a} \in \mathcal{A}$. In Section 2, the optimal control derivation returned the form of the policy, which was derived to be discrete or time-varying linear. In the context of learning a robot policy by reinforcement learning (Section 6) or imitation learning (Section 7), the choice of the parameterized model to represent a policy is no longer defined and will directly influence in the behaviour of our robot. We desire the policy to have properties that facilitate safe and effective learning in continuous state and action spaces. Desirable properties for such policies could include *stability*, *smoothness* or *adaptability* to name a few.

5.1 From Torque Control to Motion Generation

To properly impose the desired properties in the policies, it is common to abstract from the robot control signals and model our policies as motion generators. In

robotics, the common action space is the torque signal τ applied on each joint of the robot. The state $\boldsymbol{s} = [\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{c}]^\top$ is composed of both the joints position and velocity information, $\boldsymbol{q}, \dot{\boldsymbol{q}}$ and the additional sensors input $\boldsymbol{c}$. Thus, a robot policy is a function that computes a desired torque signal given both the joint's and the sensors input, $\tau = \pi(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{c})$. To properly integrate the desired properties of stability, smoothness or adaptability, we can abstract from the torque signal and control the robot directly in the motion space, $\ddot{\boldsymbol{q}} = \pi(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{c})$. Given the robot's dynamics model is usually known,

$$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} = \tau - \boldsymbol{c}(\boldsymbol{q}, \dot{\boldsymbol{q}}) - \boldsymbol{g}(\boldsymbol{q})$$

with $\boldsymbol{M}$ is the inertia matrix, $\boldsymbol{c}$ the Coriolis and centrifugal forces and $\boldsymbol{g}$ the gravity vector; we can smartly model the torque policy to set the desired acceleration in the robot. With $\tau = \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}}_{\text{des}} + \boldsymbol{c}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{g}(\boldsymbol{q})$, the robot's joint acceleration is $\ddot{\boldsymbol{q}} = \ddot{\boldsymbol{q}}_{\text{des}}$ as long as the inertia matrix is invertible. Given this change of variables, the policy becomes a dynamical system. As we will show along this section, imposing a set of desired properties in our policy models becomes trivial when modelling the policies as dynamical systems. These motion generator type policies are widely known as *movement primitives*.

5.2 Types of Movement Primitives

The literature in movement primitives is vast. Depending on the desired properties, a wide set of policy models are proposed that can cover different desired properties from stroke-motion primitives, periodic primitives, orientation primitives or space-constrained primitives. In this section we introduce three motion primitives that are representative of different classes of primitives. We will introduce, phase and state dependant primitives $\pi(\boldsymbol{q}, \dot{\boldsymbol{q}}, \zeta)$, phase dependant primitives $\pi(\zeta)$ and state dependant primitives $\pi(\boldsymbol{q}, \dot{\boldsymbol{q}})$.

Dynamic Movement Primitives [97, 39] represent a globally-stable, robust and adaptable parameterized dynamic system that can learn highly nonlinear dynamics. The model is composed of two elements: a state-dependant second order linear attractor and a time dependant nonlinear forcing term that reshapes that linear dynamics to follow the desired nominal behaviour.

DEFINITION 6: DYNAMIC MOVEMENT PRIMITIVES

A dynamic movement primitive for a single degree-of-freedom trajectory q, of a point-to-point movement is defined by

$$\tau\ddot{q} = \alpha(\beta(g - q) - \dot{q}) + f(\zeta), \quad \tau\dot{\zeta} = \gamma\zeta,$$

with α and β the gains of the linear attractor, g the target position, $gamma$ the gain for the phase dynamics and f the nonlinear forcing term. The forcing term is modelled to become zero for large ζ values.

A common choice to represent the forcing term is by a linear weighting of a set of basis functions, ϕ

$$f(\zeta) = \frac{\sum_{k=0}^{K} w_k \phi_k(\zeta)}{\sum_{k=0}^{K} \phi_k(\zeta)}$$

with each basis function modelled by an exponential basis function

$$\phi_k(\zeta) = \exp(-(\zeta - \mu_k)^2/\sigma_k).$$

The exponential basis function tends to zero the further ζ is from the center μ_k, so the forcing term will tend to zero the bigger the phase is, $\lim_{\zeta\to\infty} f(\zeta) = 0$. Additionally, dynamic movement primitives allows to adapt to different target positions by changing the target state g while mantaining the nonlinear shape due to the forcing term f.

Probabilistic Movement Primitives [78] represent phase dependant, probabilistic motion primitives for both stroke and periodic movements. The proposed model is composed of two parts. The time dependant desired position is computed by a linearly-weighted sum of basis functions

$$p(\boldsymbol{q} \mid \zeta, \boldsymbol{w}) = \psi(\zeta)^\top \boldsymbol{w} + \epsilon_q, \quad \epsilon_q \sim \mathcal{N}(0, \sigma^2)$$

with additional white noise. Then, rather than having fix values for the weights $\boldsymbol{w}$ as in DMP, we assume they are sampled from a Gaussian distribution

$$\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{w}|\boldsymbol{\mu}_{\boldsymbol{w}}, \Sigma_{\boldsymbol{w}})$$

This is closely related to the Bayesian linear regression model from Section 3, with phase ζ as the input and the trajectory as the target, but it is used as *generative* model [5] rather than a regression model. Training these models is similar to direct regression when a unique latent trajectory is assumed. For modelling a more diverse distribution over trajectories, an EM procedure is used instead.

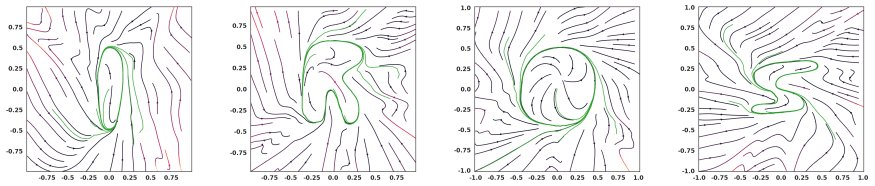


Fig. 5: A visual example of learned stable vector fields given a set of letter shape trajectories [118].

Definition 7: Probabilistic Movement Primitives

A probabilistic movement primitive (ProMP) represents a distribution in the position, $\boldsymbol{q}$ in terms of the weights $\boldsymbol{w}$, that belongs to a distribution parameterized by $\boldsymbol{\theta}$

$$p(\boldsymbol{q}|\zeta,\boldsymbol{\theta}) = \int_{\boldsymbol{w}} p(\boldsymbol{q}|\zeta,\boldsymbol{w})p(\boldsymbol{w}|\boldsymbol{\theta})\mathrm{d}\boldsymbol{w} = \int_{\boldsymbol{w}} \mathcal{N}(\boldsymbol{q}|\psi(\zeta)\boldsymbol{w},\sigma^2\boldsymbol{I})\mathcal{N}(\boldsymbol{w}|\mu_{\boldsymbol{w}},\Sigma_{\boldsymbol{w}})\mathrm{d}\boldsymbol{w}$$
$$= \mathcal{N}(\boldsymbol{q}|\psi(\zeta)\mu_{\boldsymbol{w}},\psi(\zeta)\Sigma_{\boldsymbol{w}}\psi^{\intercal}(\zeta)+\sigma^2\boldsymbol{I})$$

In contrast with dynamic movement primitives that directly provides the desired acceleration; ProMP proposes to learn a distribution in the position. Therefore, to control the robot we assume a tracking controller that exactly reproduces the given trajectory distribution.

Stable Vector Fields (SVF) [44, 45] are a set of state dependant dynamic systems that are stable by construction. Diffeomorphism based SVF can represent nonlinear dynamics while remaining stable towards a target or a limit cycle (Fig. 5). In contrast with DMPs or ProMPs, these motion primitives depend only in the robot state and do not require an additional phase variable. This property makes them an ideal choice for reactive motion generation or human-robot interaction, given they will naturally adapt their movements without the requirement of a phase estimation. One possible approach to build SVF is by exploiting a set of diffeomorphic functions. Diffeomorphism-based SVFs [73, 118, 89] are a family of dynamic systems that are composed of two parts. First, a globally stable linear dynamic system is defined in a latent space $\dot{z} = -z$. Then, a parameterize diffeomorphic function, Φ is given between the observation space $\mathcal{Q}$ and the latent space $\mathcal{Z}$.

Definition 8: Diffeomorphism-based Stable Vector Fields

Given a globally stable dynamic system $\dot{z} = g(z)$ in a latent space $\mathcal{Z}$ and a diffeomorphic map from the observation space $\mathcal{Q}$ to the latent space $\mathcal{Z}$, $\Phi : \mathcal{Q} \rightarrow \mathcal{Z}$, then the dynamics in the observation space

$$\dot{\boldsymbol{q}} = \boldsymbol{J}_{\Phi}^{-1} g(\Phi(\boldsymbol{q}))$$

are also globally stable.

SVF usually represent first-order stable dynamic systems. To control the robot, we can apply an error correction in the acceleration

$$\ddot{\boldsymbol{q}}_{\text{des}} = \alpha(\dot{\boldsymbol{q}}_{\text{des}} - \dot{\boldsymbol{q}}).$$

To model the diffeomorphic function Φ, several representations has been proposed from Gaussian processes to invertible neural networks [118], such as those used in normalizing flows.

6 Reinforcement Learning

A compelling notion from cognitive science and artificial intelligence is the ability to learn optimal decision making from experience, improving performance through trial and error in a systematic fashion [109]. Section 2 showed how to solve optimal control using value function methods, namely value iteration (Algorithm 1), policy iteration (Algorithm 2). These methods recover the optimal policy by maximizing the state-action value function $\pi^*(s) = \arg\max_a Q^*(s, a)$. Although theoretically sound, these methods make strong assumptions that do not always hold in practice. Section 2.3 discussed the challenges and approximations required for nonlinear optimal control, when the value function and optimal policy can not be obtained exactly. Section 4 highlighted the strong assumption of knowing the dynamics model, and the challenges faced when combining optimal control solvers with approximate dynamics models.

This section looks at model-free reinforcement learning, which covers a broad range of methods. These approaches can include value-based methods (Section 6.1), which develop on the idea of approximate dynamic programming. An alternative strategy can be to optimize the policy directly, using a broad of numerical methods (Section 6.2). Policy gradient approaches use gradient estimation methods to obtain low variance updates from policy evaluation data, and policy search methods in general can use a range of numerical methods for optimizing the policy. We will discuss episodic relative entropy policy search, which uses inference-based methods for proximal optimization. When moving from optimal control to reinforcement learning, another key aspect is *exploration*, which is arguable orthogonal to optimal control but essential for effective exploration.

6.1 Value Function Methods

In Section 2.3, we discussed approximate dynamic programming, where value functions and policies are obtained using function approximation. As this approach is already a combination of optimal control and machine learning methods, it translates naturally into a reinforcement learning approach.

Definition 9: Infinite Horizon Objective

$$J_\pi = \mathbb{E}_{\boldsymbol{s}_0\sim\mu_0,\, \boldsymbol{s}_{t+1}\sim\mathcal{P}(\boldsymbol{s}_t,\boldsymbol{a}_t),\, \boldsymbol{a}_t\sim\pi(\boldsymbol{s}_t)}\left[\sum_{t=0}^{\infty}\gamma^t r(\mathbf{s}_t,\mathbf{a}_t)\right] = \mathbb{E}_{\boldsymbol{s}_0\sim\mu_0}\left[V^\pi(\boldsymbol{s}_0)\right],$$

with initial state distribution μ_0, transition probability distribution $\mathcal{P}$, and discount factor $0 \leq \gamma < 1$ that trades off the short term and long-term rewards.

In contrast to Section 2, classical reinforcement learning is typically directly formulated for the infinite horizon objective as in Definition 9 [109], which is typically easier in the discounted formulation introduced here (It can be introduced straightforwardly in Section 2 but was omitted for simplicity). Definition 10 introduces, the state and state-action value functions for continuous spaces, and the corresponding optimal ones from Bellman's principle of optimally.

Definition 10: Value Functions of Arbitrary Policies

The value functions induced by policy π, used in policy iteration

$$V^\pi(\mathbf{s}) = \mathbb{E}_{\boldsymbol{a}\sim\pi(\boldsymbol{s})}\left[r(\mathbf{s},\mathbf{a}) + \gamma\mathbb{E}_{\boldsymbol{s}'\sim\mathcal{P}(\boldsymbol{s},\boldsymbol{a})}\left[V^\pi(\mathbf{s}')\right]\right],$$
$$Q^\pi(\mathbf{s},\mathbf{a}) = r(\mathbf{s},\mathbf{a}) + \gamma\mathbb{E}_{\mathbf{s}'\sim\mathcal{P}(\mathbf{s},\mathbf{a}),\, \boldsymbol{a}\sim\pi(\boldsymbol{s})}\left[Q^\pi(\mathbf{s}',\mathbf{a}')\right].$$

The optimal value functions, used in value iteration

$$V^*(\mathbf{s}) = \max_{\mathbf{a}} r(\mathbf{s},\mathbf{a}) + \gamma\mathbb{E}_{\mathbf{s}'\sim\mathcal{P}(\mathbf{s},\mathbf{a})}\left[V^*(\mathbf{s}')\right],$$
$$Q^*(\mathbf{s},\mathbf{a}) = r(\mathbf{s},\mathbf{a}) + \gamma\mathbb{E}_{\mathbf{s}'\sim\mathcal{P}(\mathbf{s},\mathbf{a})}\left[\max_{\mathbf{a}'} Q^*(\mathbf{s}',\mathbf{a}')\right].$$

With ADP, given a set of N transition samples from a policy acting on an environment $\mathcal{D} = \{\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}'_i\}_{i=1,\dots,N}$, and without access to the transition model, we can make use of the samples to learn the V and Q functions using *temporal difference* (TD) learning [108]. To understand TD learning let us start first with its tabular version. Given a single transition $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}'_t)$, we want to update the V function. If we have an estimate for the value function, and using definition 10 we can write $V(\mathbf{s}_t) = \hat{V}(\mathbf{s}_t) = r_t + \gamma V(\mathbf{s}_{t+1})$. Let us define the TD error as $\delta_t = \hat{V}(\mathbf{s}_t) - V(\mathbf{s}_t) = r_t + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t)$, which is then used to update the current value as

$$V_{\text{new}}(\mathbf{s}_t) = V(\mathbf{s}_t) + \alpha\delta_t = (1-\alpha)V(\mathbf{s}_t) + \alpha(r_t + \gamma V(\mathbf{s}_{t+1})),$$

Algorithm 4 Temporal Difference Learning for acquiring V^π

Init: $V_0(\mathbf{s}) \leftarrow 0$, $t = 0$, $\alpha \in [0, 1]$
repeat $t = t + 1$
  Observe transition $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}'_t)$ using policy π
  /* Compute TD error
  $\delta_t = r_t + \gamma V_t(\mathbf{s}_{t+1}) - V_t(\mathbf{s}_t)$
  /* Update value function
  $V_{t+1}(\mathbf{s}_t) = V_t(\mathbf{s}_t) + \alpha\,\delta_t$
until Convergence of V
return Value function V^π

where $\alpha \in [0, 1]$ is an hyperparameter to control the update. With α=1 the new value is solely based on the prediction. Note that if the TD error is positive, then the value increases, and otherwise it decreases. The TD learning sampled-based version for computing the value function of a policy π in discrete spaces is presented in algorithm 4.

To obtain an optimal control policy from a value function we can instead learn the state-action value function Q, whose TD update equations for Q-functions are

$$\begin{aligned} \delta_t &= r_t + \gamma Q_t(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q_t(\mathbf{s}_t, \mathbf{a}_t) \\ Q_{t+1}(\mathbf{s}_t, \mathbf{a}_t) &= Q_t(\mathbf{s}_t, \mathbf{a}_t) + \alpha\,\delta_t\,. \end{aligned}$$

There are two ways to choose the action $\mathbf{a}_{t+1}$ used for the next-state estimate. In the SARSA[4] algorithm, $\mathbf{a}_t \sim \pi(\cdot|\mathbf{s}_t)$ [95]. These samples are *on-policy* and produce estimates of the current Q-function. In the Q-learning algorithm, $\mathbf{a}_t = \arg\max_{\mathbf{a}} Q_t(\mathbf{s}_t, \mathbf{a})$ [121]. These samples are *off-policy*, because $\mathbf{a}_{t+1}$ is obtained using Q rather than π, and produce Q-function estimates of the *optimal* policy. Note that this action selection only tells us which action we should select to estimate the Q-function at the next-state.

As the agent is traversing the environment, we still need to decide which action to take in the current state. Should we take the best action according to our current estimate, or should we take another action at random? Behaving optimally and using the best action is known as 'greedy' behaviour, and may result in converging to local optima. Random actions perform exploration, but are suboptimal policies. This phenomenon is a well-established problem in reinforcement learning literature, known as the *exploration-exploitation* trade-off [109]. Two common strategies to ensure exploration are the ε-greedy and softmax policies (Definition 11).

[4] SARSA stands for state–action–reward–state–action.

Algorithm 5 Q-Learning

Init: $Q(\mathbf{s}, \mathbf{a}) \leftarrow$ arbitrarily, $\forall \mathbf{s} \in \mathcal{S}$, $\forall \mathbf{a} \in \mathcal{A}$, $Q(\mathbf{s}$ is terminal, $\cdot) = 0$, $\alpha \in [0, 1]$
repeat for each episode
  /* Sample an initial state
  $\mathbf{s} \leftarrow \mathbf{s}_0$
  repeat for each step
    Sample $\mathbf{a}$ from exploration policy (ε-greedy, softmax)
    Observe transition $(\mathbf{s}, \mathbf{a}, r, \mathbf{s}')$
    /* Update the Q-function
    $Q(\mathbf{s}, \mathbf{a}) \leftarrow Q(\mathbf{s}, \mathbf{a}) + \alpha \left[r + \gamma \max_{\mathbf{a}'} Q(\mathbf{s}', \mathbf{a}') \right]$
    $\mathbf{s} \leftarrow \mathbf{s}'$
  until $\mathbf{s}$ is a terminal state
until Convergence of Q
return Q^*

DEFINITION 11: EXPLORATION POLICIES

ε-**greedy** policy: $\pi(\mathbf{a}|\mathbf{s}) = \begin{cases} \arg\max_{\mathbf{a}'} Q(\mathbf{s}, \mathbf{a}') & \text{with probability } 1 - \varepsilon \\ \text{other action} & \text{with probability } \varepsilon \end{cases}$

Randomly switches between optimal and random actions.

Softmax policy: $\pi(\mathbf{a}|\mathbf{s}) = \dfrac{\exp\left(\beta Q(\mathbf{s}, \mathbf{a})\right)}{\sum_{\mathbf{a}'} \exp\left(\beta Q(\mathbf{s}, \mathbf{a}')\right)}$

Sample from a Boltzmann distribution, which originated from statistical thermodynamics, $p(\mathbf{a}|\mathbf{s}) \propto \exp(-E(\mathbf{s})/T)$ with energy $E(\boldsymbol{s})$ and temperature T, where the Q-function is the negative energy function and β is the inverse temperature.

Using these exploration strategies, the Q-learning algorithm [121] (Algorithm 5) is a classic method to learn an optimal Q-function, and the basis for many of the recent deep reinforcement learning algorithms such as deep Q-learning (DQN). In the tabular version, this algorithm is proven to converge as long as all states and actions are infinitely visited and $\alpha \to 0$ as training progresses.

Due to the curse-of-dimensionality, we cannot use the tabular version of Q-learning for continuous state spaces. For policy evaluation, we can approximate the value function with a function approximator, e.g. a neural network with parameters $\boldsymbol{\phi}$, $Q^{\pi}(\mathbf{s}, \mathbf{a}) \approx Q_{\boldsymbol{\phi}}(\mathbf{s}, \mathbf{a})$. Assuming $Q_{\boldsymbol{\phi}}$ is differentiable w.r.t. $\boldsymbol{\phi}$, to find the optimal parameters we can use a combination of TD learning and stochastic gradient descent on a mean squared error (MSE) of sampled transitions $\mathcal{D} = \{\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}'_i\}_{i=1,\dots,N}$ following a policy π. Using bootstrapping, we iteratively optimize the value function parameters $\boldsymbol{\phi}^* = \arg\min_{\boldsymbol{\phi}} \mathcal{L}_{\text{BS}}(\boldsymbol{\phi})$,

$$\mathcal{L}_{\mathrm{BS}}(\boldsymbol{\phi}) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\hat{Q}^{\pi}(\mathbf{s}_i, \mathbf{a}_i) - Q_{\boldsymbol{\phi}}(\mathbf{s}_i, \mathbf{a}_i) \right)^2,$$
$$\hat{Q}^{\pi}(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \mathbb{E}_{\mathbf{a}' \sim \pi(\cdot|\mathbf{s})} \left[Q_{\boldsymbol{\phi}}(\mathbf{s}'_i, \mathbf{a}') \right],$$
$$\boldsymbol{\phi}_{k+1} = \boldsymbol{\phi}_k - \alpha_k \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\phi}} \mathcal{L}_{\mathrm{BS}}(\boldsymbol{\phi}),$$

where $\hat{Q}^{\pi}$ is a bootstrapped target. In the context of value function learning, bootstrapping (BS) means that we use the old approximation to get the target values for a new approximation. Given an offline dataset of transitions, and a parametrized approximation $Q_{\boldsymbol{\phi}}$, the fitted Q-iteration algorithm uses a similar approach to Q-learning to find the (local) optimal parameters $\boldsymbol{\phi}^*$ that minimize the bootstrapped MSE [24, 92].

Learning value function approximations with TD learning and stochastic gradient descent presents several challenges. One is that it is not a 'proper' gradient descent routine, because the target values change after each parameter update. Nevertheless, for the target we ignore this aspect and keep the parameters $\boldsymbol{\phi}$ fixed [67]. This approach introduces a bias in the optimization, which means we are optimizing a different objective than the mean squared error. In certain cases the optimization procedure can even diverge, for example when training with off-policy samples [119]. One workaround is to maintain two value functions, where one is used for the target values and another is updated, and the target function is updated at a slower pace [67].

6.2 Policy Search Methods

Value function methods are well motivated and developed, but if we are only interested in the optimal policy, the added complexity of modelling the approximate value functions and ill-posed bootstrap training can introduce many issues. Many of these problems can be avoided by performing policy search [21, 81]. Instead of learning value functions to obtain a policy, we can learn the policy directly. Moreover, using task-specific policies is possible and beneficial, as discussed in Section 5. For instance, for a robotic task it seems more reasonable to learn the parameters of a dynamic movement primitive instead of a generic Q-function [97, 70].

For value-based methods, we used the Q-function to sample from a Boltzmann distribution. For policy search methods, we can parametrize the policy distribution explicitly. Similarly, for a Gaussian policy, which are popular for continuous control, we have $\pi(\mathbf{a}|\mathbf{s}; \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}(s), \boldsymbol{\Sigma}_{\boldsymbol{\theta}}(s))$ [31], a Gaussian process as seen in Section 3.

To make our explanation of model-free policy search clearer, we describe a generic procedure in Algorithm 6.

Algorithm 6 Generic model-free algorithm to learn π^*

Init: current policy π_k
repeat
  /* Exploration
  Generate (multiple) trajectories τ following the current policy π_k
  /* Policy Evaluation
  Assess the quality of the whole trajectory or single actions
  /* Policy Update
  Compute new policy π_{k+1} using the trajectories and evaluations
until Convergence of π
return π^*

An important distinction in policy search methods is whether we learn the policy parameters $\boldsymbol{\theta}$ in an episodic or step-based manner. The distinction of both methods is in the policy evaluation and update parts in Algorithm 6.

6.3 Episodic Policy Search

In episodic policy search, we learn a high-level policy that encodes a search distribution $\pi(\boldsymbol{\theta};\boldsymbol{\omega})$ over the parameters of a lower-level policy $\pi(\mathbf{a}|\mathbf{s};\boldsymbol{\theta})$. For instance, $\pi(\boldsymbol{\theta};\boldsymbol{\omega})$ can be a Gaussian distribution $\pi(\boldsymbol{\theta};\boldsymbol{\omega}) = \mathcal{N}(\boldsymbol{\theta};\boldsymbol{\omega}=\{\boldsymbol{\mu},\boldsymbol{\Sigma}\})$. Often in this context the lower-level policy is a deterministic policy $\mathbf{a} = \pi(\mathbf{s};\boldsymbol{\theta})$, which reduces the variance in the returns and allows for smooth exploration in the real robot [21]. Algorithm 7 shows the generic approach to episodic policy search.

DEFINITION 12: EPISODIC POLICY SEARCH

Formally, episodic policy search solves the problem

$$\boldsymbol{\omega}^* = \arg\max_{\boldsymbol{\omega}} J_{\boldsymbol{\omega}} = \arg\max_{\boldsymbol{\omega}} \mathbb{E}_{\boldsymbol{\theta}\sim\pi(\boldsymbol{\theta};\boldsymbol{\omega})}\left[R(\boldsymbol{\theta})\right],$$

where $R(\theta)$ is the expected episodic return of the lower-level policy.

Using the Gaussian process perspective from Section 3, we can see the high-level as the weight distribution that's iterative updated using the data (episodic experience) and the lower-level policy is the regression model itself.

Policy Gradient. Solving the policy update in Algorithm 7 without any constraints involves maximizing an expectation. For that we can resort to gradient-based methods, which include the computation of a gradient estimate, followed by a policy update by applying that gradient with a learning rate α, $\boldsymbol{\omega}_{k+1} = \boldsymbol{\omega}_k + \alpha\nabla_{\boldsymbol{\omega}} J_{\boldsymbol{\omega}}$, where the gradient can be estimated using methods such as finite differences or likelihood ratio gradients (Definition 13) [80, 69].

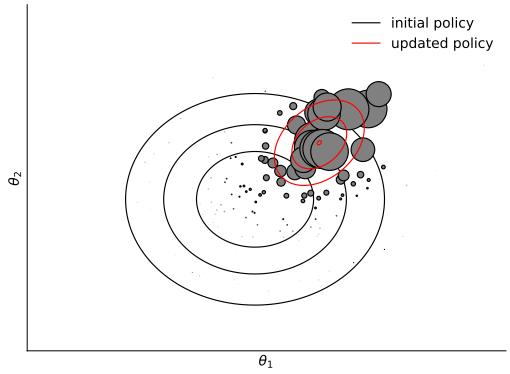


Fig. 6: Illustration of a policy update in parameter space under a Gaussian parameterization. The grey circles represent samples from the initial policy, and their size the corresponding reward. The updated policy Gaussian (red) was approximated from the weighted samples. The REPS algorithm applies a constraint to this update in terms of the KL divergence between the posterior and prior parameter distributions.

DEFINITION 13: LIKELIHOOD RATIO EPISODIC POLICY GRADIENT

The likelihood ratio gradient of the expected return $J_\omega = \mathbb{E}_{\boldsymbol{\theta}\sim\pi(\boldsymbol{\theta};\omega)}\left[R(\boldsymbol{\theta})\right]$ is given by

$$\nabla_\omega J_\omega = \nabla_\omega \int \pi(\boldsymbol{\theta};\boldsymbol{\omega}) R(\boldsymbol{\theta}) \mathrm{d}\boldsymbol{\theta} \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\omega \log \pi(\boldsymbol{\theta}^i;\boldsymbol{\omega}) R^i.$$

Usually an $\boldsymbol{\omega}$-independent baseline b is subtracted to R^i to obtain a lower variance gradient estimate, while keeping the estimator unbiased.

Probabilistic Search. To overcome the premature convergence seen with some policy gradient methods, an alternative is to perform a gradient step by considering

Algorithm 7 Generic episodic policy search algorithm

Init: initial upper-level policy $\pi(\boldsymbol{\theta};\boldsymbol{\omega}_0)$
repeat
  /* Exploration
  Sample lower-level policies from the upper-level distribution $\boldsymbol{\theta}^i \sim \pi(\boldsymbol{\theta};\boldsymbol{\omega}_k)$, $i = 1,\ldots,N$
  /* Policy Evaluation
  Collect returns $\mathcal{D}_{\text{episode}} = \{\boldsymbol{\theta}^i, R^i\}_{i=1,\ldots,N}$, with $R^i = \sum_{t=0}^{T-1} r_t^i$
  /* Policy Update
  Obtain a new policy $\pi(\boldsymbol{\theta};\boldsymbol{\omega}_{k+1})$ based on $\mathcal{D}_{\text{episode}}$
until Convergence of $\pi(\boldsymbol{\theta};\boldsymbol{\omega})$
return $\pi(\boldsymbol{\theta};\boldsymbol{\omega}^*)$

the underlying probability metric [124]. However, in this approach a learning rate is still an hyperparameter to be tuned. A different direction is to write the optimization to ensure that policy updates are only ε-apart in distribution space, according to the KL-divergence. This idea is behind the episodic version of the relative entropy policy search (REPS) algorithm (Definition 14) [79], which does not solve the policy update by gradient updates, but rather by solving the constrained optimization problem in closed-form using Lagrangian multipliers. The update resembles the recursive Bayesian update, see Section 3, where the exponential return acts as an unnormalized likelihood. As a result, approximate inference methods such as importance sampling as used to update the parameter distribution (Figure 6).

DEFINITION 14: RELATIVE ENTROPY POLICY SEARCH

Episodic policy search can be formulated as a relative entropy search problem

$$\max_{\pi(\boldsymbol{\theta};\boldsymbol{\omega})} \int \pi(\boldsymbol{\theta};\boldsymbol{\omega}) R(\boldsymbol{\theta}) \mathrm{d}\boldsymbol{\theta} \quad \text{s.t. } D_{\mathrm{KL}}\left[\pi(\boldsymbol{\theta};\boldsymbol{\omega}) \| q(\boldsymbol{\theta})\right] \leq \varepsilon, \int \pi(\boldsymbol{\theta};\boldsymbol{\omega}) \mathrm{d}\boldsymbol{\omega} = 1,$$

where in a sequential policy update q is usually the previous (or initial) policy. The closed-form solution to this maximization problem is

$$\pi(\boldsymbol{\theta};\boldsymbol{\omega}) \propto q(\boldsymbol{\theta}) \exp\left(R(\boldsymbol{\theta})/\eta^*\right),$$

where η^* is the Lagrangian multiplier that results from minimizing the dual function $\eta^* = \arg\min_\eta \eta\varepsilon + \eta \log \int q(\boldsymbol{\theta}) \exp\left(R(\boldsymbol{\theta})/\eta\right) \mathrm{d}\boldsymbol{\theta}$.

6.4 Step-based Policy Search

Contrary to episodic policy search, which explores in parameter space, in step-based methods the exploration is done in the action space by the lower-level policy $\pi(\mathbf{a}|\mathbf{s})$ [21]. Algorithm 8 shows the generic approach to step-based policy search.

DEFINITION 15: STEP-BASED POLICY SEARCH

Formally, step-based policy search solves the problem

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}, \quad J_{\boldsymbol{\theta}} = \mathbb{E}_{\boldsymbol{\tau}\sim\mu_0,\mathcal{P},\pi_{\boldsymbol{\theta}}}\left[R(\boldsymbol{\tau})\right] = \int p(\boldsymbol{\tau};\boldsymbol{\theta}) R(\boldsymbol{\tau}) \mathrm{d}\boldsymbol{\tau},$$

where $\boldsymbol{\tau} = \mathbf{s}_0\mathbf{a}_0 \ldots \mathbf{s}_{T-1}\mathbf{a}_{T-1}$ is a state-action trajectory, μ_0 is the initial state distribution, $\mathcal{P}$ the transition dynamics, $\pi_{\boldsymbol{\theta}}$ the parametrized policy, and $R(\boldsymbol{\tau})$ the trajectory return.

Algorithm 8 Generic step-based policy search algorithm

Init: initial lower-level policy $\pi(\mathbf{a}|\mathbf{s};\boldsymbol{\theta})$
repeat
  /* Exploration
  Sample trajectories $\boldsymbol{\tau}^i$ using the (stochastic) lower-level policy
  /* Policy Evaluation
  Collect step-based returns $\mathcal{D}_{\text{step}} = \{\mathbf{s}_t^i, \mathbf{a}_t^i, Q_t^i\}_{t=0,\dots,T-1;i=1,\dots,N}$, with $Q_t^i = \sum_{h=t}^{T-1} r_h^i$
  /* Policy Update
  Obtain a new policy $\pi(\mathbf{a}|\mathbf{s};\boldsymbol{\theta}_{k+1})$ based on $\mathcal{D}_{\text{step}}$
until Convergence of $\pi(\mathbf{a}|\mathbf{s};\boldsymbol{\theta})$
return $\pi(\mathbf{a}|\mathbf{s};\boldsymbol{\theta}^*)$

The step-based exploration and policy evaluation are straight forward to compute, but how do we approach the policy update? We can again use a gradient-based approach and perform gradient ascent on the objective $J_{\boldsymbol{\theta}}$. Again, we make use of the likelihood ratio trick to compute the gradient of an expectation. With the Markovian structure of the MDP we write the trajectory distribution as $p(\boldsymbol{\tau};\boldsymbol{\theta}) = p(\mathbf{s}_0)\prod_{t=0}^{T-1} p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)\pi(\mathbf{a}_t|\mathbf{s}_t;\boldsymbol{\theta})$. The likelihood ratio gradient results in

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}} &= \int p(\boldsymbol{\tau};\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\tau};\boldsymbol{\theta}) R(\boldsymbol{\tau}) \mathrm{d}\boldsymbol{\tau} \\ &= \int p(\boldsymbol{\tau};\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}} \left(\sum_{t=0}^{T-1} \log \pi(\mathbf{a}_t|\mathbf{s}_t;\boldsymbol{\theta}) \right) R(\boldsymbol{\tau}) \mathrm{d}\boldsymbol{\tau}, \end{aligned} \tag{4}$$

where we used the fact that the transition probability does not depend on $\boldsymbol{\theta}$.

Hence, to compute the policy gradient, it suffices to be able to sample trajectories from the environment, without requiring the analytical form of the model. The return of the trajectory $R(\boldsymbol{\tau})$ can be decomposed further, by noticing that rewards that appear before the state and action at time t can be dropped, since they were not caused by future states and actions [7]. After some algebraic manipulation and noticing that rewards to come after a state-action pair are simply the Q-function, we arrive at the policy gradient theorem in Definition 16 [110].

DEFINITION 16: POLICY GRADIENT THEOREM

The policy gradient theorem computes the gradient of the infinite horizon objective from Definition 9 w.r.t. the policy parameters as

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \mathbb{E}_{\mathbf{s}\sim\mu_\gamma^\pi}\left[\int \nabla_{\boldsymbol{\theta}}\pi(\mathbf{a}|\mathbf{s};\boldsymbol{\theta})Q^\pi(\mathbf{s},\mathbf{a})\mathrm{d}\mathbf{a}\right] \\ &= \mathbb{E}_{\mathbf{s}\sim\mu_\gamma^\pi}\left[\int \pi(\mathbf{a}|\mathbf{s};\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}}\log\pi(\mathbf{a}|\mathbf{s};\boldsymbol{\theta})Q^\pi(\mathbf{s},\mathbf{a})\mathrm{d}\mathbf{a}\right],\end{aligned}$$

where μ_γ^π is the discounted state distribution, $\mu_\gamma^\pi(\mathbf{s}) = (1-\gamma)d_\gamma^\pi(\mathbf{s}) = (1-\gamma)\sum_{t=0}^\infty \gamma^t P(\mathbf{s}_t = \mathbf{s}|\mathbf{s}_0, \pi)$, with $\mathbf{s}_0$ the initial state.
An important conclusion of this theorem is that we need only to be able to sample from the discounted state distribution, but do not need to compute its gradient. In practice, to reduce the estimator's variance, while keeping its unbiasedness, the Q function estimate is replaced by the advantage function $A^\pi(\mathbf{s},\mathbf{a}) = Q^\pi(\mathbf{s},\mathbf{a}) - V^\pi(\mathbf{s})$.

6.5 Episodic- or Step-based Policy Search?

There are clear benefits and drawbacks of both approaches, and both methods should be considered taking into account the robotic application.

Some characteristics of the episode-based methods are the exploration in parameter space, which allows for more sophisticated exploration strategies of deterministic policies that can be deployed in a real robot [47, 84]. It is typically efficient for a small amount of parameters (up to ∼100); generalization and multi-task learning of open-loop policies such as DMPs. It is also a structure-less optimization problem and it can be solved with any black-box optimizer, such as genetic algorithms or Bayesian optimization [14].

Some characteristics of the step-based methods are exploration in action space allows for finer grade exploration at every time step. A stochastic exploration in action space is difficult to deploy in the real system and correlated exploration is needed [120]. Future rewards can be traced to a particular action, which leads to better credit assignment [107]. There is less variance in policy evaluation, and there are more data points to update the policy. It is less likely to create unstable policy updates, as it uses the structure of the MDP. The policy gradient theorem shows that we can update the policy immediately after collecting a single sample [110, 58, 25, 31].

7 Imitation Learning

In contrast to reinforcement learning, in imitation learning we optimize a policy to *imitate* a set of provided demonstrations. It is common to classify imitation learning algorithms between those that aim to learn an explicit action generator (policy) $\pi : \mathcal{S} \rightarrow \mathcal{A}$, that generates actions similar to the ones from the demonstrations and the algorithms that aim to learn an implicit state-action function (reward) $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ that will learn to set a high reward to those state-action pairs that are similar to the ones in the demonstrations. We will refer to the first as imitation learning and to the second as inverse reinforcement learning algorithms.

7.1 Learning Policies by Imitation

In this subsection, we introduce a set of algorithms to learn a robot policy π given a set of state-action demonstrations $\mathcal{D}$. *Behavioral cloning* (BC) refers to the approach of directly learning a conditioned distribution from a set of demonstrations

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\boldsymbol{s},\boldsymbol{a} \sim p(\mathcal{D})} [\log \pi(\boldsymbol{a}|\boldsymbol{s})], \tag{5}$$

which is closely related to regression or density estimation. An alternative approach considers the problem of learning a policy that matches the state-action occupancy distribution of the expert demonstrations

$$\pi^* = \min_{\pi} \psi(\rho_\pi, \rho_\mathcal{D}) - \lambda \mathbb{H}(\pi) \tag{6}$$

with ψ being a divergence distance that measures how close the stationary distribution of the expert demonstrations and the policy are. $\mathbb{H}(\pi)$ is the 'causal entropy' of the policy that helps during the learning process by regularizing π. We call this approach *apprenticeship learning* (AL) [35][5].

Behavioral Cloning. Revising Equation 5, the BC approach directly models the condition action-state distribution using regression or density estimation. As a result, it can be used offline and does not require online data collection. While this method is very effective for many problems, the limitations of naive behavioral cloning are clear from the formulation. The approach needs many demonstrations to generalize effectively and it may easily suffer from *covariate shift* [85]. In regression, covariate shift refers to the performance degredation when a model is used outside its training data distribution. For BC, it manifests as a policy that cannot act or recover in unseen states. Given the problem of behavioral cloning only optimizes over the instant state-action pairs; if the robot finds itself in a state that does not belong to

[5] Note, apprenticeship learning is sometimes used interchangably with imitation learning in the older literature.

Fig. 7: Performance of a learned policy for helicopter acrobacy. The policy was trained by apprenticeship learning to mimic a set of expert demonstrations doing acrobatics [17].

the demonstrations, the robot has to rely on extrapolation and interpolation to know what action to take.

A possible solution to this problem is given by DAGGER [93]. This algorithm proposes to measure how certain the robot is in a particular state. By computing the state distribution, if the robot lies in a state that was not in the demonstrations, the robot will be uncertain on which action to take. If the robot is uncertain, it will ask the teacher for additional demonstrations. These demonstrations will be added to the set of demonstrations and the robot will re-learn the policy. DAGGER proposes to solve the imitation learning problem by actively increasing the demonstrations dataset.

Another solution is proposed by DART [49]. This algorithm, rather than querying additional demonstrations from the teacher, it will inject additional noise in the action while the teacher provides demonstrations. This noise injection allows to generate more diverse demonstrations and encourages robust policies. The recorded demonstrations will provide information on how to recover from states close to the optimal trajectories.

Apprenticeship Learning. In contrast with behavioral cloning, apprenticeship learning proposes to learn a policy that matches the stationary state-action pairs rather than the conditioned distribution. The obtained policy is going to be more robust than the policies learned by behavioral cloning. On the other hand, apprenticeship learning methods require a dynamic model from which new samples can be generated, or online samples.

Different apprenticeship learning algorithms have proposed different divergence distances ψ, to measure the distance between the occupancy distributions $\rho(\boldsymbol{s}, \boldsymbol{a})$. The classical apprenticeship learning [1] proposes to model the distance metric ψ by contrastive divergence

$$\psi(\rho_\pi, \rho_\mathcal{D}) = \max_c \mathbb{E}_{\boldsymbol{s},\boldsymbol{a}\sim\rho_\pi}[c(\boldsymbol{s}, \boldsymbol{a})] - \mathbb{E}_{\boldsymbol{s},\boldsymbol{a}\sim\rho_\mathcal{D}}[c(\boldsymbol{s}, \boldsymbol{a})]$$

while a more modern approach, generative adversarial imitation learning (GAIL) [35], gets inspiration from generative adversarial networks (GAN) and substitute the contrastive divergence by the Jensen-Shannon divergence

$$\psi(\rho_\pi, \rho_\mathcal{D}) = \max_c \mathbb{E}_{\boldsymbol{s},\boldsymbol{a}\sim\rho_\pi}[\log(c(\boldsymbol{s},\boldsymbol{a}))] + \mathbb{E}_{\boldsymbol{s},\boldsymbol{a}\sim\rho_\mathcal{D}}[\log(1 - c(\boldsymbol{s},\boldsymbol{a}))].$$

In practice, apprenticeship learning algorithms combine a reinforcement learning phase with a cost learning phase. Given an initial policy π a set of trajectory demonstrations are generated by evaluating the policy on the system. The cost function $c(\boldsymbol{s},\boldsymbol{a})$ is trained to set high cost to the samples generated by the policy π and low cost to the samples generated by the demonstrations. Then, the policy is trained by reinforcement learning to maximize over the cost function $c(\boldsymbol{s},\boldsymbol{a})$. By iteratively training both the cost and the policy, the policy will learn to generate trajectories that match the stationary distribution of the expert demonstrations. One of the most successful use cases of apprenticeship learning is for helicopter acrobacy learning (Fig. 7). A policy trained on human expert demonstrations was able to perfectly mimic the expert on doing acrobacies in the sky.

7.2 Learning Rewards from Data: Inverse Reinforcement Learning

Inverse reinforcement learning (IRL) deals with the problem of learning the reward function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ that the expert demonstrations are trying to maximize [74]. While most of the apprenticeship learning algorithms learn a proxy reward function, in this section we will focus in the learning of the reward itself.

The problem of inverse reinforcement learning can be framed as finding the true reward $r^*(\boldsymbol{s}_t,\boldsymbol{a}_t)$ where

$$\mathbb{E}_{\boldsymbol{a}_t\sim\pi_\mathcal{D}(\boldsymbol{s}_t),\,\boldsymbol{s}_{t+1}\sim p(\cdot|\boldsymbol{s}_t,\boldsymbol{a}_t),\,\boldsymbol{s}_0\sim p(\boldsymbol{s}_0)}\left[\sum_t \gamma^t r^*(\boldsymbol{s}_t,\boldsymbol{a}_t)\right] \geq$$
$$\mathbb{E}_{\boldsymbol{a}_t\sim\pi(\boldsymbol{s}_t),\,\boldsymbol{s}_{t+1}\sim p(\cdot|\boldsymbol{s}_t,\boldsymbol{a}_t),\,\boldsymbol{s}_0\sim p(\boldsymbol{s}_0)}\left[\sum_t \gamma^t r^*(\boldsymbol{s}_t,\boldsymbol{a}_t)\right], \quad \forall\ \pi \in \Pi.$$

such that the reward r^* is maximized for the behavioral cloned policy $\pi_\mathcal{D}$ compared to the other policies in a set Π. The problem definition is challenging given it is an ill-posed problem. Setting $r(\boldsymbol{s}_t,\boldsymbol{a}_t) = 0$ satisfies the inequality, which is not a meaningful reward function. Moreover, we have only a finite set of expert demonstrations and not the whole expert state-action distribution. We assume the demonstrations come from an optimal expert policy rather than a suboptimal one, and finally we assume we can consider all possible policies through Π.

In the following, we introduce two methods to tackle the problem: Maximum margin [91] and maximum entropy IRL [130, 129]. In both cases, we will consider a reward function modelled as a linearly-weighted sum of features $r(\boldsymbol{s}) = \boldsymbol{w}^\intercal\phi(\boldsymbol{s})$. Given the features are linear,

$$\hat{R}^* = \mathbb{E}_{\boldsymbol{a}_t \sim \pi_{\mathcal{D}}(\boldsymbol{s}_t), \boldsymbol{s}_{t+1} \sim p(\cdot|\boldsymbol{s}_t, \boldsymbol{a}_t), \boldsymbol{s}_0 \sim p(\boldsymbol{s}_0)} \left[\sum_t \gamma^t r^*(\boldsymbol{s}_t) \right],$$
$$= \mathbb{E}\left[\sum_t \gamma^t \boldsymbol{w}^\intercal \phi(\boldsymbol{s}_t) \right] = \boldsymbol{w}^\top \mathbb{E}\left[\sum_t \gamma^t \phi(\boldsymbol{s}_t) \right] = \boldsymbol{w}^\top \Phi(\pi_{\mathcal{D}}),$$

where $\Phi(\pi_{\mathcal{D}})$ represents the average sum of discounted features induced by $\pi_{\mathcal{D}}$.

Maximum Margin Inverse Reinforcement Learning [91] proposes to solve the IRL problem by optimizing

$$\min_{\boldsymbol{w}} \boldsymbol{w}^\top \boldsymbol{w} \quad \text{s.t.} \quad \boldsymbol{w}^\intercal \Phi(\pi_{\mathcal{D}}) \geq \boldsymbol{w}^\intercal \Phi(\pi) + m(\pi_{\mathcal{D}}, \pi), \quad \forall\, \pi \in \Pi.$$

The optimization problem will search for the smallest weights that satisfy that the reward in the demonstrations policy $\pi_{\mathcal{D}}$ is bigger than the reward for the rest of the policies up to a margin term $m(\pi_{\mathcal{D}}, \pi)$. The chosen margin should increase for policies that are very different from $\pi_{\mathcal{D}}$. This approach is closely related to support vector machines, which are maximum margin models for classification. The connection here is that our reward function is acting as a classifier to differentiate $\pi_{\mathcal{D}}$ from the other policies in Π.

Maximum Entropy Inverse Reinforcement Learning [130] frames the problem as a maximum entropy problem. Given τ is a state-action trajectory and given an expert policy $\pi_{\mathcal{D}}$, we aim to find

$$\arg\max_p \mathbb{H}(p) = -\int_\tau p(\tau) \log p(\tau) \mathrm{d}\tau \quad \text{s.t.} \int_\tau p(\tau)\Phi(\tau)\mathrm{d}\tau = \Phi(\pi_{\mathcal{D}}).$$

with p being a distribution over the trajectories. The optimal solution of the maximum entropy problem is $p(\tau) \propto \exp\{\boldsymbol{w}^{*\top}\Phi(\tau)\}$, that is the maximally uncertain distribution subject to it agrees with all known constraints (expert policy). The optimal return for a trajectory is $R(\tau) = \boldsymbol{w}^{*\top}\Phi(\tau)$.

8 Outlook

This chapter has introduced Robot Learning in the context of optimizing controllers from sampled data. By building on ideas from optimal control, we have motivated model-based and model-free methods for policy optimization by incorporating tools from machine learning, including function approximation and gradient estimation. Moreover, we have extended these ideas to imitation learning, where policies and rewards can be estimated from expert human demonstration.

Due to the breadth and depth of Robot Learning, many topics and settings were not discussed in this chapter. This includes discussing the learning challenges on specific robot platforms, such as autonomous vehicles and legged robots. We also

omitted specific tasks that are important robotic problems, such as navigation, robotic grasping, state estimation and human-robot interaction. As an outlook, we wish to briefly discuss some key topics and open problems in the field of Robot Learning.

Deep Reinforcement Learning considers the combination of reinforcement learning algorithms with deep neural networks [68, 101, 32]. This approach has demonstrated impressive performance on complex tasks and has been widely adopted by the research community. However, these methods have seen less adoption in practical applications and products, due to their performance variance and lack of performance guarantees. Due to their sample efficiency and lack of safety guarantees, their use is mainly limited to learning in simulators rather than physical systems.

Inductive Biases for Deep Robot Learning. One means of improving sample efficiency and safety is to incorporate useful structures into machine learning models that enforce the desired characteristics [6]. The flexibility of automatic differentiation means that a broad range of structure can be incorporated into deep learning models, such as computer vision models [33], state estimation algorithms [30, 43], non-holonomic constraints [63], stability guarantees [118] and energy conservation [61]. Striking the right balance with inductive biases, providing enough useful structure without underfitting to the data, can provide Robot Learning methods with a greater degree of interpretability, safety and sample efficiency for deployment on real-world systems.

Parallelized and Differential Simulators. Extending the notion of inductive biases can eventually result in a feature-complete differentiable simulator, that can model rigid-body physics, contact and rendering [19, 115, 42, 128]. This enables simulators that themselves can learn from data. Moreover, gradients from the simulator can be used to accelerate learning in a model-based fashion. Finally, implementing these simulators on hardware accelerators such as GPUs provides dramatic speed-ups for sample-based methods such as deep reinforcement learning, which in turn accelerates Robot Learning research and deployment.

Transferring from Simulation to Reality. The widespread adoption of simulators in Robot Learning methods introduces the issue of the 'reality gap', the difference between the real-world and the simulated replication. Methods for 'sim2real' transfer consider means of designing or perturbing the simulator during training to minimize the adverse effects of the reality gap [114, 71, 88]. Effective sim2real methods, combined with advances in simulation, would facilitate more real-world deployment of many powerful RL algorithms.

Task and Motion Planning (TAMP) considers higher-level robot control that requires many subsystems, such as perception, motor control and navigation [15, 127]. While this chapter has focused on ML for motor control, ML methods can also be combined with classical approaches for TAMP to improve performance in a data-driven fashion [37, 54]. For example, it enables joint optimization of each system, allowing representations from perception to be optimized to aid planning, and navigation to to be fine-tuned for motion control. Advances in TAMP are crucial if robots are to be able to carry out complex tasks in everyday, unstructured environments.

Safe Learning for Robotics. The restriction of many Robot Learning methods to simulators is in part due to safety concerns when learning on physical systems. While 'safe' is an ambiguous term, it generally refers to algorithms that can learn with certain guarantees that prevent catastrophic failure of the robot [9, 34]. Safe methods can be developed by considering inductive biases, such as motion constraints of the robot, as seen in Section 5 with motion primitives. Safety can also be incorporated by considering the statistical uncertainty of the learned models.

Multimodal Robot Learning. A quality of deep learning models are their ability to easily combine different 'data modalities', such as vision, sound, speech and other telemetry [29]. Advances in vision and natural language processing can enable robots that can learn to integrate diverse sensor inputs [53], and also interpret and communicate with humans via speech or gestures for richer human-robot interaction [64, 65, 117].

In conclusion, Robot Learning represents an exciting paradigm for robotics, with many opportunities for robots to learn, adapt and improve the skills they need to carry out complex tasks and integrate with society.

References

1. Abbeel, P., and Ng, A. Y. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning* (2004), p. 1.
2. Aggarwal, C. C., et al. Neural networks and deep learning. *Springer 10* (2018), 978–3.
3. Åström, K. J., and Eykhoff, P. System identification—a survey. *Automatica* (1971).
4. Astrom, K. J., and Wittenmark, B. *Adaptive Control*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., USA, 1994.
5. Barber, D. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2011.
6. Baxter, J. A model of inductive bias learning. *Journal of artificial intelligence research* (2000).
7. Baxter, J., and Bartlett, P. L. Infinite-horizon policy-gradient estimation. *J. Artif. Int. Res. 15*, 1 (Nov. 2001), 319–350.
8. Bellman, R. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.
9. Berkenkamp, F., Turchetta, M., Schoellig, A. P., and Krause, A. Safe model-based reinforcement learning with stability guarantees. In *Advances in Neural Information Processing Systems* (2017), p. 908–919.
10. Bertsekas, D. *Dynamic programming and optimal control*. Athena scientific, 2012.
11. Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer New York, 2006.
12. Boyd, S., and Vandenberghe, L. *Convex optimization*. Cambridge university press, 2004.
13. Bryson, A. E. *Applied optimal control: Optimization, estimation and control*. Routledge, 2018.
14. Calandra, R., Seyfarth, A., Peters, J., and Deisenroth, M. Bayesian optimization for learning gaits under uncertainty. *Annals of Mathematics and Artificial Intelligence (AMAI) 76* (06 2015).
15. Cambon, S., Alami, R., and Gravot, F. A hybrid approach to intricate motion, manipulation and task planning. *The International Journal of Robotics Research 28*, 1 (2009), 104–126.
16. Chua, K., Calandra, R., McAllister, R., and Levine, S. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems* (2018).

17. Coates, A., Abbeel, P., and Ng, A. Y. *Autonomous Helicopter Flight Using Reinforcement Learning*. Springer US, Boston, MA, 2017, pp. 75–85.
18. Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems 2*, 4 (1989), 303–314.
19. Degrave, J., Hermans, M., Dambre, J., et al. A differentiable physics engine for deep learning in robotics. *Frontiers in neurorobotics 13* (2019), 6.
20. Deisenroth, M., and Rasmussen, C. E. Pilco: A model-based and data-efficient approach to policy search. In *International Conference on Machine Learning* (2011).
21. Deisenroth, M. P., Neumann, G., Peters, J., et al. A survey on policy search for robotics. *Foundations and Trends® in Robotics* (2013).
22. Dellaert, F. Factor graphs: Exploiting structure in robotics. *Annual Review of Control, Robotics, and Autonomous Systems 4*, 1 (2021), 141–166.
23. Dempster, A. P., Laird, N. M., and Rubin, D. B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)* (1977).
24. Ernst, D., Geurts, P., and Wehenkel, L. Tree-based batch mode reinforcement learning. *J. Mach. Learn. Res. 6* (Dec. 2005), 503–556.
25. Fujimoto, S., van Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. In *Proceedings of the 35th International Conference on Machine Learning* (2018), vol. 80, pp. 1587–1596.
26. Funk, N., Schaff, C., Madan, R., Yoneda, T., De Jesus, J. U., Watson, J., Gordon, E. K., Widmaier, F., Bauer, S., Srinivasa, S. S., Bhattacharjee, T., Walter, M. R., and Peters, J. Benchmarking structured policies and policy optimization for real-world dexterous object manipulation. *IEEE Robotics and Automation Letters 7*, 1 (2022), 478–485.
27. Geist, A. R., and Trimpe, S. Structured learning of rigid-body dynamics: A survey and unified view from a robotics perspective. *GAMM-Mitteilungen 44*, 2 (2021).
28. Ghahramani, Z., and Roweis, S. T. Learning nonlinear dynamical systems using an em algorithm. *Advances in neural information processing systems* (1999), 431–437.
29. Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
30. Haarnoja, T., Ajay, A., Levine, S., and Abbeel, P. Backprop kf: Learning discriminative deterministic state estimators. In *Advances in neural information processing systems* (2016).
31. Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of International Conference on Machine Learning (ICML)* (2018), vol. 80, pp. 1856–1865.
32. Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning* (2018), PMLR, pp. 1861–1870.
33. Handa, A., Bloesch, M., Pătrăucean, V., Stent, S., McCormac, J., and Davison, A. gvnn: Neural network library for geometric computer vision. In *European Conference on Computer Vision (ECCV)* (2016).
34. Hewing, L., Wabersich, K. P., Menner, M., and Zeilinger, M. N. Learning-based model predictive control: Toward safe learning in control. *Annual Review of Control, Robotics, and Autonomous Systems 3*, 1 (2020), 269–296.
35. Ho, J., and Ermon, S. Generative adversarial imitation learning. *Advances in neural information processing systems 29* (2016), 4565–4573.
36. Hofmann, T., Schölkopf, B., and Smola, A. J. Kernel methods in machine learning. *Annals of Statistics 36* (2008).
37. Hogg, C., Kuter, U., and Munoz-Avila, H. Learning methods to generate good plans: Integrating htn learning and reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Jul. 2010), vol. 24, pp. 1530–1535.
38. Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. *Neural networks 2*, 5 (1989), 359–366.

39. Ijspeert, A. J., Nakanishi, J., Hoffmann, H., Pastor, P., and Schaal, S. Dynamical movement primitives: learning attractor models for motor behaviors. *Neural computation 25*, 2 (2013), 328–373.
40. Izmailov, P., Vikram, S., Hoffman, M. D., and Wilson, A. G. G. What are bayesian neural network posteriors really like? In *Proceedings of the 38th International Conference on Machine Learning* (18–24 Jul 2021), M. Meila and T. Zhang, Eds., vol. 139 of *Proceedings of Machine Learning Research*, PMLR, pp. 4629–4640.
41. Jacobson, D. H., and Mayne, D. Q. *Differential dynamic programming*. North-Holland, 1970.
42. Jatavallabhula, K. M., Macklin, M., Golemo, F., Voleti, V., Petrini, L., Weiss, M., Considine, B., Parent-Levesque, J., Xie, K., Erleben, K., Paull, L., Shkurti, F., Nowrouzezahrai, D., and Fidler, S. gradsim: Differentiable simulation for system identification and visuomotor control. In *International Conference on Learning Representations (ICLR)* (2021).
43. Jonschkowski, R., Rastogi, D., and Brock, O. Differentiable particle filters: End-to-end learning with algorithmic priors. In *Proceedings of Robotics: Science and Systems* (Pittsburgh, Pennsylvania, June 2018).
44. Khansari-Zadeh, S. M., and Billard, A. Learning stable nonlinear dynamical systems with gaussian mixture models. *IEEE Transactions on Robotics 27*, 5 (2011), 943–957.
45. Khansari-Zadeh, S. M., and Billard, A. Learning control lyapunov function to ensure stability of dynamical system-based robot reaching motions. *Robotics and Autonomous Systems 62*, 6 (2014), 752–765.
46. Kober, J., Bagnell, J. A., and Peters, J. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research (IJRR)* (2013).
47. Kober, J., and Peters, J. Learning motor primitives for robotics. In *IEEE International Conference on Robotics and Automation (ICRA)* (2009), pp. 2112–2118.
48. Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems 25* (2012), 1097–1105.
49. Laskey, M., Lee, J., Fox, R., Dragan, A., and Goldberg, K. Dart: Noise injection for robust imitation learning. In *Conference on robot learning* (2017), PMLR, pp. 143–156.
50. Lázaro-Gredilla, M., and Figueiras-Vidal, A. R. Marginalized neural network mixtures for large-scale regression. *Transactions on Neural Networks 21*, 8 (2010).
51. LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature 521*, 7553 (2015), 436–444.
52. Lee, M. A., Zhu, Y., Srinivasan, K., Shah, P., Savarese, S., Fei-Fei, L., Garg, A., and Bohg, J. Making sense of vision and touch: Self-supervised learning of multimodal representations for contact-rich tasks. In *2019 International Conference on Robotics and Automation (ICRA)* (2019), IEEE, pp. 8943–8950.
53. Lee, M. A., Zhu, Y., Zachares, P., Tan, M., Srinivasan, K., Savarese, S., Fei-Fei, L., Garg, A., and Bohg, J. Making sense of vision and touch: Learning multimodal representations for contact-rich tasks. *IEEE Transactions on Robotics 36*, 3 (2020), 582–596.
54. Leonetti, M., Iocchi, L., and Stone, P. A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. *Artificial Intelligence 241* (September 2016), 103 – 130.
55. Levine, S., and Abbeel, P. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems* (2014).
56. Li, Q., Kroemer, O., Su, Z., Veiga, F. F., Kaboli, M., and Ritter, H. J. A review of tactile information: Perception and action through touch. *IEEE Transactions on Robotics 36*, 6 (2020), 1619–1634.
57. Li, W., and Todorov, E. Iterative linear quadratic regulator design for nonlinear biological movement systems. In *1st International Conference on Informatics in Control, Automation and Robotics* (2004).
58. Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. In *International Conference on Learning Representations, (ICLR)* (2016).

59. Lioutikov, R., Kroemer, O., Peters, J., and Maeda, G. Learning manipulation by sequencing motor primitives with a two-armed robot. In *Proceedings of the 13th International Conference on Intelligent Autonomous Systems* (2014), vol. 302 of *Advances in Intelligent Systems and Computing*, Springer, pp. 1601–1611.
60. Ljung, L. *System Identification - Theory For the User*. Prentice Hall, Upper Saddle River, N.J., 1999.
61. Lutter, M., Ritter, C., and Peters, J. Deep Lagrangian Networks: Using physics as model prior for deep learning. In *International Conference on Learning Representations (ICLR)* (2019).
62. Lutter, M., Silberbauer, J., Watson, J., and Peters, J. A differentiable newton euler algorithm for multi-body model learning. In *Robotics: Science and Systems Conference (RSS), Workshop on Structured Approaches to Robot Learning for Improved Generalization* (2020).
63. Lutter, M., Silberbauer, J., Watson, J., and Peters, J. Differentiable physics models for real-world offline model-based reinforcement learning. In *IEEE International Conference on Robotics and Automation (ICRA)* (2021).
64. Lynch, C., and Sermanet, P. Language conditioned imitation learning over unstructured data. *Robotics: Science and Systems* (2021).
65. Matuszek, C., Bo, L., Zettlemoyer, L., and Fox, D. Learning from unscripted deictic gesture and language for human-robot interactions. *Proceedings of the AAAI Conference on Artificial Intelligence 28*, 1 (Jun. 2014).
66. Mesbah, A. Stochastic model predictive control: An overview and perspectives for future research. *IEEE Control Systems Magazine 36*, 6 (2016), 30–44.
67. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR abs/1312.5602* (2013).
68. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature 518*, 7540 (2015), 529–533.
69. Mohamed, S., Rosca, M., Figurnov, M., and Mnih, A. Monte carlo gradient estimation in machine learning. *Journal of Machine Learning Research 21*, 132 (2020), 1–62.
70. Muelling, K., Kober, J., Kroemer, O., and Peters, J. Learning to select and generalize striking movements in robot table tennis. In *AAAI Fall Symposium on Robots that Learn Interactively from Human Teachers* (2012), pp. 263–279.
71. Muratore, F., Gienger, M., and Peters, J. Assessing transferability from simulation to reality for reinforcement learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2019).
72. Neal, R. M. *Bayesian Learning for Neural Networks*. PhD thesis, University of Toronto, CAN, 1995.
73. Neumann, K., and Steil, J. J. Learning robot motions with stable dynamical systems under diffeomorphic transformations. *Robotics and Autonomous Systems 70* (2015), 1–15.
74. Ng, A. Y., Russell, S. J., et al. Algorithms for inverse reinforcement learning. In *International Conference on Machine Learning* (2000), vol. 1, p. 2.
75. Nguyen-Tuong, D., and Peters, J. Using model knowledge for learning inverse dynamics. In *International Conference on Robotics and Automation* (2010).
76. Nguyen-Tuong, D., and Peters, J. Model learning for robot control: a survey. *Cognitive Processing* (2011).
77. Ober, S. W., and Rasmussen, C. E. Benchmarking the neural linear model for regression. In *Symposium on Advances in Approximate Bayesian Inference* (2019).
78. Paraschos, A., Daniel, C., Peters, J., Neumann, G., et al. Probabilistic movement primitives. *Advances in neural information processing systems* (2013).
79. Peters, J., Mülling, K., and Altün, Y. Relative entropy policy search. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence* (2010), p. 1607–1612.
80. Peters, J., and Schaal, S. Policy gradient methods for robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2006), pp. 2219–2225.

81. Peters, J., and Schaal, S. Reinforcement learning of motor skills with policy gradients. *Neural Networks 21*, 4 (May 2008), 682–697.
82. Peters, J., Tedrake, R., Roy, N., and Morimoto, J. *Robot Learning*. Springer US, Boston, MA, 2010, pp. 865–869.
83. Ploeger, K., Lutter, M., and Peters, J. High acceleration reinforcement learning for real-world juggling with binary rewards. In *Conference on Robot Learning (CoRL)* (2020).
84. Ploeger, K., Lutter, M., and Peters, J. High acceleration reinforcement learning for real-world juggling with binary rewards. In *Conference on Robot Learning (CoRL)* (2020).
85. Pomerleau, D. A. Alvinn: An autonomous land vehicle in a neural network. Tech. rep., Carnegie-Mellon University Pittsburgh PA Artificial Intelligence and Psychology Department, 1989.
86. Puterman, M. L. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
87. Rahimi, A., and Recht, B. Random features for large-scale kernel machines. In *Advances in Neural Information Processing Systems* (2008), J. Platt, D. Koller, Y. Singer, and S. Roweis, Eds., vol. 20, Curran Associates, Inc.
88. Ramos, F., Possas, R., and Fox, D. Bayessim: Adaptive domain randomization via probabilistic inference for robotics simulators. In *Robotics: Science and Systems* (2019).
89. Rana, M. A., Li, A., Fox, D., Boots, B., Ramos, F., and Ratliff, N. Euclideanizing flows: Diffeomorphic reduction for learning stable dynamical systems. In *Learning for Dynamics and Control* (2020), PMLR, pp. 630–639.
90. Rasmussen, C. E., and Williams, C. K. I. *Gaussian Processes for Machine Learning*. The MIT Press, 2005.
91. Ratliff, N. D., Bagnell, J. A., and Zinkevich, M. A. Maximum margin planning. In *Proceedings of the 23rd international conference on Machine learning* (2006), pp. 729–736.
92. Riedmiller, M. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. In *Proceedings of the 16th European Conference on Machine Learning* (Berlin, Heidelberg, 2005), ECML'05, Springer-Verlag, p. 317–328.
93. Ross, S., Gordon, G., and Bagnell, D. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (2011), JMLR Workshop and Conference Proceedings, pp. 627–635.
94. Rubinstein, R. Y., and Kroese, D. P. *The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-Carlo Simulation (Information Science and Statistics)*. Springer New York, 2004.
95. Rummery, G. A., and Niranjan, M. On-line Q-learning using connectionist systems. Tech. Rep. TR 166, Cambridge University Engineering Department, Cambridge, England, 1994.
96. Särkkä, S. *Bayesian Filtering and Smoothing*. Cambridge University Press, 2013.
97. Schaal, S. Dynamic movement primitives-a framework for motor control in humans and humanoid robotics. In *Adaptive motion of animals and machines*. Springer, 2006, pp. 261–280.
98. Schmidhuber, J. Deep learning in neural networks: An overview. *Neural networks 61* (2015), 85–117.
99. Schneider, J. G. Exploiting model uncertainty estimates for safe dynamic control learning. *Advances in Neural Information Processing Systems* (1997).
100. Schön, T. B., Lindsten, F., Dahlin, J., Wågberg, J., Naesseth, C. A., Svensson, A., and Dai, L. Sequential monte carlo methods for system identification. *IFAC-PapersOnLine 48*, 28 (2015), 775–786.
101. Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. Trust region policy optimization. In *International conference on machine learning* (2015), PMLR, pp. 1889–1897.
102. Shumway, R. H., and Stoffer, D. S. An approach to time series smoothing and forecasting using the em algorithm. *Journal of time series analysis 3*, 4 (1982), 253–264.

103. Sideris, A., and Bobrow, J. E. An efficient sequential linear quadratic algorithm for solving nonlinear optimal control problems. In *Proceedings of the 2005, American Control Conference, 2005.* (2005), IEEE, pp. 2275–2280.
104. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature 529*, 7587 (2016), 484–489.
105. Sutanto, G., Wang, A., Lin, Y., Mukadam, M., Sukhatme, G., Rai, A., and Meier, F. Encoding physical constraints in differentiable Newton-Euler Algorithm. *Learning for Dynamics Control (L4DC)* (2020).
106. Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (2014), pp. 3104–3112.
107. Sutton, R. S. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst, 1984. AAI8410337.
108. Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine Learning 3*, 1 (1988), 9–44.
109. Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.
110. Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems (NIPS)* (1999), pp. 1057–1063.
111. Tangirala, A. *Principles of System Identification: Theory and Practice*. CRC Press, 2018.
112. Tassa, Y. *Theory and Implementation of Biomimetic Motor Controllers*. PhD thesis, Hebrew University of Jerusalem, 2011.
113. Titsias, M. Variational learning of inducing variables in sparse gaussian processes. In *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics* (16–18 Apr 2009), vol. 5 of *Proceedings of Machine Learning Research*, PMLR, pp. 567–574.
114. Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. Domain randomization for transferring deep neural networks from simulation to the real world. In *International Conference on Intelligent Robots and Systems (IROS)* (2017).
115. Toussaint, M. A., Allen, K. R., Smith, K. A., and Tenenbaum, J. B. Differentiable physics and stable modes for tool-use and manipulation planning. In *Robotics: Science and Systems* (2018).
116. Traversaro, S., Brossette, S., Escande, A., and Nori, F. Identification of fully physical consistent inertial parameters using optimization on manifolds. In *International Conference on Intelligent Robots and Systems (IROS)* (2016).
117. Tucker, M., Aksaray, D., Paul, R., Stein, G. J., and Roy, N. Learning unknown groundings for natural language interaction with mobile robots. In *Robotics Research*. Springer, 2020, pp. 317–333.
118. Urain, J., Ginesi, M., Tateo, D., and Peters, J. Imitationflow: Learning deep stable stochastic dynamic systems by normalizing flows. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2020), IEEE, pp. 5231–5237.
119. van Hasselt, H., Doron, Y., Strub, F., Hessel, M., Sonnerat, N., and Modayil, J. Deep reinforcement learning and the deadly triad. *CoRR abs/1812.02648* (2018).
120. van Hoof, H., Tanneberg, D., and Peters, J. Generalized exploration in policy search. In *Machine Learning* (2017), pp. 1705–1724.
121. Watkins, C. J. C. H., and Dayan, P. Q-learning. *Machine Learning 8*, 3 (May 1992), 279–292.
122. Watson, J., Lin, J. A., Klink, P., Pajarinen, J., and Peters, J. Latent derivative bayesian last layer networks. In *Proceedings of the 24th International Conference on Artificial Intelligence and Statistics* (2021).
123. Watson, J., Lin, J. A., Klink, P., and Peters, J. Neural linear models with functional gaussian process priors. In *Third Symposium on Advances in Approximate Bayesian Inference* (2021).
124. Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J., and Schmidhuber, J. Natural evolution strategies. *J. Mach. Learn. Res. 15*, 1 (Jan. 2014), 949–980.

125. Williams, C. K. Computing with infinite networks. In *Advances in neural information processing systems* (1997), pp. 295–301.
126. Williams, G., Wagener, N., Goldfain, B., Drews, P., Rehg, J. M., Boots, B., and Theodorou, E. A. Information theoretic mpc for model-based reinforcement learning. In *2017 IEEE International Conference on Robotics and Automation (ICRA)* (2017), IEEE, pp. 1714–1721.
127. Wolfe, J., Marthi, B., and Russell, S. Combined Task and Motion Planning for Mobile Manipulation. In *International Conference on Automated Planning and Scheduling (ICAPS)* (2010).
128. Zhao, S., Jakob, W., and Li, T.-M. Physics-based differentiable rendering: From theory to implementation. In *ACM SIGGRAPH 2020 Courses* (New York, NY, USA, 2020), SIGGRAPH '20, Association for Computing Machinery.
129. Ziebart, B. D., Bagnell, J. A., and Dey, A. K. Modeling interaction via the principle of maximum causal entropy. In *ICML* (2010).
130. Ziebart, B. D., Maas, A. L., Bagnell, J. A., Dey, A. K., et al. Maximum entropy inverse reinforcement learning. In *Aaai* (2008), vol. 8, Chicago, IL, USA, pp. 1433–1438.