Robot Learning

Summary

Fabian Damken

November 9, 2021

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Contents

List of Figures

List of Tables

List of Algorithms

1. Statistics, Linear Algebra and Calculus Refresher

1.1. Basics

1.1.1. Entropy

1.1.2. Gaussian Distribution and Properties

1.2. Kullback-Leibler Divergence

1.2.1. Fisher Information Matrix

1.3. Monte-Carlo Integration and Gradient Estimation

1.3.1. Gradient Estimation

1.4. Linear Algebra and Calculus

1.4.1. Moore-Penrose Pseudo-Inverse

2. Robotics

This chapter covers the basics of robotics, covering modeling position, velocity, acceleration and forces as well as representing trajectories. Also the main concepts of linear and model-based control are covered.

The definition of what a robot are quite diverse. The robotics institute of America defines a robot as follows: "A robot is a reprogrammable multi-functional manipulator designed to move material, parts, tools, or specialized devices through variable programmed motions for the performance of a variety of tasks." Another, rather inverse, definition is from G. Randelov: "A computer is just an amputee robot."

2.1. Modeling Robots

Modeling of a robot can be split into two separate categories: kinematics and dynamics. In *kinematics*, only the geometric properties of the robot are modeled, e.g. the link length. In *dynamics*, the forces acting on the links and joints are analyzed. There are two types of joints that can be used to model every joint out there: revolute and prismatic joints, whilst revolute joints are the most common ones. The displacement of a revolute joint is an angle (typically in radians), the displacement of a prismatic joint is a distance (typically in meters).

The displacements of all joints in a robot is denoted with a vector $\boldsymbol{q}$, the task-space (e.g. the Cartesian coordinate system of the world) is denoted by $\boldsymbol{x}$ and the state (i.e. the variables of the robot and the environment) is denoted by $\boldsymbol{s}$. The set of places $\boldsymbol{x}$ in the task-space that the robot can reach is called the *workspace*. That is, everything outside the workspace is not reachable.

Actions that can be taken by the controlled are denoted by $\boldsymbol{u}$ or $\boldsymbol{a}$ and are most often in some way related to torques in the joints. A (control) policy π then maps the state $\boldsymbol{s}$ onto some action $\boldsymbol{u}$ to take, either in a deterministic or stochastic way:

- Deterministic: $\boldsymbol{u} = \pi(\boldsymbol{s})$
- Stochastic: $\boldsymbol{u} \sim \pi(\boldsymbol{u} \mid \boldsymbol{s})$

The complete system of a robot including the generation of the trajectory to perform as well as the controller is shown in Figure 2.1.

2.1.1. Kinematics

Forward kinematics tackle the problem to get the position of the end-effector given the current joint positions or, more general, the position of any point in reference to a set coordinate system. Forward kinematics therefore describe a mapping from joint- to task-space:

$$\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{q})$$

For simple robots, e.g. a two-dimensional robot with only few revolute joints or a robot with only prismatic joints, the forward kinematics can be found straightforwardly with geometric knowledge. For more complex robots, however, more sophisticated methods are needed to not waste a lot of time finding the forward kinematics model.

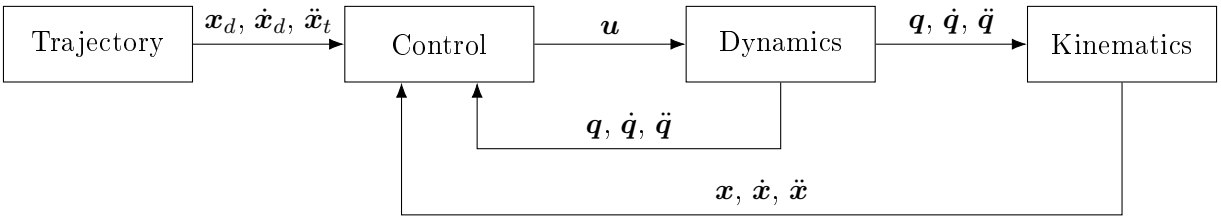


Figure 2.1.: Block diagram of a complete system with the desired values $\boldsymbol{x}_d$, $\dot{\boldsymbol{x}}_d$, $\ddot{\boldsymbol{x}}_t$, the joint angles/velocities/accelerations $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, $\ddot{\boldsymbol{q}}$, the end-effector positions/velocities/accelerations $\boldsymbol{x}$, $\dot{\boldsymbol{x}}$, $\ddot{\boldsymbol{x}}$, and the motor commands/torques $\boldsymbol{u}$.

Rotations and Euler Angles

To model revolute joints, a decent modeling of rotations is needed. In two-dimensional settings, a rotary transformation around an arbitrary angle α is described by the following matrix:

$$\boldsymbol{R}(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}$$

In three-dimensional settings, one transformation is needed for a rotation around one of the Cartesian axis. Letting x, y and z be the first, second and third Cartesian axis, respectively, the corresponding rotation matrices around an angle α are given as:

$$\boldsymbol{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \quad \boldsymbol{R}_y(\alpha) = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \quad \boldsymbol{R}_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

One of many methods of representing arbitrary rotations are *Euler angles* which are parameterized by roll ψ, pitch θ and yaw ϕ. They transform from a coordinate system 1 into a coordinate system 0 as follows:

$$\boldsymbol{R}_1^0 = \boldsymbol{R}_z(\phi)\boldsymbol{R}_y(\theta)\boldsymbol{R}_x(\psi)$$

Problems with Euler angles are that they are not unique, i.e. different Euler angles may describe the same rotation, and it is hard to quantify differences between Euler angles. An alternative for describing rotations are *unit quaternions* and angle-axis formulations, where both a rotation axis as well as a rotation angle are given. Both of them solve the singularities with Euler angles and it is easier to computer differences of orientations.

Homogeneous Transformations

Homogeneous transformations are a ways to represent combined translation and rotation transformations, e.g.

$$\boldsymbol{p}^1 = \boldsymbol{R}_2^1\boldsymbol{p}^2 + \boldsymbol{\delta}^1 \quad \Longrightarrow \quad \boldsymbol{p}^0 = \boldsymbol{R}_1^0\boldsymbol{p}^1 + \boldsymbol{\delta}^0 = \boldsymbol{R}_1^0\left(\boldsymbol{R}_2^1\boldsymbol{p}^2 + \boldsymbol{\delta}^1\right) + \boldsymbol{\delta}^0$$

into a single matrix-vector multiplication, making the computation less clumsy:

$$\boldsymbol{p}^0 = \boldsymbol{R}_1^0\boldsymbol{p}^1 + \boldsymbol{\delta}^0 \quad \Longrightarrow \quad \underbrace{\begin{bmatrix} \boldsymbol{p}^1 \\ 1 \end{bmatrix}}_{\tilde{\boldsymbol{p}}^1} = \underbrace{\begin{bmatrix} \boldsymbol{R}_1^0 & \boldsymbol{\delta}^1 \\ \boldsymbol{0} & 1 \end{bmatrix}}_{\boldsymbol{H}_1^0} \underbrace{\begin{bmatrix} \boldsymbol{p}^0 \\ 1 \end{bmatrix}}_{\tilde{\boldsymbol{p}}^0}$$

Hence, multiple translation-rotation transformations can be stacked as follows:

$$\tilde{\boldsymbol{p}}^0 = \boldsymbol{H}_1^0\boldsymbol{H}_2^1\cdots\boldsymbol{H}_n^{n-1}\tilde{\boldsymbol{p}}^n$$

Denavit-Hartenberg Convention

A common convention for describing the coordinate systems of a robot is the *Denavit-Hartenberg convention* which describes the kinematics of a complete robot with only four parameters per joint. These four parameters are:

- θ_i, the angle between x_{i-1} and x_i, measured around z_{i-1}. Variable if i is a revolute joint.
- d_i, the distance of the origin of S_{i-1} along z_{i-1} to the intersection with x_i. Variable if i is a prismatic joint.
- a_i, the distance between the intersection of z_{i-1} and x_i along x_i towards the origin of S_i.
- α_i, the angle between z_{i-1} and z_i, measured around x_i.

Along with that, the following conditions have to hold for the coordinate systems S_i:

- The origins of the coordinate systems lie on the movement axis.
- The z_{i-1}-axis lies along the movement axis of the i-th joint.
- The x_i-axis is perpendicular to the z_{i-1}-axis and point away from it.
- The x_i-axis and the z_{i-1}-axis have an intersection.

Placing the coordinate system can be formulated as an algorithm, check out the foundations of robotics summary[1] for more details.

2.1.2. Differential Forward Kinematics (Velocities and Accelerations)

Often it is necessary to get the velocities $\dot{\boldsymbol{x}}$ and accelerations $\ddot{\boldsymbol{x}}$ of the end-effector. These can be computed straightforwardly using the chain rule

$$\begin{aligned} \dot{\boldsymbol{x}} &= \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{f}(\boldsymbol{q}) = \frac{\mathrm{d}\boldsymbol{f}(\boldsymbol{q})}{\mathrm{d}\boldsymbol{q}}\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = \underbrace{\boldsymbol{J}(\boldsymbol{q})}_{\text{Jacobian}}\dot{\boldsymbol{q}} \\ \ddot{\boldsymbol{x}} &= \dot{\boldsymbol{J}}(\boldsymbol{q})\dot{\boldsymbol{q}} + \boldsymbol{J}(\boldsymbol{q})\ddot{\boldsymbol{q}} \end{aligned} \tag{2.1}$$

where the end-effector velocities and accelerations are computed from the joint velocities and accelerations.

Singularities

In some cases, the Jacobian $\boldsymbol{J}(\boldsymbol{q})$ might get rank-deficient, i.e. $\det \boldsymbol{J}(\boldsymbol{q}) = 0$. In this case, infinite velocities in the joints are required to reach a desired end-effector velocities as the Jacobian is not invertible anymore. These positions are called *singularities* of the robot and correspond to a loss in the degrees of freedom, e.g. when the robot stretches out its arm.

[1] `https://projects.frisp.org/documents/29` (German)

Computing the Jacobians

There are two main ways of computing the Jacobian:

Analytical As before, the Jacobian is derived analytically and has the problem of singularities.

Geometric These are derived from geometric insights, maybe circumventing the "representational singularities".

Both of these ways are just different representations of the same concept.

2.1.3. Inverse Kinematics

The obvious next problem is how to get the required joint positions to reach a given end-effector pose[2]. Hence, the *inverse kinematics* are a mapping from the task- to the joint-space:

$$\boldsymbol{q} = \boldsymbol{f}^{-1}(\boldsymbol{x})$$

For really simple robots, this can again be solved from a geometric perspective. However, there is most likely more than one solution for a given end-effector position! With certain redundancies in the robot, it is even possible to get infinite solutions.

An additional problem is that the inverse kinematics equations are often not solvable analytically, hence numerical methods have to be used. But those do not guarantee to find all solutions which might be necessary to move into an "optimal" joint configuration. However, for a lot of industrial robots, invertible solutions are possible due to intelligent joint and link design.

2.1.4. Dynamics

For the dynamics of a robot, a forward model

$$\ddot{\boldsymbol{q}} = \boldsymbol{f}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{u})$$

is wanted that gives the joint accelerations given the joint positions and velocities and torques/forces (in case of revolute/prismatic joints, respectively). Usually the dynamics are represented in the general form

$$\boldsymbol{u} = \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{c}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{g}(\boldsymbol{q}) \tag{2.2}$$

with the motor commands $\boldsymbol{u}$, the joint positions/velocities/accelerations $\boldsymbol{q}/\dot{\boldsymbol{q}}/\ddot{\boldsymbol{q}}$, the mass matrix $\boldsymbol{M}(\boldsymbol{q})$, the Coriolis and centripetal forces $\boldsymbol{c}(\boldsymbol{q}, \dot{\boldsymbol{q}})$, and the gravity $\boldsymbol{g}(\boldsymbol{q})$. This general form can be easily inverted to get the joint accelerations, as the mass matrix is always positive definite and hence invertible:

$$\ddot{\boldsymbol{q}} = \boldsymbol{M}^{-1}(\boldsymbol{q})\big(\boldsymbol{u} - \boldsymbol{c}(\boldsymbol{q}, \dot{\boldsymbol{q}}) - \boldsymbol{g}(\boldsymbol{q})\big) \tag{2.3}$$

Computing the Forces

To compute the rigid body forces, there are two central methods:

1. Newton-Euler Method
    - Force-dissection-based approach.

[2] A pose refers to both the position and the orientation at the same time.

- Can be formalized nicely, check out the foundations of robotics summary[3] for more details.

2. Lagrangian Method
   - Energy-based approach.
   - Based on Lagrangian mechanics, check out the theoretical physics: classical mechanics summary[4] for more details.

Other forces than rigid body forces, e.g. friction, are a lot harder to model as there is not general recipe for modeling them (friction is not so well understood).

Newton-Euler Method The Newton-Euler method is based on *force dissection* and exploiting of the required force equilibrium. It yields the *recursive Newton-Euler algorithm* that can be used to straightforwardly compute the torques/forces in the joints.

Lagrangian Methods The *Lagrangian method* is energy-based and works by defining the *Lagrangian*

$$L = T - V$$

where T is the kinetic energy and V is the potential energy. For a one-dimensional point mass with mass m in a gravity field with the gravity constant g, the energies are given as $T = \frac{1}{2}m\dot{y}^2$ and $V = mgy$, where y is the one degree of freedom. The Lagrangian is hence given as:

$$L = \frac{1}{2}m\dot{y}^2 - mgy$$

Let f be the force of e.g. a motor that is somehow attached to the mass, the equations of motion are given as the *Lagrangian equations of motion*:

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{y}} - \frac{\partial L}{\partial y} = f \quad \Longrightarrow \quad m\ddot{y} + mg = f$$

For multi-dimensional systems, this equation can be generalized straightforwardly as

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{x}_i} - \frac{\partial L}{\partial x_i} = f_i$$

where x_i is one degree of freedom as f_i is the respective force.

For robots, finding the potential energy is straightforward as it is just the sum of all potential energies in the links, i.e.

$$V = \sum_{i=1}^{n} V_i = \sum_{i=1}^{n} m_i \boldsymbol{g}^T \boldsymbol{r}_{c_i},$$

where $\boldsymbol{g}$ is the vector describing the gravitational acceleration and $\boldsymbol{r}_{c_i}$ is the position of the center of mass of the i-th link. The kinetic energy is a tad more complicated as it involves rotational energy and interactions between the different joints, e.g. Coriolis forces. It is given as

$$T = \frac{1}{2}\dot{\boldsymbol{q}}^T \left[\sum_{i=1}^{n} m_i \boldsymbol{J}_{\boldsymbol{v}_i}^T(\boldsymbol{q}) \boldsymbol{J}_{\boldsymbol{v}_i}(\boldsymbol{q}) + \boldsymbol{J}_{\boldsymbol{\omega}_i}^T(\boldsymbol{q}) \boldsymbol{R}_i(\boldsymbol{q}) \boldsymbol{I}_i \boldsymbol{R}_i^T(\boldsymbol{q}) \boldsymbol{J}_{\boldsymbol{\omega}_i}(\boldsymbol{q}) \right] \dot{\boldsymbol{q}},$$

where

[3] `https://projects.frisp.org/documents/29` (German)
[4] `https://projects.frisp.org/documents/31` (German)

$\boldsymbol{q}/\dot{\boldsymbol{q}}$ are the positions/velocities of the joints,

m_i is the mass of the i-th link,

$\boldsymbol{J}_{\boldsymbol{v}_i}$ is the linear Jacobian of the i-th joint,

$\boldsymbol{J}_{\boldsymbol{\omega}_i}$ is the rotational Jacobian of the i-th joint

$\boldsymbol{I}_i$ is the inertia tensor of the i-th link, and

$\boldsymbol{R}_i(\boldsymbol{q})$ is the orientation of the i-th link.

Comparison of Newton-Euler and Lagrangian Using the Newton-Euler approach manually, i.e. to find the equations directly, is quite tedious and therefore not suitable for large robots. The Lagrangian approach, on the other hand, is a lot faster to execute by hand and therefore appropriate for finding the equations explicitly.

In most cases, however, the algorithm is used computationally without formulating the equations explicitly. In this case, the Newton-Euler method is better suited as it has a computational complexity of $\mathcal{O}(n)$ whilst the Lagrangian approach has a complexity of $\mathcal{O}(n^3)$.

General Forms

If somehow an inverse dynamics model $\boldsymbol{u} = \boldsymbol{f}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}})$ is achieved in the general form (2.2), it is easy to compute the forward dynamics model $\ddot{\boldsymbol{q}} = \boldsymbol{f}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{u})$ by inverting the mass matrix as shown in (2.3). By integrating $\ddot{\boldsymbol{q}}$, the velocities and positions can be recovered:

$$\dot{\boldsymbol{q}}(t) = \int_0^t \ddot{\boldsymbol{q}}(\tau)\,\mathrm{d}\tau \qquad\qquad \boldsymbol{q}(t) = \int_0^t \dot{\boldsymbol{q}}(\tau)\,\mathrm{d}\tau$$

But this is (in most cases) not possible in closed form, so numerical integration techniques are required. One suitable method is for example the *symplectic Euler method*, also called the *semi-implicit Euler method*. It first integrates for the velocity with an explicit Euler method and subsequently integrates for the position with an implicit Euler method:

$$\dot{\boldsymbol{q}}_{k+1} = \dot{\boldsymbol{q}}_k + h\ddot{\boldsymbol{q}}(t_k) \qquad\qquad \boldsymbol{q}_{k+1} = \boldsymbol{q}_k + h\dot{\boldsymbol{q}}_{k+1}$$

Here, k is the step, $h \in (0, 1)$ is the step size with $t_k \coloneqq kh$, and $\boldsymbol{q}_k$ and $\dot{\boldsymbol{q}}_k$ are the respective approximations of the positions and velocities.

2.2. Representing Trajectories

A trajectory $\boldsymbol{q}_d(t)/\dot{\boldsymbol{q}}_d(t)/\ddot{\boldsymbol{q}}_d(t)$ specifies the joint positions/velocities/accelerations at any point in time and is used to specify movements for the robot to execute. A common way to specify trajectories is to specify via-points the robot has to reach and to interpolate between them as shown in Figure 2.2. But as the motor commands can only influence the acceleration direction and the velocities and positions are found with integration, the trajectory and the time-derivative of the trajectory must not jump! This can be achieved with polynomial spline-interpolation which will be discussed further in the next sections.

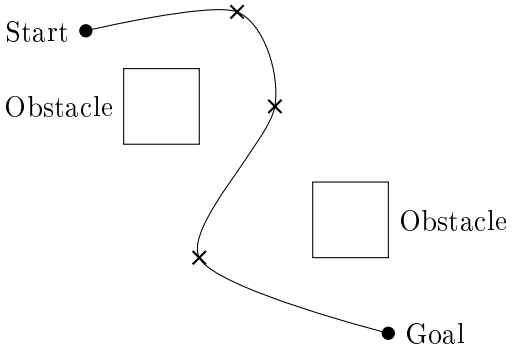


Figure 2.2.: Illustration of trajectory planning with via-points (the crosses) to avoid obstacles. The trajectory between the via-points is found with interpolation.

2.2.1. Splines

As already discussed, polynomial splines can be used to avoid jumps in the positions and velocities. The spline must be at least cubic as linear and quadratic splines do not have enough parameters to encode all the required information (and they have jumps in the velocities).

Cubic Splines

A cubic spline

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

has four parameters that can be found using the boundary conditions $q(t_0) = q_0$, $\dot{q}(t_0) = v_0$, $q(t_f) = q_f$ and $\dot{q}(t_f) = v_f$ where t_0 and t_f are the initial and final time step, respectively. This yields a system of linear equations,

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 0 & 1 & 2t_0 & 3t_0^2 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} q_0 \\ v_0 \\ q_f \\ v_f \end{bmatrix}$$

that can be solved for the parameters a_0, a_1, a_2 and a_3. The via-points are the initial and final position of the spline between them. In case of multiple degrees of freedom, multiple splines have to be used, i.e. the parameters become vectors.

But cubic splines still exhibit jumps in the acceleration which are dangerous at high speeds and might damage the robot!

Quintic Splines

An alternative are quintic splines

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

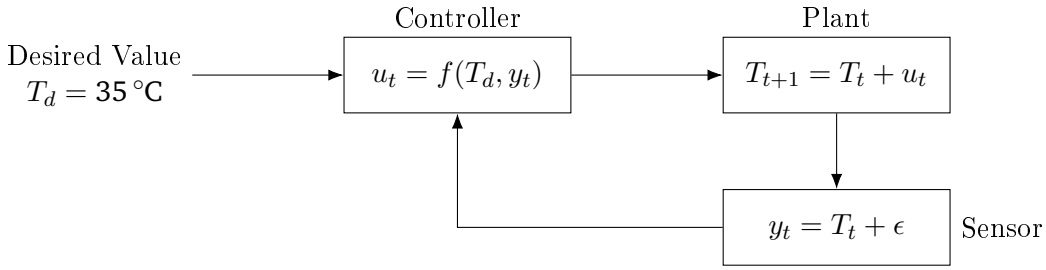


Figure 2.3.: Illustration of the feedback control loop in shower with a simple linear model of the temperature and the measurement errors ϵ.

which do not have jumps in the acceleration by imposing additional constraints on the accelerations, $\ddot{q}(t_0) = a_0$ and $\ddot{q}(t_f) = a_f$. This again yields a system of linear equations:

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 & t_0^4 & t_0^5 \\ 0 & 1 & 2t_0 & 3t_0^2 & 4t_0^3 & 5t_0^4 \\ 0 & 0 & 2 & 6t_0 & 12t_0^2 & 20t_0^3 \\ 1 & t_f & t_f^2 & t_f^3 & t_f^4 & t_f^5 \\ 0 & 1 & 2t_f & 3t_f^2 & 4t_f^3 & 5t_f^4 \\ 0 & 0 & 2 & 6t_f & 12t_f^2 & 20t_f^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} q_0 \\ v_0 \\ a_0 \\ q_f \\ v_f \\ a_f \end{bmatrix}$$

By equalizing the accelerations at the start and end of each spline, the acceleration becomes continuous, i.e. no jumps are present anymore.

2.2.2. Alternatives

Alternatives to splines are for example

- linear segments with parabolic blends,
- trapezoidal minimum time trajectories,
- potential fields $V(\boldsymbol{q})$ with $\dot{\boldsymbol{q}} = \frac{\partial V(\boldsymbol{q})}{\partial \boldsymbol{q}}$, and
- nonlinear dynamical systems $\ddot{\boldsymbol{q}} = \boldsymbol{f}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta}$.

2.3. Control in Joint-Space

Given a desired trajectory $\boldsymbol{q}_d(t)/\dot{\boldsymbol{q}}_d(t)/\ddot{\boldsymbol{q}}_d(t)$, it is still necessary to find the corresponding motor inputs $\boldsymbol{u}$ to follow this trajectory. This problem is called *control*.

The generic idea of *feedback control* is to take an action, measure the current state, compare it with the desired state, and adjust the action over and over again. A simple example for this is controlling the water temperature when taking a shower. An illustration of the control loop is shown in Figure 2.3 with a simple model of the actual temperature where the control input is purely additive. Additionally, the measured data is noisy, illustrated by the measurement errors ϵ. The function $f(T_d, y_t)$ is called the *control law* and determines the control inputs. A special case for this is linear feedback control, which will be discussed in the next section.

2.3.1. Linear Feedback Control

For *linear feedback control*, the control law $f(T_d, y_t)$ is a linear function $f(T_d, y_t) = K(T_d - y_t)$ that determines the control input based on the difference of the desired and the actual temperatures (the error). The parameter K is called the *gain* which amplifies the error for the control input. Obviously, too high or too low gains would cause an uncomfortable way to shower, either by overshooting around the desired temperature (too high gain) or by never reaching the desired temperature (too small gain). Also, messing up the sign (by choosing a negative K) can be catastrophic, as in this case it would drive the temperature towards absolute zero.

The following sections will generalize the idea of a linear feedback controller to multiple variables, where K becomes a matrix.

P-Controller

For a desired joint position $\boldsymbol{q}_d$, given the actual joint position $\boldsymbol{q}_t$, a *P-controller* just includes terms proportional to the positioning error:

$$\boldsymbol{u}_t = \boldsymbol{K}_P(\boldsymbol{q}_d - \boldsymbol{q}_t)$$

But a simple P-controller causes high oscillations in the position, making it not suitable for real control.

PD-Controller

An extended version, the *PD-controller*, additionally includes differential values in terms of the joint velocities. The control law is then given as

$$\boldsymbol{u}_t = \boldsymbol{K}_P(\boldsymbol{q}_d - \boldsymbol{q}_t) + \boldsymbol{K}_D(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}_t)$$

with the gains $\boldsymbol{K}_P$ and $\boldsymbol{K}_D$. This control law eliminates the oscillations, but a steady-state error remains due to the gravitational force acting on the robot.

With Gravity Compensation If a model of the gravitational force is available, it can be used to remove the steady-state error by adding the gravitational acceleration to the control low:

$$\boldsymbol{u}_t = \boldsymbol{K}_P(\boldsymbol{q}_d - \boldsymbol{q}_t) + \boldsymbol{K}_D(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}_t) + \boldsymbol{g}(\boldsymbol{q})$$

But of course this required a reasonable model which is not always available.

PID-Controller

Another commonly used control law is the *PID-controller* which, in addition to the proportional and differential terms, adds an integral part that keeps track of the error from the beginning of time:

$$\boldsymbol{u}_t = \boldsymbol{K}_P(\boldsymbol{q}_d - \boldsymbol{q}_t) + \boldsymbol{K}_D(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}_t) + \boldsymbol{K}_I \int_0^t (\boldsymbol{q}_d - \boldsymbol{q}_\tau)\,\mathrm{d}\tau$$

This approach also removes the steady-state error for steady-state control, i.e. when the robot should move to a single point. However, for tracking control, this approach is not suitable as there may always be a positioning error, letting the integral part become large very quick. This results in high motor commands may causing damage to the robot.

Also, there is the problem of wind-up where if the position is not reached, the controller might get saturated due to limitations of the motor. This causes the control law to be effectively useless as it cannot react to other errors anymore.

2.3.2. Model-Based Feedback Control

Some problems with PD-control with gravity compensation include that there always needs to be an error in either the position of the velocity to generate a control signal. This implies that high gains are needed in order to be accurate. But high gains cause the robot to be really stiff and dangerous as it moves with higher power!

If a model of the dynamics is present (e.g. using the recursive Newton-Euler algorithm), it can be used to compute the motor inputs for a desired acceleration. Being able to only set the acceleration is no limitation since dynamical systems are second-order systems anyway, so only the accelerations are directly controllable. This is called *model-based feedback control* and it is described by the following equations:

$$\begin{aligned}\ddot{\boldsymbol{q}}_t^{\text{ref}} &= \boldsymbol{K}_P(\boldsymbol{q}_d - \boldsymbol{q}_t) + \boldsymbol{K}_D(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}_t) + \ddot{\boldsymbol{q}}_d \\ \boldsymbol{u}_t &= \boldsymbol{M}(\boldsymbol{q}_t)\ddot{\boldsymbol{q}}_t^{\text{ref}} + \boldsymbol{c}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{g}(\boldsymbol{q})\end{aligned}$$

A major drawback is of course that this needs an accurate model of the dynamics!

2.3.3. Feedforward Control

In feedforward control, it is assumed that $\boldsymbol{q}_t \approx \boldsymbol{q}_d$ and $\dot{\boldsymbol{q}}_t \approx \dot{\boldsymbol{q}}_d$. This directly yields the following control law:

$$\begin{aligned}\boldsymbol{u}_t^{\text{FF}} &= \boldsymbol{M}(\boldsymbol{q}_d)\ddot{\boldsymbol{q}}_d + \boldsymbol{c}(\boldsymbol{q}_d, \dot{\boldsymbol{q}}_d) + \boldsymbol{g}(\boldsymbol{q}_d) \\ \boldsymbol{u}_t^{\text{FB}} &= \boldsymbol{K}_P(\boldsymbol{q}_d - \boldsymbol{q}) + \boldsymbol{K}_D(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}) \\ \boldsymbol{u}_t &= \boldsymbol{u}_t^{\text{FF}} + \boldsymbol{u}_t^{\text{FB}}\end{aligned}$$

As the inverse model does not play such a high role in feedforward control, it can also be used if the model is only approximate or even bad. Hence, feedforward control is highly relevant in practice as models are often not so good that model-based feedback control can be used. Additionally, it is often possible to pre-compute the feedforward terms $\boldsymbol{u}_t^{\text{FF}}$ such that less real-time computations are needed.

2.4. Control in Task-Space

So far it was assumed that the space to plan the movements in was the joint-space. However, sometimes it is more practical to plan in task-space, e.g. when trying to hit a ball with a table tennis racket. Hence, an inverse kinematics model $\boldsymbol{q} = \boldsymbol{f}^{-1}(\boldsymbol{x})$ as discussed in subsection 2.1.3 is needed. But is is often hard if not impossible to find such a model!

2.4.1. Differential Inverse Kinematics

As already seen, the inverse kinematics problem is hard to solve. But assuming the robot is non-redundant (i.e. $n = r$, where r is the dimensionality of the task-space), the Jacobian of the differential forward kinematics model in (2.1) becomes square. Assuming not to be in a singularity, the joint velocities can therefore be obtained by inverting $\boldsymbol{J}(\boldsymbol{q})$:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}^{-1}(\boldsymbol{q})\dot{\boldsymbol{x}}$$

By (numerically) integrating these velocities, the joint positions can be recovered (if the initial position is known).

Jacobian Transpose

If the Jacobian is not square, it is not possible to invert it directly. Hence, the task-space error

$$E = \frac{1}{2}\big(\boldsymbol{x}_d - \boldsymbol{f}(\boldsymbol{q})\big)^T\big(\boldsymbol{x}_d - \boldsymbol{f}(\boldsymbol{q})\big)$$

has to be minimized (where $\boldsymbol{f}(\boldsymbol{q})$ is the forward kinematics model). This can be done by gradient decent "in the system", i.e. the desired joint velocities are computed by minimizing the error. The desired joint positions can then again be recovered by numerical integration. Computing the gradient

$$\frac{\partial E}{\partial \boldsymbol{q}} = -\frac{\mathrm{d}\boldsymbol{f}(\boldsymbol{q})}{\mathrm{d}\boldsymbol{q}}\big(\boldsymbol{x} - \boldsymbol{f}(\boldsymbol{q})\big) \doteq -\boldsymbol{J}^T(\boldsymbol{q})\big(\boldsymbol{x}_d - \boldsymbol{f}(\boldsymbol{q})\big)$$

therefore yields the following desired velocity, where γ is the step size:

$$\dot{\boldsymbol{q}}_d = -\gamma\boldsymbol{J}^T(\boldsymbol{q})\big(\boldsymbol{x}_d - \boldsymbol{f}(\boldsymbol{q})\big)$$

This is called the *Jacobian transpose method*. As already said, the desired joint position $\boldsymbol{q}_d$ can then be recovered by numerically integrating $\dot{\boldsymbol{q}}_d$. All of this can then be fed into an joint-space controller, e.g. a PD- or model-based controller.

By adding numerical differentiation, it is also possible to use joint-space controllers that control the acceleration.

Jacobian Pseudo-Inverse

Under the assumption that the robot is not too far away from the solution, it is possible to also set desired task space velocities whilst finding the smallest $\dot{\boldsymbol{q}}_d$ that keeps that velocity. This can be formulated as the optimization problem

$$\begin{aligned} \dot{\boldsymbol{q}}_d = \arg\min_{\dot{\boldsymbol{q}}} \frac{1}{2}\dot{\boldsymbol{q}}^T\dot{\boldsymbol{q}} \\ \text{s.t.} \quad \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} = \dot{\boldsymbol{x}}_d \end{aligned} \tag{2.4}$$

This can be solved with Lagrangian optimization with the multipliers $\boldsymbol{\lambda}$. For brevity, let $\boldsymbol{J} \coloneqq \boldsymbol{J}(\boldsymbol{q})$. Given the Lagrangian

$$\mathcal{L} = \frac{1}{2}\dot{\boldsymbol{q}}^T\dot{\boldsymbol{q}} - \boldsymbol{\lambda}^T\big(\boldsymbol{J}\dot{\boldsymbol{q}} - \dot{\boldsymbol{x}}_d\big),$$

setting the derivative w.r.t. $\dot{\boldsymbol{q}}$ to zero yields

$$\frac{\partial \mathcal{L}}{\partial \dot{\boldsymbol{q}}} = \dot{\boldsymbol{q}} - \boldsymbol{J}^T\boldsymbol{\lambda} \stackrel{!}{=} \boldsymbol{0} \quad \Longrightarrow \quad \dot{\boldsymbol{q}} = \boldsymbol{J}^T\boldsymbol{\lambda}. \tag{2.5}$$

Inserting this into the Lagrangian, taking the derivative w.r.t. $\boldsymbol{\lambda}$ and setting it to zero then yields the solution of the optimization problem:

$$\begin{aligned} & & \mathcal{L} &= \frac{1}{2}\dot{\boldsymbol{q}}^T\dot{\boldsymbol{q}} - \boldsymbol{\lambda}^T\big(\boldsymbol{J}\dot{\boldsymbol{q}} - \dot{\boldsymbol{x}}_d\big) = \frac{1}{2}\boldsymbol{\lambda}^T\boldsymbol{J}\boldsymbol{J}^T\boldsymbol{\lambda} - \boldsymbol{\lambda}^T\big(\boldsymbol{J}\boldsymbol{J}^T\boldsymbol{\lambda} - \dot{\boldsymbol{x}}_d\big) \\ \Longrightarrow & & \frac{\partial \mathcal{L}}{\partial \boldsymbol{\lambda}} &= \boldsymbol{J}\boldsymbol{J}^T\boldsymbol{\lambda} - 2\boldsymbol{J}\boldsymbol{J}^T\boldsymbol{\lambda} + \dot{\boldsymbol{x}}_d = -\boldsymbol{J}\boldsymbol{J}^T\boldsymbol{\lambda} + \dot{\boldsymbol{x}}_d \stackrel{!}{=} 0 \\ \Longrightarrow & & \boldsymbol{\lambda} &= \big(\boldsymbol{J}\boldsymbol{J}^T\big)^{-1}\dot{\boldsymbol{x}}_d \end{aligned}$$

Plugging this result back into (2.5) yields the desired joint velocity and the solution of the optimization problem:

$$\dot{\boldsymbol{q}}_d = \underbrace{\boldsymbol{J}^T\big(\boldsymbol{J}\boldsymbol{J}^T\big)^{-1}}_{\boldsymbol{J}^\dagger}\dot{\boldsymbol{x}}_d \doteq \boldsymbol{J}^\dagger\dot{\boldsymbol{x}}_d$$

2.4.2. Task-Prioritization with Null-Space Movements

It is possible to modify the task-space control law to simultaneously execute another action in the *null-space*, a space that does not contradict the constraints of the optimization problem (2.4). This can be, for example, to push the robot into a rest position $\boldsymbol{q}_{\text{rest}}$ where it does not consume energy. This base task can be formulated with a P-controller

$$\dot{\boldsymbol{q}}_0 = \boldsymbol{K}_P(\boldsymbol{q}_{\text{rest}} - \boldsymbol{q}).$$

The optimization problem then is as following:

$$\begin{aligned} \dot{\boldsymbol{q}}_d = \arg\min_{\dot{\boldsymbol{q}}} \frac{1}{2}(\dot{\boldsymbol{q}} - \dot{\boldsymbol{q}}_0)^T(\dot{\boldsymbol{q}} - \dot{\boldsymbol{q}}_0) \\ \text{s.t.} \quad \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} = \dot{\boldsymbol{x}}_d \end{aligned}$$

The result of this optimization problem is

$$\dot{\boldsymbol{q}}_d = \boldsymbol{J}^\dagger \dot{\boldsymbol{x}}_d + \left(\boldsymbol{I} - \boldsymbol{J}^\dagger \boldsymbol{J}\right)\dot{\boldsymbol{q}}_0$$

where again $\boldsymbol{J} \coloneqq \boldsymbol{J}(\boldsymbol{q})$. The null-space is characterized by $\left(\boldsymbol{I} - \boldsymbol{J}^\dagger \boldsymbol{J}\right)$ which includes all movements $\dot{\boldsymbol{q}}_{\text{null}}$ that do not contradict the constraint $\boldsymbol{J}\dot{\boldsymbol{q}} = \dot{\boldsymbol{x}}_d$, i.e. $\dot{\boldsymbol{x}}_d = \boldsymbol{J}(\dot{\boldsymbol{q}} + \dot{\boldsymbol{q}}_{\text{null}})$ or equivalent $\boldsymbol{J}\dot{\boldsymbol{q}}_{\text{null}} = \boldsymbol{0}$ holds.

2.4.3. More Advanced Solutions

It is also possible to use an acceleration formulation which has the solution

$$\ddot{\boldsymbol{q}}_d = \boldsymbol{J}^\dagger(\ddot{\boldsymbol{x}}_d - \dot{\boldsymbol{J}}\dot{\boldsymbol{q}}) + \left(\boldsymbol{I} - \boldsymbol{J}^\dagger \boldsymbol{J}\right)\ddot{\boldsymbol{q}}_0.$$

There is a whole class of so-called *operational space control laws*, i.e. task-space control laws, that can all be derived from the following most general optimization problem:

$$\begin{aligned} \min_{\boldsymbol{u}} \frac{1}{2}(\boldsymbol{u} - \boldsymbol{u}_0)^T(\boldsymbol{u} - \boldsymbol{u}_0) \\ \text{s.t.} \quad \boldsymbol{A}(\boldsymbol{q}, \dot{\boldsymbol{q}}, t)\ddot{\boldsymbol{q}} = \dot{\boldsymbol{b}}(\boldsymbol{q}, \dot{\boldsymbol{q}}, t) \\ \boldsymbol{u}_0 = \boldsymbol{g}(\boldsymbol{q}, \dot{\boldsymbol{q}}, t) \\ \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} = \boldsymbol{u} + \boldsymbol{c}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{g}(\boldsymbol{q}) \end{aligned}$$

2.4.4. Singularities and Damped Pseudo-Inverse

If the Jacobian gets rank-deficient, i.e. singular, it is not longer possible to evaluate the pseudo-inverse $\boldsymbol{J}^\dagger = \boldsymbol{J}^T\left(\boldsymbol{J}\boldsymbol{J}^T\right)^{-1}$ due to the last inversion. It is therefore numerically more stable to use a damped pseudo-inverse $\boldsymbol{J}^{\dagger(\lambda)} = \boldsymbol{J}^T\left(\boldsymbol{J}\boldsymbol{J}^T + \lambda\boldsymbol{I}\right)^{-1}$ as a replacement of the regular pseudo-inverse. This avoids singularities in the inversion.

3. Machine Learning Foundations

This chapter is a basic introduction into the foundations of machine learning, focusing on the tools that are important for robot learning. Machine learning has become the best approach to various problems (e.g. in robotics and computer vision) in terms of speed, engineering time and robustness due to the shear amount of data that is generated on a daily basis. The goal of machine learning is to *describe* data make *predictions*, and make *decisions* based on data. To learn from previously seen data, several assumptions has to be made like all training data is i.i.d., queries are drawn from the same distribution as the training data or that the answer comes from a set of possible answers known in advance. Of course different machine learning models and algorithms impose different assumptions and some impose severe while others impose mild conditions. In general, there are two core problems when building a machine learning model:

Estimation Is it possible to obtain an uncertain quantity of interest?

Generalization Is it possible to predict results for previously unseen situations?

3.1. The Six Machine Learning Choices

When building a model, there are six choices to be made, the former three of them describing the problem and the latter three describing the solution:

1. **Problem Class**
   What is the nature of the training data and what kind of queries can be made?

2. **Problem Assumptions**
   What is known about the source of the data or the form of the solution?

3. **Evaluation Criteria**
   What is the goal? How to evaluate answers to specific queries?
   How to measure the overall performance?

4. **Model Type**
   Will there be an intermediate model?
   What aspects of the problem will be modeled?
   How will the model be used?

5. **Model Class**
   What (parametric) class of models will be used?
   What criteria are used to pick a particular model from the class?

6. **Algorithm**
   What process is used to fit the model to the data and to make predictions?

3.1.1. Problem Class

3.1.2. Problem Assumptions

Typical assumptions are for example:

- Learning is actually possible (e.g. there exists a causal structure).
- There is some smoothness in the solution, e.g. for regression.
- The process generating the data is stationary.
- Process assumptions, e.g. i.i.d., Markovian, …
- Observability.
- Frequentist assumption: there exists a true model.

Finite Differences

Automatic Differentiation

3.7.4. Efficient Gradient Descent

Stochastic Gradient Descent

Mini-Batch Gradient Descent

3.7.5. Choosing the Learning Rate

Plateaus and Valleys

Adaptive Learning Rates

Momentum

Adadelta

Adam

3.7.6. Choosing the Descent Direction

Hessian Approaches

Conjugate Gradient

Levenberg-Marquardt

3.7.7. Initialization of the Parameters

3.7.8. Overfitting

Weight Decay

Early Stopping

Input Noise Augmentation

Dropout

Batch Normalization

3.7.9. Theoretical Analysis

Universal Function Approximation Theorem

3.7.10. Network Architectures

Convolutional Neural Networks (CNNs)

Recurrent Neural Networks (RNNs)

3.7.11. Neural Networks in Robotics

Value Functions

Policies

3.8. Wrap-Up

4. Optimal Control

In lots of scenarios it is too costly to program a desired behavior. This phenomenon is none from e.g. image understanding where the state-of-the-art approach is to use machine learning. So supervised learning is not enough for acquiring certain behaviors, e.g. due to imperfect demonstrations, the correspondence problem and the simple fact that it is impossible to demonstrate everything. The correspondence problem describes the fact that the robot works differently than humans, i.e. the transfer from a human motion to joint movements or a robot is not straightforward.

Hence, self-improvement is needed! The robot explores the environment by trial-and-error and the environment (or the engineers of the environment) give feedback based on the current states and the executed actions. This is called the *reward*. In optimal control and reinforcement learning settings, it is common to talk about the *reward*, whereas in other literature it is more common to talk about the *cost*, which is just an inverted notion of the same concept, so $\textit{Reward} = -\textit{Cost}$ and maximizing the reward is equivalent to minimizing the cost.

This chapter is split into two sections: optimal control for discrete and continuous state-action spaces. In both settings the time is assumed to be discrete (controlling a robot is most often done with a certain sampling rate anyway).

The vast field of optimal control is based on the *principle of optimality* by Richard Bellman (1957): "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." This principle can be broken done in two main ideas:

1. Breaking down the optimal solutions in small parts, each part must be optimal.
2. The initial and previous states do not matter for the optimal policy, i.e. only the current state is relevant for determining the optimal action.

This principle leads to the principle of *dynamic programming*. It breaks down the problem into small subproblems which are then solved optimally. All sub-solutions together than constitute an optimal (global) policy. Famous examples for algorithms implementing the paradigm of dynamic programming are for example Dijkstra's algorithm for the shortest-path-problem.

4.1. Discrete State-Action Space: Dynamic Programming

Optimal control for discrete state-action spaces are based on *Markov decision processes* (MDPs) which are defined by

- state space $\boldsymbol{s} \in \mathcal{S}$,
- action space $\boldsymbol{a} \in \mathcal{A}$,
- transition dynamics $p_t(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t)$,

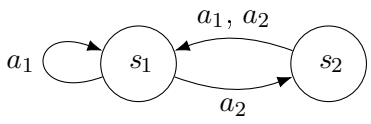


Figure 4.1.: Simple Markov decision process with two states and actions.

- reward function $r_t : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and
- initial state distribution $\mu(\boldsymbol{s})$.

The *Markov property* describes that the transition dynamics are only dependent on the previous state and the taken action, but not on any previous states or actions, i.e.

$$p_t(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1}, \cdots) = p_t(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t)$$

holds. MDPs can be visualized nicely as transition diagrams as illustrated in Figure 4.1.

A *policy* π in an MDPs describes the action to take given a state. It can either be *deterministic*, i.e. $\boldsymbol{a} = \pi(\boldsymbol{s})$, $\pi : \mathcal{S} \to \mathcal{A}$ or *stochastic*, i.e. $\boldsymbol{a} \sim \pi(\cdot \mid \boldsymbol{s})$, $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^+$. As deterministic policies can also be represented as stochastic policies with an action probability of 1, all of the following assumes a stochastic policy if not stated otherwise. But this stochastic policy might encode deterministic behavior.

4.1.1. Finite-Horizon Optimal Control

For optimal control with a finite horizon T, the cumulative reward over T time steps is maximized. That is, a policy $\pi(\boldsymbol{a} \mid \boldsymbol{s})$ is searched that maximizes the expected reward

$$J_\pi = \mathbb{E}_{\mu,p,\pi}\left[r_T(\boldsymbol{s}_T) + \sum_{t=1}^{T-1} r_t(\boldsymbol{s}_t, \boldsymbol{a}_t)\right], \tag{4.1}$$

where $\mathbb{E}_{\mu,p,\pi}[\cdot]$ is a short form for $\mathbb{E}_{\boldsymbol{s}_1 \sim \mu(\cdot),\, \boldsymbol{s}_{t+1} \sim p(\cdot \mid \boldsymbol{s}_t, \boldsymbol{a}_t),\, \boldsymbol{a}_t \sim \pi(\cdot \mid \boldsymbol{s}_t)}$. The final reward $r_T(\boldsymbol{s}_T)$ does not depend on the action as there is no action to take in the last state. In all of the following, the final reward might also be referred to as $r_T(\boldsymbol{s}_T, \boldsymbol{a}_T)$, but it does still only depend on $\boldsymbol{s}_T$ as there is no $\boldsymbol{a}_T$! It is convenient as it clears up some of the summations.

Value and State-Action Value Functions

To assess the "quality" of a state or state-action pair, two functions are defined: The value and the state-action value functions, where the latter is often called the Q-function. For some policy π, they are defined as follows:

$$V_t^\pi(\boldsymbol{s}) = \mathbb{E}_{p,\pi}\left[\sum_{\tau=t}^{T} r_\tau(\boldsymbol{s}_\tau, \boldsymbol{a}_\tau) \,\middle|\, \boldsymbol{s}_t = \boldsymbol{s}\right]$$

$$Q_t^\pi(\boldsymbol{s}, \boldsymbol{a}) = \mathbb{E}_{p,\pi}\left[\sum_{\tau=t}^{T} r_\tau(\boldsymbol{s}_\tau, \boldsymbol{a}_\tau) \,\middle|\, \boldsymbol{s}_t = \boldsymbol{s},\, \boldsymbol{a}_t = \boldsymbol{a}\right]$$

The intuition of these functions is as follows:

Value Function How "good" is it to be in state $\boldsymbol{s}$ under the policy π?

Q-Function How "good" is it to take action $\boldsymbol{a}$ in state $\boldsymbol{s}$ when subsequently following the policy π?

Given the optimal policy π^*, i.e. the policy that maximizes (4.1), the respective value and Q-functions are usually called V_t^* and Q_t^*. For the optimal policy, the value and Q-function can be calculated from each other straightforwardly:

$$V_t^*(\boldsymbol{s}) = \max_{\boldsymbol{a}} Q_t^*(\boldsymbol{s}, \boldsymbol{a})$$
$$Q_t^*(\boldsymbol{s}, \boldsymbol{a}) = r_t(\boldsymbol{s}, \boldsymbol{a}) + \mathbb{E}_{\boldsymbol{s}' \sim p(\cdot \mid \boldsymbol{s}, \boldsymbol{a})}\left[V_{t+1}^*(\boldsymbol{s}')\right]$$

Value Iteration

Applying the principle of dynamic programming to an MDP is straightforward. As in the last time step there is no action, the value function for the last time step is simply

$$V_T^*(\boldsymbol{s}) = r_T(\boldsymbol{s})$$

for every state $\boldsymbol{s}$ (this has to be evaluated). Then *value iteration* iterates backwards in time, by maximizing the Q-function that is computed from the value function for $t = T-1, \cdots, 1$:

$$V_t^*(\boldsymbol{s}) = \max_{\boldsymbol{a}} Q_t(\boldsymbol{s}, \boldsymbol{a}) = \max_{\boldsymbol{a}} \left(r_t(\boldsymbol{s}, \boldsymbol{a}) + \mathbb{E}_{\boldsymbol{s}' \sim p(\cdot \mid \boldsymbol{s}, \boldsymbol{a})}\left[V_{t+1}^*(\boldsymbol{s}') \mid \boldsymbol{s}, \boldsymbol{a}\right]\right)$$

Following this pattern, the optimal value function for time step t is obtained after $T - t + 1$ iterations. The (deterministic) optimal policy $\pi_t^* : \mathcal{S} \to \mathcal{A}$ is then obtained by maximizing the Q-function

$$\pi_t^*(\boldsymbol{s}) = \arg\max_{\boldsymbol{a}} Q_t^*(\boldsymbol{s}, \boldsymbol{a})$$

which can either be obtained from the value function or, ideally, has been stored whilst executing the value iteration.

A pseudo-code version of value iteration is shown in algorithm 1. Note that the computation of the value function was split into computing the Q-function on line 3 and then computing the value function on line 4 for to simplify the final computation of the policy on line 5.

Algorithm 1: Value Iteration for Finite-Horizon Problems

1 $V_T^*(\boldsymbol{s}) \leftarrow r_T(\boldsymbol{s})$ for all $\boldsymbol{s} \in \mathcal{S}$
2 for $t = T-1, \cdots, 1$ do
  // Compute Q-Function for time step t for all $\boldsymbol{s} \in \mathcal{S}$ and $\boldsymbol{a} \in \mathcal{A}$:
3 $Q_t^*(\boldsymbol{s}, \boldsymbol{a}) \leftarrow r_t(\boldsymbol{s}, \boldsymbol{a}) + \sum_{\boldsymbol{s}'} p_t(\boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a}) V_{t+1}^*(\boldsymbol{s}')$
  // Compute V-Function for time step t for all $\boldsymbol{s} \in \mathcal{S}$:
4 $V_t^*(\boldsymbol{s}) \leftarrow \max_{\boldsymbol{a}} Q_t^*(\boldsymbol{s}, \boldsymbol{a})$
  // Return optimal policy for each time step t for all $\boldsymbol{s} \in \mathcal{S}$:
5 return $\pi_t^*(\boldsymbol{s}) \leftarrow \arg\max_{\boldsymbol{a}} Q_t^*(\boldsymbol{s}, \boldsymbol{a})$

Consequences of a Finite Time Horizon

Choosing a finite time horizon over an infinite one has some major consequences on the environment and the e.g. the value function. First of all, it matters how many time steps are left. This implies that not only the state transition model, but also the policy and the reward function might be time-dependent as denoted by the index t. This leads to also the value and Q-function being time-dependent! However, having a finite amount of steps makes it possible to find the optimal value function in a finite amount of steps.

4.1.2. Infinite-Horizon Optimal Control

Using an infinite time horizon, i.e. $T = \infty$, the time index is not part of the state! Hence, also the optimal policy as well as the value and Q-function are time-independent and also the reward function and the state transition model are time-independent. But this comes at the cost of the sum of rewards, i.e. the optimization objective, being infinite and hence divergent.

The simplest and most straightforward approach is to introduce a *discount factor* $\gamma \in [0, 1)$ that trades of the long term vs. the immediate reward. The optimization objective, the discounted sum of rewards, now is

$$J_\pi = \mathbb{E}_{\mu,p,\pi}\left[\sum_{t=1}^{\infty} \gamma^t r(\boldsymbol{s}_t, \boldsymbol{a}_t)\right]$$

analogous to the objective for the finite horizon (4.1). The value and state-action value functions are defined accordingly as

$$V^\pi(\boldsymbol{s}) = \mathbb{E}_{p,\pi}\left[\sum_{t=1}^{\infty} \gamma^t r(\boldsymbol{s}_t, \boldsymbol{a}_t) \,\middle|\, \boldsymbol{s}_1 = \boldsymbol{s}\right]$$
$$Q^\pi(\boldsymbol{s}, \boldsymbol{a}) = \mathbb{E}_{p,\pi}\left[\sum_{t=1}^{\infty} \gamma^t r(\boldsymbol{s}_t, \boldsymbol{a}_t) \,\middle|\, \boldsymbol{s}_1 = \boldsymbol{s},\, \boldsymbol{a}_1 = \boldsymbol{a}\right],$$

with the relations

$$V^\pi(\boldsymbol{s}) = \mathbb{E}_\pi\big[Q^\pi(\boldsymbol{s}, \boldsymbol{a}) \,\big|\, \boldsymbol{a}\big]$$
$$Q^\pi(\boldsymbol{s}, \boldsymbol{a}) = r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \mathbb{E}_{\boldsymbol{s}' \sim p(\cdot \,|\, \boldsymbol{s}, \boldsymbol{a})}\big[V^\pi(\boldsymbol{s}')\big]$$

between them. These are analogous to the finite-horizon case relations in $(4.1.1)$, however in the last formula the expectation is multiplied with the discount factor. Also the maximization in the first equation was replaced with an expectation as now the policy might be suboptimal. In the finite case, only the optimal policy was used which equals using the maximum operator.

Value Iteration

Finding the optimal Q-function is again possible using value iteration with $T \to \infty$ as shown in algorithm 2. The only slight change (except for making the value function time-independent) is again that the expectation for computing the Q-function has to be multiplied with the discount factor in line 3.

Algorithm 2: Value Iteration for Infinite-Horizon Problems

1 $V^*(\boldsymbol{s}) \leftarrow 0$ for all $\boldsymbol{s} \in \mathcal{S}$
2 repeat
  // Compute Q-Function for all $\boldsymbol{s} \in \mathcal{S}$ and $\boldsymbol{a} \in \mathcal{A}$:
3 $Q^*(\boldsymbol{s}, \boldsymbol{a}) \leftarrow r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \sum_{\boldsymbol{s}'} p(\boldsymbol{s}' \,|\, \boldsymbol{s}, \boldsymbol{a}) V^*(\boldsymbol{s}')$
  // Compute V-Function for all $\boldsymbol{s} \in \mathcal{S}$:
4 $V^*(\boldsymbol{s}) \leftarrow \max_{\boldsymbol{a}} Q^*(\boldsymbol{s}, \boldsymbol{a})$
5 until convergence of V^*
  // Return optimal policy for all $\boldsymbol{s} \in \mathcal{S}$:
6 return $\pi^*(\boldsymbol{s}) \leftarrow \arg\max_{\boldsymbol{a}} Q^*(\boldsymbol{s}, \boldsymbol{a})$

Policy Iteration

One major drawback of value iteration as introduced before is that a lot of redundant maximization operations have to be performed when calculating the Q-function values. This drawback can be overcome by *policy iteration*, a similar approach for computing the optimal value and Q-function. It is composed of two steps that are after each other until convergence:

1. *Policy Evaluation:* Estimation of the quality of the states and actions for the current policy.
2. *Policy Improvement:* Improve the policy by taking the actions with the highest quality.

The process of policy evaluation is shown in algorithm 3. The main difference is that the value and Q-functions are not the optimal ones any more but the ones for the given policy π. This results in the computation of the value function on line 4 to become an expectation (the policy is not optimal anymore and so is the Q-function, so using the maximum action would be wrong). The next step for policy iteration is to improve the policy by taking the actions with the highest quality. This corresponds to choosing the policy as

$$\pi(\boldsymbol{a} \mid \boldsymbol{s}) = \begin{cases} 1 & \text{if } \boldsymbol{a} = \arg\max_{\boldsymbol{a}'} Q^\pi(\boldsymbol{s}, \boldsymbol{a}') \\ 0 & \text{otherwise} \end{cases}$$

where Q^π is the Q-function found via policy evaluation. Putting it all together, policy iteration is shown in algorithm 4.

Algorithm 3: Policy Evaluation for Infinite-Horizon Problems

Input: Policy π

1 $V^\pi(\boldsymbol{s}) \leftarrow 0$ for all $\boldsymbol{s} \in \mathcal{S}$

2 repeat

  // Compute Q-Function for all $\boldsymbol{s} \in \mathcal{S}$ and $\boldsymbol{a} \in \mathcal{A}$:

3 $Q^\pi(\boldsymbol{s}, \boldsymbol{a}) \leftarrow r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \sum_{\boldsymbol{s}'} p(\boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a}) V^\pi(\boldsymbol{s}')$

  // Compute V-Function for all $\boldsymbol{s} \in \mathcal{S}$:

4 $V^\pi(\boldsymbol{s}) \leftarrow \sum_{\boldsymbol{a}} \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})$

5 until convergence of V^π

6 return V^π and/or Q^π as needed

4.2. Continuous State-Action Space: Linear Quadratic Regulator

In this section the application of the principle of dynamic programming will be applied to continuous state-action spaces. For continuous systems, this is often just called *optimal control* and a slightly different notation is used, namely $\boldsymbol{x}$ for the state and $\boldsymbol{u}$ for the action. This section will stick to this convention.

Sadly, the optimal control problem of maximizing the expected long-term reward is not tractable for almost all systems. In fact, the only system where it is is for linear transition dynamics with additive Gaussian noise and a quadratic reward. The resulting optimal policy is called the *Linear Quadratic Regulator* (LQR). The first part of this section will focus on these tractable systems, following by an approach for approximating nonlinear systems and optimal control with learned models.

Algorithm 4: Policy Iteration for Infinite-Horizon Problems

Input: Policy π

$V^\pi(\boldsymbol{s}) \leftarrow 0$ for all $\boldsymbol{s} \in \mathcal{S}$

repeat

  repeat

    // Compute Q-Function for all $\boldsymbol{s} \in \mathcal{S}$ and $\boldsymbol{a} \in \mathcal{A}$:

    $Q^\pi(\boldsymbol{s}, \boldsymbol{a}) \leftarrow r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \sum_{\boldsymbol{s}'} p(\boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a}) V^\pi(\boldsymbol{s}')$

    // Compute V-Function for all $\boldsymbol{s} \in \mathcal{S}$:

    $V^\pi(\boldsymbol{s}) \leftarrow \sum_{\boldsymbol{a}} \pi(\boldsymbol{a} \mid \boldsymbol{s}) Q^\pi(\boldsymbol{s}, \boldsymbol{a})$

  until convergence of V^π

  // Improve policy for all $\boldsymbol{s} \in \mathcal{S}$ and $\boldsymbol{a} \in \mathcal{A}$:

  $\pi(\boldsymbol{a} \mid \boldsymbol{s}) = \begin{cases} 1 & \text{if } \boldsymbol{a} = \arg\max_{\boldsymbol{a}'} Q^\pi(\boldsymbol{s}, \boldsymbol{a}') \\ 0 & \text{otherwise} \end{cases}$

until convergence of π

return π

4.2.1. Linear Quadratic Regulator (LQR)

LQR systems is defined by the state and action spaces, $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{u} \in \mathbb{R}^m$, the linear (possible time-dependent) transition dynamics

$$p_t(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}) = \mathcal{N}\big(\boldsymbol{x}_{t+1} \,\big|\, \boldsymbol{A}_t \boldsymbol{x}_t + \boldsymbol{B}_t \boldsymbol{u}_t + \boldsymbol{b}_t,\, \boldsymbol{\Sigma}_t\big)$$

with (additive) Gaussian noise, the quadratic reward function

$$r_t(\boldsymbol{x}, \boldsymbol{u}) = -(\boldsymbol{x} - \boldsymbol{x}_d)^T \boldsymbol{R}_t (\boldsymbol{x} - \boldsymbol{x}_d) - \boldsymbol{u}_t^T \boldsymbol{H}_t \boldsymbol{u}_t,$$

with symmetric and positive-definite matrices $\boldsymbol{R}_t$ and $\boldsymbol{H}_t$, and the initial state distribution $\mu(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$. The final reward is given as $r_T(\boldsymbol{x}) = r_T(\boldsymbol{x}, \boldsymbol{0})$. The objective is to maximize the expected long-term reward with a finite time horizon T:

$$J_\pi = \mathbb{E}_{\mu,p,\pi}\left[r_T(\boldsymbol{x}) + \sum_{t=1}^{T-1} r_t(\boldsymbol{x}_t, \boldsymbol{u}_t) \right]$$

The value $\boldsymbol{x}_d$ can be seen as the desired state. Deviations from $\boldsymbol{x}_d$ punish the system with a highly negative reward.

4.2.2. Solving the Optimal Control Problem

Applying the principle of optimal control now says to start with the last time step T by setting $V_T^*(\boldsymbol{x}) = r_T(\boldsymbol{x})$ for all states. Subsequently, iterate over $t = T-1, \cdots, 1$ and compute

$$V_t^*(\boldsymbol{x}) = \max_{\boldsymbol{u}} \Big(r_t(\boldsymbol{x}_t, \boldsymbol{u}_t) + \mathbb{E}_{\boldsymbol{x}' \sim p(\cdot \mid \boldsymbol{x}, \boldsymbol{u})} \big[V_{t+1}^*(\boldsymbol{x}') \,\big|\, \boldsymbol{x}, \boldsymbol{u} \big] \Big).$$

For applying this to continuous state-action spaces, the expectation and the maximization have to be solved. And this step is only possible for LQR problems! Applying dynamic programming can be split further into the following steps:

1. Compute the value function for the last time step:

$$V_T^*(\boldsymbol{x}) = r_T(\boldsymbol{x})$$

2. To get from $t+1$ to t, compute the Q-function for all states and actions:

$$Q_t^*(\boldsymbol{x}, \boldsymbol{u}) = r_t(\boldsymbol{x}_t, \boldsymbol{u}_t) + \mathbb{E}_{\boldsymbol{x}' \sim p(\cdot \mid \boldsymbol{x}, \boldsymbol{u})}\left[V_{t+1}^*(\boldsymbol{x}') \,\middle|\, \boldsymbol{x}, \boldsymbol{u}\right]$$

3. Compute the optimal policy for time step t for all states:

$$\pi_t^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}} Q_t^*(\boldsymbol{x}, \boldsymbol{u})$$

4. Compute the optimal value function for time step t for all states:

$$V_t^*(\boldsymbol{x}) = Q_t^*\big(\boldsymbol{x}, \pi_t^*(\boldsymbol{x})\big)$$

5. Repeat from 2 until $t = 1$ is reached.

The next paragraphs will focus on these steps separately.

1. Compute the Value Function for the Last Time Step This can be computed straightforwardly as

$$V_T^*(\boldsymbol{x}) = r_T(\boldsymbol{x}) = -(\boldsymbol{x} - \boldsymbol{x}_d)^T \boldsymbol{R}_T (\boldsymbol{x} - \boldsymbol{x}_d) \doteq -(\boldsymbol{x} - \boldsymbol{x}_d)^T \boldsymbol{V}_T (\boldsymbol{x} - \boldsymbol{x}_d)$$

with $\boldsymbol{V}_t \coloneqq \boldsymbol{R}_T$.

2. Compute the Q-Function To compute the Q-function

$$Q_t^*(\boldsymbol{x}, \boldsymbol{u}) = r_t(\boldsymbol{x}_t, \boldsymbol{u}_t) + \mathbb{E}_{\boldsymbol{x}' \sim p(\cdot \mid \boldsymbol{x}, \boldsymbol{u})}\left[V_{t+1}^*(\boldsymbol{x}') \,\middle|\, \boldsymbol{x}, \boldsymbol{u}\right],$$

firstly the expectation has to be computed. By assuming a quadratic structure for the value function of time step $t+1$, i.e. $V_{t+1}^*(\boldsymbol{x}) = -(\boldsymbol{x}-\boldsymbol{x}_d)^T \boldsymbol{V}_{t+1} (\boldsymbol{x}-\boldsymbol{x}_d)$ with a symmetric and positive-definite $\boldsymbol{V}_{t+1}$, the expectation becomes

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{x}' \sim p(\cdot \mid \boldsymbol{x}, \boldsymbol{u})}\left[V_{t+1}^*(\boldsymbol{x}') \,\middle|\, \boldsymbol{x}, \boldsymbol{u}\right] &= \int V_{t+1}^*(\boldsymbol{x}')\, p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u})\, \mathrm{d}\boldsymbol{x}' \\
&= -\int (\boldsymbol{x} - \boldsymbol{x}_d)^T \boldsymbol{V}_{t+1} (\boldsymbol{x} - \boldsymbol{x}_d)\, \mathcal{N}\big(\boldsymbol{x}' \,\big|\, \boldsymbol{A}_t \boldsymbol{x} + \boldsymbol{B}_t \boldsymbol{u}_t + \boldsymbol{b}_t,\, \boldsymbol{\Sigma}_t\big)\, \mathrm{d}\boldsymbol{x}' \\
&= -\big(\boldsymbol{A}_t \boldsymbol{x} + \boldsymbol{B}_t \boldsymbol{u}_t + \boldsymbol{b}_t - \boldsymbol{x}_d\big)^T \boldsymbol{V}_{t+1} \big(\boldsymbol{A}_t \boldsymbol{x} + \boldsymbol{B}_t \boldsymbol{u}_t + \boldsymbol{b}_t - \boldsymbol{x}_d\big) - \operatorname{tr}(\boldsymbol{V}_{t+1} \boldsymbol{\Sigma}_t)
\end{aligned}$$

where in the last step equation (380) from the matrix cookbook[1] was used. Plugging in the reward function yields the Q-function:

$$\begin{aligned}
Q_t^*(\boldsymbol{x}, \boldsymbol{u}) = &- (\boldsymbol{x} - \boldsymbol{x}_d)^T \boldsymbol{R}_t (\boldsymbol{x} - \boldsymbol{x}_d) - \boldsymbol{u}^T \boldsymbol{H}_t \boldsymbol{u} \\
&- \big(\boldsymbol{A}_t \boldsymbol{x} + \boldsymbol{B}_t \boldsymbol{u}_t + \boldsymbol{b}_t - \boldsymbol{x}_d\big)^T \boldsymbol{V}_{t+1} \big(\boldsymbol{A}_t \boldsymbol{x} + \boldsymbol{B}_t \boldsymbol{u}_t + \boldsymbol{b}_t - \boldsymbol{x}_d\big) \\
&- \operatorname{tr}(\boldsymbol{V}_{t+1} \boldsymbol{\Sigma}_t)
\end{aligned}$$

[1] `https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf`

3. Compute the Optimal Policy Computing the optimal policy is done by maximizing the Q-function w.r.t. the action $\boldsymbol{u}$. This means taking the derivative of $(4.2.2)$ and setting it to zero:

$$\begin{aligned}
\frac{\partial}{\partial \boldsymbol{u}} Q_t^*(\boldsymbol{x}, \boldsymbol{u}) &= -2\boldsymbol{H}_t\boldsymbol{u} - 2\boldsymbol{B}_t^T\boldsymbol{V}_{t+1}\big(\boldsymbol{A}_t\boldsymbol{x} + \boldsymbol{B}_t\boldsymbol{u}_t + \boldsymbol{b}_t - \boldsymbol{x}_d\big) \\
&= -2\boldsymbol{H}_t\boldsymbol{u} - 2\boldsymbol{B}_t^T\boldsymbol{V}_{t+1}\boldsymbol{A}_t\boldsymbol{x} - 2\boldsymbol{B}_t^T\boldsymbol{V}_{t+1}\boldsymbol{B}_t\boldsymbol{u}_t - 2\boldsymbol{B}_t^T\boldsymbol{V}_{t+1}\boldsymbol{b}_t + 2\boldsymbol{B}_t^T\boldsymbol{V}_{t+1}\boldsymbol{x}_d \\
&= -2\big(\boldsymbol{H}_t + \boldsymbol{B}_t^T\boldsymbol{V}_{t+1}\boldsymbol{B}_t\big)\boldsymbol{u} - 2\boldsymbol{B}_t^T\boldsymbol{V}_{t+1}\big(\boldsymbol{A}_t\boldsymbol{x} + \boldsymbol{b}_t - \boldsymbol{x}_d\big) \stackrel{!}{=} \boldsymbol{0} \\
\implies \qquad \pi_t^*(\boldsymbol{x}) = \boldsymbol{u}^* &= -\big(\boldsymbol{H}_t + \boldsymbol{B}_t^T\boldsymbol{V}_{t+1}\boldsymbol{B}_t\big)^{-1}\boldsymbol{B}_t^T\boldsymbol{V}_{t+1}\big(\boldsymbol{A}_t\boldsymbol{x} + \boldsymbol{b}_t - \boldsymbol{x}_d\big)
\end{aligned}$$

4. Compute the Value Function To compute the value function, plug the obtained optimal policy into the Q-function:

4.2.3. Optimal Control with Learned Models

The problem with local linearization is that it only really works in simulation and not on the real robot. This is caused by the fact that the models for robots are often really bad. So learning models instead of hand-crafting them would be a good alternative[2]!

Major challenges in model-based policy learning are that the methods mainly work for balancing tasks where the state space is highly restricted. For more complex problems, using learned models becomes a lot harder. The main cause for this is that the model is likely to be inaccurate the further the robot moves away from known locations. Such inaccuracies can be exploited by the optimizer, causing the policy to not work on the real system. Jumping to an area off the state space by exploiting the model can again make the policy inherently unstable.

Questions in Model Learning

When learning or designing models, a lot of questions have to be answered, for example what the actions $\boldsymbol{u}$ are. They might be motor torques, but this requires a good forward dynamics model for control. Other options are e.g. the joint accelerations, which requires a good inverse dynamics model, or the accelerations in task space. The latter is ideal for control, but requires a good task-space control law.

Another big question is how to design the reward. It can encode basic information like whether the task was successful or not (using a binary reward) or more task knowledge like how far the end-effector is away from the goal. Typically a mixture is needed with usually hand-crafted weighting between them.

Human Motor Cost Functions

Research in cognitive science shows that also human movements can be explained very well using cost (or reward) functions! Reaching movements for example include minimum jerk, minimum torque change, minimum end-point variance. Even complicated things like locomotion can be explained with minimum metabolic energy consumption.

State-Of-The-Art Approaches

Probabilistic Inference for Learning Control (PILCO):

[2] Model learning will be covered in more detail in chapter 7.

- Learn Gaussian process forward models.
- Use uncertainty predicted by the GP model to predict the long-term reward.
- Policy optimization with analytical gradient of the expected reward.

Guided Policy Search by Trajectory Optimization:

- Learn time-dependent linear forward models.
- LQR-like algorithm for trajectory optimization with the additional constraint that the new trajectory should stay close to the data. This increases the stability!
- Use the optimized trajectories to train a generalizing neural network policy.

5. Approximate Optimal Control

In contrast to the discrete and linear systems covered in chapter 4, optimal control problems are usually not solvable in closed form for nonlinear continuous systems. Hence, approximations are needed to control these systems.

5.1. Discrete State-Action Space: Approximate Dynamic Programming

This section covers *approximate dynamic programming*, so an extension of section 4.1 using function approximators for the V- or Q-function. These are needed as dynamic programming suffers from both the curse of dimensionality in the number of states and dimensionality of the state representation and in the "curse of modeling" as there often is no model. The idea is to use an approximation $\hat{V}_{\boldsymbol{\theta}}(\boldsymbol{s})$ to approximate the true $V(\boldsymbol{s})$ where $\boldsymbol{\theta} \in \Theta$ are the parameters to be optimized. This way, the size of the state space $|\mathcal{S}|$ is reduced to the size of the parameter space, $|\Theta|$, so less parameters have to be estimated. But this advantage comes at the drawback of questionable expressiveness of $\hat{V}_{\boldsymbol{\theta}}(\boldsymbol{s})$, i.e. there might not be a $\boldsymbol{\theta} \in \Theta$ to approximate an arbitrary $V(\boldsymbol{s})$. As this is essentially a regression problem, lots of models from machine learning can be used with the usual drawbacks like linear models being easier to learn, but neural networks are more expressive but prone to overfit.

5.1.1. Approximate Value Iteration

In order to get the target values for fitting the regression model (by minimizing the MSE), value iteration with value function approximation invokes classical value iteration for a subset of the states. Subsequently the value function is fitted and the procedure starts over. A procedural description is shown on algorithm 5.

Algorithm 5: Approximate Value Iteration

1 Initialize $\boldsymbol{\theta}$ somehow.
2 repeat
3 | Get a subset $\mathcal{S}' \subseteq \mathcal{S}$ with $|\mathcal{S}'| \ll |\mathcal{S}|$.
4 | Calculate $V(\boldsymbol{s}')$ for all $\boldsymbol{s}' \in \mathcal{S}$ using value/policy iteration.
5 | Minimize MSE: $\boldsymbol{\theta}^{\text{new}} \leftarrow \arg\min_{\boldsymbol{\theta}'} \frac{1}{|\mathcal{S}'|} \sum_{\boldsymbol{s}' \in \mathcal{S}'} \left(\hat{V}_{\boldsymbol{\theta}'}(\boldsymbol{s}') - V(\boldsymbol{s}')\right)^2$
6 until convergence of $\boldsymbol{\theta}$

Convergence Analysis

While the algorithm seems simple, a big question is whether it actually converges. To analyze convergence, the *Bellman operator* T has to be defined. Applied to a value function V, the Bellman operator has the form

$$TV(\boldsymbol{s}) = \max_{\boldsymbol{a} \in \mathcal{A}} \Big(r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \sum_{\boldsymbol{s}' \in \mathcal{S}} p(\boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a}) V\big(\boldsymbol{s}'\big) \Big),$$

i.e. an application of the Bellman equation. If the Bellman operator is a contraction w.r.t. the infinite norm

$$\|V\|_\infty \coloneqq \max_{\boldsymbol{s} \in \mathcal{S}} \big|V(\boldsymbol{s})\big|$$

with contraction number γ^k,

$$\big\|TV_k - TV^*\big\|_\infty \leq \gamma^k \big\|V_k - V^*\big\|_\infty,$$

it ensures convergence to a fixed point:

$$\lim_{k \to \infty} V_k = V^*$$

So convergence is guaranteed if the function approximator ensures that the Bellman operator is a contraction! Extensive research focused on the fixed-point view of certain algorithms. e.g. TD, Q-Learning, SARSA, …Recent work from Dai et al. (2018) proposed the SBEED algorithm that uses nonlinear function approximations with convergence guarantees.

Approximation Error and Performance Loss

Another big question when dealing with function approximations is how the approximation error

$$\big\|\hat{V} - V^*\big\|_\infty$$

relates to the performance loss

$$\big\|V^\pi - V^*\big\|_\infty$$

with the greedy policy

$$\pi(\boldsymbol{s}) = \arg\max_{\boldsymbol{a} \in \mathcal{A}} \Big(r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \sum_{\boldsymbol{s}' \in \mathcal{S}}^{p} (\boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a}) \hat{V}\big(\boldsymbol{s}'\big) \Big).$$

For approximate value iteration, an upper bound on the performance loss is

$$\big\|V^\pi - V^*\big\|_\infty \leq \frac{2\gamma}{1-\gamma} \big\|\hat{V} - V^*\big\|_\infty,$$

so if for any ε, $\big\|V^\pi - V^*\big\|_\infty \leq \varepsilon$ can be achieved, the policy is optimal.

5.1.2. Approximate Policy Iteration

For policy iteration, only policy evaluation has to be altered from the vanilla policy iteration algorithm. But given a policy π, it is only possible to estimate V^π from samples or estimate the Bellman operator $T^\pi \hat{V}_{\boldsymbol{\theta}}$, which are both not in the function approximation space Θ, so the fixed point theorem is not applicable! Hence, projection back to the space Θ is needed. Two methods are possible for this:

- *Direct Policy Evaluation:* Directly minimize the projected cost function (e.g. MSE).

- *Indirect Policy Evaluation:* Solve the projected form of the value function $\Pi V = \Pi T_\Pi V$, where Π is the projection.

The approximation error again draws an upper bound on the performance loss:

$$\limsup_{k\to\infty} \left\|V^{\pi_k} - V^*\right\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k\to\infty} \left\|V^{\pi_k} - V_k\right\|_\infty$$

5.2. Continuous State-Action Space: Differential Dynamic Programming

For continuous optimal control problems, the LQR case (i.e. linear dynamics and quadratic reward functions) is the only solvable system. However, in robotics the dynamics as well as the cost functions are often nonlinear (e.g. by defining the reward function in task-space). So a big question is whether it is possible to perform LQR-like control for nonlinear systems with approximations?

The simplest idea for approximating nonlinear systems is to locally linearize them. This means Taylor-expanding the dynamics $\boldsymbol{x}_{t+1} = \boldsymbol{f}_t(\boldsymbol{x}_t, \boldsymbol{a}_t)$ around a point $(\tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{u}}_t)$ as

$$\boldsymbol{f}_t(\boldsymbol{x}, \boldsymbol{u}) \approx \underbrace{\boldsymbol{f}_t(\tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{u}}_t)}_{\tilde{\boldsymbol{f}}_t \coloneqq} + \underbrace{\frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{x}}\bigg|_{\boldsymbol{x}=\tilde{\boldsymbol{x}}_t,\, \boldsymbol{u}=\tilde{\boldsymbol{u}}_t}}_{\boldsymbol{B}_t \coloneqq} \boldsymbol{\Delta x}_t + \underbrace{\frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{u}}\bigg|_{\boldsymbol{x}=\tilde{\boldsymbol{x}}_t,\, \boldsymbol{u}=\tilde{\boldsymbol{u}}_t}}_{\boldsymbol{B}_t \coloneqq} \boldsymbol{\Delta u}_t \doteq \tilde{\boldsymbol{f}}_t + \boldsymbol{A}_t \boldsymbol{\Delta x}_t + \boldsymbol{B}_t \boldsymbol{\Delta u}_t$$

with $\boldsymbol{\Delta x}_t \coloneqq \boldsymbol{x}_t - \tilde{\boldsymbol{x}}_t$ and $\boldsymbol{\Delta u}_t \coloneqq \boldsymbol{u}_t - \tilde{\boldsymbol{u}}_t$. Similarly the reward can be linearized using a second-order Taylor expansion

$$\begin{aligned}
r_t(\boldsymbol{x}_t, \boldsymbol{u}_t) \approx & \underbrace{r_t(\tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{u}}_t)}_{r_t \coloneqq} + \frac{\partial r_t}{\partial \boldsymbol{x}}\bigg|_{\boldsymbol{x}=\tilde{\boldsymbol{x}}_t,\, \boldsymbol{u}=\tilde{\boldsymbol{u}}_t} \boldsymbol{\Delta x}_t + \boldsymbol{\Delta x}_t^T \frac{\partial^2 r_t}{\partial \boldsymbol{x}^2}\bigg|_{\boldsymbol{x}=\tilde{\boldsymbol{x}}_t,\, \boldsymbol{u}=\tilde{\boldsymbol{u}}_t} \boldsymbol{\Delta x}_t \\
& + \frac{\partial r_t}{\partial \boldsymbol{u}}\bigg|_{\boldsymbol{x}=\tilde{\boldsymbol{x}}_t,\, \boldsymbol{u}=\tilde{\boldsymbol{u}}_t} \boldsymbol{\Delta u}_t + \boldsymbol{\Delta u}_t^T \frac{\partial^2 r_t}{\partial \boldsymbol{u}^2}\bigg|_{\boldsymbol{x}=\tilde{\boldsymbol{x}}_t,\, \boldsymbol{u}=\tilde{\boldsymbol{u}}_t} \boldsymbol{\Delta u}_t \\
& + \boldsymbol{\Delta x}_t^T \frac{\partial^2 r_t}{\partial \boldsymbol{x} \partial \boldsymbol{u}}\bigg|_{\boldsymbol{x}=\tilde{\boldsymbol{x}}_t,\, \boldsymbol{u}=\tilde{\boldsymbol{u}}_t} \boldsymbol{\Delta u}_t + \boldsymbol{\Delta u}_t^T \frac{\partial^2 r_t}{\partial \boldsymbol{u} \partial \boldsymbol{x}}\bigg|_{\boldsymbol{x}=\tilde{\boldsymbol{x}}_t,\, \boldsymbol{u}=\tilde{\boldsymbol{u}}_t} \boldsymbol{\Delta x}_t \\
\doteq & r_t + \begin{bmatrix} \boldsymbol{\Delta x}_t^T & \boldsymbol{\Delta u}_t^T \end{bmatrix} \begin{bmatrix} r_t^{\boldsymbol{x}} \\ r_t^{\boldsymbol{u}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{\Delta x}_t^T & \boldsymbol{\Delta u}_t^T \end{bmatrix} \underbrace{\begin{bmatrix} r_t^{\boldsymbol{xx}} & r_t^{\boldsymbol{xu}} \\ r_t^{\boldsymbol{xu},T} & r_t^{\boldsymbol{uu}} \end{bmatrix}}_{\boldsymbol{R} \coloneqq} \begin{bmatrix} \boldsymbol{\Delta x}_t \\ \boldsymbol{\Delta u}_t \end{bmatrix}
\end{aligned}$$

with the gradients $r_t^{\boldsymbol{x}}$ and $r_t^{\boldsymbol{u}}$ and Hessians $r_t^{\boldsymbol{xx}}$, $r_t^{\boldsymbol{uu}}$, and $r_t^{\boldsymbol{xu}}$ w.r.t. the respective parameters evaluated at $(\tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{u}}_t)$.

This is again the linear optimal control problem[1] which is solvable (the LQR problem), see subsection 4.2.1. But it suffers from one big problem: the LQR problem can be constructed locally around the optimal trajectory, but the local LQR parameters are needed to compute the optimal trajectory! Hence, the optimal solver is iterative, updating the approximation until it converges to a (locally) optimal solution. This algorithm is known as *differential dynamic programming* (DDP).

[1] Note that in this case the matrix $\boldsymbol{R}$ has to be negative definite as there is no minus in front of the quadratic term!

5.2.1. Differential Dynamic Programming

In *differential dynamic programming* (DDP) it is, in contrast to LQR, often useful to work with the Q-function instead of the value function. For an arbitrary value function $V_t(\boldsymbol{s})$ and an arbitrary reward function $r_t(\boldsymbol{s}, \boldsymbol{a})$, the Q-function is given as

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

The first- and second-order derivatives are therefore:

$$\begin{aligned}
Q_t^{\boldsymbol{x}} &\coloneqq \frac{\partial Q_t}{\partial \boldsymbol{x}} = \frac{\partial r_t}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{x}} \frac{\partial V_{t+1}}{\partial \boldsymbol{s}} \\
Q_t^{\boldsymbol{u}} &\coloneqq \frac{\partial Q_t}{\partial \boldsymbol{u}} = \frac{\partial r_t}{\partial \boldsymbol{u}} + \frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{u}} \frac{\partial V_{t+1}}{\partial \boldsymbol{s}} \\
Q_t^{\boldsymbol{xx}} &\coloneqq \frac{\partial^2 Q_t}{\partial \boldsymbol{x}^2} = \frac{\partial^2 r_t}{\partial \boldsymbol{x}^2} + \left(\frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{x}}\right)^T \frac{\partial^2 V_{t+1}}{\partial \boldsymbol{x}^2} \frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{x}} + \frac{\partial V_{t+1}}{\partial \boldsymbol{s}} \cdot \frac{\partial^2 \boldsymbol{f}_t}{\partial \boldsymbol{x}^2} \\
Q_t^{\boldsymbol{xu}} &\coloneqq \frac{\partial^2 Q_t}{\partial \boldsymbol{u}^2} = \frac{\partial^2 r_t}{\partial \boldsymbol{u}^2} + \left(\frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{u}}\right)^T \frac{\partial^2 V_{t+1}}{\partial \boldsymbol{u}^2} \frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{u}} + \frac{\partial V_{t+1}}{\partial \boldsymbol{s}} \cdot \frac{\partial^2 \boldsymbol{f}_t}{\partial \boldsymbol{u}^2} \\
Q_t^{\boldsymbol{xu}} &\coloneqq \frac{\partial^2 Q_t}{\partial \boldsymbol{x} \partial \boldsymbol{u}} = \frac{\partial^2 r_t}{\partial \boldsymbol{x} \partial \boldsymbol{u}} + \left(\frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{x}}\right)^T \frac{\partial^2 V_{t+1}}{\partial \boldsymbol{x} \partial \boldsymbol{u}} \frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{u}} + \frac{\partial V_{t+1}}{\partial \boldsymbol{x}} \cdot \frac{\partial^2 \boldsymbol{f}_t}{\partial \boldsymbol{x} \partial \boldsymbol{u}}
\end{aligned} \tag{5.1}$$

These derivatives can now be used to solve the optimization problem

$$\boldsymbol{\delta u}^*(\boldsymbol{\delta x}) = \arg\max_{\boldsymbol{\delta u}} \tilde{Q}_t(\boldsymbol{\delta x}, \boldsymbol{\delta u}),$$

which describes the optimal action perturbation $\boldsymbol{\delta u}^*$ at a point $(\boldsymbol{x}, \boldsymbol{u})$ with the Q-function for a perturbation defined as

$$\begin{aligned}
\tilde{Q}_t(\boldsymbol{\delta x}, \boldsymbol{\delta u}) &\coloneqq Q(\boldsymbol{x} + \boldsymbol{\delta x}, \boldsymbol{u} + \boldsymbol{\delta u}) - Q(\boldsymbol{x}, \boldsymbol{u}) \\
&= r_t(\boldsymbol{x} + \boldsymbol{\delta x}, \boldsymbol{u} + \boldsymbol{\delta u}) - r_t(\boldsymbol{x}, \boldsymbol{u}) + V_{t+1}\big(\boldsymbol{f}(\boldsymbol{x} + \boldsymbol{\delta x}, \boldsymbol{u} + \boldsymbol{\delta u})\big) - V_{t+1}\big(\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})\big)
\end{aligned}$$

and the second-order Taylor approximation

$$\approx \frac{1}{2} \begin{bmatrix} 1 & \boldsymbol{\delta x}^T & \boldsymbol{\delta u}^T \end{bmatrix} \begin{bmatrix} 0 & Q_t^{\boldsymbol{x},T} & Q_t^{\boldsymbol{u},T} \\ Q_t^{\boldsymbol{x}} & Q_t^{\boldsymbol{xx}} & Q_t^{\boldsymbol{xu}} \\ Q_t^{\boldsymbol{u}} & Q_t^{\boldsymbol{ux}} & Q_t^{\boldsymbol{uu}} \end{bmatrix} \begin{bmatrix} 1 \\ \boldsymbol{\delta x} \\ \boldsymbol{\delta u} \end{bmatrix} \tag{5.2}$$

This optimization problem can be solved in closed form by taking the derivative of the quadratic expansion (5.2) and setting it to zero:

$$\begin{aligned}
\frac{\partial \tilde{Q}_t}{\partial \boldsymbol{\delta u}} &\approx \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & Q_t^{\boldsymbol{x},T} & Q_t^{\boldsymbol{u},T} \\ Q_t^{\boldsymbol{x}} & Q_t^{\boldsymbol{xx}} & Q_t^{\boldsymbol{xu}} \\ Q_t^{\boldsymbol{u}} & Q_t^{\boldsymbol{ux}} & Q_t^{\boldsymbol{uu}} \end{bmatrix} \begin{bmatrix} 1 \\ \boldsymbol{\delta x} \\ \boldsymbol{\delta u} \end{bmatrix} = \begin{bmatrix} Q_t^{\boldsymbol{u}} & Q_t^{\boldsymbol{ux}} & Q_t^{\boldsymbol{uu}} \end{bmatrix} \begin{bmatrix} 1 \\ \boldsymbol{\delta x} \\ \boldsymbol{\delta u} \end{bmatrix} \\
&= Q_t^{\boldsymbol{u}} + Q_t^{\boldsymbol{ux}} \boldsymbol{\delta x} + Q_t^{\boldsymbol{uu}} \boldsymbol{\delta u} \overset{!}{=} \mathbf{0} \\
\implies \quad Q_t^{\boldsymbol{uu}} \boldsymbol{\delta u} &= -Q_t^{\boldsymbol{u}} - Q_t^{\boldsymbol{ux}} \boldsymbol{\delta x} \\
\iff \quad \boldsymbol{\delta u}^* &= -\big(Q_t^{\boldsymbol{uu}}\big)^{-1} \big(Q_t^{\boldsymbol{u}} + Q_t^{\boldsymbol{ux}} \boldsymbol{\delta x}\big) \doteq \boldsymbol{k}_t + \boldsymbol{K}_t \boldsymbol{\delta x}
\end{aligned} \tag{5.3}$$

Here, $\boldsymbol{k}_t \coloneqq -\big(Q_t^{\boldsymbol{uu}}\big)^{-1} Q_t^{\boldsymbol{u}}$ and $\boldsymbol{K}_t \coloneqq -\big(Q_t^{\boldsymbol{uu}}\big)^{-1} Q_t^{\boldsymbol{ux}} \boldsymbol{\delta x}$. Plugging this result back into (5.2) yields a quadratic model for the value function:

$$V_t(\boldsymbol{x}) = -\frac{1}{2} \boldsymbol{k}_t^T Q_t^{\boldsymbol{uu}} \boldsymbol{k} \qquad V_t^{\boldsymbol{x}}(\boldsymbol{x}) = Q_t^{\boldsymbol{x}} - \boldsymbol{K}_t^T Q_t^{\boldsymbol{uu}} \boldsymbol{k} \qquad V_t^{\boldsymbol{xx}}(\boldsymbol{x}) = Q_t^{\boldsymbol{xx}} - \boldsymbol{K}^T Q_t^{\boldsymbol{uu}} \boldsymbol{K}$$

This can be iterated backwards in time to compute a local approximation of the value function.
The DDP algorithm then consists of three steps:

1. Execute $\boldsymbol{u}_{1:T-1}$ forward in time, linearize the dynamics $\boldsymbol{f}$ and quadratize the reward r.
2. Update the local Q- and V-functions backwards in time, via dynamic programming. Compute the time-varying control law.
3. Update the new action sequence using line search.

Implementation Details

Computing the Quadratic Expansion There are multiple different methods of computing the quadratic expansion (5.2):

- Analytical solution (exact, computationally efficient but personally expensive).
- Automatic differentiation (exact, computationally expensive but personally cheap).
- Finite difference approximation (inexact, computationally okay-ish and personally cheap).

Line Search As the update is only valid locally, a line search is used to optimize the step size of the update. Each DDP iteration i gives a local update $\boldsymbol{\delta u}^{(i)}$, so the updated control law is

$$\hat{\boldsymbol{u}}^{(i)} = \boldsymbol{u}^{(i)} + \alpha \boldsymbol{k}^{(i)} + \boldsymbol{K}^{(i)}\big(\hat{\boldsymbol{x}}^{(i)} - \boldsymbol{x}^{(i)}\big)$$

where usually α is evaluated over a range of values to maximize the reward improvement

$$\boldsymbol{\nabla} J(\alpha) = \alpha \sum_{i=1}^{N-1} \boldsymbol{k}^{(i),T} Q_t^{\boldsymbol{u},(i)} + \frac{\alpha^2}{2} \sum_{i=1}^{N} \boldsymbol{k}^{(i),T} Q_t^{\boldsymbol{uu},(i)} \boldsymbol{k}^{(i)}.$$

Issues with DDP

Differential dynamic programming has two major issues:

1. The local approximation is only valid along the optimized trajectory. So if the state diverges (e.g. due to modeling errors or disturbances), future controllers are invalid.
   This can be solved by updating the controllers online using model-predictive control.
2. The Hessian terms are expensive to compute, prohibiting online computation.
   This can be solved by using Hessian approximations (e.g. BFGS).

5.2.2. Iterative LQR

When using Gauss-Newton optimization, the Hessians are approximated from the Jacobians (i.e. the Gradients). By leaving out the second-order derivatives of the state dynamics in the derivatives of the Q-function (5.1), computation time is saved and the DDP model is effectively transformed into a Gauss-Newton equivalent. This is called *iterative LQR* (iLQR). In DDP it is often necessary to regularize $Q_t^{\boldsymbol{uu}}$: $\hat{Q}_t^{\boldsymbol{uu}} \coloneqq Q_t^{\boldsymbol{uu}} + \mu \boldsymbol{I}$. But this regularization affects the controller as $Q_t^{\boldsymbol{uu}}$ defines the feedback control law (5.3). As $\mu \to \infty$, $\boldsymbol{K}_t \to \boldsymbol{O}$. In iLQR the regularization is directly applied to the Hessian of the value function in (5.1) with the motivation of reducing the trajectory update rather than the controller specifically.

5.2.3. Stochastic DDP

In stochastic DDP, a state-action dependent disturbance $F(\boldsymbol{x}, \boldsymbol{u})$ is considered. This yields equations similar to DDP with additional differential terms due to the disturbances. For constant noise, however, the solution is equivalent to DDP and for state-action dependent noise, the optimal solution seeks lower disturbance regions.

5.2.4. Guided Policy Search

Guided Policy Search (GPS) mitigates the problem of DDP only being valid along a trajectory by using a DDP-like algorithm for generating multiple controllers which are then used to train a neural network controller. GPS will be discussed in more detail in subsection 9.5.2.

5.3. Wrap-Up

This chapter covered two methods for solving nonlinear optimal control problems: differential and approximate dynamic programming. The former applies LQR to nonlinear problems by iteratively linearizing the dynamics around the current trajectory to get time-varying linear controllers. As the algorithm is a bit brittle and only valid along a single trajectory, it is often used online in a model-predictive control fashion. In practice, approximations are used to compute the Hessians which is expensive to do otherwise (e.g. iLQR). These local controllers can then also be used to fit a global controller (guided policy search). The latter uses function approximators to find a global value function while using vanilla value/policy iteration to find local value functions.

6. State Estimation

6.1. Kalman Filter as an Optimal Filter

6.1.1. Observers

6.1.2. Optimal Observers

6.1.3. Geometric Perspective

6.2. Kalman Filter as Bayesian Inference

6.3. Partially Observed Optimal Control

6.4. Extended, Unscented and Particle Filter

6.4.1. Extended Kalman Filter (EKF)

6.4.2. Cubature Kalman Filter (CKF)

6.4.3. Unscented Kalman Filter (UKF)

6.4.4. Particle Filter / Sequential Monte Carlo (PF/SMC)

Importance Sampling

Sequential Importance Sampling

Sequential Importance Resampling

6.4.5. Examples

Approximate Message Passing

Pendulum

6.5. Wrap-Up

7. Model Learning

This chapter focuses on why learning models is needed in robotics and how this can be done. Also a specific algorithm for linear Gaussian dynamical systems is studied in subsection 7.3.1. Models have a wide range of applications in robotics, e.g. to simulate control and motion planning algorithms, detect faults, assess stability and safety, use them for state estimations (see chapter 6) and to design controllers using control theory and trajectory optimization. One of the first decisions to make is which model to learn as there are a lot of models that are frequently used in robotics, for example:

- continuous-time dynamics: $\boldsymbol{F}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}, \boldsymbol{u}) = \boldsymbol{0}$
- discrete-time forward dynamics: $\boldsymbol{x}_{t+1} = \boldsymbol{f}(\boldsymbol{x}_t, \boldsymbol{u}_t)$
- discrete-time inverse dynamics: $\boldsymbol{u}_t = \boldsymbol{f}^{-1}(\boldsymbol{x}_t, \boldsymbol{x}_{t+1})$
- kinematics: $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{q})$
- sensors: $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$

But getting a good model is hard! There are three major approaches for this:

1. Use the data-sheet of the robot and derive the dynamics using physics.
2. Disassemble the system and measure every component.
3. Learn the model from data.

The first approach might be the fastest, but often the data given from the manufacturer is not correct or has a wide confidence region, e.g. in the friction. Disassembling the systems and measuring everything might yield the best model, but is also time-intensive. This also has to be redone quite often as the systems changes (e.g. due to wear of the motor). This chapter focuses on learning the model from data, which might be a good tradeoff between the former two approaches.

7.1. Modeling Assumptions: White, Black and Gray

When creating models, assumptions have to be made. The spectrum of assumptions ranges from white-box models (lots of usage of prior knowledge) to black-box models (no usage of prior knowledge) with gray-box models in between. The spectrum of different modeling strategies is quite large with lots of intermediate stages.

7.1.1. White-Box Strategy

For white-box models, lots of prior literature and domain knowledge is used, e.g. in the recursive Newton-Euler algorithm. The physical parameters (e.g. the center of mass) is retrieved from the data-sheet or by disassembling and measuring the system (with all the caveats discussed before). These parameters can then be fine-tuned using data from the real system. Some up- and downsides of this strategy are shown in Table 7.1.

Positives	Negatives
Leverages prior work	Lots of manual labor
Interpretable	Unmodeled phenomena (e.g. friction) may cause issues
Should generalize well	

Table 7.1.: Positives and negatives of the white-box modeling strategy.

Positives	Negatives
Only required training data	Amount of data needed is unknown
Quick to implement	Generalization is questionable
	May be computationally expensive to train and use

Table 7.2.: Positives and negatives of the black-box modeling strategy.

7.1.2. Black-Box Strategy

The black-box strategy is the opposite end of the spectrum compared to white-box models. No prior work or domain knowledge is used, instead lots of training data is collected and then a model (e.g. a neural network, Gaussian process) is fit to the data. Some up- and downsides of this strategy are shown in Table 7.2.

7.1.3. Gray-Box Strategy

Gray-box models combine aspects of the white- and black-box models, for example:

- Fit a white-box model and handle the remaining error with a black-box model (e.g. a Gaussian process with a physics-based mean).
- Fit a black-box model with special constraints (e.g. ensure energy conservation in a closed system).
- Construct a model with white- and black-box components that can be trained "end-to-end" (e.g. a differentiable physics model combined with a neural network friction model, jointly trained with gradient decent).

7.2. Collecting the Data for Training the Model

One big problem in model learning is to get the data from the real robot. The main question is what input signal is the best for learning models, i.e. what controls should the robot be given to perform interesting movements?

For linear systems, so-called *responses* are identified, which are the output given a certain signal as an input. These signals used for identifying the responses usually are the simplest component of a broader class of signals. For analog signals, usually delta functions are a good signal. For digital and periodic signals, step functions and sinusoidal harmonics are used, respectively.

7.2.1. Impulse Response

The delta function (an *impulse*) is widely used in physics and signal processing. It is defined to have the following properties:

$$\delta(t-\tau) = \begin{cases} 1 & \text{if } t = \tau \\ 0 & \text{otherwise} \end{cases} \qquad \int_{-\infty}^{\infty} \delta(t-\tau)\,\mathrm{d}t = 1 \qquad \int_{-\infty}^{\infty} f(t)\delta(t-\tau)\,\mathrm{d}t = f(\tau)$$

Impulse functions can be used to approximate an arbitrary function as a sequence of impulses as

$$f(t) \approx \sum_{n=-\infty}^{\infty} f(n\Delta_t)\delta(t-n\Delta_t)$$

where Δ_t is a time step. As $\Delta_t \to 0$, the function is approximated perfectly, i.e. the approximation becomes an equality. This is the main idea of a *convolution*, which is an important part of signal processing and system identification theory.

7.2.2. Step Response

The step function (also called *Heaviside step function*) is closely related to the impulse, however ot contains a jump and is defined as:

$$\mathit{step}(t-\tau) = \begin{cases} 1 & \text{if } t \geq \tau \\ 0 & \text{otherwise} \end{cases} \qquad \frac{\mathrm{d}}{\mathrm{d}t}\mathit{step}(t-\tau) = \delta(t-\tau)$$

Similarly to the impulse, the step function can be used to represent digital (quantized) signals.

7.2.3. Characterization of Dynamical Systems

Dynamical systems are usually characterized using the terms *damping* and *resonant frequencies*, which both origin in linear system theory:

Damping Describes how the system dissipates energy.

- Heavily damped: Energy is lost quickly, the dynamics are "slow".
- Lightly damped: Energy is lost fast, the dynamics are "quick" and oscillations are common.
- "Critically" damped: Energy is lost "ideally", that is, as fast as possible without oscillations.

Resonant Frequencies These are the frequencies that maximally excite a system. This is important when designing structures like bridges, where hitting the resonant frequencies may cause catastrophic accidents.

This intuition holds locally for nonlinear system by linear approximation (Taylor-expansion), but the properties vary with the state (and thus the time)!

7.2.4. Frequency Analysis

Another viewpoint to analyze whether a signal is good for system identification is their frequency spectrum. The broader the bandwidth is, the more informative the response becomes. These frequencies can be obtained with a Fourier transform. For an impulse signal, all frequencies are covered and for sinusoidal harmonics only specific frequencies are covered. Motivate from this, other signal types are studied, for example chirps and white Gaussian noise. Chirps describe an oscillation of which the frequency gets higher and higher, like $\sin(x^2)$. White Gaussian noise also covers a wide range of frequencies, however both of these signals are impractical in reality: Physical systems require smooth inputs for safety, limiting the frequency. A practical solution to this problem is to low-pass the original signal, but this obviously causes a loss in the amount of frequencies that are covered.

7.2.5. Ornstein-Uhlenbeck Process and Active Learning

The frequency analysis motivates the *Ornstein-Uhlenbeck process* for identification, which is smoothed noise:

$$\boldsymbol{u}_{t+1} = \alpha \boldsymbol{u}_t + \boldsymbol{\eta}, \quad \boldsymbol{\eta} \sim p(\cdot)$$

For higher α, this causes more movements in the actions.

Another approach in machine learning is to use the model itself to identifying the signals. This task of choosing the data points which inform the model the most is called *active learning* and it is highly coupled with *information theory*. There are several objectives that can be used for selecting the data point, for example:

- Uncertainty: Variance/Entropy: $\mathrm{Var}[\boldsymbol{x}_t]$, $H[\boldsymbol{x}_t]$
- Expected Variance Reduction: $\mathrm{Var}\left[\mathcal{M} \,\middle|\, \mathcal{D} \cup (\boldsymbol{x}_{1:T}, \boldsymbol{u}_{1:T-1})\right]$
  "Variance reduction in model $\mathcal{M}$ by adding the trajectory to the dataset $\mathcal{D}$."
- Information Gain: $\frac{1}{2} \log \frac{\mathrm{Var}[\boldsymbol{x}_t]}{\mathrm{Var}_{\text{noise}}}$
  Equivalent to $H[\boldsymbol{x}] - H_{\text{min}}$ for Gaussian beliefs.

Active learning require a model that is capable of expressing the model uncertainty, hence it requires a Bayesian model, e.g. a Gaussian process. Optimizing the trajectories with active learning is still an open research topic!

In practice, out-of-phase sinuses on every joint create very diverse end-effector trajectories that are useful for learning.

7.3. Learning Models

Now given good training data, what is the best objective L for learning a good model? There are multiple options, e.g.

- One-step regression, i.e. $L = \left\| \boldsymbol{x}_{t+1} - \boldsymbol{f}(\boldsymbol{x}_t) \right\|_2^2$
- Multi-step regression, i.e. $L = \left\| \boldsymbol{x}_{t+k} - \boldsymbol{f}^k(\boldsymbol{x}_t) \right\|_2^2$
- Inference over the trajectory, i.e. $L = \log p(\boldsymbol{x}_{1:T})$

where $\boldsymbol{f}^k(\boldsymbol{x}_k)$ refers to applying $\boldsymbol{f}(\boldsymbol{x}_k)$ k times to the state $\boldsymbol{x}_t$, or rather the respective result of the previous application. In general an objective that considers a longer prediction horizon (a higher k) should yield models that have a better long-horizon prediction, but are more expensive to train. It becomes even harder when the problem is only partially observed. One such interesting (and solvable!) case are Linear Gaussian Dynamical Systems (LGDS) which are covered in the next section.

7.3.1. Linear Gaussian Dynamical Systems (LGDS)

Linear Gaussian dynamical systems are given as linear state transitions in an unobserved (latent) space $\boldsymbol{x}_t$ from which measurements $\boldsymbol{y}_t$ are taken (also linearly). Both of these parts underlie additive Gaussian noise, forming the following probabilistic model

$$\begin{aligned} p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t) &= \mathcal{N}\big(\boldsymbol{x}_{t+1} \,\big|\, \boldsymbol{A}\boldsymbol{x}_t,\, \boldsymbol{\Sigma}_\eta\big) \\ p(\boldsymbol{y}_t \mid \boldsymbol{x}_t) &= \mathcal{N}\big(\boldsymbol{y}_t \,\big|\, \boldsymbol{C}\boldsymbol{x}_t,\, \boldsymbol{\Sigma}_\zeta\big) \end{aligned}$$

with the initial state distribution $\boldsymbol{x}_1 \sim \mathcal{N}(\cdot \mid \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$. The model parameters are defined as

$$\boldsymbol{\theta} \coloneqq \big\{\, \boldsymbol{A},\, \boldsymbol{C},\, \boldsymbol{\Sigma}_\eta,\, \boldsymbol{\Sigma}_\zeta,\, \boldsymbol{\mu}_1,\, \boldsymbol{\Sigma}_1 \,\big\}$$

which are to be estimated from a sequence of measurements $\boldsymbol{y}_{1:T}$. This is done using an EM algorithm that in the E-step uses the model to infer the hidden states $\boldsymbol{x}_t$ using a Kalman filter and in the M-step maximizes the expected log-likelihood to improve the model.

As the system behaves Markovian (the next state only depends on the current state and not the past), the likelihood can be factorized as

$$p(\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T}) = p(\boldsymbol{x}_1) \prod_{t=1}^{T-1} p(\boldsymbol{x}_{t+1} \mid \boldsymbol{s}_t) \prod_{t=1}^{T} p(\boldsymbol{y}_t \mid \boldsymbol{x}_t).$$

Hence, the log-likelihood is

$$\log p(\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T}) = \log p(\boldsymbol{x}_1) + \sum_{t=1}^{T-1} \log p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t) + \sum_{t=1}^{T} \log p(\boldsymbol{y}_t \mid \boldsymbol{x}_t).$$

As the latent states $\boldsymbol{x}_t$ are unknown, it is not possible to maximize the log-likelihood directly, but rather the *expected* log-likelihood over $\boldsymbol{x}_t$, conditioned on the observations, has to be maximized. This can be done straightforwardly by exploiting the invariance of the trace under cycling permutations. This (rather ugly)

process[1] yields the expected log-likelihood

$$\begin{aligned}\mathbb{E}_{\boldsymbol{x}_{1:T}}\left[\log p(\boldsymbol{x}_{1:T}, \boldsymbol{y}_{1:T}) \,\middle|\, \boldsymbol{y}_{1:T}\right] = &\underbrace{-\frac{T(k+p)}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}_1| - \frac{T-1}{2}\log|\boldsymbol{\Sigma}_\eta| - \frac{T}{2}\log|\boldsymbol{\Sigma}_\zeta|}_{\text{Entropies}} \\ &\underbrace{-\frac{1}{2}\operatorname{tr}\left(\boldsymbol{P}_1^{-1}\left(\boldsymbol{\Sigma}_1 - \hat{\boldsymbol{x}}_1\boldsymbol{\mu}_1^T - \boldsymbol{\mu}_1\hat{\boldsymbol{x}}_1^T + \boldsymbol{\mu}_1\boldsymbol{\mu}_1^T\right)\right)}_{\text{Initial State Error}} \\ &\underbrace{-\frac{1}{2}\operatorname{tr}\left(\boldsymbol{\Sigma}_\eta^{-1}\left(\sum_{t=2}^T \boldsymbol{P}_t - \boldsymbol{P}_{t,t-1}\boldsymbol{A}^T - \boldsymbol{A}\boldsymbol{P}_{t,t-1} + \boldsymbol{A}\boldsymbol{P}_{t-1}\boldsymbol{A}^T\right)\right)}_{\text{Dynamics Error}} \\ &\underbrace{-\frac{1}{2}\operatorname{tr}\left(\boldsymbol{\Sigma}_\zeta^{-1}\left(\sum_{t=1}^T \boldsymbol{y}_t\boldsymbol{y}_t^T - \boldsymbol{y}_t\hat{\boldsymbol{x}}_t^T\boldsymbol{C}^T - \boldsymbol{C}\hat{\boldsymbol{x}}_t\boldsymbol{y}_t^T + \boldsymbol{C}\boldsymbol{P}_t\boldsymbol{C}^T\right)\right)}_{\text{Observation Error}}\end{aligned}$$

with the expectations

$$\boldsymbol{P}_t \coloneqq \mathbb{E}_{\boldsymbol{x}_{1:T}}\left[\boldsymbol{x}_t\boldsymbol{x}_t^T \,\middle|\, \boldsymbol{y}_{1:T}\right] \qquad \boldsymbol{P}_{t,t-1} \coloneqq \mathbb{E}_{\boldsymbol{x}_{1:T}}\left[\boldsymbol{x}_t\boldsymbol{x}_{t-1}^T \,\middle|\, \boldsymbol{y}_{1:T}\right] \qquad \hat{\boldsymbol{x}}_t \coloneqq \mathbb{E}_{\boldsymbol{x}_{1:T}}[\boldsymbol{x}_t \,|\, \boldsymbol{y}_{1:T}]$$

that encode the self and cross correlation and the expected states, respectively. By maximizing the expected log-likelihood, closed-form update rules for the parameters can be derived given an expected latent trajectory. This yields the following update rules:

$$\begin{aligned}\boldsymbol{A}^{\text{new}} &= \left(\sum_{t=2}^T \boldsymbol{P}_{t,t-1}\right)\left(\sum_{t=2}^T \boldsymbol{P}_{t-1}\right)^{-1} \\ \boldsymbol{C}^{\text{new}} &= \left(\sum_{t=1}^T \boldsymbol{y}_t\hat{\boldsymbol{x}}_t^T\right)\left(\sum_{t=1}^T \boldsymbol{P}_t\right) \\ \boldsymbol{\Sigma}_\eta^{\text{new}} &= \frac{1}{T-1}\left(\sum_{t=2}^T \boldsymbol{P}_t - \boldsymbol{A}^{\text{new}}\sum_{t=2}^T \boldsymbol{P}_{t,t-1}\right) \\ \boldsymbol{\Sigma}_\zeta^{\text{new}} &= \frac{1}{T}\sum_{t=1}^T \boldsymbol{y}_t\boldsymbol{y}_t^T - \boldsymbol{C}^{\text{new}}\hat{\boldsymbol{x}}_t\boldsymbol{y}_t^T \\ \boldsymbol{\mu}_1^{\text{new}} &= \hat{\boldsymbol{x}}_1 \\ \boldsymbol{\Sigma}_1^{\text{new}} &= \boldsymbol{P}_1 - \hat{\boldsymbol{x}}_1\hat{\boldsymbol{x}}_1^T\end{aligned}$$

Plugging this together with the Kalman smoother, this yields an expectation-maximization (EM) algorithm with the E-step being the Kalman smoother and the M-step being the above update rules. For the LGDS, this guarantees monotonic improvement in the expected log-likelihood! For nonlinear settings, it is possible to approximate the expectations (e.g. using cubature rules or particle methods) and update the parameters using gradient descent. This is called *approximate* EM.

7.4. Case Studies

This section looks into some specific examples of model learning that go beyond LGDS.

[1] See `https://www.ias.informatik.tu-darmstadt.de/uploads/Team/JoeWatson/Damken_Bsc_2020.pdf`, pages 75 to 82 for the full derivation.

7.4.1. Combining Rigid Body Dynamics and Gaussian Processes

Inverse rigid body dynamics can be expressed as a linear equation $\boldsymbol{\tau} = \boldsymbol{\Phi}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}})\boldsymbol{\beta}$ where the matrix $\boldsymbol{\Phi}$ contains nonlinear functions of the joint angles/velocities/accelerations. The weights $\boldsymbol{\beta}$ contain physical parameters like masses. However, this model misses nonlinear effects like friction! One method to handle this is to combine the rigid body model with a Gaussian process to fit the residual error (with $\boldsymbol{x} = \{\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}\}$ for brevity):

$$\boldsymbol{\tau}(\boldsymbol{x}) \sim \boldsymbol{\Phi}(\boldsymbol{x})\boldsymbol{\beta} + \mathcal{GP}\big(\mathbf{0},\, k(\boldsymbol{x}, \boldsymbol{x}')\big)$$

While standard kernels are not well-suited for this problem, it is possible to define a linear kernel using the feature functions $\boldsymbol{\phi}_k$ of $\boldsymbol{\Phi}$:

$$k^k_{\text{rbd}}(\boldsymbol{x}_p, \boldsymbol{x}_q) = \boldsymbol{\phi}_k^T(\boldsymbol{x}_p)\boldsymbol{W}_k\boldsymbol{\phi}_k(\boldsymbol{x}_q) + \sigma_k^2\delta_{pq}$$

This kernel was found to have superior performance in handling the residual error.

7.4.2. Deep Lagrangian Networks

Deep Lagrangian networks (DeLaNs) use neural networks to model the Lagrangian equations of motions, resulting in the inductive bias of energy conservation. This should yield to more physically plausible model predictions. See Lutter et al., "Deep Lagrangian Networks: Using Physics as Model Prior for Deep Learning" (2019) for more information on this.

7.4.3. The Differentiable Recursive Newton-Euler Algorithm

Implementing the recursive Newton-Euler algorithm that was introduced in section 2.1.4 in an automatic differentiation library, physical models can be trained with gradient decent! However, care has to be taken on the parameters, e.g. to enforce positive masses (this can be done, for example, by training the square-root instead of the parameter itself). As it is trained with gradient decent, black-box models can be easily integrated. Some example (where f_{NN} is a feedforward neural network) are:

Friction $\tau = \tau_d - \text{sign}(\dot{q})\left\|f_{\text{NN}}(q, \dot{q}; \psi_F)\right\|_1$

Residual Error $\tau = \tau_d - f_{\text{NN}}(q, \dot{q}; \psi_R)$

Full NN $\tau = f_{\text{NN}}(\tau - d, q, \dot{q}; \psi_M)$

Nonholonomic systems can also be modeled by using maximal rather than intrinsic coordinates:

Holonomic Only constrained in position.

Non-Holonomic Also constrained on differential terms, e.g. velocities.

Intrinsic Coordinates Internal reference frame.

Maximal Coordinates External reference frame.

One example for a non-holonomic constraint is the string in the ball-in-cup environment. Black-box models like neural networks or recurrent neural networks fail on this, and the learned policies are not transferable to the real system. The differential recursive Newton-Euler algorithm can plan with the string constraint, and the resulting policies work on the real robot!

8. Policy Representations

To train a policy $\pi(\boldsymbol{a} \mid \boldsymbol{s})$, this policy has to be represented somehow. In control theory, this policy is commonly known as the *controller* and has the form of a PID controller, for example. In robot learning, however, these policies have to be represented in a "learnable" way. This chapter covers parametric policies with off-the-shelf methods like neural networks and movement primitives which can encode complex behavior in a modular fashion. This chapter focuses on parametric policies, i.e. policies that is represented by a conditional probability distribution $\pi(\boldsymbol{a} \mid \boldsymbol{s}; \boldsymbol{\theta})$ with the parameters $\boldsymbol{\theta}$. Desirable properties for policy representations are:

Compactness The number of parameters should be low.

Learnability Easy to learn from demonstrations and by reinforcement learning.

Stochasticity Ability to encode exploration and variability.

Optimality Should be able to encode the optimal behavior.

Scalability Usable for a high number of degrees of freedom.

Modularity Adaptability for a new environment and co-activation and blending of movements.

Usability Should be usable for stroke-based (point-to-point) and rhythmic movement.

A stochastic policy is useful as it can be used for exploration in reinforcement learning and can capture variability in the movements. Different exploration models are for example:

- No exploration: $\boldsymbol{a} = \pi(\boldsymbol{s}) = \boldsymbol{f}_{\boldsymbol{w}}(\boldsymbol{s}), \qquad \boldsymbol{\theta} = \boldsymbol{w}$
- Uncorrelated: $\boldsymbol{a} \sim \pi(\cdot \mid \boldsymbol{s}; \boldsymbol{\theta}) = \mathcal{N}\big(\cdot \mid \boldsymbol{f}_{\boldsymbol{w}}(\boldsymbol{s}), \sigma^2 \boldsymbol{I}\big), \quad \boldsymbol{\theta} = \{\boldsymbol{w}, \sigma^2\}$
- Correlated: $\boldsymbol{a} \sim \pi(\cdot \mid \boldsymbol{s}; \boldsymbol{\theta}) = \mathcal{N}\big(\cdot \mid \boldsymbol{f}_{\boldsymbol{w}}(\boldsymbol{s}), \boldsymbol{\Sigma}\big), \quad \boldsymbol{\theta} = \{\boldsymbol{w}, \boldsymbol{\Sigma}\}$
- Complex distributions like Gaussian mixture models, normalizing flows, …

8.1. Off-The-Shelf Policies

Some examples for off-the-shelf policies are neural networks, radial basis functions (RBF) networks, Gaussian processes and locally weighted regression models. While they are easy to use, off-the-self policies have some major drawbacks: they usually generalize bad if the controller has not been trained in the requested area of the state space, they are not robust (so also small changes in the system may lead to catastrophic failures), and they are often really sample-inefficient.

8.1.1. Linear Basis Functions

Linear controllers $\boldsymbol{a} = \boldsymbol{f_w}(\boldsymbol{s}) = \boldsymbol{\phi}^T(\boldsymbol{s})\boldsymbol{w}$, in the simplest case a PD-controller with $\boldsymbol{\phi}^T(\boldsymbol{s}) = \begin{bmatrix} 1 & \boldsymbol{s}^T \end{bmatrix}$, may yield really good policies using a very compact representation. Also they are easy to learn with linear regression. Whoever, a good feature representation has to be used! This representation was successfully applied to balancing a pole on a cart, but even learning the swing-up can be hard.

8.1.2. Radial Basis Functions (RBFs)

A more expressive policy representation are weighted radial basis functions (RBFs) $\boldsymbol{f_w}(\boldsymbol{s}) = \boldsymbol{\phi}^T(\boldsymbol{s})\boldsymbol{\beta}$ with the features

$$\phi_i(\boldsymbol{s}) = \exp\left\{ -\frac{1}{2} \sum_{j=1}^{D} \frac{(s_j - \mu_{ij})^2}{h_i j} \right\}.$$

The parameters in are $\boldsymbol{w} = \{\boldsymbol{\beta}, \boldsymbol{\mu}_{1:K}, \boldsymbol{h}_{1:K}\}$, where K is the number of RBFs and $\boldsymbol{\mu}_{1:K}$ and $\boldsymbol{h}_{1:K}$ are the locations and bandwidths of the RBFs, respectively. But as the features are really localized, it might be possible that no basis function is active at some point. This can be fixed by using normalized RBFs,

$$\boldsymbol{f}_{\boldsymbol{w}}^{norm}(\boldsymbol{s}) = \frac{\sum_{i=1}^{K} \phi_i(\boldsymbol{s}\beta_i)}{\sum_{j=1}^{K} \phi_i(\boldsymbol{s})},$$

which do not suffer from this problem. However, this can lead to numerically issues as the denominator gets really small if every basis function is far from being active. Also, this problem is really hard to optimize as the optimization for $\boldsymbol{h}_{1:K}$ is non-convex!

8.1.3. Trajectory Following Controller

Another method for policy representation is to use the previously discussed methods for representing a trajectory (see section 2.2) and use a feedback controller

$$\pi(\boldsymbol{q}, \dot{\boldsymbol{q}}, t; \boldsymbol{w}) = \boldsymbol{K}_P\big(\boldsymbol{q}_d(\boldsymbol{q}, \dot{\boldsymbol{q}}, t; \boldsymbol{w}) - \boldsymbol{w}\big) + \boldsymbol{K}_D\big(\dot{\boldsymbol{q}}_d(\boldsymbol{q}, \dot{\boldsymbol{q}}, t; \boldsymbol{w}) - \dot{\boldsymbol{q}}\big)$$

to follow the desired trajectory $\big(\boldsymbol{q}_d(\boldsymbol{q}, \dot{\boldsymbol{q}}, t; \boldsymbol{w}), \dot{\boldsymbol{q}}_d(\boldsymbol{q}, \dot{\boldsymbol{q}}, t; \boldsymbol{w})\big)$. Usually the matrices $\boldsymbol{K}_P$ and $\boldsymbol{K}_D$ are hand-tuned, but they may also be learned.

8.2. Movement Primitives

Movement primitives are a compact representation of movements that are often represented as trajectory generators $\boldsymbol{\tau} = \boldsymbol{f}(\boldsymbol{w})$ with the parameters $\boldsymbol{w}$ of the primitive. They can represent both stroke-based and rhythmic movements and can be time-dependent, state-dependent or both. The basic idea is to use dynamical systems to represent trajectories as integrating it results in a trajectory. Dynamical systems (defined by a second-order differential equation) can encode various different movements and the whole class of stability analysis from ODEs can be used to generate e.g. stroke-based movements by exploiting a single-point attractor. These movement primitives are known as *dynamic* movement primitives (DMPs).

8.2.1. Dynamic Movement Primitives (DMPs)

Dynamic movement primitives can encode lots of desirable properties like stability, perturbation robustness and stroke-based and rhythmic behaviors. They are also really easy to learn and scale well with a high number of degrees of freedom. By intelligent parametrization, they can also encode temporal scaling, i.e. it is possible to execute a movement slower.

Starting from the model of a spring-damper system, $\ddot{y} = \alpha\big(\beta(g - y) - \dot{y}\big)$, which always converges to the goal attractor g as time goes by[1], a *forcing function* $f_{\boldsymbol{w}}(t)$ can be added to obtain a moving attractor

$$\ddot{y} = \alpha\big(\beta(g - y) - \dot{y}\big) + f_{\boldsymbol{w}}(t) = \alpha\left(\beta\left(g + \frac{f_{\boldsymbol{w}}(t)}{\alpha\beta} - y\right) - \dot{y}\right) \tag{8.1}$$

that converges to $g + f_{\boldsymbol{w}}(t)/(\alpha\beta)$. The forcing function $f_{\boldsymbol{w}}$ is a learnable function with parameters $\boldsymbol{w}$ to adapt to all kinds of movements.

Using DMPs for multiple degrees of freedom is also possible by using an individual DMP per DoF, while sharing the phase variable z. For periodic movements, periodic phase variables can be used as these control the trajectory.

Temporal Scaling

To achieve temporal scaling, a phase variable $z(t)$ is added to the system to replace the time t in (8.1):

$$\begin{aligned}\ddot{y} &= \tau^2\alpha\big(\beta(g - y) - \dot{y}/\tau\big) + \tau^2 f_{\boldsymbol{w}}(z)\\ \dot{z} &= -\tau\alpha_z z\end{aligned}$$

The variable τ is the *temporal scaling variable* and z is called the *phase*. Higher values of τ correspond to higher movement speeds and a value of $\tau = 1$ corresponds to the "original" movement speed.

Representation of the Forcing Function

One approach for representing the forcing function is to use the function

$$f_{\boldsymbol{w}} = \boldsymbol{\psi}^T(z)\boldsymbol{w}$$

with K normalized RBF basis functions[2]

$$\psi_i(z) = \frac{\phi_i(z)z}{\sum_{j=1}^{K}\phi_j(z)} \qquad\qquad \phi_i(z) = \exp\left\{-\frac{1}{2}\frac{(z - c_i)^2}{h_i}\right\} \tag{8.2}$$

with the locations $c_{1:K}$ and the bandwidths $h_{1:K}$. These dynamics movement primitives are stable by design as for $t \to \infty$ the forcing function vanishes[3] and the becomes a PD controller.

[1] More formally, $\lim_{t\to\infty} y(t) = g$.

[2] Notice the multiplication with the phase variable z in the RBF! This is used to make the DMP stable and is just more convenient to put it there instead of in the forcing function itself.

[3] This is due to $z \to 0$ for $t \to \infty$ as $\dot{z} = -\tau\alpha_z z$ and $\tau, \alpha_z > 0$.

Imitation Learning

Dynamic movement primitives with radial basis functions can be easily used for imitation learning using linear (ridge) regression. Given a trajectory $(\boldsymbol{q}_{1:T}, \dot{\boldsymbol{q}}_{1:T}, \ddot{\boldsymbol{q}}_{1:T})$, a goal attractor g, parameters α, β, α_z and a temporal scaling variable τ, first compute the target values of the forcing function for each t:

$$f_t = \frac{\ddot{\boldsymbol{q}}_t}{\tau^2} - \alpha\big(\beta(g - q_t) - \dot{q}_t/\tau^2\big),$$

Subsequently, compute the weights $\boldsymbol{w}$ of $f_{\boldsymbol{w}}$ with

$$\boldsymbol{w} = \big(\boldsymbol{\Psi}^T\boldsymbol{\Psi} + \sigma^2\boldsymbol{I}\big)^{-1}\boldsymbol{\Psi}\boldsymbol{f}$$

where $\boldsymbol{\Psi} = \begin{bmatrix}\boldsymbol{\psi}_1 & \boldsymbol{\psi}_2 & \cdots & \boldsymbol{\psi}_T\end{bmatrix}$ is the feature matrix and $\boldsymbol{f}$ is the vector of target values. The parameter σ^2 is used for regularization and can also be chosen $\sigma^2 = 0$ for non-ridge regression.

8.2.2. Probabilistic Movement Primitives (ProMPs)

Another approach on movement primitives are *probabilistic movement primitives* (ProMPs), a stochastic representation of trajectories $\boldsymbol{\tau} \sim p(\cdot \mid \boldsymbol{w})$. This is represented by learning a (parameterized) distribution $p(\boldsymbol{w} \mid \boldsymbol{\theta})$ over the parameters $\boldsymbol{w}$, each representing a single trajectory

$$\boldsymbol{\tau} = \boldsymbol{f}(\boldsymbol{w}) + \boldsymbol{\epsilon}$$

with noise ϵ. The weights can then be integrated out to obtain $p(\boldsymbol{\tau} \mid \boldsymbol{\theta})$. Stochastic movement primitives are useful as it allows to represent uncertainty which gives information on the importance of time points. Also probabilistic operations can be applied.

One possible representation of a trajectory is to use multiple normalized RBF basis, one for each degree of freedom. Let

$$\psi_i^{(d)}(z) = \frac{\phi_i^{(d)}(z)z}{\sum_{j=1}^K \phi_j^{(d)}(z)} \qquad \text{with} \qquad \phi_i^{(d)}(z) = \exp\left\{-\frac{1}{2}\frac{(z-c_i)^2}{h_i}\right\}$$

be the i-th RBF feature ($i = 1, \cdots, K$) of the d-th degree of freedom ($d = 1, \cdots, D$). Then let

$$\boldsymbol{\Psi}_t^T \coloneqq \begin{bmatrix} \psi_1^{(1)}(z_t) & \psi_2^{(1)}(z_t) & \cdots & \psi_K^{(1)}(z_t) \\ \psi_1^{(2)}(z_t) & \psi_2^{(2)}(z_t) & \cdots & \psi_K^{(2)}(z_t) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1^{(D)}(z_t) & \psi_2^{(D)}(z_t) & \cdots & \psi_K^{(D)}(z_t) \end{bmatrix}$$

be the phase-dependent basis for the ProMP. One trajectory is given as

$$\boldsymbol{y}_t \sim \mathcal{N}\big(\cdot \;\big|\; \boldsymbol{\Psi}_t^T\boldsymbol{w},\, \boldsymbol{\Sigma}_y\big),$$

with additive Gaussian noise, where $\boldsymbol{y}_t$ is the value of the trajectory at time step t. This can e.g. be the positions and velocities of the joints. Then the probabilistic model for the whole trajectory therefore is given as

$$p(\boldsymbol{\tau} \mid \boldsymbol{w}) = \prod_t \mathcal{N}\big(\boldsymbol{y}_t \;\big|\; \boldsymbol{\Psi}_t^T\boldsymbol{w},\, \boldsymbol{\Sigma}_y\big).$$

Given the distribution $p(\boldsymbol{w} \mid \boldsymbol{\theta})$ of the weights, the trajectory distribution $p(\boldsymbol{\tau} \mid \boldsymbol{\theta})$ can be obtained by marginalization:

$$p(\boldsymbol{\tau} \mid \boldsymbol{\theta}) = \int p(\boldsymbol{\tau} \mid \boldsymbol{w}) p(\boldsymbol{w} \mid \boldsymbol{\theta})\, \mathrm{d}\boldsymbol{w}$$

By using a Gaussian $p(\boldsymbol{w} \,|\, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{w} \,|\, \boldsymbol{\mu}_w, \boldsymbol{\Sigma}_w)$ with $\boldsymbol{\theta} = \{\boldsymbol{\mu}_w, \boldsymbol{\Sigma}_w\}$ for the weight distribution, the trajectory distribution can be computed in closed form for a time step t:

$$\begin{aligned} p(\boldsymbol{y}_t \,|\, \boldsymbol{\theta}) &= \int p(\boldsymbol{y}_t \,|\, \boldsymbol{w}) p(\boldsymbol{w} \,|\, \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{w} \\ &= \int \mathcal{N}\big(\boldsymbol{y}_t \,\big|\, \boldsymbol{\Psi}_t^T \boldsymbol{w},\, \boldsymbol{\Sigma}_y\big) \, \mathcal{N}(\boldsymbol{w} \,|\, \boldsymbol{\mu}_w, \boldsymbol{\Sigma}_w) \, \mathrm{d}\boldsymbol{w} \\ &= \mathcal{N}\big(\boldsymbol{y}_t \,\big|\, \boldsymbol{\Psi}_t^T \boldsymbol{\mu}_w,\, \boldsymbol{\Psi}_t^T \boldsymbol{\Sigma}_w \boldsymbol{\Psi}_t + \boldsymbol{\Sigma}_y\big) \end{aligned}$$

Conditioning

It is possible to condition the distribution over the parameters on a position $\boldsymbol{y}_{t^*}^*$ at time step t^* with covariance $\boldsymbol{\Sigma}_y^*$ by using Bayes rule:

$$p(\boldsymbol{w} \,|\, \boldsymbol{y}_{t^*}^*, \boldsymbol{\Sigma}_y^*) \propto \mathcal{N}\big(\boldsymbol{y}_{t^*}^* \,\big|\, \boldsymbol{\Psi}_{t^*}^T \boldsymbol{w},\, \boldsymbol{\Sigma}_y^*\big) p(\boldsymbol{w})$$

This allows ProMPs to generalize to other end-points of the movement or reaching intermediate points. Using Gaussian weight distributions, Bayes rule can be computed in closed form:

$$\begin{aligned} \boldsymbol{\mu}_w^{\text{new}} &= \boldsymbol{\mu}_w + \boldsymbol{\Sigma} \boldsymbol{\Psi}_{t^*} \big(\boldsymbol{\Psi}_{t^*}^T \boldsymbol{\Sigma}_w \boldsymbol{\Sigma}_{t^*} + \boldsymbol{\Sigma}_y^*\big)^{-1} \big(\boldsymbol{y}_{t^*}^* - \boldsymbol{\Psi}_{t^*}^T \boldsymbol{\mu}_w\big) \\ \boldsymbol{\Sigma}_w^{\text{new}} &= \boldsymbol{\Sigma}_w - \boldsymbol{\Sigma}_w \boldsymbol{\Psi}_{t^*} \big(\boldsymbol{\Psi}_{t^*}^T \boldsymbol{\Sigma}_w \boldsymbol{\Psi}_{t^*} + \boldsymbol{\Sigma}_y^*\big)^{-1} \boldsymbol{\Psi}_{t^*}^T \boldsymbol{\Sigma}_w \end{aligned}$$

Combination

It is possible to combine multiple ProMPs into one movement to solve a combination of tasks. For the distributions $\{p_i(\boldsymbol{y}_t)\}_{i=1}^N$, the combined ProMP $p_{\text{co}}(\boldsymbol{y}_t)$ is given as

$$p^{\text{co}}(\boldsymbol{y}_t) \propto \prod_{i=1}^{N} p_i(\boldsymbol{y}_t)^{\alpha_i(t)}$$

with the activation factors $\alpha_i(t)$. For Gaussian distributions $p_i(\boldsymbol{y}_t) = \mathcal{N}\big(\boldsymbol{y}_t \,\big|\, \boldsymbol{\mu}_t^{[i]}, \boldsymbol{\Sigma}_y^{[i]}\big)$, the combined distribution has the following mean and covariance:

$$\begin{aligned} \boldsymbol{\mu}_t^{\text{co}} &= \big(\boldsymbol{\Sigma}_t^{\text{co}}\big)^{-1} \left(\sum_{i=1}^{N} \left(\boldsymbol{\Sigma}_t^{[i]} / \alpha_t^{[i]} \right)^{-1} \boldsymbol{\mu}_t^{[i]} \right) \\ \boldsymbol{\Sigma}_t^{\text{co}} &= \left(\sum_{i=1}^{N} \left(\boldsymbol{\Sigma}_t^{[i]} / \alpha_t^{[i]} \right)^{-1} \right)^{-1} \end{aligned}$$

Using the time-varying activation factors $\alpha_i(t)$, it is also possible to achieve *blending*, i.e. shifting smoothly from one primitive (movement) to another.

8.2.3. Time-Independent Stable Movement Primitives

Both the DMPs and ProMPs consider the time as part of the state, however, knowing the exact time is hard and required external algorithms for phase estimation. It is possible to eliminate the dependency on the time in the state by learning *nonlinear stable dynamical systems*

$$\ddot{\boldsymbol{q}} = \boldsymbol{f}(\boldsymbol{q}, \dot{\boldsymbol{q}}; \boldsymbol{w}).$$

For linear dynamical systems, it is easy to construct them stable by choosing the state dynamics matrix to be negative definite. For nonlinear systems this is harder. But by combining a linear system with an invertible transformation $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{z})$, it is possible to create stable nonlinear systems from linear systems. This is due to the fact that the potential energy will be the same in both systems, i.e. $V(\boldsymbol{z}) = V(\boldsymbol{y})$. Examples for such invertible transformations are scalar multiplication or translations in the state-space (this will change the equilibrium!).

Imitation Flows

A wide range of such invertible transformations can be found by using invertible flows. Invertible flows are a set of neural networks that can be computed forward and backward, i.e. they are bijective. These can represent very complex nonlinear, yet stable, dynamics and are capable of representing both stroke-based and rhythmic movements.

8.2.4. Libraries of Primitives

Usually, one primitive is not enough. Ideally, the robot has a library of primitives which are all good at what they do and the robot "just" has to decide which movement to make by choosing the corresponding primitive.

9. Model-Based Reinforcement Learning

This chapter introduces model-based reinforcement learning methods. In reinforcement learning, the goal is to maximize the expected long-term reward like in optimal control (see chapter 4 and 5), however, no knowledge about the environment is given in comparison to solving a given MDP. This has lead to two strategies: model-based RL and model-free RL. In the former a dynamics model of the environment is learned which is then used for solving the optimal control problem. In the latter to (explicit) model is learned but rather the policy is learned directly from data. But the distinction of model-based and model-free RL is not very clean: model-based RL could still use model-free methods to optimize the policy, e.g. by using the learned model to improve accuracy (e.g. DYNA algorithm), use the dynamics to generate data for MFRL or to perturb the parameters to generate robust MFRL policies (sim-to-real). Analogously, MFRL methods often also use models: the V- and Q-function implicitly encode the dynamics of the system, but through the cumulative reward rather than the state.

A typical distinction between MBRL and MFRL is as follows: "MBRL is more sample-efficient than MFRL, but MFRL has better asymptotic performance, due to modeling errors introducing performance bias. Modeling errors can be catastrophic for MBRL so the algorithms are brittle." This gives rise to three questions which will be addresses in the following sections:

- *When* is MBRL more sample-efficient than MFRL?
- *Why* does MBRL have a performance bias?
- *When* are modeling errors catastrophic?

9.1. Sample Efficiency

The key distinction of MBRL and MFRL is the availability of an inductive bias to generalize outside of the data (OOD). While in a discrete MDP the transition matrix usually has to structure, an LQR system has – and this structure can be exploited by MBRL. Therefore, MBRL only benefits in terms of sample efficiency if there is domain knowledge that can be exploited.

Discrete MDP For a discrete MDP with state space $\mathcal{S}$ and action space $\mathcal{A}$ and a time horizon T, the *minimum*[1] amount of samples required to solve the MDP are:

- for model-free RL: $\mathcal{O}\big(|\mathcal{S}|^2 \cdot |\mathcal{A}|\big)$
- for model-based RL: $\mathcal{O}\big(|\mathcal{S}|^2 \cdot |\mathcal{A}|\big)$

So both MFRL and MBRL have the same sample efficiency as they both have to traverse the whole state-action space!

[1] The actual amount of samples required is more evolved and an active are of research.

Continuous Linear System (LQR) For a continuous linear LQR system with state dimensionality n_s and action dimensionality n_a and time horizon T, the *minimum* amount of samples required to solve the MDP are:

- for model-free RL: $\mathcal{O}\big((n_s+1)\cdot n_a \cdot T\big)$ [optimal trajectory]
- for model-based RL: $\mathcal{O}\big(n_s \cdot (n_s+n_a+1)\big)$ [any trajectory]

As usually $T \gg n_s$, MBRL is considerable more sample-efficient than MFRL for the LQR case as the amount of samples needed is independent from the time horizon!

9.2. Models in Reinforcement Learning

To use a model in RL, there has to be useful domain knowledge. This is available in a variety of settings, for example in well understood physics-based problems (e.g. it lots of robotics applications), for "smooth" dynamical systems where Gaussian processes and neural networks can extrapolate locally, or games with simple transition rules, e.g. using Monte Carlo tree search for Go. Some shallow domain knowledge can also be used to encode symmetries in feature transformations, e.g. using sine features for angles to encode the "full rotation" symmetry. But there are also various settings where no or little domain knowledge is available: e.g. most discrete MDPs that have arbitrary structure and complex, stochastic, partially observed environments (Atari games from pixels, table tennis with low-frequency tracking, …).

Problems in MBRL include that a performance bias might get introduced by modeling errors or even the optimizer itself. Modeling errors are especially bad as it is never possible to remove them completely. The famous quote "All models are wrong, but some are useful." (George E. P. Box) is no less true in MBRL: There might be models that are useful for reinforcement learning, but they always have the problem of being wrong. If the optimizer gets to exploit these errors, the resulting policy will not work on a real system! This directly leads to the second source of performance biases: problematic optimizers. Issues in the optimizer are mainly produced by two problems: local optima and numerical stability. The following sections will address these problems separately.

9.2.1. Local Optima and Sample-Based Methods

In section 5.2, differential dynamic programming (DDP) was discussed. This is a method of dynamic programming that used linearizations to find an optimal policy. But these linearizations are susceptible to local optima! A common method for escaping local optima are sample-based methods of which some are discussed in the following sections. But they have the major problem of not scaling well to high-dimensional search (state-action) spaces which makes them infeasible for most problems. Some of these scaling issues can be overcome by combining the method with replanning which will be covered in section 9.4.

The Cross-Entropy Method (CEM)

The *cross-entropy method* is a relatively simple reinforcement learning method that works similar to evolutionary algorithms: first sample parameters $\boldsymbol{\theta}_{k\,k=1,\cdots,K}$ from a parametric distribution $p_{\boldsymbol{\omega}}(\boldsymbol{\theta})$ and evaluate them somehow (e.g. by generating a corresponding trajectory and calculating the total reward) using an objective $J(\boldsymbol{\theta})$. Then select the best K_{top} samples and fit the distribution parameters $\boldsymbol{\omega}$ accordingly (e.g. using a maximum likelihood estimator). This converges to the optimal parameters $\boldsymbol{\omega}^*$ parameters for the parameter distribution. For (optimal) control, the parameters are a sequence of actions. A procedural description for a Gaussian distribution $p_{\boldsymbol{\omega}}(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta} \,|\, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ with $\boldsymbol{\omega} = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$ is shown in algorithm 6.

The CEM can also be extended for using a more complex policy, e.g. a neural network. Then the parameters are updated using gradient descent instead of directly fitting the new data. To avoid overfitting, only a single GD step is done per iteration.

Algorithm 6: Cross-Entropy Method for Gaussian Parameter Distribution

$\boldsymbol{\mu} \leftarrow \mathbf{0},\ \boldsymbol{\Sigma} \leftarrow \boldsymbol{I}$ // Exemplary initialization.
repeat
  $\boldsymbol{\theta}_{k=1,\cdots,K} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ // Sample parameters.
  $J_k \leftarrow J(\boldsymbol{\theta}_k)$ // Evaluate parameters for $k = 1, \cdots, K$
  $\boldsymbol{\theta}_{(k)=(1),\cdots,(K)} \leftarrow \text{sort}_J\ \boldsymbol{\theta}_{k=1,\cdots,K}$ // Sort parameters w.r.t. the performance.
  // Fit new distribution parameters:
  $\boldsymbol{\mu}^{\text{new}} \leftarrow K_{\text{top}}^{-1} \sum_{k=1}^{K_{\text{top}}} \boldsymbol{\theta}_{(k)}$
  $\boldsymbol{\Sigma}^{\text{new}} \leftarrow K_{\text{top}}^{-1} \sum_{k=1}^{K_{\text{top}}} (\boldsymbol{\theta}_k - \boldsymbol{\mu}^{\text{new}})(\boldsymbol{\theta}_k - \boldsymbol{\mu}^{\text{new}})^T$
until convergence of $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$

(Model-Predictive) Path Integral Control (MPPI)

Path integral control is a control-as-inference method initially derived for continuous time. For discrete time it is really similar to CEM: first sample a control sequence from a prior distribution, then compute a rollout of the dynamics and compute the "likelihood". Afterwards the parameters can be updated in a weighted fashion.

9.2.2. Numerical Sensitivity

Previously, the optimal control problem

$$\begin{aligned} &\max_{\boldsymbol{a}_{1:T-1}} \sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t) \\ &\text{s.t.} \quad \boldsymbol{s}_{t+1} = \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{a}_t) \end{aligned}$$

with $r(\boldsymbol{s}_T, \boldsymbol{a}_T) \coloneqq r(\boldsymbol{s}_T)$ was assessed from a dynamic programming perspective (DDP). Other techniques are for example *backpropagation-through-time* or *sequential dynamic programming* (check out the optimization of static and dynamical systems summary[2] for more details on SQP methods).

In backpropagation-through-time, the trajectory is seen as a computation graph

$$\boldsymbol{s}_2 = \boldsymbol{f}(\boldsymbol{s}_1, \boldsymbol{a}_1) \quad \Longrightarrow \quad \boldsymbol{s}_3 = \boldsymbol{f}(\boldsymbol{s}_2, \boldsymbol{a}_2) = \boldsymbol{f}\big(\boldsymbol{f}(\boldsymbol{s}_1, \boldsymbol{a}_1), \boldsymbol{a}_2\big) \quad \Longrightarrow \quad \cdots$$

with the objective $J = \sum_{t=0}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t)$. To optimize the objective, it is possible to take the derivative w.r.t. the actions through time. For example for $\boldsymbol{a}_1$:

$$\frac{\partial J}{\partial \boldsymbol{a}_1} = \frac{\partial r}{\partial \boldsymbol{a}_1} + \left(\frac{\partial r}{\partial \boldsymbol{s}_2} + \frac{\partial r}{\partial \boldsymbol{s}_3} \frac{\partial \boldsymbol{s}_3}{\partial \boldsymbol{s}_2} + \cdots + \frac{\partial r}{\partial \boldsymbol{s}_T} \frac{\partial \boldsymbol{s}_T}{\partial \boldsymbol{s}_{T-1}} \cdots \frac{\partial \boldsymbol{s}_3}{\partial \boldsymbol{s}_2} \frac{\partial \boldsymbol{s}_2}{\partial \boldsymbol{s}_1} \right) \frac{\partial \boldsymbol{s}_2}{\partial \boldsymbol{a}_1}$$

But this gradient is very poorly conditioned as very small changed in $\boldsymbol{a}_1$ have large effects in $\boldsymbol{s}_t$ if t is large!

[2] https://projects.frisp.org/documents/32

9.3. Optimism, Pessimism and Uncertainty in Reinforcement Learning

In reinforcement learning, *optimism* is similar to *greediness*, describing for aggressive the algorithm pursues the optimal solution compared to exploring the environment more. For model-based RL, too much optimism leads to exploitation of modeling errors that benefit control, for example energy generation if a friction parameter is negative and constraint violation (e.g. solving a maze by walking through walls). Such exploitation may have catastrophic consequences, e.g. robots that accelerate into walls. This issue of algorithms being too optimistic has to be resolved for practical methods. Three main approaches are currently used to tackle this:

- Inductive biases to impose additional structure, e.g. energy conservation.
- Regularization to dissuade the optimizer from exploiting modeling errors.
- Replanning over short horizons to constantly adapt to the state at additional computational costs.

Another method for dealing with too optimistic models is to "automatically" regularize the policy by incorporating uncertainty into the model. There are two types of uncertainty:

Aleatoric Inherent statistical uncertainty, e.g. additive noise like in the LGDS.

Epistemic Model uncertainty of a Bayesian model arising from a lack of data to estimate the parameters, e.g. when putting a distribution on the state dynamics matrix of a LGDS.

9.3.1. Optimism Under Uncertainty

Adding uncertainty to the model has the effect of softening the reward for uncertain states. For example in a Gaussian setting with the reward $r(x) = -x^2$ where $x \sim \mathcal{N}(\mu, \sigma^2)$, the expected reward include the variance: $\mathbb{E}\big[r(x)\big] = -(\mu^2 + \sigma^2)$. Therefore the higher the (epistemic) uncertainty, the lower the reward. This is the idea behind Bayesian dynamics models: the epistemic uncertainty should anticipate prediction errors. For Bayesian linear regression models, including Gaussian processes, the decomposition of the uncertainty into aleatoric and epistemic uncertainty is obvious. Take for example Gaussian Bayesian linear regression with the features $\boldsymbol{\phi}(\boldsymbol{x})$:

$$
\begin{aligned}
y(\boldsymbol{x}) &= \boldsymbol{\phi}^T(\boldsymbol{x})\boldsymbol{w} + \boldsymbol{\eta}, \quad \boldsymbol{w} \sim \mathcal{N}(\boldsymbol{\mu}_w, \boldsymbol{\Sigma}_w), \quad \boldsymbol{\eta} \sim \mathcal{N}(\mathbf{0}, \sigma_\eta^2 \boldsymbol{I}) \\
\mu_y(\boldsymbol{x}) &= \boldsymbol{\phi}^T(\boldsymbol{x})\boldsymbol{\mu}_w \\
\sigma_y^2 &= \underbrace{\sigma_\eta^2}_{\text{aleatoric}} + \underbrace{\sigma_\eta^2 \boldsymbol{\phi}^T(\boldsymbol{x})\boldsymbol{\Sigma}_w\boldsymbol{\phi}(\boldsymbol{x})}_{\text{epistemic}}
\end{aligned}
$$

Some other approaches are:

- *Bayesian Neural Networks:* Require approximate inference which limits theirs performance. They are an active area of research!
- *Neural Linear Models:* Bayesian linear regression with neural features. These are trained to maximize the marginal likelihood, so they overfit the data and uncertainty quantification is poor outside of data.
- *Variational Inference:* The predictive distribution is a sample estimate generated by sampling the network weights from a variational weight distribution. Mean field variational inference (MFVI) methods scale better than Markov chain Monte-Carlo (MCMC) methods, but the inference is less accurate.

- *Ensembles:* Several neural networks, each trained with different mini-batches. The predictive distribution is computed over all the models. While these are popular, how "Bayesian" this method really is is questionable.

9.4. Replanning

For replanning, the optimization problem for a single action $\boldsymbol{a}_t$ takes the next T time steps into account[3]:

$$\begin{aligned} \boldsymbol{a}_t^* = \arg\max_{\boldsymbol{a}_{t:t+T}} & \sum_{\tau=t}^{t+T} r(\boldsymbol{s}_\tau, \boldsymbol{a}_\tau) \\ \text{s.t.} \quad & \boldsymbol{s}_{t+1} = \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{a}_t) \end{aligned}$$

This is known as *Model-Predictive Control* (MPC) or *Receding Horizon Control* (RHC).

A huge benefit of replanning is that the controllers can react to disturbances and model errors. But doing the replanning is really costly and limits real-time execution. Also often a good terminal reward function is needed to prevent overly greedy planning.

9.5. Case Studies

9.5.1. Probabilistic Learning for Control (PILCO)

The PILCO algorithm uses a Gaussian process for the dynamics model with decent sample efficiency and good uncertainty quantification. The policy is represented using e.g. an RBF network and a backpropagation-through-time algorithm is used for computing the gradients. The key idea is that the epistemic uncertainty of the dynamics model regularizes the expected reward. But as backpropagation-through-time is used, the algorithm is a little brittle for large planning horizons, so only small T may be used.

9.5.2. Guided Policy Search (GPS)

9.5.3. Probabilistic Ensemble Trajectory Sampling (PETS)

PETS uses neural network ensembles with CEM MPC where instead of propagating the distributions with moment matching, particles and randomly selected models from the ensemble are propagated.

9.6. Wrap-Up

This chapter covered some topics of MBRL. The core of MBRL is to combine model learning with methods from optimal control in various ways. It is often useful if the RL task has some kind of structure that can be incorporated into the model specification, e.g. energy conservation. In practice, regularization is often needed to prevent the optimizer from exploiting beneficial modeling errors like energy generation. Popular methods include inductive biases in these models and uncertainty-based regularization using Bayesian models and replanning.

[3] The notation of arg max is abused a bit here as only the first action is used as the result of the optimization problem. But the other steps can be used as a "warm start" for the upcoming optimization problems.

For a comparison with the other discussed reinforcement learning methods (value function methods, see chapter 10 and policy search, see chapter 11), the following list gives an overview over the main benefits and drawbacks of *model-based reinforcement learning*:

Model Complexity Very High

- A forward model $\boldsymbol{f} : \mathbb{R}^{n_s+n_a} \rightarrow \mathbb{R}^{n_s}$ has to be learned.
- Dynamic programming has to be applicable (e.g. LQR).
- Small errors in the model can have a big impact on the policy.

Scalability Poor (with some positive exceptions)

- Learning high-dimensional (or discontinuous) models is very hard.

Data Efficiency Excellent

- Every transition is used to learn the model.
- Model can be reused for different tasks.

Other Limitations Distance between two policies is hard to control; huge computation times.

10. Value Function Methods

This chapter covers *value function methods*, a variant of model-free reinforcement learning. Often learning a good model is too hard, and the controller may only achieve good results by exploiting the model. Additionally, most optimal control methods are based on linearizations that only work moderately for nonlinear tasks. *Model-free reinforcement learning* does not impose any assumptions on the structure on the model but rather learns the policy directly. In most model-free RL methods, this means learning the value function to solve the optimal control problem. The value function is learned from samples[1] $\mathcal{D} = \{\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}'_i\}_{i=1}^N$ with the state $\boldsymbol{s}_i$, action $\boldsymbol{a}_i$, immediate reward r_i, and the next state $\boldsymbol{s}'_i$.

10.1. Temporal Difference Tabular Learning

For discrete states and actions, the Q-function can simply be tabulated. This is called *tabular learning*. Given a transition $(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}'_i)$, the idea of *temporal difference learning* is to use the current approximation $V(\boldsymbol{s}_i)$ and the one-step prediction of the current value,

$$\hat{V}(\boldsymbol{s}_i) \coloneqq \mathbb{E}_{\boldsymbol{a}\sim\pi(\cdot|\boldsymbol{s}_i)}\Big[r(\boldsymbol{s}_i, \boldsymbol{a}) + \mathbb{E}_{\boldsymbol{s}'\sim p(\cdot|\boldsymbol{s}_i,\boldsymbol{a})}\big[V(\boldsymbol{s}')\big]\Big] \approx r_i + \gamma V(\boldsymbol{s}'_i),$$

to update the current value function. This is done with the temporal difference (TD) error

$$\delta_i^V \coloneqq \hat{V}(\boldsymbol{s}_i) - V(\boldsymbol{s}_i) \doteq r_i + \gamma V(\boldsymbol{s}'_i) - V(\boldsymbol{s}_i) \tag{10.1}$$

by updating the value function with a "step size" $\alpha \in [0, 1]$:

$$V^{\text{new}}(\boldsymbol{s}_i) = V(\boldsymbol{s}_i) + \alpha\delta_i^V \doteq (1-\alpha)V(\boldsymbol{s}_i) + \alpha\hat{V}(\boldsymbol{s}_i) \tag{10.2}$$

The value α describes the amount of trust in the prediction. For $\alpha = 1$, the value function directly jumps to the prediction, and for $\alpha = 0$ the value function always stays the same. This part of the algorithm can be seen as the *policy evaluation* with temporal differences. To form a complete RL algorithm, also policy improvement is needed. As the algorithm has to also reach areas unseen before, exploration is needed in the policy. Two basic approaches for this problem are ϵ-greedy policies

$$\pi(\boldsymbol{a} \mid \boldsymbol{s}) = \begin{cases} \arg\max_{\boldsymbol{a}'} Q^\pi(\boldsymbol{s}, \boldsymbol{a}') & \text{with probability } 1-\epsilon \\ \text{any action } \boldsymbol{a} & \text{with probability } \epsilon \end{cases}$$

and soft-max policies, enabling also learning the exploration (with a parameter β):

$$\pi(\boldsymbol{a} \mid \boldsymbol{s}) = \frac{\exp\{\beta Q(\boldsymbol{s}, \boldsymbol{a})\}}{\sum_{\boldsymbol{a}'} \exp\{\beta Q(\boldsymbol{s}, \boldsymbol{a}')\}}$$

[1] The samples can (and should) be drawn from multiple episodes or sequences, hence the index is called i rather than t. It does not necessarily refer to the time step!

Both of these policies need a Q-function which can also be learned in a TD-fashion, similar to (10.2). Let $(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}'_i)$ again be a transition of the system. Then the TD-error for the Q-function is defined as

$$\delta_i^Q := \hat{Q}(\boldsymbol{s}_i, \boldsymbol{a}_i) - Q(\boldsymbol{s}_i) \doteq r_i + \gamma Q(\boldsymbol{s}'_i, \boldsymbol{a}_?) - Q(\boldsymbol{s}_i) \tag{10.3}$$

with the one-step prediction $\hat{Q}(\boldsymbol{s}_i) := r_i + \gamma Q(\boldsymbol{s}'_i, \boldsymbol{a}_?)$, analogous to (10.1). The update of the Q-function is then

$$Q^{\text{new}}(\boldsymbol{s}_i, \boldsymbol{a}_i) = Q(\boldsymbol{s}_i, \boldsymbol{a}_i) + \alpha \delta_i^Q \doteq (1-\alpha) Q(\boldsymbol{s}_i, \boldsymbol{a}_i) + \alpha \hat{Q}(\boldsymbol{s}_i, \boldsymbol{a}_i).$$

Other than for updating the V-function, now an action $\boldsymbol{a}_?$ has to be selected to compute the TD-error (10.3). There are two common methods for estimating $\boldsymbol{a}_?$, both forming separate algorithms:

Q-Learning $\boldsymbol{a}_? = \arg\max_{\boldsymbol{a}} Q(\boldsymbol{s}'_i, \boldsymbol{a})$

- Estimates the Q-function of the optimal policy.
- Generates off-policy samples, i.e. $\boldsymbol{a}_? \neq \boldsymbol{a}'_i$.
- Equivalent to learning the value function.
- See algorithm 7 for an algorithmic description.

SARSA $\boldsymbol{a}_? = \boldsymbol{a}'_i$, where $\boldsymbol{a}'_i \sim \pi(\cdot \mid \boldsymbol{s}'_i)$, so quintuples $(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}'_i, \boldsymbol{a}'_i)$ are used for learning

- Estimates the Q-function of the exploration policy.
- Generates on-policy samples.
- *SARSA* stands for *state-action-reward-state-action*.

Note that, as the policy depends on the Q-function, the policy is non-stationary.

Algorithm 7: Tabular Q-Learning for Discrete States

$Q^*(\boldsymbol{s}, \boldsymbol{a}) \leftarrow 0$ for all $\boldsymbol{s} \in \mathcal{S}$ and $\boldsymbol{a} \in \mathcal{A}$
$i \leftarrow 0$
repeat
  $i \leftarrow i + 1$
  if episode finished then
    Start new episode.
  Observe a transition $(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}'_i)$.
  $\boldsymbol{a}_? \leftarrow \arg\max_{\boldsymbol{a}} Q^*(\boldsymbol{s}'_i, \boldsymbol{a})$ // Compute the exploration action.
  $\hat{Q}^*(\boldsymbol{s}_i, \boldsymbol{a}_i) \leftarrow r_i + \gamma Q^*(\boldsymbol{s}'_i, \boldsymbol{a}_?)$ // Compute the one-step prediction.
  $\delta_i \leftarrow \hat{Q}^*(\boldsymbol{s}_i, \boldsymbol{a}_i) - Q^*(\boldsymbol{s}_i, \boldsymbol{a}_i)$ // Compute the TD-error.
  $Q^*(\boldsymbol{s}_i, \boldsymbol{a}_i) \leftarrow Q^*(\boldsymbol{s}_i, \boldsymbol{a}_i) + \alpha \delta_i$ // Update the Q-Function.
until convergence of Q^*

10.2. Approximate Temporal Difference Learning

If the states are continuous, it is no longer possible to tabulate them[2]. To keep it simple, use a linear function approximator for the value function:

$$V(\boldsymbol{s}) \approx V_{\boldsymbol{w}}(\boldsymbol{s}) = \boldsymbol{\phi}^T(\boldsymbol{s}) \boldsymbol{w}$$

[2] Of course discretizing is an option, but the state space becomes really big really quick.

The optimal weights $\boldsymbol{w}$ can again be found using temporal difference learning! With *bootstrapping*, that is using the old approximation values to measure the new approximation error as for tabular TD learning, the goal is to minimize the mean-squared error

$$\text{MSE}(\boldsymbol{w}) \approx \text{MSE}_{\text{BS}}(\boldsymbol{w}) = \sum_{i=1}^{N} \left(\hat{V}^{\pi}(\boldsymbol{s}_i) - V_{\boldsymbol{w}}(\boldsymbol{s}_i)\right)^2$$

for samples $\left\{(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}'_i)\right\}_{i=1}^{N}$ of $\boldsymbol{w}$ with

$$\hat{V}^{\pi}(\boldsymbol{s}_i) = \mathbb{E}_{\boldsymbol{a}\sim\pi(\cdot|\boldsymbol{s}_i)}\Big[r(\boldsymbol{s}_i, \boldsymbol{a}) + \mathbb{E}_{\boldsymbol{s}'\sim p(\cdot|\boldsymbol{s}_i,\boldsymbol{a})}\left[V_{\boldsymbol{w}_{\text{old}}}(\boldsymbol{s}')\right]\Big] \approx r_i + \gamma V_{\boldsymbol{w}_{\text{old}}}(\boldsymbol{s}'_i).$$

This objective can be optimized with (stochastic) gradient descent (see section 3.7.4) by approximating the bootstrapped objective (10.2) with a single sample $(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}'_i)$:

$$\text{MSE}_{\text{BS}}(\boldsymbol{w}) \approx \left(\hat{V}^{\pi}(\boldsymbol{s}_i) - V_{\boldsymbol{w}}(\boldsymbol{s}_i)\right)^2 \approx \left(r_i + \gamma V_{\boldsymbol{w}_{\text{old}}}(\boldsymbol{s}'_i) - V_{\boldsymbol{w}}(\boldsymbol{s}_i)\right)^2 \tag{10.4}$$

The gradient descent step is then given as

$$\begin{aligned}
\boldsymbol{w}_{i+1} &= \boldsymbol{w}_i + \tilde{\alpha}_i \frac{\partial\, \text{MSE}_{\text{BS}}}{\partial \boldsymbol{w}}\bigg|_{\boldsymbol{w}=\boldsymbol{w}_i,\, \boldsymbol{w}_{\text{old}}=\boldsymbol{w}_i} \\
&= \boldsymbol{w}_i + \alpha_i \underbrace{\left(r_i + \gamma V_{\boldsymbol{w}_i}(\boldsymbol{s}'_i) - V_{\boldsymbol{w}_i}(\boldsymbol{s}_i)\right)}_{\delta_i} \boldsymbol{\phi}(\boldsymbol{s}_i) \\
&= \boldsymbol{w}_i + \alpha_i \delta_i \boldsymbol{\phi}(\boldsymbol{s}_i)
\end{aligned}$$

for a linear approximation $V_{\boldsymbol{w}}(\boldsymbol{s}) = \boldsymbol{\phi}^T(\boldsymbol{s})\boldsymbol{w}$ and the derivative

$$\frac{\partial\, \text{MSE}_{\text{BS}}}{\partial \boldsymbol{w}} = 2\big(r_i + \gamma V_{\boldsymbol{w}_{\text{old}}}(\boldsymbol{s}'_i) - V_{\boldsymbol{w}}(\boldsymbol{s}_i)\big)\frac{\partial V_{\boldsymbol{w}}(\boldsymbol{s}_i)}{\partial \boldsymbol{w}} = 2\big(r_i + \gamma V_{\boldsymbol{w}_{\text{old}}}(\boldsymbol{s}'_i) - V_{\boldsymbol{w}}(\boldsymbol{s}_i)\big)\boldsymbol{\phi}(\boldsymbol{s}_i) \tag{10.5}$$

of the function approximator[3]. Of course this can be readily extended for deep function approximators like neural networks by replacing the derivative $\frac{\partial V_{\boldsymbol{w}}(\boldsymbol{s}_i)}{\partial \boldsymbol{w}}$ in (10.5) with e.g. an autograd engine or backpropagation. As for tabular TD methods, similar update rules can be achieved for the Q-function, enabling approximate Q-learning and SARSA with $Q_{\boldsymbol{w}}(\boldsymbol{s}, \boldsymbol{a}) = \boldsymbol{\phi}^T(\boldsymbol{s}, \boldsymbol{a})\boldsymbol{w}$:

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i + \alpha\big(r_i + \gamma Q_{\boldsymbol{w}_i}(\boldsymbol{s}'_i, \boldsymbol{a}_?) - Q_{\boldsymbol{w}_i}(\boldsymbol{s}_i, \boldsymbol{a}_i)\big)\boldsymbol{\phi}(\boldsymbol{s}_i, \boldsymbol{a}_i)$$

Note that, even though an index t is used here, it is possible (and often needed) to train over multiple episodes by resetting the environment once an episode is done. This process of resetting the environment, observing samples, and training the V-/Q-function is repeated until convergence. A procedural description of approximate Q-learning is shown in algorithm 8

Remarks on Approximate TD Learning

- When using a tabular encoding for $\boldsymbol{\phi}(\boldsymbol{s})$, the function approximation is equivalent with discretizing the state-space and using tabular TD learning methods.
- Approximate TD learning is not proper stochastic gradient descent!

[3] The multiplication with two is absorbed into $\alpha_i = 2\tilde{\alpha}_i$.

Algorithm 8: Approximate Q-Learning for Continuous States

$\boldsymbol{w}_0 \leftarrow \mathbf{0}$
$t \leftarrow 0$
repeat
  $t \leftarrow t + 1$
  if episode finished then
    Start new episode.
  Observe a transition $(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}'_i)$.
  $\boldsymbol{a}_? \leftarrow \arg\max_{\boldsymbol{a}} Q_{\boldsymbol{w}}(\boldsymbol{s}'_i, \boldsymbol{a})$ // Compute the exploration action.
  $\hat{Q}_{\boldsymbol{w}}(\boldsymbol{s}_i, \boldsymbol{a}_i) \leftarrow r_i + \gamma Q_{\boldsymbol{w}}(\boldsymbol{s}'_i, \boldsymbol{a}_?)$ // Compute the one-step prediction.
  $\delta_i \leftarrow \hat{Q}_{\boldsymbol{w}}(\boldsymbol{s}_i, \boldsymbol{a}_i) - Q_{\boldsymbol{w}}(\boldsymbol{s}_i, \boldsymbol{a}_i)$ // Compute the TD-error.
  $\boldsymbol{w}_{i+1} \leftarrow \alpha_i \delta_i \boldsymbol{\phi}(\boldsymbol{s}_i, \boldsymbol{a}_i)$ // Update the Q-Function.
until convergence of $\boldsymbol{w}_i$

  - Firstly, the target values $\hat{V}^{\pi}(\boldsymbol{s})$ change after a parameter update due to bootstrapping. This actually introduces a bias and changes the optimization objective such that not the mean-squared error is optimized anymore.
  - Secondly, the samples are not statistically independent if draw from a sequence of states and actions. Instead, a *replay memory* should be built that holds samples from multiple trajectories and in each iterative step, one sample should be drawn from the memory.

- It is possible to draw multiple samples at once and to a combined GD (mini-batch) step to improve convergence.
- Due to e.g. off-policy updates, convergence is not guaranteed every time. For linear approximations, however, certain convergence properties hold given that the convergence properties of SGD (see section 3.7.4) are fulfilled.
- Temporal difference learning is fast in computation time (i.e. fast convergence), but it is not sample-efficient as every sample is only used once!

10.3. Batch Reinforcement Learning Methods

One major problem of online temporal difference methods is that they are typically highly data-inefficient as every sample is only used once. This property can be improved by using a batch of data to increase the data-efficiency. This section will look into two exemplary algorithms,

- Least-Squares Temporal Differences (LSTD), and
- Fitted Q-Iteration,

that make use of this technique. But compared to TD methods, batch methods are usually computationally much more expensive, so there is a tradeoff here.

10.3.1. Least-Squares Temporal Differences (LSTD)

For *least-squares temporal differences*, the bootstrapped mean-squared error (10.2) is solved analytically for N samples. Let

$$\boldsymbol{r} \coloneqq \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_N \end{bmatrix} \qquad \boldsymbol{\Phi}^T \coloneqq \begin{bmatrix} \boldsymbol{\phi}^T(\boldsymbol{s}_1) \\ \boldsymbol{\phi}^T(\boldsymbol{s}_2) \\ \vdots \\ \boldsymbol{\phi}^T(\boldsymbol{s}_N) \end{bmatrix} \qquad \boldsymbol{\Phi}_+^T \coloneqq \begin{bmatrix} \boldsymbol{\phi}^T(\boldsymbol{s}'_1) \\ \boldsymbol{\phi}^T(\boldsymbol{s}'_2) \\ \vdots \\ \boldsymbol{\phi}^T(\boldsymbol{s}'_N) \end{bmatrix}$$

be the rewards, steps and next steps, respectively. Then the bootstrapped objective (10.2) can be rewritten as

$$\begin{aligned} \text{MSE}_{\text{BS}}(\boldsymbol{w}) &\approx \sum_{i=1}^{N} \left(r_i + \gamma V_{\boldsymbol{w}_{\text{old}}}(\boldsymbol{s}'_i) - V_{\boldsymbol{w}}(\boldsymbol{s}_i)\right)^2 \\ &= \sum_{i=1}^{N} \left(r_i + \gamma \boldsymbol{\phi}^T(\boldsymbol{s}'_i)\boldsymbol{w}_{\text{old}} - \boldsymbol{\phi}^T(\boldsymbol{s}_i)\boldsymbol{w}\right)^2 \\ &= \left(\boldsymbol{r} + \gamma \boldsymbol{\Phi}_+^T \boldsymbol{w}_{\text{old}} - \boldsymbol{\Phi}^T \boldsymbol{w}\right)^T \left(\boldsymbol{r} + \gamma \boldsymbol{\Phi}_+^T \boldsymbol{w}_{\text{old}} - \boldsymbol{\Phi}^T \boldsymbol{w}\right). \end{aligned}$$

By taking the derivative of the mean-squared error and setting it to zero[4], the least-squares solution is

$$\boldsymbol{w} = \left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}\left(\boldsymbol{r} + \gamma \boldsymbol{\Phi}_+^T \boldsymbol{w}_{\text{old}}\right). \tag{10.6}$$

However, this equation still contains $\boldsymbol{w}_{\text{old}}$. For convergence, $\boldsymbol{w} = \boldsymbol{w}_{\text{old}}$ should hold. Plugging this into the least-squares solution (10.6) yields the solution to the complete least-squares temporal differences algorithm:

$$\begin{aligned} & & \boldsymbol{w} &= \left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}\left(\boldsymbol{r} + \gamma \boldsymbol{\Phi}_+^T \boldsymbol{w}_{\text{old}}\right) \\ \boldsymbol{w}_{\text{old}} \overset{!}{=} \boldsymbol{w} \quad & & \boldsymbol{w} &= \left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}\left(\boldsymbol{r} + \gamma \boldsymbol{\Phi}_+^T \boldsymbol{w}\right) \\ \iff & & \boldsymbol{w} &= \left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}\boldsymbol{r} + \gamma\left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}_+^T \boldsymbol{w} \\ \iff & & \left(\boldsymbol{I} - \gamma\left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}_+^T\right)\boldsymbol{w} &= \left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}\boldsymbol{r} \\ \iff & & \left(\left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}\boldsymbol{\Phi}^T - \left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \gamma \boldsymbol{\Phi}_+^T\right)\boldsymbol{w} &= \left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}\boldsymbol{r} \\ \iff & & \left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}\left(\boldsymbol{\Phi}^T - \gamma \boldsymbol{\Phi}_+^T\right)\boldsymbol{w} &= \left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\right)^{-1} \boldsymbol{\Phi}\boldsymbol{r} \\ \iff & & \boldsymbol{\Phi}\left(\boldsymbol{\Phi}^T - \gamma \boldsymbol{\Phi}_+^T\right)\boldsymbol{w} &= \boldsymbol{\Phi}\boldsymbol{r} \\ \iff & & \boldsymbol{w} &= \left(\boldsymbol{\Phi}\left(\boldsymbol{\Phi}^T - \gamma \boldsymbol{\Phi}_+^T\right)\right)^{-1} \boldsymbol{\Phi}\boldsymbol{r} \end{aligned} \tag{10.7}$$

Equation (10.7) is the solution of LSTD. By analyzing the convergence properties of TD-learning it can be seen that this is the same solution as the fixed point of TD-learning. Hence, with enough samples, it is possible to calculate the V-function approximation weights in one shot without iterations. As for TD learning, it is also possible to adapt this for learning the Q-function by altering the matrices $\boldsymbol{\Phi}$ and $\boldsymbol{\Phi}_+$ accordingly. The algorithm is then called least-squares Q-learning (LSQ) and is used for least-squares policy iteration (LSPI).

However, as opposed to TD learning, LSTD is computationally much more expensive!

[4] $-\frac{1}{2}\frac{\partial}{\partial \boldsymbol{w}}\left\|\boldsymbol{r} + \gamma \boldsymbol{\Phi}_+^T \boldsymbol{w}_{\text{old}} - \boldsymbol{\Phi}_+^T \boldsymbol{w}\right\|_2^2 = \boldsymbol{\Phi}\left(\boldsymbol{r} + \gamma \boldsymbol{\Phi}_+^T \boldsymbol{w}_{\text{old}} - \boldsymbol{\Phi}^T \boldsymbol{w}\right) = \boldsymbol{\Phi}\left(\boldsymbol{r} + \gamma \boldsymbol{\Phi}_+^T \boldsymbol{w}_{\text{old}}\right) - \boldsymbol{\Phi}\boldsymbol{\Phi}^T \boldsymbol{w} \overset{!}{=} \boldsymbol{0}$

10.3.2. Fitted Q-Iteration

When using batch RL it is also easier to use nonlinear function approximators as many of them only work in a batch setup, e.g. regression trees. Also using just a single sample could lead to catastrophic forgetting, e.g. for neural networks that do not "remember" the states seen previously. *Fitted Q-iteration* is a method that uses some nonlinear regression model Regress$(\boldsymbol{X}, \boldsymbol{y})$ with input data $\boldsymbol{X}$ and target values $\boldsymbol{y}$ for approximate value iteration. A procedural representation is shown in algorithm 9. Invoking the regression model performs the required expectation for the Q-function implicitly. But is it still hard to solve the maximization operator on line 4 for continuous actions. See for example Neumann and Peters: "Fitted Q-Iteration by Advantage Weighted Regression", NIPS (2008).

Algorithm 9: Fitted Q-Iteration with Regression Model Regress$(\boldsymbol{X}, \boldsymbol{y})$

Input: Dataset $\mathcal{D} = \left\{(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}'_i)\right\}_{i=1}^{N}$

1 $\quad \boldsymbol{X} \leftarrow \begin{bmatrix} \boldsymbol{s}_1 & \boldsymbol{s}_2 & \cdots & \boldsymbol{s}_N \\ \boldsymbol{a}_1 & \boldsymbol{a}_2 & \cdots & \boldsymbol{a}_N \end{bmatrix}^T$

2 $Q(\boldsymbol{s}, \boldsymbol{a}) \leftarrow 0$ for all $\boldsymbol{s} \in \mathcal{S}$ and $\boldsymbol{a} \in \mathcal{A}$

3 repeat

4 $\quad q_i \leftarrow r_i + \gamma \max_{\boldsymbol{a}'} Q(\boldsymbol{s}'_i, \boldsymbol{a}')$ // Generate target values for each $i = 1, \cdots, N$.

5 $\quad Q(\boldsymbol{s}, \boldsymbol{a}) \leftarrow$ Regress$(\boldsymbol{X}, \boldsymbol{q})$ // Learn new Q-Function.

6 until convergence of Q

10.4. Wrap-Up

This chapter covered value function methods with have been the driving reinforcement learning approach in the 90s. They can be applied to lots of problems, e.g. chess at professional level, robot soccer, etc. However, they are not always the method of choice as the state-action space has to be filled up with a sufficient amount of samples (the amount of data needed grows exponentially with the dimension) and errors in the value function approximation can cause drastic failures that can be very hard to control.

For a comparison with the other discussed reinforcement learning methods (model-based, see chapter 9 and policy search, see chapter 11), the following list gives an overview over the main benefits and drawbacks of *value function methods*:

Model Complexity OK

- Only the Q-function $Q : \mathbb{R}^{|\mathcal{S}|+|\mathcal{A}|} \rightarrow \mathbb{R}$ is learned.
- But small function approximation errors can have a big effect on the policy.

Scalability Poor (with some positive exceptions)

- Approximating the Q-function in high dimensions is difficult.
- Policy is hard to obtain in high-dimensional action spaces due to the maximization.

Data Efficiency OK (online TD learning) to good (batch methods)

- Online: every transition is only used once.
- Batch: every transition is reused.

Other Limitations Policy update is unbounded, might lead to oscillations.

11. Policy Search Methods

This chapter covers *policy search methods* which do not use value functions or similar as a intermediate step for finding an optimal policy but rather search the policy directly. This is motivated by the problems that arise with value function methods as they do not handle continuous actions well (due to the required maximization of the Q-function) and the amount of samples needed grows exponentially with the dimensionality of the state-space. Also, an error in the value function propagates and arbitrarily distorts the policy update. Many of these problems can be fixed by using parametric policies and policy search, i.e. finding the policy directly instead of using a value function. The policies are improved upon demonstrations and it is even possible to use task-appropriate policies or to train on the real system.

11.1. Categorization of Policy Search

In value-based methods, the action to execute is often chosen using an ϵ-greedy of soft-max approach, both requiring to either maximize w.r.t. or sum over the actions, both of which is difficult for continuous state spaces. Policy search methods instead use parameterized policies for action selection, for example Gaussian policies

$$\pi(\boldsymbol{a} \mid \boldsymbol{s}; \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{a} \mid \boldsymbol{f}_{\boldsymbol{w}}(\boldsymbol{s}),\, \sigma^2 \boldsymbol{I}), \quad \boldsymbol{\theta} = \{\boldsymbol{w}, \sigma^2\}$$

with a learnable mean function $\boldsymbol{f}_{\boldsymbol{w}}(\boldsymbol{s})$. The problem if policy search not is to find parameters $\boldsymbol{\theta}$ that constitute a good policy. Model-free approaches can be broken down into three main steps:

1. Exploration: Sample trajectories $\boldsymbol{\tau}^{[i]}$ following the current policy π_k.
2. Policy Evaluation: Asses the quality of the trajectories, either *step-based* or *episode-based*.
3. Policy Improvement: Compute the next policy π_{k+1} from the trajectories and evaluations.
4. If not converged, go to 1.

There are two types of policy evaluation: *step-based* and *episode-based*, which will be explained further in the following sections.

11.1.1. Episode-Based Policy Search

In *episode-based* policy evaluation, the quality of parameters $\boldsymbol{\theta}^{[i]}$ is directly assessed using the (undiscounted) total reward

$$R^{[i]} = \sum_{t=1}^{T} r_t^{[i]}.$$

The data for the policy update is

$$\mathcal{D} = \left\{ \boldsymbol{\theta}^{[i]}, R^{[i]} \right\}_{i=1,\cdots,N}.$$

This but causes a high variance in $R^{[i]}$ as it is the sum of T random variables, but works well for a moderate amount of parameters. Other properties of the episode-based strategy are:

- Explores in the parameter space (the data to learn from contains the parameters).
- Allows more sophisticated exploration strategies.
- Often efficient for a small amount of parameters.
- Generalization and multi-task learning, e.g. for policies like DMPs.
- Structure-less black-box optimization of the policy.

For episodic policy search, an upper-level policy $\pi(\boldsymbol{\theta};\boldsymbol{\omega})$ over the parameters of the low-level control policy $\pi(\boldsymbol{a} \mid \boldsymbol{s};\boldsymbol{\theta})$ is learned. This can, for example, be a Gaussian $\pi(\boldsymbol{\theta};\boldsymbol{\omega}) = \mathcal{N}(\boldsymbol{\theta} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$ with the parameters $\boldsymbol{\omega} = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$. To reduce the variance in the returns, the upper-level policy is often chosen to be deterministic. Then a policy $\pi(\boldsymbol{\theta};\boldsymbol{\omega})$ is searched that maximizes the expected reward

$$J(\boldsymbol{\omega}) = \mathbb{E}_{\pi(\boldsymbol{\theta};\boldsymbol{\omega})}\big[R_{\boldsymbol{\theta}}\big] = \int \pi(\boldsymbol{\theta};\boldsymbol{\omega}) R_{\boldsymbol{\theta}} \,\mathrm{d}\boldsymbol{\theta} \tag{11.1}$$

with the expected long-term reward

$$R_{\boldsymbol{\theta}} = \mathbb{E}_{p,\pi}\left[\sum_{t=1}^{T} r_t \,\middle|\, \boldsymbol{\theta}\right].$$

The exploration is encoded in the upper-level policy $\pi(\boldsymbol{\theta};\boldsymbol{\omega})$ from which parameters are sampled to generate the trajectories in the exploration step of policy search.

11.1.2. Step-Based Policy Search

In *step-based* policy evaluation, the quality of single state-action pairs is assessed using the (undiscounted) reward to come

$$Q_t^{[i]} = \sum_{\tau=r}^{T} r_\tau^{[i]}. \tag{11.2}$$

The data for the policy update is

$$\mathcal{D} = \left\{ \boldsymbol{s}_t^{[i]}, \boldsymbol{a}_t^{[i]}, Q_t^{[i]} \right\}_{t=1,\cdots,T}^{i=1,\cdots,N}.$$

This causes less variance in $Q_t^{[i]}$ as it is the sum of only $T - t + 1$ random variables. Other properties of the step-based strategy are:

- Explores in the action space (the data to learn from contains the actions).
- Less variance in the quality assessment compared to episode-based evaluation.
- More data points to fit the policy (one for each time step and episode).
- Less likely to create unstable policies.
- Uses the structure of the RL problem.
- Decomposed into single time steps.

For step-based policy search, the lower-level policy $\pi(\boldsymbol{a} \mid \boldsymbol{s}; \boldsymbol{\theta})$ is learned directly. This is done by maximizing the expected reward

$$J(\boldsymbol{\theta}) = \mathbb{E}_{p(\boldsymbol{\tau};\boldsymbol{\theta})}\big[R_{\boldsymbol{\tau}}\big] = \int p(\boldsymbol{\tau};\boldsymbol{\theta}) R_{\boldsymbol{\tau}} \,\mathrm{d}\boldsymbol{\tau} \tag{11.3}$$

with the trajectory distribution

$$p(\boldsymbol{\tau};\boldsymbol{\theta}) = p(\boldsymbol{s}_1) \prod_{t=1}^{T-1} p(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t) \pi(\boldsymbol{a}_t \mid \boldsymbol{s}_t; \boldsymbol{\theta}) \tag{11.4}$$

and the return for a single trajectory $R_{\boldsymbol{\tau}} = \sum_{t=1}^{T} r_t$. The exploration has to be encoded directly into the lower-level policy $\pi(\boldsymbol{a} \mid \boldsymbol{s}; \boldsymbol{\theta})$ in contrast to episode-based policy search.

11.1.3. Exploration vs. Exploitation

The exploration vs. exploitation tradeoff describes the tradeoff between continuing to explore in the next iteration (exploration) and maximizing the reward on the existing samples (exploitation). The fundamental question for policy search is how to control this tradeoff. This requires a quantification of the difference of two policies which will be discussed in more detail in subsection 11.2.2. Usually this leads to limiting the distance between two subsequent policies for the policy update as greedy updates

$$\boldsymbol{\theta}^{\text{new}} = \arg\max_{\tilde{\boldsymbol{\theta}}} \mathbb{E}_{p,\pi_{\tilde{\boldsymbol{\theta}}}}\big[Q^{\pi}(\boldsymbol{s}, \boldsymbol{a})\big]$$

will usually lead to an unstable learning process with large jumps.

11.2. Policy Gradient Methods

This section covers *policy gradient methods*, a class of policy search methods that use the gradient of the expected long-term reward w.r.t. parameters for updating the policy. The first subsection focuses on plain gradient methods and the second covers more sophisticated methods using relative entropy and natural gradients.

11.2.1. Policy Gradients

The basic idea of policy gradients is to optimize the policy in a gradient descent-fashion using an estimate of the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ of the expected long-term reward. This splits policy gradient methods into two steps: computing the gradient and subsequently improving the policy.

Gradient Computation

Computing (and approximating) the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is an important part of policy gradient methods. This section covers two methods of computing the gradient: finite differences and likelihood ratio gradients. Additionally, baselines are introduces that reduce the variance in the gradient to improve learning.

Finite Differences Finite difference gradient approximation is the most straightforward way of approximating the derivative using a first-order Taylor approximation around $\boldsymbol{\theta}$:

$$
\begin{aligned}
& J(\boldsymbol{\theta} + \boldsymbol{\delta}_\theta) \approx J(\boldsymbol{\theta}) + \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \boldsymbol{\delta}_\theta \\
\iff \quad & \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \boldsymbol{\delta}_\theta \approx J(\boldsymbol{\theta} + \boldsymbol{\delta}_\theta) - J(\boldsymbol{\theta}) \\
\iff \quad & \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \boldsymbol{\delta}_\theta \boldsymbol{\delta}_\theta^T \approx \boldsymbol{\delta}_\theta^T \big(J(\boldsymbol{\theta} + \boldsymbol{\delta}_\theta) - J(\boldsymbol{\theta}) \big) \\
\iff \quad & \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \approx \boldsymbol{\delta}_\theta^T \big(\boldsymbol{\delta}_\theta \boldsymbol{\delta}_\theta^T \big)^{-1} \big(J(\boldsymbol{\theta} + \boldsymbol{\delta}_\theta) - J(\boldsymbol{\theta}) \big)
\end{aligned}
$$

This can be used to update a single parameter estimate, e.g. the mean. A large class of algorithms uses this technique: Kiefer-Wolfowitz Procedure, Robbins-Monroe, Simultaneous Perturbation Stochastic Approximation (SPSA), Random Search, …

Episode-Based Likelihood Ratio Gradient For an upper-level policy $\pi(\boldsymbol{\theta}; \boldsymbol{\omega})$ and the expected long-term reward $R_{\boldsymbol{\theta}}$ for a single $\boldsymbol{\theta}$, it is possible to approximate the gradient of the expected reward (11.1) using the log-ratio trick. That is, for an arbitrary function $f(\boldsymbol{x})$, the following holds:

$$
\nabla_{\boldsymbol{x}} \ln f(\boldsymbol{x}) = \frac{1}{f(\boldsymbol{x})} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \quad \implies \quad \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = f(\boldsymbol{x}) \nabla_{\boldsymbol{x}} \ln f(\boldsymbol{x})
$$

BY applied this to the gradient of the expected return, the gradient turns into an expectation that can be approximated using Monte-Carlo integration

$$
\begin{aligned}
\nabla_{\boldsymbol{\omega}} J(\boldsymbol{\omega}) &= \nabla_{\boldsymbol{\omega}} \int \pi(\boldsymbol{\theta}; \boldsymbol{\omega}) R_{\boldsymbol{\theta}} \, \mathrm{d}\boldsymbol{\theta} = \int \nabla_{\boldsymbol{\omega}} \pi(\boldsymbol{\theta}; \boldsymbol{\omega}) R_{\boldsymbol{\theta}} \, \mathrm{d}\boldsymbol{\theta} \\
&= \int \pi(\boldsymbol{\theta}; \boldsymbol{\omega}) R_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\omega}} \ln \pi(\boldsymbol{\theta}; \boldsymbol{\omega}) \, \mathrm{d}\boldsymbol{\theta} \\
&\approx \frac{1}{N} \sum_{i=1}^{N} R^{[i]} \, \nabla_{\boldsymbol{\omega}} \ln \pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}\big) =: \nabla_{\boldsymbol{\omega}}^{\mathrm{MC}} J(\boldsymbol{\omega})
\end{aligned} \tag{11.5}
$$

with N samples $\boldsymbol{\theta}^{[i]} \sim \pi(\cdot; \boldsymbol{\omega})$ and the respective rewards $R^{[i]} := R_{\boldsymbol{\theta}^{[i]}}$. This gradient is called the *Parameter Exploring Policy Gradient* (PGPE). The log-ratio trick is needed and Monte-Carlo estimation can not directly be applied to the expectation as that would require differentiation of the sampling which is not possible.

Step-Based Likelihood Ratio Policy Gradient As mentioned in subsection 11.1.1, the total undiscounted reward $R_{\boldsymbol{\theta}}$ in episode-based evaluation exhibits a very high variance. This variance can be reduced by using step-based evaluation and the reward to come $Q_t^{[i]}$ as defined in (11.2) instead. This requires maximizing the expected reward $J(\boldsymbol{\theta})$ as defined in (11.3) instead as step-based policy gradient methods to not work with upper- and lower-level policies but rather optimize the policy $\pi(\boldsymbol{a} \,|\, \boldsymbol{s}; \boldsymbol{\theta})$ directly. By again using the log-ratio trick, the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ can be approximated as

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \int p(\boldsymbol{\tau}; \boldsymbol{\theta}) R_{\boldsymbol{\tau}} \, \mathrm{d}\boldsymbol{\tau} = \int \nabla_{\boldsymbol{\theta}} p(\boldsymbol{\tau}; \boldsymbol{\theta}) R_{\boldsymbol{\tau}} \, \mathrm{d}\boldsymbol{\tau} \\
&= \int p(\boldsymbol{\tau}; \boldsymbol{\theta}) R_{\boldsymbol{\tau}} \nabla_{\boldsymbol{\theta}} \ln p(\boldsymbol{\tau}; \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\tau} \\
&\approx \frac{1}{N} \sum_{i=1}^{N} R^{[i]} \nabla_{\boldsymbol{\theta}} \ln p\big(\boldsymbol{\tau}^{[i]}; \boldsymbol{\theta}\big) =: \nabla_{\boldsymbol{\omega}}^{\mathrm{MC}} J(\boldsymbol{\omega})
\end{aligned} \tag{11.6}
$$

with the trajectory distribution $p(\boldsymbol{\tau};\boldsymbol{\theta})$ and N samples $\boldsymbol{\tau}^{[i]} \sim p(\cdot;\boldsymbol{\theta})$ and the respective rewards $R^{[i]} \coloneqq R_{\boldsymbol{\tau}^{[i]}}$. What remains is to calculate the gradient of logarithm of the trajectory distribution. Applying the logarithm to the distribution (11.4) yields

$$\ln p(\boldsymbol{\tau};\boldsymbol{\theta}) = \sum_{t=1}^{T-1} \ln \pi(\boldsymbol{a}_t \mid \boldsymbol{s}_t;\boldsymbol{\theta}) + \underbrace{\ln p(\boldsymbol{s}_1) + \sum_{t=1}^{T-1} \ln p(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t)}_{\text{Constant}}$$

where the last terms are constant w.r.t. $\boldsymbol{\theta}$. Hence, these terms vanish when taking the gradient w.r.t. $\boldsymbol{\theta}$. Plugging this result into the Monte-Carlo approximation (11.6) yields

$$\nabla_{\boldsymbol{\omega}}^{\text{MC}} J(\boldsymbol{\omega}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T-1} R^{[i]} \nabla_{\boldsymbol{\theta}} \ln \pi\big(\boldsymbol{a}_t^{[i]} \,\big|\, \boldsymbol{s}_t^{[i]};\boldsymbol{\theta}\big)$$

which is called the REINFORCE algorithm. But this still used the total rewards with high variance! As the rewards in the past are not correlated with the actions in the future, i.e.

$$\mathbb{E}_{p(\boldsymbol{\tau})}\big[r_t \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}_h \mid \boldsymbol{s}_h;\boldsymbol{\theta})\big] \overset{(1)}{=} \mathbb{E}_{p(\boldsymbol{\tau})}[r_t] \underbrace{\mathbb{E}_{p(\boldsymbol{\tau};\boldsymbol{\theta})}\big[\nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a}_h \mid \boldsymbol{s}_h;\boldsymbol{\theta})\big]}_{= \nabla_{\boldsymbol{\theta}} \mathbb{E}_{p(\boldsymbol{\tau};\boldsymbol{\theta})}[1] = \nabla_{\boldsymbol{\theta}} 1 = 0} = 0$$

holds for all $h > t$, replacing the total reward with the reward to come only scales the gradient. This is called the *policy gradient theorem*. The result gradient then is

$$\nabla_{\boldsymbol{\omega}}^{\text{MC}} J(\boldsymbol{\omega}) \propto \nabla_{\boldsymbol{\theta}}^{\text{PG}} J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T-1} Q_t^{[i]} \nabla_{\boldsymbol{\theta}} \ln \pi\big(\boldsymbol{a}_t^{[i]} \,\big|\, \boldsymbol{s}_t^{[i]};\boldsymbol{\theta}\big).$$

Baselines A big problem with likelihood ratio policy gradients is that they exhibit a high variance. But this variance can be reduced by subtracting a baseline b_i from the reward:

$$\nabla_{\boldsymbol{\omega}}^{\text{MC}} J(\boldsymbol{\omega}) = \frac{1}{N} \sum_{i=1}^{N} \big(R^{[i]} - b^{[i]}\big) \nabla_{\boldsymbol{\omega}} \ln \pi\big(\boldsymbol{\theta}^{[i]};\boldsymbol{\omega}\big) \tag{11.7}$$

If the baseline is good, this trick can lower the variance of the estimate. It is still an unbiased estimator as the expectation of the baseline b vanishes:

$$\begin{aligned}\mathbb{E}_{p(\boldsymbol{\theta};\boldsymbol{\omega})}\big[\nabla_{\boldsymbol{\omega}} \ln p(\boldsymbol{\theta};\boldsymbol{\omega}) b\big] &= \int p(\boldsymbol{\theta};\boldsymbol{\omega}) \nabla_{\boldsymbol{\omega}} \ln p(\boldsymbol{\theta};\boldsymbol{\omega}) b \,\mathrm{d}\boldsymbol{\theta} = b \int p(\boldsymbol{\theta};\boldsymbol{\omega}) \nabla_{\boldsymbol{\omega}} \ln p(\boldsymbol{\theta};\boldsymbol{\omega}) \,\mathrm{d}\boldsymbol{\theta} \\ &= b \int \nabla_{\boldsymbol{\omega}} p(\boldsymbol{\theta};\boldsymbol{\omega}) \,\mathrm{d}\boldsymbol{\theta} = b \nabla_{\boldsymbol{\omega}} \int p(\boldsymbol{\theta};\boldsymbol{\omega}) \,\mathrm{d}\boldsymbol{\theta} = b \nabla_{\boldsymbol{\omega}} 1 = b \cdot 0 = 0\end{aligned}$$

A usually good baseline is the average reward, but there also exists an optimal baseline for each algorithm that minimizes the variance of the gradient estimate. The principle of subtracting baselines can analogously be applied to step-based gradients

$$\nabla_{\boldsymbol{\theta}}^{\text{PG}} J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T-1} \big(Q_t^{[i]} - b_i\big(\boldsymbol{s}_t^{[i]}\big)\big) \nabla_{\boldsymbol{\theta}} \ln \pi\big(\boldsymbol{a}_t^{[i]} \mid \boldsymbol{s}_t^{[i]};\boldsymbol{\theta}\big).$$

where the baseline might even vary with the state. As long as the baseline does not change with the action, the expectation vanishes and the estimator is unbiased.

[1] As r_t and $\boldsymbol{a}_h$ are independent for $h > t$.

11.2.2. Step Sizes, Metrics, Relative Entropy and Natural Gradient

An important factor when using policy gradient methods is the step size α in the policy update

$$\boldsymbol{\omega}_{k+1} = \boldsymbol{\omega}_k + \alpha \nabla_{\boldsymbol{\omega}} J(\boldsymbol{\omega}) \qquad\qquad \boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

for episode- or step-based evaluation, respectively. The simplest (naive) idea is to constrain the policy update using a distance metric in the parameter space, for example the Euclidean distance $L_2(\pi_{k+1}, \pi_k)$ defined as

$$L_2(\pi_{k+1}, \pi_k) \coloneqq \|\boldsymbol{\omega}_{k+1} - \boldsymbol{\omega}_k\| \qquad\qquad L_2(\pi_{k+1}, \pi_k) \coloneqq \|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k\|$$

for the episode- and step-based evaluation, respectively. With a constraint

$$L_2(\pi_{k+1}, \pi_k) \leq \epsilon$$

on the policy update, the step size is given as $\alpha_k = \|\nabla J\|^{-1}\epsilon$. But the Euclidean metric has the big problem if not being invariant to scaling of the variables! Take for example the policy $\pi(a \,|\, \boldsymbol{s}; \boldsymbol{\theta}) = \mathcal{N}(a \,|\, \boldsymbol{s}^T\boldsymbol{\theta}, \sigma^2)$ with $\boldsymbol{s} \in [0, 1] \times [0, 1000]$ and the different parameters $\boldsymbol{\theta}_1 = [1, 1]^T$, $\boldsymbol{\theta}_2 = [1.1, 1]^T$ and $\boldsymbol{\theta}_3 = [1, 1.1]$. Then the distances $\|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2\|$ and $\|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_3\|$ are equivalent, but the policies $\pi(a \,|\, \boldsymbol{s}; \boldsymbol{\theta}_1)$ and $\pi(a \,|\, \boldsymbol{s}; \boldsymbol{\theta}_2)$ are less different than $\pi(a \,|\, \boldsymbol{s}; \boldsymbol{\theta}_1)$ and $\pi(a \,|\, \boldsymbol{s}; \boldsymbol{\theta}_3)$ due to the different scaling in $\boldsymbol{s}$.

Hence, a metric that is invariant to transformations of the parameters would be better. Let $\boldsymbol{M}$ be a matrix that captures the "influence" of the parameters of the policy, then $\boldsymbol{M}$ can be used to defined a new metric that incorporates this influence:

$$L_M(\pi_{k+1}, \pi_k) \coloneqq (\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k)^T \boldsymbol{M} (\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k)$$

By choosing a good matrix $\boldsymbol{M}$, different influence like in the example above can be captured. Using this metric can be incorporated into the gradient ascent with a separate optimization problem for the ascent direction $\boldsymbol{\delta}_\theta$:

$$\begin{aligned} \boldsymbol{\delta}_\theta^* = \arg\max_{\boldsymbol{\delta}_\theta} \left(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})\right)^T \boldsymbol{\delta}_\theta \\ \text{s.t.} \quad \boldsymbol{\delta}_\theta^T \boldsymbol{M} \boldsymbol{\delta}_\theta \leq \epsilon \end{aligned} \tag{11.8}$$

This optimization problem encodes that the actual ascent direction should stay as close as possible to the steepest ascent direction, i.e. the gradient, but should not create jumps in the policy. The solution to this optimization problem is

$$\boldsymbol{\delta}_\theta^* = \lambda \boldsymbol{M}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \boldsymbol{M}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

with the Lagrangian multiplier λ.

Relative Entropy and Natural Gradient

An alternative way of measuring the distance between two policies is to use the "distance" of the distributions. A well-known measure for this is *relative entropy*, also known as the Kullback-Leibler divergence

$$D_{\mathrm{KL}}(p \| q) = \int p(\boldsymbol{x}) \ln \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} \, \mathrm{d}\boldsymbol{x}$$

introduced in section 1.2. An important property of the KL divergence is that is that the distance of two parametric distributions $p_{\boldsymbol{\theta}+\boldsymbol{\delta}_\theta}$ and $p_{\boldsymbol{\theta}}$ can be approximated by

$$D_{\mathrm{KL}}(p_{\boldsymbol{\theta}+\boldsymbol{\delta}_\theta} \| p_{\boldsymbol{\theta}}) \approx \boldsymbol{\delta}_\theta^T \boldsymbol{G}(\boldsymbol{\theta}) \boldsymbol{\delta}_\theta$$

with the Fisher information matrix $\boldsymbol{G}(\boldsymbol{\theta})$ (see subsection 1.2.1 for a thorough introduction). The *natural gradient* uses the fisher information matrix as a metric, i.e. $\boldsymbol{M} = \boldsymbol{G}(\boldsymbol{\theta})$, in the optimization problem (11.8). As every parameter has the same influence under this metric, the natural gradient is invariant to linear transformations of the parameter space! The ascent direction computed with this optimization problem corresponds to the steepest ascent in the policy space and not in the parameter space, hence the name *natural gradient*. It also guarantees converges to a local minimum!

Computing the Natural Gradient In the episode-based setting, it is easy to introduce the natural Gradient if the upper-level policy is a Gaussian. First the gradient $\nabla_{\boldsymbol{\omega}}^{\text{MC}} J(\boldsymbol{\omega})$ and subsequently the natural gradient $\nabla_{\boldsymbol{\omega}}^{\text{NG}} J(\boldsymbol{\omega}) = \boldsymbol{G}^{-1}(\boldsymbol{\omega}) \nabla_{\boldsymbol{\omega}} J(\boldsymbol{\omega})$ is computed, where the latter can be done in closed form for Gaussians. The natural gradient is then used for updating the parameters

$$\boldsymbol{\omega}_{k+1} = \boldsymbol{\omega}_k + \eta \nabla_{\boldsymbol{\omega}}^{\text{NG}} J(\boldsymbol{\omega})$$

with a step size η. For the step-based setting, this is not so easy and will be covered in section 11.2.2.

Compatible Function Approximation

For the step-based policy gradient with a baseline

$$\nabla_{\boldsymbol{\theta}}^{\text{PG}} J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T-1} \left(Q_t^{[i]} - b_i\left(\boldsymbol{s}_t^{[i]}\right) \right) \nabla_{\boldsymbol{\theta}} \ln \pi\left(\boldsymbol{a}_t^{[i]} \mid \boldsymbol{s}_t^{[i]}; \boldsymbol{\theta}\right),$$

the gradient estimate can be improved by estimating the reward to come with a function approximator

$$f_{\boldsymbol{w}}(\boldsymbol{s}, \boldsymbol{a}) = \boldsymbol{\psi}^T(\boldsymbol{s}, \boldsymbol{a}) \boldsymbol{w} \approx Q_t^{[i]} - b_i\left(\boldsymbol{s}_t^{[i]}\right).$$

This approximator works only on the states and actions and does not depend on the time step or the sample sequence. The function approximation (FA) gradient is

$$\nabla_{\boldsymbol{\theta}}^{\text{FA}} J(\boldsymbol{\theta}) \coloneqq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T-1} f_{\boldsymbol{w}}\left(\boldsymbol{s}_t^{[i]}, \boldsymbol{a}_t^{[i]}\right) \nabla_{\boldsymbol{\theta}} \ln \pi\left(\boldsymbol{a}_t^{[i]} \mid \boldsymbol{s}_t^{[i]}; \boldsymbol{\theta}\right),$$

which is still unbiased for the features $\boldsymbol{\psi}(\boldsymbol{s}, \boldsymbol{a}) = \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{a} \mid \boldsymbol{s}; \boldsymbol{\theta})$.

Episodic Natural Actor-Critic

11.3. Probabilistic Policy Search

Probabilistic policy search (a subset of episodic policy search methods) addresses the problems of policy gradient methods, namely the high amount of samples that are needed and the fact that the learning rate still has to be tuned. It is based on the *success matching principle* by Arrow, 1958: "When learning from a set of their own trials in iterated decision problems, humans attempt to match not the best taken action but the reward-weighted frequency of their actions and outcomes." This means that not only the reward, but also the frequency of this reward and the action is relevant. If a state is not very common, it might not be worth it to optimize the policy here.

This idea can be formalized as *episode-based success matching*. In each iteration k, first sample parameters $\boldsymbol{\theta}^{[i]} \sim \pi(\cdot; \boldsymbol{\omega}_k)$ and evaluate them: $R^{[i]} = \sum_{t=1}^{T} r_t^{[i]}$. Then somehow compute the *success probability*

$$w^{[i]} = f\big(R^{[i]}\big)$$

of each sample. This is essentially a transformation of the reward to a non-negative weight (an improper probability distribution). Next, compute the success-weighted policy

$$p\big(\boldsymbol{\theta}^{[i]}\big) \propto w^{[i]} \pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}_k\big)$$

of each sample. This is only a proportionality to indicate that the distribution has to be normalized. Finally, fit a new policy $\pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}_{k+1}\big)$ that best approximates $p\big(\boldsymbol{\theta}^{[i]}\big)$. The two open issues in this algorithm are:

- how to compute $w^{[i]} = f\big(R^{[i]}\big)$ and
- how to fit the policy $\pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}_{k+1}\big)$.

11.3.1. Policy Fitting by Weighted Maximum Likelihood

A first idea to find the policy $\pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}_{k+1}\big)$ that best fits the distribution $p\big(\boldsymbol{\theta}^{[i]}\big) \propto w^{[i]} \pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}_{k+1}\big)$ is to minimize the KL divergence between the two:

$$\begin{aligned}
\boldsymbol{\omega}_{k+1} &= \arg\min_{\boldsymbol{\omega}} D_{\mathrm{KL}}\big(p(\boldsymbol{\theta}) \| \pi(\boldsymbol{\theta}; \boldsymbol{\omega})\big) = \arg\min_{\boldsymbol{\omega}} \int p(\boldsymbol{\theta}) \ln \frac{p(\boldsymbol{\theta})}{\pi(\boldsymbol{\theta}; \boldsymbol{\omega})} \,\mathrm{d}\boldsymbol{\theta} \\
&= \arg\min_{\boldsymbol{\omega}} \underbrace{\int p(\boldsymbol{\theta}) \ln p(\boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{\theta}}_{\text{constant w.r.t. } \boldsymbol{\omega}} - \int p(\boldsymbol{\theta}) \ln \pi(\boldsymbol{\theta}; \boldsymbol{\omega}) \,\mathrm{d}\boldsymbol{\theta} \\
&= \arg\max_{\boldsymbol{\omega}} \int p(\boldsymbol{\theta}) \ln \pi(\boldsymbol{\theta}; \boldsymbol{\omega}) \,\mathrm{d}\boldsymbol{\theta} = \arg\max_{\boldsymbol{\omega}} \int \pi(\boldsymbol{\theta}; \boldsymbol{\omega}) \frac{p(\boldsymbol{\theta})}{\pi(\boldsymbol{\theta}; \boldsymbol{\omega})} \ln \pi(\boldsymbol{\theta}; \boldsymbol{\omega}) \,\mathrm{d}\boldsymbol{\theta} \\
&\approx \arg\max_{\boldsymbol{\omega}} \frac{1}{N} \sum_{i=1}^{N} \underbrace{\frac{p\big(\boldsymbol{\theta}^{[i]}\big)}{\pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}\big)}}_{= w^{[i]}} \ln \pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}\big) = \arg\max_{\boldsymbol{\omega}} \frac{1}{N} \sum_{i=1}^{N} w^{[i]} \ln \pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}\big)
\end{aligned} \tag{11.9}$$

This estimate of $\pi^{\text{new}}(\boldsymbol{\theta})$ is called the *weighted maximum likelihood estimate*. For a Gaussian policy $\pi(\boldsymbol{\theta}) = \pi(\boldsymbol{\theta}; \boldsymbol{\omega}) = \mathcal{N}(\boldsymbol{\theta} \,|\, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ with $\boldsymbol{\omega} = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$, the estimates can be computed in closed form

$$\boldsymbol{\mu}_{k+1} = \frac{1}{W} \sum_{i=1}^{N} w^{[i]} \boldsymbol{\theta}^{[i]} \qquad\qquad \boldsymbol{\Sigma}_{k+1} = \frac{1}{W} \sum_{i=1}^{N} w^{[i]} \big(\boldsymbol{\theta}^{[i]} - \boldsymbol{\mu}_{k+1}\big) \big(\boldsymbol{\theta}^{[i]} - \boldsymbol{\mu}_{k+1}\big)^T$$

with $W \coloneqq \sum_{i=1}^{N} w^{[i]}$. Closed-form solutions also exist for more complicated policies like Gaussian mixture models and Gaussian processes. With closed-form solutions, no learning rates are needed!

Gradient Ascent for Non-Closed-Forms and Similarity to Policy Gradients

If no closed-form solution exists for a parametric policy $\pi(\boldsymbol{\theta}; \boldsymbol{\omega})$, it is always possible to take the gradient of the objective (11.9) as

$$\boldsymbol{\nabla}_{\boldsymbol{\omega}} \left(\frac{1}{N} \sum_{i=1}^{N} w^{[i]} \ln \pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}\big) \right) = \frac{1}{N} \sum_{i=1}^{N} w^{[i]} \boldsymbol{\nabla}_{\boldsymbol{\omega}} \ln \pi\big(\boldsymbol{\theta}^{[i]}; \boldsymbol{\omega}\big)$$

and maximize the objective with gradient ascent. The form of the gradient is very similar to the gradient to (11.5), the gradient of the episodic policy gradient method. The difference is that instead of the total reward $R^{[i]}$, the success probabilities $w^{[i]}$ are used for weighting the samples. If the rewards are directly used for the weights (e.g. if the reward already is an improper probability distribution), policy gradient and weighted maximum likelihood are actually equivalent.

Computing the Weights

The next problem is to compute the weights (success probabilities) $w^{[i]} = f\big(R^{[i]}\big)$. The simplest way is to use an exponential transformation

$$w^{[i]} = \exp\Big\{\big(R^{[i]} - R^{[i]}_{\text{max}}\big)\beta\Big\}$$

with a *temperature* β. This temperature is usually set by heuristics, e.g. $\beta = 10/\big(R^{[i]}_{\text{max}} - R^{[i]}_{\text{min}}\big)$. Different algorithms emerged from this idea, for example:

- EM-Algorithms: PoWER, Reward-Weighted Regression
- Optimal Control: PI2
- Relative Entropy Policy Search (REPS)

A downside of the exponential transformation is that in a stochastic environment, the no longer the expected reward is maximized as

$$\mathbb{E}_{p(\boldsymbol{\tau})}\Big[\exp\big\{R(\boldsymbol{\tau})\big\}\Big] \neq \exp\Big\{\mathbb{E}_{p(\boldsymbol{\tau})}\big[R(\boldsymbol{\tau})\big]\Big\}.$$

This has the effect that the objective gets "risk attracted". However, it still works well for moderately stochastic environments.

Notes on Expectation Maximization The expectation-maximization (EM) algorithm is a method for maximum likelihood estimation in the presence of latent variables. In this setting, the observed variables are the reward and the unobserved variables are the trajectories (or parameters) that lead to the reward. The algorithm is split into two steps:

E-Step Estimate the new desired distribution.

M-Step Estimate new (policy) parameters from weighted samples.

Algorithms that are based on the EM-principle are Reward-Weighted Regression (Peters, 2007) and PoWER[2] (Kober, 2008).

11.3.2. Relative Entropy Policy Search (REPS)

Choosing the next policy inherently requires choosing a got temperature β. A too low temperature would converge too slow and a too high temperature would directly jump to a single sample and does not explore enough. This is a variant of the exploration-exploitation tradeoff. As for vanilla policy gradients, this can again be addressed using a metric to control the jump in the policy. The metric of choice for *relative entropy policy search* (REPS) is, as obvious from the name, the KL divergence

$$D_{\text{KL}}\big(\pi(\boldsymbol{\theta}) \,\|\, q(\boldsymbol{\theta})\big) \leq \epsilon$$

[2] PoWER stands for "Policy Learning by Weighting Exploration with the Returns".

constraining the information loss between the new policy[3] $\pi(\boldsymbol{\theta})$ and the old policy $q(\boldsymbol{\theta})$ to be at most ϵ. The REPS optimization problem is:

$$\begin{aligned} \max_{\pi} \; & \int \pi(\boldsymbol{\theta}) R(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} \\ \text{s.t.} \quad & D_{\mathrm{KL}}\big(\pi(\boldsymbol{\theta}) \,\|\, q(\boldsymbol{\theta})\big) \leq \epsilon \\ & \int \pi(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} = 1 \end{aligned} \tag{11.10}$$

The analytical solution to this optimization problem is

$$\pi(\boldsymbol{\theta}) \propto q(\boldsymbol{\theta}) \exp\left\{ \frac{R(\boldsymbol{\theta})}{\eta} \right\}$$

where $R(\boldsymbol{\theta})$ is the total reward for a parameter $\boldsymbol{\theta}$ and η is the Lagrangian multiplier for the KL constraint. This the same result as for the weighted maximum likelihood approach using the exponential transformation with temperature $\beta = 1/\eta$. The scaling factor η is determined by the optimization problem and specified by the KL bound ϵ. To get η, the dual objective (approximated with Monte-Carlo integration, $\boldsymbol{\theta}_i \sim q(\cdot)$)

$$\eta\epsilon + \ln \int q(\boldsymbol{\theta}) \exp\left\{ \frac{R(\boldsymbol{\theta})}{\eta} \right\} \mathrm{d}\boldsymbol{\theta} \approx \eta\epsilon + \ln \left(\frac{1}{N} \sum_{i=1}^{N} \exp\left\{ \frac{R(\boldsymbol{\theta}_i)}{\eta} \right\} \right)$$

has to be minimized (subject to $\eta \geq 0$). This can be done with standard numerical optimization algorithms like trust-region methods or similar.

For discrete $\boldsymbol{\theta}$, the optimization problem is

$$\begin{aligned} \max_{\pi} \; & \sum_{\boldsymbol{\theta}} \pi(\boldsymbol{\theta}) R(\boldsymbol{\theta}) \\ \text{s.t.} \quad & D_{\mathrm{KL}}\big(\pi(\boldsymbol{\theta}) \,\|\, q(\boldsymbol{\theta})\big) \leq \epsilon \\ & \sum_{\boldsymbol{\theta}} \pi(\boldsymbol{\theta}) = 1 \end{aligned} \tag{11.11}$$

where now the KL divergence is discrete too.

Derivation of REPS

With the multipliers λ and η, the Lagrangian of this problem is

$$\begin{aligned} L[\pi, \lambda, \eta] &= \int \pi(\boldsymbol{\theta}) R(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} + \lambda \left(1 - \int \pi(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} \right) + \eta \big(\epsilon - D_{\mathrm{KL}}\big(\pi(\boldsymbol{\theta}) \,\|\, q(\boldsymbol{\theta})\big) \big) \\ &= \int \pi(\boldsymbol{\theta}) R(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} + \lambda \left(1 - \int \pi(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} \right) + \eta \left(\epsilon - \int \pi(\boldsymbol{\theta}) \ln \frac{\pi(\boldsymbol{\theta})}{q(\boldsymbol{\theta})} \, \mathrm{d}\boldsymbol{\theta} \right) \\ &= (\lambda + \eta\epsilon) + \int \pi(\boldsymbol{\theta}) R(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} - \lambda \int \pi(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} - \eta \int \pi(\boldsymbol{\theta}) \ln \frac{\pi(\boldsymbol{\theta})}{q(\boldsymbol{\theta})} \, \mathrm{d}\boldsymbol{\theta} \\ &= (\lambda + \eta\epsilon) + \int \pi(\boldsymbol{\theta}) \left[R(\boldsymbol{\theta}) - \lambda - \eta \ln \frac{\pi(\boldsymbol{\theta})}{q(\boldsymbol{\theta})} \right] \mathrm{d}\boldsymbol{\theta} \, . \end{aligned} \tag{11.12}$$

[3] Note REPS does not assume the policy to be parametric. In the following, an upper-level policy is optimized, but it is also possible to directly optimize a policy that produces actions dependent on states. See Peters, Mülling, Altün: "Relative Entropy Policy Search" (2010).

To maximize this Lagrangian w.r.t. the primal variable $\pi(\boldsymbol{\theta})$, the term can be discarded as it does not depend on the primal variable. The remaining integral is a function over the primal variable that can be solved using variational calculus. Let

$$\mathcal{L}[\pi, \lambda, \eta] = \pi(\boldsymbol{\theta}) \left[R(\boldsymbol{\theta}) - \lambda - \eta \ln \frac{\pi(\boldsymbol{\theta})}{q(\boldsymbol{\theta})} \right]$$

be the respective Lagrangian. As $\mathcal{L}$ does not depend on $\pi'(\boldsymbol{\theta})$, the Euler-Lagrange equation becomes $\frac{\partial \mathcal{L}}{\partial \pi(\boldsymbol{\theta})} = 0$. This yields the algebraic equation

$$\frac{\partial \mathcal{L}}{\partial \pi(\boldsymbol{\theta})} = R(\boldsymbol{\theta}) - \lambda - \eta \ln \frac{\pi(\boldsymbol{\theta})}{q(\boldsymbol{\theta})} - \eta\pi(\boldsymbol{\theta}) \frac{1}{\pi(\boldsymbol{\theta})} = R(\boldsymbol{\theta}) - \eta \ln \pi(\boldsymbol{\theta}) + \eta \ln q(\boldsymbol{\theta}) - (\eta + \lambda) \stackrel{!}{=} 0$$

that is easily solvable for $\pi(\boldsymbol{\theta})$:

$$\begin{aligned}
& 0 = R(\boldsymbol{\theta}) - \eta \ln \pi(\boldsymbol{\theta}) + \eta \ln q(\boldsymbol{\theta}) - (\eta + \lambda) \\
\iff \quad & \eta \ln \pi(\boldsymbol{\theta}) = R(\boldsymbol{\theta}) + \eta \ln q(\boldsymbol{\theta}) - (\eta + \lambda) \\
\iff \quad & \ln \pi(\boldsymbol{\theta}) = \frac{R(\boldsymbol{\theta})}{\eta} + \ln q(\boldsymbol{\theta}) - \frac{\eta + \lambda}{\eta} \\
\iff \quad & \pi(\boldsymbol{\theta}) = q(\boldsymbol{\theta}) \exp\left\{ \frac{R(\boldsymbol{\theta})}{\eta} \right\} \exp\left\{ -\frac{\eta + \lambda}{\eta} \right\}
\end{aligned}$$

But the last exponential term is just a normalization factor! Hence, it can also be written as

$$\exp\left\{ \frac{\eta + \lambda}{\eta} \right\} = \int q(\boldsymbol{\theta}) \exp\left\{ \frac{R(\boldsymbol{\theta})}{\eta} \right\} \mathrm{d}\boldsymbol{\theta} \quad \iff \quad \frac{\eta + \lambda}{\eta} = \ln \int q(\boldsymbol{\theta}) \exp\left\{ \frac{R(\boldsymbol{\theta})}{\eta} \right\} \mathrm{d}\boldsymbol{\theta} \,.$$

To get the dual objective $h(\lambda, \eta)$, the solution of the primal variable has to be plugged into the Lagrangian (11.12):

$$\begin{aligned}
h(\lambda, \eta) = L[\pi^*, \lambda, \eta] &= (\lambda + \eta\epsilon) + \int \pi^*(\boldsymbol{\theta}) R(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} - \lambda \underbrace{\int \pi^*(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}}_{=1} - \eta \int \pi(\boldsymbol{\theta}) \ln \frac{\pi^*(\boldsymbol{\theta})}{q(\boldsymbol{\theta})} \, \mathrm{d}\boldsymbol{\theta} \\
&= \eta\epsilon + \int \pi^*(\boldsymbol{\theta}) \left[R(\boldsymbol{\theta}) - \eta \ln \frac{\pi^*(\boldsymbol{\theta})}{q(\boldsymbol{\theta})} \right] \mathrm{d}\boldsymbol{\theta} \\
&= \eta\epsilon + \int \pi^*(\boldsymbol{\theta}) \left[R(\boldsymbol{\theta}) - \eta \ln \frac{q(\boldsymbol{\theta}) \exp\left\{ \frac{R(\boldsymbol{\theta})}{\eta} \right\} \exp\left\{ -\frac{\eta + \lambda}{\eta} \right\}}{q(\boldsymbol{\theta})} \right] \mathrm{d}\boldsymbol{\theta} \\
&= \eta\epsilon + \int \pi^*(\boldsymbol{\theta}) \left[R(\boldsymbol{\theta}) - \eta \ln \left(\exp\left\{ \frac{R(\boldsymbol{\theta})}{\eta} \right\} \exp\left\{ -\frac{\eta + \lambda}{\eta} \right\} \right) \right] \mathrm{d}\boldsymbol{\theta} \\
&= \eta\epsilon + \int \pi^*(\boldsymbol{\theta}) \left[R(\boldsymbol{\theta}) - \eta \frac{R(\boldsymbol{\theta})}{\eta} + \eta \frac{\eta + \lambda}{\eta} \right] \mathrm{d}\boldsymbol{\theta} \\
&= \eta\epsilon + \eta \frac{\eta + \lambda}{\eta} \underbrace{\int \pi^*(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}}_{=1} = \eta\epsilon + \eta \frac{\eta + \lambda}{\eta} = \eta\epsilon + \ln \int q(\boldsymbol{\theta}) \exp\left\{ \frac{R(\boldsymbol{\theta})}{\eta} \right\} \mathrm{d}\boldsymbol{\theta}
\end{aligned}$$

By approximating this integral with Monte-Carlo integration, the Lagrangian multiplier η can be calculated.

11.3.3. REPS for Contextual Policy Search

In contextual policy search, the task is dependent on a context $\boldsymbol{c}$ that is fixed before executing (e.g. the target location when throwing a ball). This can be introduced to the policy by conditioning the upper-level policy on the context: $\pi(\boldsymbol{\theta} \mid \boldsymbol{c})$. The dataset for the policy update is changed to $\mathcal{D} = \left\{\boldsymbol{\theta}^{[i]}, \boldsymbol{c}^{[i]}, R^{[i]}\right\}_{i=1,\cdots,N}$, taking care of the context. The objective is also changed to maximize the expected reward over all parameters and all contexts with the context distribution $\mu_0(\boldsymbol{c})$ and the reward $R(\boldsymbol{c}, \boldsymbol{\theta})$:

$$J_\pi = \iint \mu_0(\boldsymbol{c})\pi(\boldsymbol{\theta} \mid \boldsymbol{c})R(\boldsymbol{c}, \boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{c} \,\mathrm{d}\boldsymbol{\theta}$$

The REPS optimization problem for contextual policy search directly optimized the joint distribution $p(\boldsymbol{c}, \boldsymbol{\theta}) = \mu_0(\boldsymbol{c})\pi(\boldsymbol{\theta} \mid \boldsymbol{c})$ instead of conditioning the optimization problem:

$$\begin{aligned}
\max_p \quad & \iint p(\boldsymbol{c}, \boldsymbol{\theta}) R(\boldsymbol{c}, \boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{c} \,\mathrm{d}\boldsymbol{\theta} \\
\text{s.t.} \quad & D_{\mathrm{KL}}\big(p(\boldsymbol{c}, \boldsymbol{\theta}) \,\|\, q(\boldsymbol{c}, \boldsymbol{\theta})\big) \leq \epsilon \\
& \iint p(\boldsymbol{c}, \boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{c} \,\mathrm{d}\boldsymbol{\theta} = 1 \\
& \forall \boldsymbol{c} : \int p(\boldsymbol{c}, \boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{\theta} = \mu_0(\boldsymbol{c})
\end{aligned}$$

Some major problems with this approach are that the context distributions cannot be freely chosen by the algorithm, there are an infinite amount of constraints and for each context a lot of parameter samples are needed.

Feature Matching

Instead of perfectly matching the context distribution which emits an infinite amount of constraints, one idea is to instead match certain features $\boldsymbol{\phi}(\boldsymbol{c})$, for example the first- and second-order moments $\boldsymbol{\phi}^T(c) = \begin{bmatrix} c & c^2 \end{bmatrix}$. This would be equivalent to matching mean and variance which, for Gaussian distributions, is exact (mean and variance are sufficient statistics). With the target features $\hat{\boldsymbol{\phi}}$, the the optimization problem is given as:

$$\begin{aligned}
\max_p \quad & \iint p(\boldsymbol{c}, \boldsymbol{\theta}) R(\boldsymbol{c}, \boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{c} \,\mathrm{d}\boldsymbol{\theta} \\
\text{s.t.} \quad & D_{\mathrm{KL}}\big(p(\boldsymbol{c}, \boldsymbol{\theta}) \,\|\, q(\boldsymbol{c}, \boldsymbol{\theta})\big) \leq \epsilon \\
& \iint p(\boldsymbol{c}, \boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{c} \,\mathrm{d}\boldsymbol{\theta} = 1 \\
& \iint p(\boldsymbol{c}, \boldsymbol{\theta}) \boldsymbol{\phi}(\boldsymbol{c}) \,\mathrm{d}\boldsymbol{c} \,\mathrm{d}\boldsymbol{\theta} = \hat{\boldsymbol{\phi}}
\end{aligned}$$

This optimization problem can again be solved in closed form

$$p(\boldsymbol{c}, \boldsymbol{\theta}) = \mu_0(\boldsymbol{c})\pi(\boldsymbol{\theta} \mid \boldsymbol{c}) \propto q(\boldsymbol{c}, \boldsymbol{\theta}) \exp\left\{\frac{R(\boldsymbol{c}, \boldsymbol{\theta}) - V(\boldsymbol{c})}{\eta}\right\}$$

with $V(\boldsymbol{c}) = \boldsymbol{\phi}^T(\boldsymbol{c})\boldsymbol{v}$ with the Lagrangian multipliers η and $\boldsymbol{v}$. Those are for the KL and the moment matching constraints, respectively. The Lagrangian multipliers are again given as the solution of the dual optimization

problem. The dual function is

$$h(\eta, \boldsymbol{v}) = \eta\epsilon + \hat{\boldsymbol{\phi}}^T\boldsymbol{v} + \eta \ln \int q(\boldsymbol{c}, \boldsymbol{\theta}) \exp\left\{\frac{R(\boldsymbol{c}, \boldsymbol{\theta}) - \boldsymbol{\phi}^T(\boldsymbol{c})\boldsymbol{v}}{\eta}\right\}$$
$$\approx \eta\epsilon + \hat{\boldsymbol{\phi}}^T\boldsymbol{v} + \eta \ln \left(\frac{1}{N}\sum_{i=1}^{N} q(\boldsymbol{c}_i, \boldsymbol{\theta}_i) \exp\left\{\frac{R(\boldsymbol{c}_i, \boldsymbol{\theta}_i) - \boldsymbol{\phi}^T(\boldsymbol{c}_i)\boldsymbol{v}}{\eta}\right\}\right)$$

with Monte-Carlo samples $(\boldsymbol{c}_i, \boldsymbol{\theta}_i)$ drawn from $q(\boldsymbol{c}, \boldsymbol{\eta})$.

For discrete $\boldsymbol{c}$ and $\boldsymbol{\theta}$, the optimization problem is

$$\begin{aligned} \max_p \sum_{\boldsymbol{c},\boldsymbol{\theta}} p(\boldsymbol{c}, \boldsymbol{\theta}) R(\boldsymbol{c}, \boldsymbol{\theta}) \\ \text{s.t.} \quad D_{\text{KL}}\big(p(\boldsymbol{c}, \boldsymbol{\theta}) \,\|\, q(\boldsymbol{c}, \boldsymbol{\theta})\big) \le \epsilon \\ \sum_{\boldsymbol{c},\boldsymbol{\theta}} p(\boldsymbol{c}, \boldsymbol{\theta}) = 1 \\ \sum_{\boldsymbol{c},\boldsymbol{\theta}} p(\boldsymbol{c}, \boldsymbol{\theta}) \boldsymbol{\phi}(\boldsymbol{c}) = \hat{\boldsymbol{\phi}} \end{aligned}$$

where now the KL divergence is discrete too.

Contextual Policies with Weighted ML

It is also possible to get contextual policies using the weighted maximum likelihood approach. For a Gaussian policy

$$\pi_{\boldsymbol{\omega}}(\boldsymbol{\theta} \,|\, \boldsymbol{c}) = \mathcal{N}(\boldsymbol{\theta} \,|\, \boldsymbol{K}\boldsymbol{c} + \boldsymbol{k},\, \boldsymbol{\Sigma}), \quad \boldsymbol{\omega} = \{\boldsymbol{K}, \boldsymbol{k}, \boldsymbol{\Sigma}\}$$

with parameters $\boldsymbol{\omega}$, the solution is given as

$$\begin{bmatrix} \boldsymbol{k}^T \\ \boldsymbol{K}^T \end{bmatrix} = \left(\boldsymbol{C}^T\boldsymbol{D}\boldsymbol{C}\right)^{-1}\boldsymbol{C}^T\boldsymbol{D}\boldsymbol{A} \qquad \boldsymbol{\Sigma} = \frac{1}{\sum_{i=1}^{N} w_i}\sum_{i=1}^{N} w^{[i]}\big(\boldsymbol{\theta}^{[i]} - \boldsymbol{\mu}^{[i]}\big)\big(\boldsymbol{\theta}^{[i]} - \boldsymbol{\mu}^{[i]}\big) \qquad \boldsymbol{\mu}^{[i]} := \boldsymbol{K}\boldsymbol{c}^{[i]} + \boldsymbol{k}$$

where the weights $w^{[i]}$ are computed some how (e.g. using the exponential transformation) and the samples $\boldsymbol{c}^{[i]}$ and $\boldsymbol{\theta}^{[i]}$ are drawn from the previous distribution. Here, $\boldsymbol{C}$ is the input data matrix (with an appended 1 for the bias), $\boldsymbol{D}$ is a diagonal weighting matrix and $\boldsymbol{A}$ is the parameter matrix.

11.3.4. Learning Versatile Solutions and Hierarchical REPS (HiREPS)

In many motor tasks, multiple solutions to the same problem are possible. One problem with the current formulation is that either only one of the solutions is found or the policy averages over multiple solutions. As these solutions may lie far apart, that creates a rather unstable policy. One idea to solve this is to use a hierarchical approach where an upper-level *gating policy* $\pi(o \,|\, \boldsymbol{c})$ chooses the option o that then selects an *option policy* $\pi(\boldsymbol{\theta} \,|\, \boldsymbol{c}, o_i)$ to handle the task.

The naive hierarchical approach is to define the joint distribution $p(\boldsymbol{c}, \boldsymbol{\theta}, o) = \mu_0(\boldsymbol{c})\pi(\boldsymbol{\theta} \,|\, \boldsymbol{c}, o)\pi(o \,|\, \boldsymbol{c})$ and

solve the optimization problem

$$\begin{aligned}\max_p \sum_o \iint p(\boldsymbol{c}, \boldsymbol{\theta}, o) R(\boldsymbol{c}, \boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{c}\,\mathrm{d}\boldsymbol{\theta} \\ \text{s.t.} \quad D_{\mathrm{KL}}\big(p(\boldsymbol{c}, \boldsymbol{\theta}, o) \,\|\, q(\boldsymbol{c}, \boldsymbol{\theta}, o)\big) \leq \epsilon \\ \sum_o \iint p(\boldsymbol{c}, \boldsymbol{\theta}, o) \,\mathrm{d}\boldsymbol{c}\,\mathrm{d}\boldsymbol{\theta} = 1 \\ \sum_o \iint p(\boldsymbol{c}, \boldsymbol{\theta}, o) \boldsymbol{\phi}(\boldsymbol{c}) \,\mathrm{d}\boldsymbol{c}\,\mathrm{d}\boldsymbol{\theta} = \hat{\boldsymbol{\phi}}\end{aligned}$$

while still matching the features. But this optimization problem does not enforce versatile solutions, i.e. multiple options are learned, but no separated between them is done. The insight to solve this is that versatile solutions are characterized by a small overlap in the solutions. This can be represented as limiting the entropy of $p(o \,|\, \boldsymbol{c}, \boldsymbol{\theta})$. This leads to the optimization problem underlying *Hierarchical REPS* (HiREPS):

$$\begin{aligned}\max_p \sum_o \iint p(\boldsymbol{c}, \boldsymbol{\theta}, o) R(\boldsymbol{c}, \boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{c}\,\mathrm{d}\boldsymbol{\theta} \\ \text{s.t.} \quad D_{\mathrm{KL}}\big(p(\boldsymbol{c}, \boldsymbol{\theta}, o) \,\|\, q(\boldsymbol{c}, \boldsymbol{\theta}, o)\big) \leq \epsilon \\ \sum_o \iint p(\boldsymbol{c}, \boldsymbol{\theta}, o) \,\mathrm{d}\boldsymbol{c}\,\mathrm{d}\boldsymbol{\theta} = 1 \\ \sum_o \iint p(\boldsymbol{c}, \boldsymbol{\theta}, o) \boldsymbol{\phi}(\boldsymbol{c}) \,\mathrm{d}\boldsymbol{c}\,\mathrm{d}\boldsymbol{\theta} = \hat{\boldsymbol{\phi}} \\ \sum_o \iint p(\boldsymbol{c}, \boldsymbol{\theta}, o) \big[- p(o \,|\, \boldsymbol{c}, \boldsymbol{\theta}) \ln p(o \,|\, \boldsymbol{c}, \boldsymbol{\theta}) \big] \,\mathrm{d}\boldsymbol{c}\,\mathrm{d}\boldsymbol{\theta} \leq \kappa\end{aligned}$$

Of course this can readily be adapted to discrete states and parameters by swapping the integrals for sums. The solution of this optimization problem actually learns versatile solutions!

11.3.5. Sequencing Movement Primitives and Sequential REPS

For many motor tasks it is also necessary to execute different elementary building blocks in a sequence to fulfill a task. This also means that the context of blocks executed later also depends on previously executed blocks! So the long-term effects have to be learned. Similar to the hierarchical approach, this changes the goal from maximizing the expected reward to maximize the expected reward over K decision steps, i.e. building blocks. For each block an upper-level policy $\pi_k(\boldsymbol{\theta} \,|\, \boldsymbol{c})$ is learned where the k-th block has to react to the outcome of block k. The objective function is

$$J_\pi = \sum_{k=1}^{K} \iint \mu_k(\boldsymbol{c}) \pi_k(\boldsymbol{\theta} \,|\, \boldsymbol{c}) R_k(\boldsymbol{c}, \boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{c}\,\boldsymbol{\theta}$$

where the context distributions $\mu_k(\boldsymbol{c})$ are specified by the previous policies $\pi_{l<k}(\boldsymbol{\theta} \,|\, \boldsymbol{c})$:

$$\mu_k(\boldsymbol{c}') = \iint \mu_{k-1}(\boldsymbol{c}) \pi_{k-1}(\boldsymbol{\theta} \,|\, \boldsymbol{c}) p(\boldsymbol{c}' \,|\, \boldsymbol{c}, \boldsymbol{\theta}) \,\mathrm{d}\boldsymbol{c}\,\mathrm{d}\boldsymbol{\theta}$$

With $p_k(\boldsymbol{c}, \boldsymbol{\theta}) = \mu_k(\boldsymbol{c})\pi(\boldsymbol{\theta} \mid \boldsymbol{c})$, the optimization problem of *Sequential REPS* is

$$\begin{aligned} \max_p \sum_{k=1}^{K} \iint p_k(\boldsymbol{c}, \boldsymbol{\theta}) R_k(\boldsymbol{c}, \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{c} \, \mathrm{d}\boldsymbol{\theta} \\ \text{s.t.} \qquad D_{\mathrm{KL}}\big(p_k(\boldsymbol{c}, \boldsymbol{\theta}) \,\|\, q_k(\boldsymbol{c}, \boldsymbol{\theta})\big) \leq \epsilon, \; \forall k \\ \iint p_k(\boldsymbol{c}, \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{c} \, \mathrm{d}\boldsymbol{\theta} = 1, \; \forall k \\ \iint p_{k-1}(\boldsymbol{c}, \boldsymbol{\theta}) p(\boldsymbol{c}' \mid \boldsymbol{c}, \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{c} \, \mathrm{d}\boldsymbol{\theta} = p_k(\boldsymbol{c}'), \; \forall k > 1 \end{aligned}$$

with the solution

$$p_k(\boldsymbol{c}, \boldsymbol{\theta}) \propto q_k(\boldsymbol{c}, \boldsymbol{\theta}) \exp\left\{ \frac{R_k(\boldsymbol{c}, \boldsymbol{\theta}) + \mathbb{E}_{p(\boldsymbol{c}' \mid \boldsymbol{c}, \boldsymbol{\theta})}\big[V_{k+1}(\boldsymbol{c}')\big] - V_k(\boldsymbol{c})}{\eta_k} \right\}$$

where $\mathbb{E}_{p(\boldsymbol{c}' \mid \boldsymbol{c}, \boldsymbol{\theta})}\big[V_{k+1}(\boldsymbol{c}')\big]$ encoded the long-term reward.

11.4. Wrap-Up

Policy search is a powerful and practical alternative to value function and model-based methods as learning a model is really hard and a small error in either the model or the value function can have catastrophic consequences. Policy search gradients have been the go-to methods for policy search for a long time, but they still need lots of samples and the learning rate has to be tuned (learning the learning rate is still an open problem). Newer methods are probabilistic policy search methods.

They reduce the policy update to a weighted maximum likelihood estimate of the policy parameters. The weights are calculated using an exponential transformation with a temperature β. This temperature has to be hand-tuned when using plain weighted ML estimates. The REPS algorithm optimizes the temperature by imposing an upper bound on the KL divergence of the new and old policy. Various extensions of REPS have been discussed like contextual REPS, HiREPS and Sequential REPS.

For a comparison with the other discussed reinforcement learning methods (model-based, see chapter 9 and value function, see chapter 10), the following list gives an overview over the main benefits and drawbacks of *episode-based policy search methods*:

Model Complexity None (no approximation errors)

- Need to evaluate total reward R.

Scalability Good

- Parameterized policies are compact.
- Also allows learning for high-dimensional robots.
- Only works for a moderate amount of parameter.

Data Efficiency Poor

- Each rollout is just one sample.
- High variance in rewards for stochastic environments.

Other Limitations Mainly used for learning single trajectories (e.g. DMPs).

And for *step-based policy search*:

Model Complexity None (no approximation errors)

- Need to evaluate reward to come Q_t.

Scalability Good

- Parameterized policies are compact.
- Also allows learning for high-dimensional robots.
- Only works for a moderate amount of parameter.

Data Efficiency Poor

- Uses every state-action pair with reward to come.
- High variance in reward to come.

Other Limitations Mainly used for learning single trajectories (e.g. DMPs).

12. Imitation Learning: Behavioral Cloning and Inverse RL

Ot is often very hard or even impossible and time-consuming to learn policies from scratch using reinforcement learning, e.g. as the search space is too large to explore it in a reasonable amount of time. A human export, for example a professional table tennis player, takes years for perfecting its policy and the robot can avoid that by cloning his policy. This is called *imitation learning* and is discussed in this chapter. Two different types of imitation learning are discussed: behavioral closing, where the actions are reproduced given the state, and inverse reinforcement learning, which is learning the reward function from demonstrations, given the optimal policy. Both of them rely on demonstrations, so a crucial question is how to demonstrate movements? A few possibilities are

- Teleoperation:
  Use a joystick to train an RC car, The steering wheel of a car to train a car, Data cloves for an artificial hand, …
- Kinesthetic Teach-In:
  Take the robot "by the hand" and demonstrate the movement, like a tennis teacher teaches a student.
- Sensuits:
  Suits with encoders and accelerometers that are attached to humans to capture the movement.
- Marker-Based Tracking:
  Use markers and a basic skeleton to obtain precise data from the movement (motion capturing suits).
- Vision:
  Video-based tracking of humans.

12.1. Distribution Matching

12.1.1. Behavioral Cloning

Behavioral cloning is the simplest form of imitation learning where demonstrations $\left\{\boldsymbol{s}_{1:T}^{[i]}, \boldsymbol{a}_{1:T}^{[i]}\right\}_{i=1,\cdots,N}$, generated by experts, that follow an optimal policy $\pi^*(\boldsymbol{a} \mid \boldsymbol{s})$ with the long-term behavior $\mu^*(\boldsymbol{s})$, are used to estimate a policy $\hat{\pi}(\boldsymbol{a} \mid \boldsymbol{s})$. This is, in principle, a supervised regression problem.

Direct behavioral cloning directly applies supervised learning methods (e.g. linear regression, Gaussian processes or neural networks) to the dataset $\mathcal{D}$ to extract a policy $\pi(\boldsymbol{a} \mid \boldsymbol{s})$, boiling down it to a regression problem. This often causes the *clean-up effect*: due to regularization in the ML model, noise in the demonstrations is often not exhibited by the reproduction. Hence, the student often surpassed the quality of the expert/teacher.

While this is a rather simple approach, it has a few major problems:

- It requires lots of demonstrations to generalize, and these demonstrations often exhibit a high variance.
- The estimated policy always has an approximation error and small errors in the policy may lead to large error in the long-term behavior $\hat{\mu}(\boldsymbol{s})$.
- The agent does not know how to recover from previously unseen states, which is problematic as the agent is prone to make mistakes due to the residual error! This problem is called the *covariate shift*.

DAGGER: New Samples to Learn to Recover

In the *DAGGER* (*Dataset Aggregation*, see Ross, Gordon and Bagnell: "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning" (2010)) algorithm, the robot is given the ability to ask for new demonstrations if it ends up in an unseen states. The teacher then provides new data and the robot aggregates it with the previously seen demonstrations.

DART: Robustness in Imitation Learning

In the *DART* (*Disturbances for Augmenting Robot Trajectories*, see Laskey, et al.: "DART: Noise Injection for Robust Imitation Learning" (2017)) method, noise is added to the demonstrations to learn how the teacher adapts. In consequence, the learned policy will be more robust with respect to unexpected variations. This is an improvement over DAGGER as it does not require a teacher to present which is time consuming.

12.1.2. Generative Adversarial Learning

In *generative adversarial learning*, not the distance between the agents and the experts policy is minimized, but the *occupancy measure*

$$\rho_\pi(\boldsymbol{s}, \boldsymbol{a}) = \pi(\boldsymbol{s} \,|\, \boldsymbol{a}) \sum_{t=1}^{\infty} \gamma^t \Pr(\boldsymbol{s}_t = \boldsymbol{s} \,|\, \pi)$$

is matched. It can be interpreted as the distribution of state-action pairs that the agent encounters when navigating through the environment with policy π. The probability $\Pr(\boldsymbol{s}_t = \boldsymbol{s} \,|\, \pi)$ is the probability of encountering state $\boldsymbol{s}$ in time step t when following policy π. The matching is done by minimizing the KL divergence between the policy π_E of the expert and the estimated policy $\hat{\pi}$:

$$\hat{\pi} = \arg\min_{\hat{\pi}'} D_{\mathrm{KL}}\big(\rho_{\pi_E}(\boldsymbol{s}, \boldsymbol{a}) \,\big\|\, \rho_{\hat{\pi}'}(\boldsymbol{s}, \boldsymbol{a})\big)$$

This is solved by GAIL (Generative Adversarial Imitation Learning) with a minmax game

$$\max_{\hat{\pi}'} \min_{D} \mathbb{E}_{\hat{\pi}'}\big[\ln D(\boldsymbol{s}, \boldsymbol{a})\big] + \mathbb{E}_{\pi_E}\big[\ln\big(1 - D(\boldsymbol{s}, \boldsymbol{a})\big)\big] - \lambda H(\hat{\pi}')$$

with a discriminator D and the policy $\hat{\pi}'$. The algorithm is composed of two steps:

1. The discriminator is trained for a high probability in the experts demonstrations.
2. The policy is trained for maximize the discriminator, considering the discriminator as the reward.

12.2. Inverse Reinforcement Learning

In *inverse reinforcement learning* (IRL), the problem if imitation learning is tackled from a different point of view: assuming the reward function provides the most informative and transferable definition of the task, learn the reward function from demonstrations given a model and a policy and subsequently recover the reward function. This reward function can then be used to further improve the policy or to learn a different policy (apprenticeship learning).

The problem setup uses demonstrations $\left\{\boldsymbol{s}_{1:T}^{[i]}, \boldsymbol{a}_{1:T}^{[i]}\right\}_{i=1,\cdots,N}$ from an expert that encode the teachers policy $\pi^*(\boldsymbol{a} \mid \boldsymbol{s})$ and the long-term behavior $\mu^*(\boldsymbol{s})$ and a transition model $r(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t)$ but no reward function as inputs. The problem of inverse RL is now to recover the reward function $r(\boldsymbol{s})$ that best explains the policy $\pi^*(\boldsymbol{a} \mid \boldsymbol{s})$ and the long-term behavior $\mu^*(\boldsymbol{s})$.

12.2.1. Basic Principle

The problem can be formulated as a feasibility problem: find a reward function $r^*(\boldsymbol{s})$ such that

$$\mathbb{E}_{p,\pi^*}\left[\sum_{t=1}^{\infty}\gamma^t r^*(\boldsymbol{s}_t)\right] \geq \mathbb{E}_{p,\pi}\left[\sum_{t=1}^{\infty}\gamma^t r^*(\boldsymbol{s}_t)\right] \tag{12.1}$$

holds for all policies π. While this problem is convex, it still has many challenges:

Limited Data Often only traces from the experts behavior are given, not the entire optimal policy.

Ill-Posed Reward $r^*(\boldsymbol{s}) \equiv 0$ is a solution, there is ambiguity in the reward function.

Expert Suboptimality Assumes that the expert is indeed optimal. If this is not the case, the problem becomes infeasible.

Computation Assumes that the policies can be enumerated.

Addressing the "Limited Data" Challenge: Feature-Based Reward Function

Assuming the reward function is linear in some features $\boldsymbol{\phi}(\boldsymbol{s})$, i.e. $r(\boldsymbol{s}) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{s})$, the right hand side of (12.1) becomes

$$\mathbb{E}_{p,\pi}\left[\sum_{t=1}^{\infty}\gamma^t r(\boldsymbol{s}_t)\right] = \mathbb{E}_{p,\pi}\left[\sum_{t=1}^{\infty}\gamma^t \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{s})\right] = \boldsymbol{w}^T \underbrace{\mathbb{E}_{p,\pi}\left[\sum_{t=1}^{\infty}\gamma^t \boldsymbol{\phi}(\boldsymbol{s})\right]}_{\boldsymbol{\psi}(\pi) :=} \doteq \boldsymbol{w}^T\boldsymbol{\psi}(\pi)$$

with $\boldsymbol{\psi}(\pi)$ being the expected discounted features of policy π. The feasibility problem (12.1) now is for find some $\boldsymbol{w}^*$ such that

$$(\boldsymbol{w}^*)^T\boldsymbol{\psi}(\pi^*) \geq (\boldsymbol{w}^*)^T\boldsymbol{\psi}(\pi)$$

holds for all policies π. This solves the *limited data* challenge as the expectation $\boldsymbol{\psi}(\pi^*)$ can be estimated from the sampled trajectories (the distributions). The amount of data obviously scales with the number of features in the reward function, but does not depend on the size of the state space of the complexity of the experts policy!

Addressing the "Ill-Posed" Challenge: (Structured) Max. Margin Solution

To deal with the challenge that the problem is ill-posed, it is possible to enforce a predefined reward margin of 1 in the optimization problem. The optimization problem can be written as

$$\begin{aligned} &\min_{\boldsymbol{w}} \boldsymbol{w}^T \boldsymbol{w} \\ &\text{s.t.} \quad \forall \pi : \boldsymbol{w}^T \boldsymbol{\psi}(\pi^*) \geq \boldsymbol{w}^T \boldsymbol{\psi}(\pi) + 1 \end{aligned}$$

where the optimization objective serves as regularization of the parameters. This enforced margin does not change the problem much as it can be absorbed in the weights/features (e.g. by introducing a bias feature), but does not allow $\boldsymbol{w} = \mathbf{0}$ as solution anymore. This is called the *max. margin* solution.

A more sophisticated approach is to use a metric $m(\pi^*, \pi)$ to measure the distance between two policies (e.g. the sum of minimum distances from the path generated by π^* for the path generated by π) and use this metric as the minimum reward margin:

$$\begin{aligned} &\min_{\boldsymbol{w}} \boldsymbol{w}^T \boldsymbol{w} \\ &\text{s.t.} \quad \forall \pi : \boldsymbol{w}^T \boldsymbol{\psi}(\pi^*) \geq \boldsymbol{w}^T \boldsymbol{\psi}(\pi) + m(\pi^*, \pi) \end{aligned}$$

The justification for this is that the reward margin should be larger for policies that are very different for π^*.

Addressing the "Expert Suboptimality" Challenge: Slack Variables

Instead of enforcing all constraints, it is possible to add slack variables ξ to the inequality constraints to allow them to be violate a bit:

$$\begin{aligned} &\min_{\boldsymbol{w}, \xi} \boldsymbol{w}^T \boldsymbol{w} + C\xi \\ &\text{s.t.} \quad \forall \pi : \boldsymbol{w}^T \boldsymbol{\psi}(\pi^*) \geq \boldsymbol{w}^T \boldsymbol{\psi}(\pi) + m(\pi^*, \pi) - \xi \end{aligned}$$

The altering of the objective ensures that the slack variables do not get too high which could lead allowing improper solutions. This resolved the challenge of expert suboptimality.

Addressing the "Computation" Challenge: Constraint Generation

One method of dealing with the too much constraints is *constraint generation*. Instead of using all policies π as constraints, only a certain set Π of policies is used as constraints. Policies are added iteratively (with the iteration variable k) to this set by computing the most-violated constraint

$$\pi^{(k)} = \arg\max_{\pi} \boldsymbol{w}^{(k),T} \boldsymbol{\psi}(\pi) + m(\pi^*, \pi).$$

If no violations have been found, the iteration stops.

12.2.2. Feature Matching by Max. Entropy

The classical approach to statistical modeling is the principle of maximum entropy. The premise is that modeling should be performed with the least commitment possible, i.e. using the distribution with the maximum entropy (subject to certain constraints).

Max. Entropy Approach to Inference

Given the first- and second-order moments m_1 and m_2, the distribution with the maximum entropy is given as the solution of

$$\begin{aligned} \max_p \int p(x)\big[-\ln p(x)\big]\,\mathrm{d}x \\ \text{s.t.} \quad \int p(x)\,\mathrm{d}x = 1 \\ \int p(x)x\,\mathrm{d}x = m_1 \\ \int p(x)x^2\,\mathrm{d}x = m_2 \end{aligned}$$

where the first constraint specifies that $p(x)$ is actually a distribution. The solution to this problem is

$$p(x) \propto \exp\big\{\lambda_1 x + \lambda_2 x^2\big\}$$

with the Lagrangian multipliers $\lambda_{1,2}$. This is a Gaussian distribution! So the Gaussian is the max. entropy distribution with given mean and variance.

Max. Entropy Approach to Inverse RL

Similarly to inference, the max. entropy approach can be applied to inverse RL with a trajectory distribution $p(\boldsymbol{\tau})$ and certain features $\boldsymbol{\psi}(\boldsymbol{\tau})$ that have to match $\boldsymbol{\psi}(\pi^*)$. The optimization problem is

$$\begin{aligned} \max_p \int p(\boldsymbol{\tau})\big[-\ln p(\boldsymbol{\tau})\big]\,\mathrm{d}\boldsymbol{\tau} \\ \text{s.t.} \quad \int p(\boldsymbol{\tau})\,\mathrm{d}\boldsymbol{\tau} = 1 \\ \int p(\boldsymbol{\tau})\boldsymbol{\psi}(\boldsymbol{\tau})\,\mathrm{d}\boldsymbol{\tau} = \boldsymbol{\psi}(\pi^*) \end{aligned}$$

and has the solution

$$p(\boldsymbol{\tau}) \propto \exp\big\{\boldsymbol{w}^T\boldsymbol{\psi}(\boldsymbol{\tau})\big\}$$

with the Lagrangian multipliers $\boldsymbol{w}$. This is a softmax distribution over the trajectories with the return $r(\boldsymbol{\tau}) = \boldsymbol{w}^T\boldsymbol{\psi}(\boldsymbol{\tau})$. The problem is that the optimization does not take the transition dynamics into account, i.e. there might be trajectories with a huge return that are unlikely due to the dynamics.

Maximum-Casual-Entropy Inverse RL

In contrast for "normal" max. entropy for inverse RL, *maximum-causal-entropy inverse RL* also includes the state dynamics as a constraint:

$$\begin{aligned} \max_\pi \sum_t \iint \mu_t(\boldsymbol{s})\pi_t(\boldsymbol{a}\,|\,\boldsymbol{s}) \ln \pi_t(\boldsymbol{a}\,|\,\boldsymbol{s})\,\mathrm{d}\boldsymbol{s}\,\mathrm{d}\boldsymbol{a} \\ \text{s.t.} \quad \forall \boldsymbol{s} : \int \pi(\boldsymbol{a}\,|\,\boldsymbol{s}) = 1 \\ \sum_t \int \mu_t(\boldsymbol{s})\boldsymbol{\phi}(\boldsymbol{s}) = \boldsymbol{\psi}(\pi^*) \\ \iint \mu_{t-1}(\boldsymbol{s})\pi_{t-1}(\boldsymbol{a}\,|\,\boldsymbol{s})p(\boldsymbol{s}'\,|\,\boldsymbol{s},\boldsymbol{a})\,\mathrm{d}\boldsymbol{s}\,\mathrm{d}\boldsymbol{a} = \mu_t(\boldsymbol{s}'_t) \end{aligned}$$

As usual, the first constraint specifies that π is actual a distribution. The second and third constraint specify that the features are matched and that the transition dynamics distribution is consistent. The solution to this optimization problem is

$$\pi_t(\boldsymbol{a} \mid \boldsymbol{s}) \propto \exp\{\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{s}) + \mathbb{E}[V_{t+1}(\boldsymbol{s}') \mid \boldsymbol{s}, \boldsymbol{a}]\}$$

where $\boldsymbol{w}$ and $V_t(\boldsymbol{s})$ are Lagrangian multipliers. If $r(\boldsymbol{s}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{s})$ is the reward, then this is a softmax over the Q-function:

$$\pi_t(\boldsymbol{a} \mid \boldsymbol{s}) \propto \exp\{Q(\boldsymbol{s}, \boldsymbol{a}; \boldsymbol{w})\}$$

As this is still a convex problem, the solution can be obtained by solving the dual relatively efficiently.

12.2.3. Reward-Parameterized Policies

Alternatively, it is also possible to assume that the policy is a softmax over the Q-function

$$\pi_t(\boldsymbol{a} \mid \boldsymbol{s}; r, \alpha) \propto \exp\{\alpha Q^*(\boldsymbol{s}, \boldsymbol{a}; r)\}$$

where Q^* is the optimal Q-function for the reward $r(\boldsymbol{s})$. Then the likelihood of a set of state-action pairs can be evaluated as

$$\log p(\mathcal{D} \mid r, \alpha) = \sum_i \log \pi(\boldsymbol{a}_i \mid \boldsymbol{s}_i; r, \alpha).$$

This is like a "smarter" approach to behavioral cloning. In fact, it can be shown that this is equivalent. It can also be extended to a Bayesian setup by introducing a prior $p(r, \alpha)$:

$$p(r, \alpha \mid \mathcal{D}) \propto p(\mathcal{D} \mid r, \alpha) p(r, \alpha)$$

12.3. Behavioral Cloning vs. Inverse RL

Behavioral cloning is

- simple to implement
- imposes no assumptions on the model, but
- it might not be possible to reproduce the long-term behavior.

For representation a *policy* is used which is hard to generalize and that needs many samples.

Inverse RL is

- harder to implement and
- requires solving a Markov decision process which is hard for many interesting MDPs.
- But if successful, reproducing the long-term behavior is possible.

For representation the reward function is used which is a compact description and easily transfers to new tasks.

13. Bayesian Reinforcement Learning

This chapter covers how to apply Bayesian methods to reinforcement learning. If applicable, they provide a principled way to handle the exploration-exploitation tradeoff by introducing (epistemic) uncertainty on the model or the policy. This can also be seen as regularization on greedy policy updates avoiding aggressive model exploitation. Active learning can also be used for guided exploration with information-theoretic objectives. Bayesian RL can also be used to create risk-sensitive and robust controllers as risk as well as perturbations of certain parameters and be encoded as uncertainty in the model. For MBRL, this was already briefly covered in section 9.3.

The major problem with Bayesian methods is that they are really complicated. In Bayesian linear regression, for example, the distribution over the output variables is given as the marginalization over the weights

$$p(\boldsymbol{y} \mid \boldsymbol{x}, \mathcal{D}) = \int p(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{w}, \mathcal{D}) p(\boldsymbol{w} \mid \mathcal{D}) \, \mathrm{d}\boldsymbol{w}$$

with the dataset $\mathcal{D}$ and the posterior

$$p(\boldsymbol{w} \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \boldsymbol{w}) p(\boldsymbol{w})}{\int p(\mathcal{D} \mid \boldsymbol{w}') p(\boldsymbol{w}') \, \mathrm{d}\boldsymbol{w}'}.$$

These two integrals are often hard to solve as they involve integrating over all dimensions of $\boldsymbol{w}$ and over nonlinear distributions. This motivates the concept of *conjugate priors*: a prior $p(\boldsymbol{w})$ is *conjugate* to the likelihood $p(\mathcal{D} \mid \boldsymbol{w})$, if both the prior and the posterior belong to the same family of distributions. This allows (per definition) closed form solutions for Bayes rule which has the advantage that an update can be computed in real-time which is important to execute policies online. Some likelihoods and their respective conjugate priors are shown in Table 13.1.

Take a Bernoulli likelihood $\text{Bernoulli}(y \mid p) = p^y (1-p)^{1-y}$ with $y \in \{0, 1\}$ for example. The conjugate prior is a Beta distribution $\text{Beta}(p; \alpha, \beta)$ distribution with the hyperparameters (α, β). The posterior then is the updated Beta distribution $p(p \mid y) = \text{Beta}(p; \alpha + y, \beta - y + 1)$.

But even when using conjugate priors, being Bayesian is still hard: most interesting Bayesian models include nonlinear transformations requiring approximate inference (e.g. a Gaussian process has no closed-form solution for regression due to the required nonlinear transformation into categorical labels). This problem has already been discussed in chapter 6.

Likelihood $p(\mathcal{D} \mid \boldsymbol{w})$	Parameter $\boldsymbol{w}$	Conjugate Prior $p(\boldsymbol{w})$
Bernoulli	p	Beta
Poisson	λ	Gamma
Gaussian	μ	Gaussian
Gaussian	σ^2	Inverse Gamma

Table 13.1.: Examples for Conjugate Likelihood-Prior-Pairs

13.1. Bandits and Thompson Sampling

Bandits are (due to their simplicity) great settings for exploring Bayesian policies and the exploration-exploitation tradeoff. They are essentially stateless MDPs with a reward distribution $p(r \mid a)$ that can be used to model various environments like targeted advertising. The optimization problem is. A k-armed bandit is a bandit with k actions $a_1, \cdots, a_k$. The idea of *Thompson sampling* is to build a model $p(r \mid a)$ of the reward and use it decision making. In Thompson sampling, the actions are sampled from a *belief*. While it was originally not motivated for a Bayesian setting, it can be used to perform Bayesian RL.

13.1.1. Restaurant Selection

Consider three restaurants a_1, a_2, and a_3 and the binary reward $r \in \{0, 1\}$ where 1 encodes that the meal was delicious and 0 encodes that it was not. Let the reward be

$$p(r = 1 \mid a) = \begin{cases} 0.05 & \text{for } a = a_1 \\ 0.1 & \text{for } a = a_2 \\ 0.85 & \text{for } a = a_3 \end{cases}$$

so every restaurant is capable of serving delicious meals, but with different odds. Three strategies for maximizing the reward are now:

1. Choose randomly every day.
2. Build an empirical model of $p(r \mid a)$ and use ϵ-greedy sampling.
3. Build a Bayesian model of $p(r \mid a)$ and sample from the posterior $p(a \mid r)$.

Obviously the first approach will not maximize the reward. This section focuses on the third approach. For the model for $p(r \mid a)$, simply calculate the frequencies of the past rewards for each restaurant – this yields a binomial likelihood distribution. Then put a Prior $\text{Beta}(p_a; \alpha_a, \beta_a)$ on the probabilities p_a for each restaurant. With $\alpha_a = \beta_a = 2$, the prior has a mean and mode of 0.5 which facilitates exploration. The posterior $p(a \mid r) = \text{Beta}(a \mid r; \tilde{\alpha}_a, \tilde{\beta}_a)$ can then be computed in closed form as the Beta distribution is a conjugate prior for the Binomial likelihood. The new parameters are

$$\tilde{\alpha}_a = \alpha_a + \sum_{i=1}^{n_a} r_a^{[i]} \qquad \tilde{\beta}_a = \beta_a + n_a - \sum_{i=1}^{n_a} r_a^{[i]}$$

where (α_a, β_a) are the parameters of the prior, n_a are the frequencies and $r_a^{[i]}$ are the reward for each restaurant a. In practice, it is easier to compute the joint distribution for each restaurant and marginalize as needed.

This Bayesian approach does not require tuning any hyperparameters and converges to the optimal policy! While ϵ-greedy methods can be tuned to have faster convergence, it is suboptimal due to constant exploration. To achieve the same optimality, more sophisticated ϵ-greedy methods are needed where the exploration rate decreases over time (e.g. linear annealed exploration).

13.2. Model-Based Bayesian RL for Discrete MDPs

In model-based Bayesian RL, an MDP with unknown model parameters (the transition and reward functions) are solved. This section will only give a shallow overview on Bayesian MBRL for discrete MDPs and for

exploration and exploitation work together to find an optimal policy. An MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, r, P)$ is build from the discrete state space $\mathcal{S}$, the discrete action space $\mathcal{A}$, the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and the transition probabilities $P : \mathcal{S} \times \mathcal{A} \to [0, 1]$ as a distribution over the states. If P is known, this is a rather easy problem where the value function and the optimal policy can be found using dynamic programming. In the RL setting, P is unknown, but it can be parameterized by $\boldsymbol{\theta}$: $P(s' \mid s, a; \boldsymbol{\theta})$. For Bayesian RL, the state space is expanded with a *belief* distribution $b(\boldsymbol{\theta})$ over the transition parameters. The modified MDP – a Bayes Adaptive MDP (BAMDP) – has the new state space $\mathcal{S} \times \mathcal{B}$ where $\mathcal{S}$ are the MDP states and $\mathcal{B}$ are the belief states. The transition distribution then becomes $P(s', b' \mid s, b, a)$ and is known! As P is known, the MDP becomes a "simple" planning problem with a value function $V : \mathcal{S} \times \mathcal{B} \to \mathbb{R}$.

The two remaining questions on BAMDPs are:

- What exactly is a belief $b(\boldsymbol{\theta})$?
- Why is the transition distribution $P(s', b' \mid s, b, a)$ known?

13.2.1. Belief

The belief $b(\boldsymbol{\theta})$ is a distribution over the parameters $\boldsymbol{\theta}$ of the transition model. For discrete states, the transition distribution is simply a multinomial and the parameters are the transition probabilities

$$\theta_{s,a,s'} = P(s' \mid s, a) \in [0, 1].$$

As $\boldsymbol{\theta}$ groups all the transitions, its dimensionality is $|\mathcal{S}|^2 \cdot |\mathcal{A}|$. The distribution $b(\boldsymbol{\theta})$ can be chosen freely, but as the transition distribution is a multinomial, it is wise to choose the conjugate prior, a Dirichlet distribution, for the belief. A Dirichlet distribution $\mathrm{Dir}(\boldsymbol{\alpha})$ is parameterized by $\alpha_i > 0$, $i = 1, \cdots, k$, $k \geq 2$ and the probability density function is

$$\mathrm{Dir}(\boldsymbol{\theta}; \boldsymbol{\alpha}) \propto \prod_{s,a,s'} \theta_{s,a,s'}^{\alpha_{s,a,s'}-1}.$$

When choosing a Dirichlet distribution as the prior for $\boldsymbol{\theta}$, i.e. the belief $b(\boldsymbol{\theta}) = \mathrm{Dir}(\boldsymbol{\theta}; \boldsymbol{\alpha})$, the parameters $\boldsymbol{\alpha}$ are hyperparameters.

13.2.2. State Transition Model

The state transition model $P(s', b' \mid s, b, a)$ decomposes into a belief and a physical model:

$$P(s', b' \mid s, b, a) = \underbrace{P(b' \mid s, b, a, s')}_{\text{Belief Model}} \underbrace{P(s' \mid s, b, a)}_{\text{Physical Model}}$$

The physical model is known as the parameters $\boldsymbol{\theta}$ of the transition model $P(s' \mid s, a; \boldsymbol{\theta})$ can be integrated out over the belief $b(\boldsymbol{\theta})$:

$$P(s' \mid s, b, a) = \mathbb{E}_{b(\boldsymbol{\theta})}\big[P(s' \mid s, a; \boldsymbol{\theta})\big] = \int b(\boldsymbol{\theta}) P(s' \mid s, a; \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}$$

This leaves the belief model to be unknown. But as $b(\boldsymbol{\theta})$ is a conjugate prior for the likelihood $P(s' \mid s, a; \boldsymbol{\theta})$, it is possible to just set the believe model to one if the next belief equals the posterior belief:

$$P(b' \mid s, b, a, s') = \begin{cases} 1 & \text{if } b'(\boldsymbol{\theta}) = b(\boldsymbol{\theta} \mid s, a, s') \\ 0 & \text{otherwise} \end{cases}$$

13.2.3. Optimal Value Function for BAMDPs

The Bellman equation for BAMDPs is

$$V^*(s,b) = \max_a \Big(r(s,a) + \gamma \sum_{\substack{s' \in \mathcal{S} \\ b' \in \mathcal{B}}} P(s' \mid s,b,a) V^*(s',b')\Big),$$

for which all components are known decently! Hence, the BE can be solved using dynamic programming. This maximization selects actions that maximize the reward and actions that further explore the environment in a natural way by incorporating the belief in the objective. If only the current belief b would be used in the maximization, pure exploitation would be done as the change in the belief would be taken into account.

The BAMDP can be recast as a POMDP with the belief as the hidden variable. In practice, these problems do not scale well with the number of states, making direct implementations infeasible for most problems. Instead different ways are used to approximate the value function and solving the Bellman equation, for example in the BEETLE algorithm[1].

13.3. Continuous MDPs and Dual Control

In continuous optimal control, the idea of trading off exploration and exploitation in a principled way is older than for discrete MDPs. Active learning was already covered in chapter 7 for optimal exploration, and this section covers the combination of optimal control and optimal exploration: *dual control*. Learning from sequential interactions is also known as *continual learning*. Similar to discrete MDPs, the dynamics model $p(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t)$ is parameterized, making it possible to put a prior on the parameters $\boldsymbol{w}$:

$$p(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t) = \int p(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{w}) p(\boldsymbol{w}) \, \mathrm{d}\boldsymbol{w}$$

By also adding a prior $p(\boldsymbol{s}_t)$ on the state, it is possible to integrate the state out too (where $\boldsymbol{z}_t \coloneqq (\boldsymbol{s}_t, \boldsymbol{w})$):

$$p(\boldsymbol{s}_{t+1}) = \iint p(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{w}) p(\boldsymbol{s}_t) p(\boldsymbol{w}) \, \mathrm{d}\boldsymbol{s}_t \, \mathrm{d}\boldsymbol{w} \doteq \int p(\boldsymbol{s}_{t+1} \mid \boldsymbol{z}_t, \boldsymbol{a}_t) p(\boldsymbol{z}_t) \, \mathrm{d}\boldsymbol{z}_t$$

This way incremental learning can be incorporated into the dynamics model

$$p(\boldsymbol{w} \mid \boldsymbol{s}_{t+1}) \propto p(\boldsymbol{s}_{t+1} \mid \boldsymbol{w}) p(\boldsymbol{w})$$

where now $\boldsymbol{w}$ changes with the state, so it becomes time-dependent too: $\boldsymbol{w} \to \boldsymbol{w}_t$, $\boldsymbol{z}_t = (\boldsymbol{s}_t, \boldsymbol{w})$. Maximizing the reward whilst anticipating the future $p(\boldsymbol{w}_t)$ then allows optimal exploration. Unfortunately, dual control is rather intractable…

13.3.1. One-Dimensional Linear Gaussian Dual Control

Assume a one-dimensional stationary LQR problem with the transition distribution

$$p(s_{t+1} \mid s_t, a_t) = \mathcal{N}\big(s_{t+1} \,\big|\, As_t + Ba_t, \, \sigma_\eta^2\big)$$

[1] Poupart et al.: "An Analytic Solution to Discrete Bayesian Reinforcement Learning" (2006)

and the reward function[2] $r(s_{t+1}, a_t) = -s_{t+1}^2 - Ha_t^2$. The optimal one-step action can be derived by maximizing $r(s_{t+1}, a_t)$:

$$\begin{aligned}\frac{\partial \mathbb{E}_{s_{t+1}}\left[r(s_{t+1}, a_t) \,\middle|\, a_t\right]}{\partial a_t} &= -\frac{\partial}{\partial a_t}\left(\mathbb{E}_{s_{t+1}}\left[s_{t+1}^2 \,\middle|\, a_t\right] + Ha_t^2\right) = -\frac{\partial}{\partial a_t}\left(\left(As_t + Ba_t\right)^2 + \sigma_\eta^2 + Ha_t^2\right) \\ &= -\frac{\partial}{\partial a_t}\left(\left(As_t + Ba_t\right)^2 + Ha_t^2\right) \propto (As_t + Ba_t)B + Ha_t \overset{!}{=} 0 \\ \implies a_t^* &= -\frac{ABs_t}{H + B^2}\end{aligned}$$

Considering a Bayesian time-varying uncertainty model $B_t \sim \mathcal{N}(\mu_{B,t}, \sigma_{B,t}^2)$, the "certainty equivalent" optimal one-step control is:

$$\begin{aligned}\frac{\partial \mathbb{E}_{s_{t+1}, B_t}\left[r(s_{t+1}, a_t) \,\middle|\, a_t\right]}{\partial a_t} &= -\frac{\partial}{\partial a_t}\left(\mathbb{E}_{s_{t+1}, B_t}\left[s_{t+1}^2 \,\middle|\, a_t\right] + Ha_t^2\right) \\ &= -\frac{\partial}{\partial a_t}\left(\mathbb{E}_{\eta, B_t}\left[(As_t + B_t a_t + \eta)^2 \,\middle|\, a_t\right] + Ha_t^2\right) \\ &\overset{(3)}{=} -\frac{\partial}{\partial a_t}\left(\left(As_t + \mu_{B,t}a_t\right)^2 + \sigma_{B,t}^2 a_t^2 + \sigma_\eta^2 + Ha_t^2\right) \\ &\propto (As_t + \mu_{B,t}a_t)\mu_{B,t} + \sigma_{B,t}^2 a_t + Ha_t \overset{!}{=} 0 \\ \implies a_t^{\text{CE}} &= -\frac{A\mu_{B,t}s_t}{H + \mu_{B,t}^2 + \sigma_{B,t}^2}\end{aligned}$$

This result has the interesting property that as the uncertainty $\sigma_{B,t}^2$ increases, the control input decreases! This is the uncertainty-based regularization that was introduced earlier on and is usually known as the "caution" or "turn-off phenomenon".

13.3.2. Practical Dual Control

Achieving dual control in practice is an open research topic due to the quasi-intractable nature of dual control. One current approaches are to perform certainty equivalence actions while computing the local uncertainties in an inner loop, and approximating dual control in an outer loop (in a MPC fashion). Another approach is to relax the continual learning requirement by focusing on the episodic setting, which is closer to a planning problem with active learning.

13.4. Wrap-Up

This chapter covered Bayesian reinforcement learning methods which have a lot to offer, especially regarding exploration, regularization and robustness of policies. But while theoretically appealing, these methods are hard to implement in practice. In a discrete setting, the MDP can be modified into an BAMDP, a kind of POMDP, with a belief over the model parameters. This makes it possible to solve the MDP in a Bayesian

[2] The reward gain R was omitted as a one-dimensional reward function can be normalized w.r.t. R by dividing both errors by R.

[3] The expectation can be computed by considering the joint distribution $\mathcal{N}(\boldsymbol{z} \,|\, \boldsymbol{\mu}_z, \boldsymbol{\Sigma}_z)$ with $\boldsymbol{z} \coloneqq \begin{bmatrix} B_t & \eta \end{bmatrix}^T$, $\boldsymbol{\mu}_z \coloneqq \begin{bmatrix} \mu_{B,t} & 0 \end{bmatrix}^T$ and $\boldsymbol{\Sigma} = \operatorname{diag}\left(\sigma_{B,t}^2, \sigma_\eta^2\right)$ and then computing the expectation $\mathbb{E}\left[(\boldsymbol{z}-\boldsymbol{s})^T \boldsymbol{M} (\boldsymbol{z}-\boldsymbol{s})\right]$ with $\boldsymbol{s} \coloneqq \begin{bmatrix} 0 & -As_t \end{bmatrix}^T$ and $\boldsymbol{M} \coloneqq \begin{bmatrix} a_t^2 & a \\ a & 1 \end{bmatrix}$ using equation (380) from the matrix cookbook.

fashion, but is computationally very expensive. In continuous control, the problem of exploring and planning simultaneously is older and is known as dual control. But this can not be achieved in closed form…

A common method for Bayesian exploration is Thompson sampling which samples from the posterior policy to achieve said optimal exploration.

14. Outlook

This "summary" covered lots of topics in the intersection of robotics and machine learning. Separated by chapter, the key terms that should now be known are:

Robotics	Kinematics, Dynamics, DH Parameters, Inverse Kinematics, Feedback, Feedforward, PID Control, Splines, Task-Space Control, Null Space, Singularities
Machine Learning	Bayes Rule, Regression, Classification, Maximum Likelihood, Bias, Variance, Gaussian Processes, Neural Networks, Backpropagation, Gradient Estimation
Optimal Control	Dynamic Programming, Bellman Equation, Policy Iteration, Value Iteration, Policy Evaluation, Linear Quadratic Regulator, Riccati Equation
Approximate OC	Differential Dynamic Programming, Gauss-Newton Approximation, Line Search, Approximate Dynamic Programming
State Estimation	
Model Learning	White-Box, Black-Box, Gray-Box, System Identification, Impulse Response, Active Learning, Linear Gaussian Dynamical System, Differentiable Physics
Policy Representations	Out-of-the-Box Policies, Dynamical Systems, Movement Primitives, Stability, Stochastic Dynamical Systems
Model-Based RL	Sample Efficiency, Optimism, Aleatoric Uncertainty, Epistemic Uncertainty, Model-Predictive Control
Value Function Methods	Temporal Difference, Q-Learning, SARSA, Batch RL, Least Squares Temporal Differences (LSTD), Fitted Q-Iteration
Policy Gradient	Step-Based, Episode-Based, Finite Differences, Likelihood Ratio, Baselines, REINFORCE, Fisher Information Matrix, Natural Gradient
Probabilistic Policy Search	Weighted Maximum Likelihood, Relative Entropy, Contexts, Hierarchy
Imitation Learning	Distribution Matching, Covariate Shift, Behavioral Cloning, Inverse Reinforcement Learning
Bayesian RL	Bandits, Thompson Sampling, Belief Space, Dual Control, Continual Learning, Caution, Turn-Off Phenomenon, Certainty Equivalence

But still, there are a lot of open research questions, for example:

- How to derive robot learning algorithms with guarantees (e.g. convergence, stability, optimality)?
- How to make interpretable robot learning algorithms (explainable AI)?

- How to combine and balance model-free and model-based methods?
- How to incorporate robustness into machine learning methods?
- What are good inductive biases and prior?
- What are good tasks for benchmarking robot learning algorithms?
- Will dual control ever be tractable?

A. Self-Test Questions

The text below also contains answers for the self-test questions! Make sure to not spoiler you!

A.1. Robotics

How to compute the racket position, orientation and velocity in a game of table tennis?

Answer The position and orientation can directly be computed using the forward kinematics model, e.g. using the Denavit-Hartenberg convention. To compute the velocity of the racket, the forward kinematics model has to be differentiated w.r.t. to the time. By using the chain rule, only the Jacobian of the model has to be computed w.r.t. the joint displacements. Multiplying this with the joint velocities gives the racket velocities.

What is an inverse dynamics model? What is a forward dynamics model?

Answer The inverse dynamics model computes the joint torques/forces given the respective accelerations. The forward dynamics model computes the accelerations from the torques/forces.

What kind of models are needed to build a robot simulator?

Answer The forward kinematics and dynamics models are needed. The latter is first used to compute the accelerations and then, after integrating the accelerations numerically two times, the former can be used to animate the robot.

How to represent trajectories in such a way that they can be tracked?

Answer Trajectories have to be at least once, better twice, continuous differentiable, to avoid jumps in the positions and velocities and possibly the accelerations. This can be achieved by modeling a trajectory as a cubic or quintic spline across given via-points. These via-points represent the support points of the spline.

What does feedback control mean?

Answer In feedback control, the actual state of the system is used to compute the control inputs. This allows for error correction if the robot does not behave exactly like the model predicts, for example.

What control laws are common for robots?

Answer It is common to use PD-controllers with gravity compensation and PID-controllers as well as model-based feedback and feedforward controllers. But the model-based controllers need a really good model.

What is model-based feedback control?

Answer In model-based feedback control, a reference acceleration is computed using a PD-controller that assesses the position, velocity and acceleration of the joints. This reference acceleration is then fed into the inverse dynamics model, giving the joint torques/forces that are then applied to the joints.

How can be inverse kinematics be computed?

Answer It is sometimes possible to compute the inverse kinematics analytically. If this is not possible, it might be possible to compute them numerically, e.g. with the Newton method. However, it is better to use the inverse differential kinematics model to compute the velocities of the joints and then integrate them to recover the positions. For square Jacobians this is possible straightforwardly, for non-square Jacobians numerical methods have to be used.

What is task-space control?

Answer In task-space control, the trajectory is planned in the task-space rather than in the joint-space. Then the task-space data has to be converted into the joint-space to then apply joint-space controllers like the PID-controller. Common methods are for example the Jacobian transpose method and the Jacobian pseudo-inverse method.

KI-Campus: Given the joint state of a robot, which model is used to compute the end-effector position?

Answer Using the forward kinematics model.

KI-Campus: Given the joint state of a robot, which model is used to compute the torques/forces applied by the physics?

Answer Using the inverse dynamics model.

KI-Campus: Given the desired end-effector state of a robot, which model is used to compute the joint positions to achieve it?

Answer Using the inverse kinematics model.

KI-Campus: How to compute the forward kinematics?

Answer The forward kinematics can be computed straightforwardly, e.g. by using the Denavit-Hartenberg convention and the respective homogeneous transformation matrices. They can also be computed by simple geometric observations in some cases.

KI-Campus: What are the limitations of the P-controller?

Answer It oscillates around the desired position and does not include velocity-control.

KI-Campus: How can model-based control deal with mismatches between the real system and the model?

Answer Using feedforward control, a model-based controller is combined with a "standard" PD-controller to eradicate modeling errors.

KI-Campus: How to compute the analytical solution for inverse kinematics?

Answer This can be done by inverting the forward kinematics or by geometric observations in the system. This is, however, rather tedious and not always possible.

KI-Campus: What are a few examples in which null-space control would make sense.

Answer For example for saving energy by being in rest postures in a redundant robot. In a redundant prismatic robot, this may be that no joint is fully stretched but all joint are located around the center.

A.2. Machine Learning Foundations

Why does statistics matter to machine learning?

Answer The real world is often not deterministic, hence it has to be necessary to model stochastic processes. Additionally, a probabilistic treatment of the models allows to quantify the uncertainty, making risk-aware predictions possible.

What are the three branches of machine learning?

Answer Supervised Learning, Unsupervised Learning, Reinforcement Learning – and various combinations.

How to derive linear regression? What is ridge regression?

Answer By minimizing the MSE between the predictions and the targets. Ridge regression is regularized linear regression where the parameters are also part of the objective to keep them small.

How do priors change the solution?

Answer Priors have the effect of regularizing the solution in a principled way. For linear regression, placing a Gaussian prior on the parameters yields the same solution as empirical ridge regression.

What is maximum a-priori? What is maximum likelihood? What is MAP?

Answer For a maximum likelihood estimator, the likelihood $p(\mathcal{D} \mid \boldsymbol{\theta})$ is maximized w.r.t. to the parameters $\boldsymbol{\theta}$. For a maximum a-posteriori estimator (MAP), the posterior $p(\boldsymbol{\theta} \mid \mathcal{D})$ is maximized. This requires placing a prior on the parameters. Maximum a-priori doesn't really make sense.

What is overfitting and how does it relate to the bias-variance tradeoff?

Answer Overfitting describes the effect if the model perfectly resembles the training data but fails to make out-of-data predictions (e.g. on a training dataset). It relates to the bias-variance tradeoff as a model that overfits has a low bias and high variance while an ideal model has low bias and low variance – which is usually not really achievable. The equation for relating the squared error with bias and variance is

$$L_{\hat{f}}(\boldsymbol{x}) = \mathbb{E}_{\mathcal{D}}\left[(y(\boldsymbol{x}) - \hat{f}_{\mathcal{D}}(\boldsymbol{x}))^2\right] = \sigma_\epsilon^2 + \text{Bias}^2\left[\hat{f}_{\mathcal{D}}(\boldsymbol{x})\right] + \text{Var}\left[\hat{f}_{\mathcal{D}}(\boldsymbol{x})\right]$$

where $y(\boldsymbol{x}) = f(\boldsymbol{x}) + \epsilon$ are the real values with noise $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$ and $\hat{f}_{\mathcal{D}}$ is the model learned from dataset $\mathcal{D}$.

How do Frequentists differ from Bayesians?

Answer Frequentists belief that there is a true model / there are true parameters while Bayesians assume parameters to be random variables.

What does non-parametric mean?

Answer A non-parametric model uses every data point as a parameter, so it really has indefinitely many parameters.

What are Kernels? What is Gaussian Process regression?

Answer Kernels are functions that represent inner products of feature transformations. Using the kernel trick (replacing all inner products with kernels), it is also possible to use infinite-dimensional feature transformations implicitly. Gaussian Process regression is a non-parametric Bayesian regression model that uses kernels for the mean and gauges the uncertainty dependent on the input variable.

What are neural networks and how relate them to the brain?

Answer Neural networks are a set of artificial neurons that are interconnected. Each input to a neuron gets weighted and the weighted sum is put through an activation function. They resemble the human brain which also has activation functions and weights the inputs. A neuron sheet in the brain the represented by a layer in the artificial neural network.

How do neural networks build stacks of feature representations?

Answer Using multiple layers. Each layer can be seen as a feature transformation, producing features for the next layer.

If a network with one layer is enough, why is it in practice not a good idea?

Answer The amount of hidden neurons in that layer rises exponentially with the complexity of the function that has to be represented. By using deep networks, the amount of parameters and the computational cost can be reduced.

How to do forward- and backpropagation?

Answer Forwardpropagation is straightforward using matrix multiplications and invoking the activation functions with the weighted vectors. Backpropagation is equivalent to an iterative application of the chain rule for each layer to compute the gradient of a loss function w.r.t. all parameters of the network.

What are different ways of doing fast gradient descent? Full, stochastic, mini-batch? Learning rate adaptation? How to initialize the parameters?

Answer Instead of using all samples for one gradient descent step (full GD), it is also possible to use only some (mini-batch GD) or even just one (stochastic GD) sample. With learning rate adaption (e.g. Adadelta, Adagrad or Adam), it is possible to accelerate learning by using a higher learning rate in flat regions and using a smaller learning rate in steep regions of the loss function. There are multiple ways of initializing the weights, e.g. by drawing them from a Gaussian distribution or using Xavier initialization which samples from a uniform distribution.

Why do neural networks overfit and what to do about it?

Answer Usually neural networks have a lot more parameters than data is available for training, making it easy for the network to memorize the data instead of learning correlations. This can be fought e.g. using dropout (disabling certain connections in the network with a certain probability) or noise augmentation (make the input data noise).

Why are convolutional neural networks used for spatially correlated data?

Answer Convolutional neural networks use convolution filters which are really good in processing spatially correlated data. For example in images in a non-ML setting, convolutions are used to detect edges in the image.

Why are recurrent neural networks used for time series data?

Answer Recurrent neural networks feed the output back as an input just like time series data usually does (e.g. in a dynamical system $\boldsymbol{s}_{t+1} = \boldsymbol{A}\boldsymbol{s}_t$, the output of time step t is feed as an input for time step $t+1$).

A.3. Optimal Control

A.3.1. Discrete Optimal Control

What is an MDP, a policy, a value function, a state-action value function?

Answer An MDP, a Markov decision process, is a system with discrete states and actions that behaves Markovian, i.e. a state only depends on the previous state and the action taken and not on any other states. The initial state is drawn from the initial state distribution. A policy (that can be either deterministic or stochastic) prescribed what action to take given an action. The value function assesses the quality of a state, i.e. it gives the expected long-term reward when following a given policy. The state-action function, also called the Q-function, assesses the quality of state-action pairs, i.e. it gives the expected long-term reward when taking the said action in said state and subsequently following a policy. Maximizing the optimal Q-function yields the optimal policy.

What is policy evaluation, policy improvement, policy iteration and value iteration?

Answer Policy evaluation estimated the (state-action) value function for a given policy by iterating the Bellman equation. Policy improvement takes a state-action value and deduces the policy from it w.r.t. the action for every state. Policy iteration iterates policy evaluation and improvement until convergence to find the optimal policy. Value iteration also finds the optimal policy but iterated the Bellman equation directly.

What is the main difference between policy iteration vs. value iteration?

Answer In value iteration a lot of redundant maximization operations are performed for computing the value function. In policy iteration this is circumvented using the embedded policy evaluation.

What is the Bellman equation? The Bellman equation describes how to compute the (optimal) value function from the (optimal) state-action value function. For infinite horizon problems, it is given as

$$V^*(\boldsymbol{s}) = \max_{\boldsymbol{a}} \left(r(\boldsymbol{s}, \boldsymbol{a}) + \gamma \sum_{\boldsymbol{s}'} p(\boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a}) V^*(\boldsymbol{s}') \right).$$

What are the differences between finite and infinite horizon objectives? Give examples of robotics problems in both settings.

Answer For finite-horizon problems, all of the transition dynamics, reward function and (state-action= value functions and therefore also the policy are time-dependent. Also there is a last reward $r_T(\boldsymbol{s}_T)$ that is independent of the action. This can be interpreted as that it is relevant how much steps are left to make decisions. For infinite-horizon objectives, this is no longer relevant and the time-dependencies is dropped for all components. An infinite-horizon problem is, for example, balancing an inverted pendulum where more reward is gained the longer the pendulum can be held upright. An example for a finite-horizon problem is ball-in-cup, where once the ball is in the cup the problem is no longer interesting.

Why is dynamic programming difficult to apply directly to robotics?

Answer Dynamic programming as discussed in this chapter is only applicable for discrete state-action spaces. This can be achieved by separating the world in buckets, but this would cause an exponential explosion in the memory required to store the value function. Using LQR (next chapter), dynamic programming can also be used for continuous state-action spaces, but this requires the system to be linear and the reward to be quadratic with Gaussian noise. But the world and therefore robots are not linear, but in most cases highly nonlinear.

A.3.2. Continuous Optimal Control

What is the LQR problem?

Answer An LQR problem is given with continuous states and actions with linear state transition dynamics with Gaussian noise, i.e. $p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t) = \mathcal{N}\big(\boldsymbol{x}_{t+1} \,\big|\, \boldsymbol{A}_t \boldsymbol{x}_t + \boldsymbol{B}_t \boldsymbol{u}_t \, \boldsymbol{\Sigma}_t\big)$. The reward function is quadratic in the states and actions, i.e. $r_t(\boldsymbol{x}, \boldsymbol{u}) = -\boldsymbol{x}^T \boldsymbol{R}_t \boldsymbol{x} - \boldsymbol{u}^T \boldsymbol{H}_t \boldsymbol{u}$ with symmetric and positive definite matrices $\boldsymbol{R}_t$ and $\boldsymbol{H}_t$. The optimal policy π^* then maximizes the cumulative reward J_π over a finite horizon T. This is, besides the discrete case, the only solvable optimal control problem.

Derive the LQR value function and optimal policy for the basic case.

Answer This can be done by following the principle of dynamic programming, i.e. first compute the value function for the last time step and subsequently calculate the Q-function, the optimal policy and the value function for the previous time step. This is done by applying the Bellman equation. See subsection 4.2.1 for the full derivation.

What is the form of the solution (value function and policy)? What is their interpretation (qualitatively)?

Answer The value function is of the form $V_t(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{V}_t \boldsymbol{x} + \boldsymbol{x}^T \boldsymbol{v}_t$, i.e. it is quadratic-linear. The optimal control input is of the form $\boldsymbol{u}_t^* = \boldsymbol{K}_t(\boldsymbol{x}_t - \boldsymbol{x}_d) + \boldsymbol{k}_t$, i.e. it is a time-varying P-controller!

What to do when dynamics and/or rewards are not linear? What the pitfalls?

Answer It is possible to linearize the dynamics and/or the rewards using a Taylor expansion and cutting off the higher-order terms. This, however, leads to oscillations and does only work for systems that are not too nonlinear. This also causes the policies to often not work on the real system due to modeling errors.

What are the potential issues when using learning models for trajectory optimization?

Answer One major issue is that the optimizer is prone to exploit error in the model. When using learned models, this might mean to jump out of the space the system learned in, causing unpredictable behaviors. Such policies will most likely never work on the real system as it does not exhibit the same errors.

A.4. Approximate Optimal Control

A.4.1. Approximate Dynamic Programming

What are the problems of dynamic programming?

Answer For large state spaces, dynamic programming becomes infeasible due to the curse of dimensionality. It is not possible to store the value function for all necessary states.

Can function approximators be used?

Answer Yes, by approximating the value function globally.

What is approximate value iteration? What is the performance loss?

Answer Approximate value iteration uses vanilla value/policy iteration to find the optimal value function for a smaller set of states and subsequently fits a global value function approximation on these values. This is repeated until convergence. The performance loss describes the difference of the optimal value function to the value function described by the greedy policy generated by the value function approximation. It is bound by the approximation error of the value function.

What is approximate policy iteration? What is the performance loss?

Answer Approximate policy iteration projects the estimated value function into the parameter space and optimized it there. Again, the performance loss is bound by the approximation error of the value function.

A.4.2. Differential Dynamic Programming

What is the difference between LQR around trajectories and DDP?

Answer In contrast to LQR, DDP is an iterative algorithm that always linearized the dynamics again after executing a set of actions. This way the algorithm converges to a local optima even for nonlinear systems.

How do you compute the quadratic expansion?

Answer The quadratic expansion can be computed analytically, with automatic differentiation or finite differences.

What is iLQR? What terms does it ignore?

Answer Iterative LQR (iLQR) is an approximation of the DDP algorithm which approximates the Hessians of the Q-function by ignoring the Hessians of the state dynamics. This can be handy as computing Hessians is usually computationally expensive which prohibits online usage of DDP.

How does stochastic DDP differ from deterministic DDP?

Answer In stochastic DDP state-action dependent noise is assumed, adding more differential term to the solution. For constant noise, however, the solution is equivalent to deterministic DDP.

What is Guided Policy Search?

Answer Guided policy search uses multiple DDP controllers to find multiple trajectories and subsequently fits a neural network to these trajectories for finding an optimal global controller. This makes the controller less brittle as it can follow more than one trajectory.

A.5. State Estimation

A.6. Model Learning

Why do we need to learn models?

Answer Building models by hand requires a lot of work (e.g. for disassembling and measuring the robot) and can often not cover complex behavior like friction. But not modeling e.g. friction may result in policies that are not transferable to the real robot, or the motion planning to fail. Learning models from data can overcome these issues.

What models exist in robotics?

Answer Forward/inverse kinematics, forward/inverse dynamics, sensor models, discrete-time models, continuous-time models, …

What's a white-box? What's a black-box? What's a gray-box?

Answer In a white-box model, lots of prior domain knowledge is used to build the model. Example are using physics-based approaches like Lagrangian mechanics to find the equations and then utilize system identification to find e.g. the masses. A black-box model is the opposite, where no prior knowledge is used and the complete model is inferred from data, e.g. using a neural network. A gray-box combined both worlds. It used some prior knowledge (like energy conservation), but also uses a black-box model for learning non-modeled behavior (e.g. friction or handling the residual error).

How to excite a system to get good data for model learning?

Answer Use impulse responses for analog signals and step functions for digital signals. As these are not practical (due to high frequencies that may damage the robot), a low-pass filter has to be used. But this again limits the diversity of signals. Another approach is to use active learning for guiding exploration, i.e. letting the model decide where to get new data is also a good option. In practice, out-of-phase sinuses create very diverse end-effector trajectories that are good for model learning.

How to learn a linear Gaussian dynamical system model?

Answer The LGDS system model can be learned using an expectation-maximization algorithm that in the E-step uses a Kalman smoother for finding the expected latent states and subsequently maximized the expected log-likelihood in the M-step. This guarantees monotonic improvement.

How to make black-box models physically safe to learn?

Answer This can be done by impose inductive biases, e.g. energy conservation, on the model. This way it is less likely for the model to be unsafe on the real system. Also for generating data, it is required to low-pass filter the frequencies as high frequencies may cause unsafe movements of the robot.

How can differentiable physics help?

Answer Differentiable physics makes it possible to use black-box models combined with white-box models like the Newton-Euler algorithm. By implementing the algorithm in an autograd engine, the parameters can be trained with gradient decent, making it simple to add a neural network, e.g. for handling the residual error by adding it to the result of the Newton-Euler algorithm.

A.7. Policy Representations

What is a policy?

Answer A policy maps a state $\boldsymbol{s}$ to an action $\boldsymbol{a}$. By using a probabilistic policy $\pi(\boldsymbol{a} \mid \boldsymbol{s})$, lot of difference behaviors like exploration can be encoded.

What off-the-shelf representations can be used for robot learning?

Answer A lot of standard machine learning regression models can be used to represent policies, e.g. neural networks, linear regression with RBF features, Gaussian process, …

How do off-the-self representations compare to trajectory-based representations?

Answer Off-the-shelf policies often do not generalize well which can lead to drastic failures. Also it is hard to encode inductive biases, achieve scalability for a high number of degrees of freedom, combine movements. Sample efficiency is also a problem as off-the-shelf methods usually require lots of training data.

What are potential advantages of movement primitives?

Answer With movement primitives, it is usually easy to learn a specific movement from demonstrations using simple linear regression. As the forcing function vanishes over time, they are stable by design, and they can encode temporal scaling to execute a movement faster/slower as needed. Also they scale really well for a higher number of degrees of freedom.

What are the main ideas of using movement primitives?

Answer The stability of a second-order linear dynamical systems can be easily assured by choosing the eigenvalues accordingly. To produce a moving attractor, a forcing function is added to the dynamical system, causing the goal attractor to move accordingly. As the forcing function is designed to vanish over time, the stability of the system is assured as it becomes a PD-controller for $t \to \infty$.

Why to use dynamical systems? What are advantages and disadvantages?

Answer Dynamical systems are highly expressive in terms of the trajectories they can generate. Also they usually only need a few parameters for generating a variety of different trajectories, exhibiting a desirable parsimonious parameter space. Advantages are that for linear systems, stability analysis is well-studied and simple. For nonlinear dynamics, however, this is different: stability analysis is not well studied and can in most cases only be analyzed using Lyapunov functions which are hard to find. Hence, instability is a real problem with using dynamical systems as trajectory representations!

Why use a probabilistic representation?

Answer Using probabilistic representations enables quantifying the uncertainty on the trajectory, marking how important certain via-points are for executing the trajectory by choosing the variance low. A high variance allows variability in the movement and allows for combining and blending trajectories using probabilistic operators. Also, using conditioning, it is possible to specialize the primitive on visiting a specific state.

How can nonlinear stable dynamics be obtained?

Answer This is possible using an (easily achievable) stable linear systems and combining it with a bijective transformation function. Because of the bijectivity, the potential energy is not changed and the nonlinear system is stable too. Using invertible flows (bijective neural networks), a huge range of different nonlinear transformations are possible, allowing for various nonlinear stable dynamics.

A.8. Model-Based Reinforcement Learning

How does model-based reinforcement learning relate to optimal control?

Answer MBRL is essentially optimal control with learned models.

How does model-free contrast to model-based? Does it really have no models?

Answer In model-based RL, an explicit dynamics model is learned. In model-free RL, the value function (or the policy) is learned directly. But these are also models of the environment and implicitly encode the dynamics. So MFRL does have models!

Is model-based RL generally better?

Answer No. While MBRL can be really sample-efficient and can even directly train on the real system due to this, it suffers in generalization. Outside of already seen regions, MBRL methods are usually brittle. Also they suffer from modeling errors which is discussed in more detail in the next questions.

Why does domain knowledge help model-based RL?

Answer Domain knowledge can be used to encode relevant properties of the environment, for example encoding rotational symmetries like in the pendulum using sine/cosine transformations. This enables better generalization.

Can modeling errors harm model-based RL?

Answer Yes. If a model has errors (which every model has), the optimizer might be able to exploit those errors in its favor. This could lead to robots walking through walls in the simulation, but just accelerating into them in a real environment. Other modeling problems are for example energy generation if some friction coefficient is negative.

What are aleatoric and epistemic uncertainty?

Answer Aleatoric uncertainty describes the noise of a variable. Epistemic uncertainty describes the uncertainty on the model itself, for example the uncertainty on the state dynamics matrix in a LGDS. The latter it often connected to Bayesian models while the former can also be found the frequentist models.

How does model-based RL relate to Model-Predictive Control?

Answer In MPC, a model is used for replanning a action by looking a few steps ahead in time. This enables the robot to adapt to new situations if some previous action was not executed correctly, for example.

What are PILCO and Guided Policy Search?

Answer PILCO uses a Gaussian process dynamics model to quantify epistemic uncertainty and use that uncertainty to regularize the reward.

A.9. Value Function Methods

Why can it be preferable to learn a value function instead of a model?

Answer Learning a model can be very hard, so model-free methods like value functions methods may be more successful. Not using a model can may also be preferable to prevent modeling errors that might get exploited by the optimizer, making the policy not transferable to a real system. Also lots of optimal control methods are based on linearization, which only works moderately for highly nonlinear tasks.

What are the V- and Q-function?

Answer The value function (V-function) describes the expected long-term reward of a state when following a policy. The state-action value function (Q-function) describes the expected long-term reward of a state when executing a given action and subsequently following a policy. The formal definition is as follows:

$$V^{\pi}(\boldsymbol{s}) = \mathbb{E}_{p,\pi}\left[\sum_{t=1}^{\infty}\gamma^{t}r(\boldsymbol{s}_t,\boldsymbol{a}_t)\,\middle|\,\boldsymbol{s}_1=\boldsymbol{s}\right] \qquad Q^{\pi}(\boldsymbol{s},\boldsymbol{a}) = \mathbb{E}_{p,\pi}\left[\sum_{t=1}^{\infty}\gamma^{t}r(\boldsymbol{s}_t,\boldsymbol{a}_t)\,\middle|\,\boldsymbol{s}_1=\boldsymbol{a},\,\boldsymbol{a}_1=\boldsymbol{a}\right]$$

What is TD-learning? How to derive it?

Answer TD-learning uses a step-based update to learn the V- or Q-function. This is done by using a one-step bootstrapping prediction of the value function for a state and computing the difference to the current prediction (the TD-error). With a step size α, the V-/Q-function is then updates in a gradient ascent-like fashion. For discrete states, the V-/Q-function can be tabulated, but for continuous states, it has to be approximated, e.g. using a linear model.

What does on-policy and off-policy mean?

Answer In an on-policy method the action for calculating the one-step prediction is chosen according to the current policy. In an off-policy method the action does not have to follow the policy. An advantage of off-policy methods is that the estimation policy (i.e. the one that is used to get the estimation action) is separated from the behavioral policy (i.e. the policy that is used to get the trajectory). This can be used for exploration in the behavioral policy, while still directly estimating the optimal Q-function using a greedy estimation policy.

What is the difference between Q-Learning and SARSA?

Answer In Q-learning, the estimation action is chosen using a greedy policy while in SARSA, the estimation action is chosen using the current behavioral policy. Q-learning is off-policy, SARSA is on-policy.

How do batch methods work?

Answer Batch methods use multiple samples at once for estimating the V-/Q-function, hence allowing for a better sample efficiency as every sample is used more than once.

How to derive LSTD?

Answer For LSTD, multiple samples are used to analytically minimize the mean-squared error introduced for approximate TD-learning. This leads to an iterative update rule for the weights which depends on the previous weights. As convergence requires a fixed point, plugging this assumption into the least-squares solution yields a solution for LSTD. This is computationally costly, but allows learning in one shot without iterating!

What was the biggest success of value functions in robot learning?

Answer In 2009, Martin Riedmiller won a robot soccer contest using batch reinforcement learning methods which was previously only won by big teams with lots of manpower.

When do value function methods work well?

Answer For small state-action spaces that can be explored fast, value functions work well. But if the state space is high-dimensional and big, the amount of samples needed grows exponentially, making it unsuitable. Also errors in the value function approximation can have catastrophic effects on the policy that are really hard to control.

Why do value function methods often fail for high-dimensional continuous actions?

Answer Value function methods require to maximize the Q-function w.r.t. the action, which is a really hard problem for itself. For continuous actions, only numerical methods can be used which are computationally expensive for high-dimensional data.

A.10. Policy Search

A.10.1. Policy Gradient Methods

When to use episode-based and when to use step-based methods?

Answer Episode-based methods exhibit a higher variance in the total reward as it is the sum of T random variables. For step-based methods, the reward to come can be used that has lower variance as it is only the sum of $T - t + 1$ random variables for the t-th reward. So for a noisy reward, step-based methods are better suited. However, episode-based methods have the advantage that task-specific policies can be used due to the upper- and lower-level policy structure. Step-based methods, on the other hand, provide more data to learn the parameters from as every state-action-reward pair is a data point as opposed to only parameter-reward pairs for episode-based methods.

How do finite difference gradient estimators work?

Answer Finite difference gradient estimators compute the function once for the value in questions and once for a perturbed value. The difference of both is then divided by the perturbation. The method can be derived by Taylor-expanding the function around the value and rearranging the terms in the equation. While this is very simple, it requires evaluation of the objective two times which might be very costly or even problematic if the function is not deterministic.

What are likelihood ratio gradient estimators?

Answer Likelihood ratio gradient estimators use the log-ratio trick that from the derivative of the logarithm, $\frac{\mathrm{d} \ln f(x)}{\mathrm{d}x} = \frac{1}{f(x)} \frac{\mathrm{d}f(x)}{\mathrm{d}x}$, the derivative of $f(x)$ can be written as $\frac{\mathrm{d}f(x)}{\mathrm{d}x} = f(x) \frac{\mathrm{d} \ln f(x)}{\mathrm{d}x}$. If $f(x)$ is a probability density under an integral, the integral can now be approximated using Monte-Carlo integration, i.e. sampling from the density and computing the gradient. This method of estimating a gradient is called the likelihood ratio gradient estimator.

Why do baselines lower the variance of the gradient estimate?

Answer Subtracting a variance can reduce the variance of the gradient estimate while leaving the estimator unbiased. Good baselines are for example the average reward or, for step-based methods, the value function.

Why is the Fisher information matrix so important? How does it relate to the KL divergence?

Answer The Fisher information matrix describes how much influence certain distribution parameters have. This is important as it can be used to make the metric for policy gradient methods invariant to linear transformations of the parameters. The KL divergence can be approximated by using the second-order Taylor-expansion where the Fisher information matrix is used.

What is a natural gradient? Why is natural gradient ascent invariant to reparametrization?

Answer Natural gradients are the ascent direction that is closest to the steepest ascent direction (the gradient) while constraining the policy to not diverge too far. The latter is done by using the Fisher information matrix as an approximation of the KL divergence. This is also the reason why it is invariant to reparametrization which is included by the FIM. It represents the steepest ascent direction in the policy space rather than the parameter space.

What is the compatible function approximation? How is it connected to natural policy gradients?

A.10.2. Probabilistic Policy Search

What is the idea behind success matching?

Answer The idea of success matching is that humans tend to not jump directly to the optimal policy, but rather make this dependent on how frequent an event is. If an event would yields a really high reward but is relatively uncommon, it might not be worth it to optimize the policy to do that event if it costs on other ends.

What is the weighted maximum likelihood solution for Gaussian policies? Gaussian policies are one of the policy types where closed-form solutions exist. With the weights $w^{[i]}$ for each sample $\boldsymbol{\theta}^{[i]}$ and the total weights $W \coloneqq \sum_{i=1}^{N} w^{[i]}$, the update rules for the mean and covariance are:

$$\boldsymbol{\mu}^{\text{new}} = \frac{1}{W} \sum_{i=1}^{N} w^{[i]} \boldsymbol{\theta}^{[i]} \qquad \boldsymbol{\Sigma}^{\text{new}} = \frac{1}{W} \sum_{i=1}^{N} w^{[i]} \left(\boldsymbol{\theta}^{[i]} - \boldsymbol{\mu}^{\text{new}}\right) \left(\boldsymbol{\theta}^{[i]} - \boldsymbol{\mu}^{\text{new}}\right)^{T}$$

What is an exponential transformation? And why and how is it used in the context of policy search?

Answer The exponential transformation $\exp\{\beta R^{[i]}\}$ is used to transform rewards $R^{[i]}$ to an improper probability distribution. In policy search, it is used to compute the weights for the weighted maximum likelihood estimate. The parameter β is the *temperature* of the transformation and has to be hand-tuned.

What is the main contribution on REPS? How is it different from gradient-based methods?

Answer The main contribution of REPS is to constrain the KL divergence of the policy update to not exceed a given threshold ϵ. This eliminates the temperature as a hyperparameter. The temperature is now chosen using an optimization problem and equals the reciprocal of the Lagrangian multiplier for the KL constraint. It differs from gradient-based methods in a way that no step-size has to be chosen.

Why does matching feature averages help in solving the constrained optimization problem in contextual policy search?

Answer Without feature matching, i.e. requiring perfect reproduction of the contextual distribution, the optimization problem has an infinite amount of constraint. Using feature matching with M features, this is reduced to M constraints for distribution reproduction. When using Gaussians, this is even exact for the first- and second-order moments as they are sufficient statistics for a Gaussian distribution.

How does HiREPS ensure learning of different solutions?

Answer HiREPS constraints the entropy of the option distribution. As a high entropy means high overlap, this constraints the overlap in the options, thus enforcing separate solutions.

A.11. Imitation Learning

What are the strengths and weaknesses of behavioral cloning?

Answer Behavioral cloning is simple to implement and imposes no assumptions on the model. However, it generalizes badly and lots of samples are needed. It also suffers from the covariate shift problem and the policies are not really transferable.

What makes DAGGER and DART different?

Answer DAGGER requires a teacher to be present to generate new samples if the robot asks for it. DART on the other hand adds noise to the inputs to achieve better generalization without the need of a teacher.

What is the covariate shift and how does it affect imitation learning?

Answer The covariate shift is the name of the phenomenon that the robot does not know how to recover if it ends up in an unseen state. The effect of this is that also imperfect samples have to be demonstrated for showing the robot how to recover.

What is inverse RL often more suitable than direct imitation learning?

Answer Direct imitation learning requires lots of samples and the teacher might be imperfect. That way, the robot often has problems of surpassing the teacher or generalize to new situations. Inverse RL tackles this by learning the reward function and building an own policy from it under the assumption that the reward function describes the task the best. Also reward functions are a lot more transferable than complete policies.

What are algorithmic challenges in IRL?

Answer

- Data Availability (often only samples and not the expert policy are available)
- Ill-Posed (e.g. the null-reward is a feasible solution)
- Expert Suboptimality
- Computation (too many constraints, impossible to enumerate policies)

What are different methods for IRL?

Answer Some methods for IRL are e.g. (Structured) Max. Margin and Max. Entropy.

Why use the maximum margin approach?

Answer One problem of IRL is that the problem is ill-posed. This can be solved by altering the constraints such that at least a margin of 1 is enforced. This margin can be absorbed into the weights, so it does not change the solution much. However, it makes the null-reward an invalid solution as it violates the constraints.

What is the max. entropy approach?

Answer In max. entropy approaches, the entropy of the policy is maximized while keeping the constraints valid.

A.12. Bayesian Reinforcement Learning

Why is Bayesian RL an interesting problem for robotics? Why is it hard?

Answer Bayesian RL is a principled way of solving the exploration-exploitation tradeoff by imposing epistemic uncertainty on the model. This uncertainty also serves as a regularization on the policy and makes the policy more robust, which is especially important in robotics. It also serves as risk-sensitive control which is important e.g. in hazardous environments. But Bayesian methods are in general hard as they often need approximate inference due to nonlinear transformations (even when using conjugate priors) requiring sampling which is not really feasible in an online fashion.

Why are Bandits so frequently covered in Bayesian RL?

Answer Bandits are really simple MDPs which can model an astonishing amount of tasks, e.g. targeted advertisement. Due to their simplicity, it is a nice setting for analyzing the exploration-exploitation tradeoff and the effect of Bayesian RL methods, i.e. how they do this optimally.

What is Thompson sampling?

Answer In Thompson sampling, exploration is done by sampling from the posterior. The data that was collected in a response to this sampling is then used for updating the likelihood and subsequently the posterior. This converges to the optimal policy (the posterior), but only works for really simple systems.

What makes Bayesian RL different from regular MBRL?

Answer Bayesian RL methods do not necessarily require a dynamics model, and MBRL is not necessarily Bayesian.

What is dual control?

Answer In dual control, a model of the environment is learned in a Bayesian fashion while simultaneously optimizing the actions to be optimal. This is useful as often a baseline controller is needed in order to get the data to learn a model and a model is needed for learning a controller. However, dual control is quasi-intractable in practice.

How does Bayesian RL relate to exploration?

Answer Bayesian RL trades exploration and exploitation off optimally by imposing an epistemic uncertainty quantification on the model. This uncertainty then "tells" the policy where to explore and where exploitation is safe.

B. Mock Exam

Question 1 What is a null space? What is task-prioritization with null space? Give a minimal example.

Answer The *null space* of a robot is the joint space of redundant solutions for a target pose in task space, i.e. all joint configurations that would not violate the constraints of the problem. Task-prioritization with null space uses that space to move the robot in a resting position (e.g. to consume less energy) while remaining in a steady position with the end-effector. A minimal example is the a one-dimensional two-join prismatic robot where the movements of one joint can be compensated by the other. If e.g. a balanced state is less energy-consuming, it would be desirable to prioritize movement that would keep the robot close to this position.

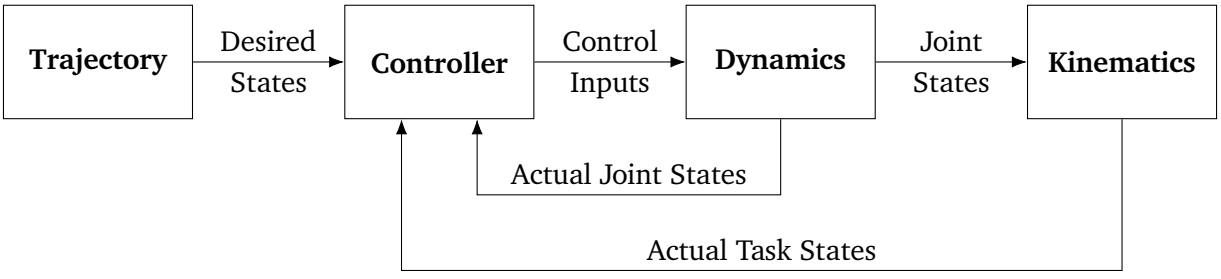
Question 2 Write down and explain the equations for: a) P-controller, b) PD-controller and c) PD + gravity compensation. Intuitively explain what are the limitations/problems of each of these controllers.

Answer Let $\boldsymbol{x}_d/\dot{\boldsymbol{x}}_d$ and $\boldsymbol{x}/\dot{\boldsymbol{x}}$, be the desired and actual position/velocity, respectively.

- The equation of the P controller is $\boldsymbol{u} = \boldsymbol{K}_P(\boldsymbol{x}_d - \boldsymbol{x})$. The control input is computed based on the position error and amplified by the gain $\boldsymbol{K}_P$. This controller has the problem that it does not handle errors in the velocity, causing oscillations around the target.
- The equation of the PD controller is $\boldsymbol{u} = \boldsymbol{K}_P(\boldsymbol{x}_d - \boldsymbol{x}) + \boldsymbol{K}_D(\dot{\boldsymbol{x}}_d - \dot{\boldsymbol{x}})$. In addition to the position error, it also handles errors in the velocity and amplifies them by the gain $\boldsymbol{K}_D$. This controller has the limitation that there remains a steady-state error caused by gravity, i.e. the robot will always remain slightly off target.
- The equation of the PD controller with gravity compensation is $\boldsymbol{u} = \boldsymbol{K}_P(\boldsymbol{x}_d - \boldsymbol{x}) + \boldsymbol{K}_D(\dot{\boldsymbol{x}}_d - \dot{\boldsymbol{x}}) + \boldsymbol{g}(\boldsymbol{x})$, where $\boldsymbol{g}(\boldsymbol{x})$ is a model of the gravitational forces. While this controller can compensate for the steady-state gravitational error, it requires an accurate model of the gravity which might not be available..

Question 3 Make an illustrative sketch that shows and connects the following terms: dynamics, control, trajectory, kinematics, desired states, and actual states. Label all connections.

Answer The following sketch shows the connection of the terms:



Question 4 Write down Bayes theorem for estimating the probability of A given the observation B. Define (i.e. name) the terms.

Answer

$$\underbrace{p(A \mid B)}_{\text{Posterior}} = \frac{\overbrace{p(B \mid A)}^{\text{Likelihood}} \overbrace{p(A)}^{\text{Prior}}}{\underbrace{p(B)}_{\substack{\text{Normalization/} \\ \text{Confidence}}}}$$

Question 5 Describe the main idea behind Bayesian linear regression. Define all the relevant distributions. How do we perform predictions? Why is it useful?

Answer Instead of assuming there are true parameters $\boldsymbol{w}$, Bayesian linear regression puts a prior $p(\boldsymbol{w})$ on the parameters and marginalizes them out as the really important thing in regression are the predictions, not the parameters. This treatment of the weights being random variables is useful to gauge the uncertainty of the model (a high variance in the prediction means large uncertainty). For a dataset $\mathcal{D}$, parameters $\boldsymbol{w}$, the input variable $\boldsymbol{x}$, and the predictive variable y, the posterior predictive distribution is

$$p(y \mid \boldsymbol{x}, \mathcal{D}) = \int p(y \mid \boldsymbol{x}, \boldsymbol{w}) p(\boldsymbol{w} \mid \mathcal{D}) \, \mathrm{d}\boldsymbol{w} \, .$$

This results in $\boldsymbol{x}$-dependent variance in mean which can be computed in closed form for linear Gaussian models.

Question 6 A dynamic motor primitive (DMP) is based on the dynamical system

$$\begin{aligned} \ddot{y} &= \tau^2 \Big(\alpha \big(\beta (g - y) - \dot{y}/\tau \big) + f_w(z) \Big) \\ \dot{z} &= -\tau \alpha_z z \end{aligned}$$

where z represents the canonical system. Give an equivalent definition in which you formalize the DMP as a PD-controller with a feedforward term that acts on the acceleration $\ddot{y}$ of the system.

Answer

$$\begin{aligned} \ddot{y} &= \tau^2 \Big(\alpha \big(\beta (g - y) - \dot{y}/\tau \big) + f_w(z) \Big) \\ &= \tau^2 \alpha \big(\beta (g - y) - \dot{y}/\tau \big) + \tau^2 f_w(z) \\ &= \tau^2 \alpha \beta (g - y) - \tau \alpha \dot{y} + \tau^2 f_w(z) \\ &= \underbrace{\tau^2 \alpha \beta}_{K_P} (\underbrace{g}_{y_d} - y) + \underbrace{\tau \alpha}_{K_D} (\underbrace{0}_{\dot{y}_d} - \dot{y}) + \underbrace{\tau^2 f_w(z)}_{u_{\text{ff}}} \\ &= K_P (y_d - y) + K_D (\dot{y}_d - \dot{y}) + u_{\text{ff}} \end{aligned}$$

Question 7 What are the advantages of using a probabilistic representation for trajectories?

Answer This allows representing how important it is to reach certain points and certain time steps by quantifying the variance at that point accordingly. Also it allows probabilistic operations on the trajectories like conditioning and blending.

Question 8 You want to train your robot to reach a desired position with reinforcement learning. You decide to use a reward of 1 when it reaches the desired position and 0 otherwise. Why is this reward not a good idea?

Answer This reward yields a large exploration problem. The robot would have to reach the exact position by chance at least once in order to recognize that there's a way to increase the reward. As that event is very unlikely, the robot would most likely not learn anything.

Question 9 Updating the Q-Function using Temporal Differences is defined with the following equation:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha\delta_t \quad \text{with} \quad \delta_t = r_t + \gamma Q_t(s_t, a_?) - Q_t(s_t, a_t)$$

In the lecture the following two different ways of choosing $a_?$ where discussed, resulting in two algorithms. Name the corresponding algorithms and discuss their differences.

A: $a_? = \arg\max_a Q_t(s_t, a)$

B: $a_? = a_{t+1}$ where $a_{t+1} \sim \pi(\cdot \mid s_{t+1})$

Answer Method (A) is called *Q-Learning* and is an off-policy method as the predictive action is not sampled from the policy. Method (B) is called *SARSA* and is an on-policy method. SARSA also requires not only state-action-reward-state data but state-action-reward-state-action data to be trained. Also it does not require an explicit maximization and has stronger convergence guarantees as opposed to Q-Learning.

Question 10 Write down (with detail) the optimization problem for the Linear-Quadratic Regulator with discrete-time and infinite-horizon, assuming there is no noise in the transition to the next state.

Answer The optimization problem is

$$\begin{aligned} \min_{\boldsymbol{u}_{1:T}} & \sum_{t=1}^{\infty} (\boldsymbol{x}_d - \boldsymbol{x}_t)^T \boldsymbol{R}_t (\boldsymbol{x}_d - \boldsymbol{x}_t) + \boldsymbol{u}_t^T \boldsymbol{H}_t \boldsymbol{u}_t \\ \text{s.t.} \quad & \boldsymbol{x}_{t+1} = \boldsymbol{A}_t \boldsymbol{x}_t + \boldsymbol{B}_t \boldsymbol{u}_t \end{aligned}$$

where $\boldsymbol{R}_t$ and $\boldsymbol{H}_t$ are symmetric and positive definite metrics for weighting the state error and to penalize taking actions. The matrices $\boldsymbol{A}_t$ and $\boldsymbol{B}_t$ define the dynamics of the system.

Question 11 When deriving the likelihood ration gradient in the class, we started with the derivative of the cost function $J(\boldsymbol{\theta})$, which is given by

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int R(\boldsymbol{\tau}) \boldsymbol{\nabla}_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\boldsymbol{\tau} \mid \pi) \, \mathrm{d}\boldsymbol{\tau}$$

where $\boldsymbol{\tau}$ is a trajectory, $\boldsymbol{\theta}$ are the parameters that we want to optimize, π is the parameterized policy, and $R(\boldsymbol{\tau})$ is the return of the trajectory. Using a log-trick, we could approximate this derivative using N sample trajectory by

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int R(\boldsymbol{\tau}) p_{\boldsymbol{\theta}}(\boldsymbol{\tau} \mid \pi) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{\tau} \mid \pi) \, \mathrm{d}\boldsymbol{\tau} \approx \frac{1}{N} \sum_{i=1}^{N} R(\boldsymbol{\tau}_i) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}_i}(\boldsymbol{\tau} \mid \pi)$$

Apart from using a sample-based approximation, what is the key benefit of the log-trick?

Answer It is often easier to take the derivative of a log-density instead of the density itself as the logarithm turns products into sums. Hence, lots of terms that are constant w.r.t. the parameters vanish, yielding a simple gradient. This is especially useful in the step-based setting where the trajectory distribution is factorized into the transition and policy distribution, where only the latter is dependent on the parameters: the former disappears in the gradient due to the logarithm.

Question 12 What is the role of the learning rate in policy gradient methods? What is the role of the temperature in probabilistic policy search methods?

Answer Both the learning rate and the temperature affect the convergence speed of the algorithm. As the learning rate is used as the step size for gradient ascent, the usual properties apply: a too low rate could lead to premature stops during the optimization while a too high rater could lead to oscillations, jumps and probably divergence. The same is true for the temperature. they define how “trustworthy” the update is.