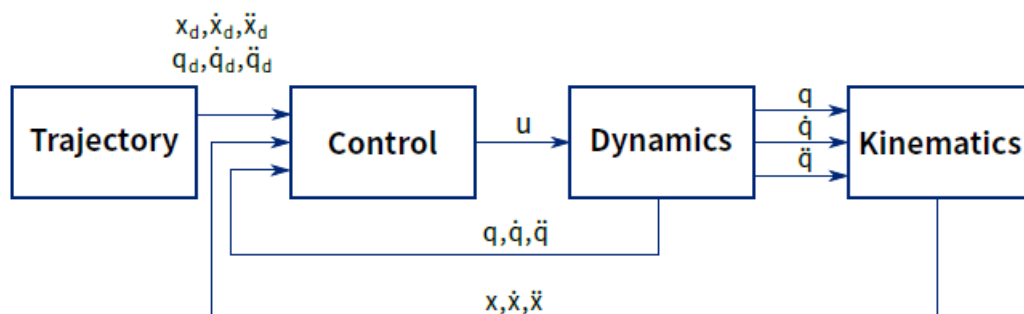


Robotics in a Nutshell

1. Objectives

- The basics about robotics:
 - Modelling position, velocity, acceleration, and forces
 - Representing trajectories
 - How to control a robot along a trajectory!
- All-important robotics intuition in a nutshell!
- To understand robot learning, we must understand the problems first.

2. Modeling Robotics



2.1. Kinematics

Position and Orientation

Position and Orientation can directly be computed using the forward kinematics model, e.g. using the Denavit-Hartenberg convention! Forward kinematics maps the joint space to the task space:

$$x = f(q)$$

The **Denavit-Hartenberg** convention is a systematic way to determine homogeneous transformations (Combining Translation and Rotation is a mess that's why we need homogenous transformations!) between a robot's frames using 4 parameters:

$$H_{i+1}^i = Rot_{z, \theta_i} Trans_{z, d_i} Trans_{x, a_i} Rot_{x, \alpha_i} = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} c_{\alpha_i} & s_{\theta_i} s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i} c_{\alpha_i} & -c_{\theta_i} s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Differential Forward Kinematics

Forward kinematics model has to be differentiated w.r.t. to the time. Note that the chain rule has to be applied! 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:

$$\dot{x} = \frac{d}{dt}f(q) = \frac{df(q)}{dq} \frac{dq}{dt} = J(q)\dot{q}, \quad J(q) \text{ is the Jacobian}$$

Acceleration is given as:

$$\ddot{x} = J(q)\ddot{q} + \dot{J}(q)\dot{q}$$

Singularities in Kinematics

What happens when I stretch my robot arm out? The columns of the Jacobian get linearly dependent, and we lose a degree of freedom and $\det(J) = 0$! These positions are called Singularities.

Computing Jacobians

Analytical Jacobians are easier to understand (as before) and can be derived by symbolic differentiation. However, the representation of the rotation matrix can cause "representational singularities".

Geometric Jacobians are derived from geometric insight (more contrived), can be implemented easier and do not have "representational singularities".

Main Difference: How the Jacobian for the orientation is represented.

2.2. Dynamics

The **forward dynamics** model gives joint accelerations given the joint positions, velocity and force's F /torques τ also called motor commands u :

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

Usually in robotics the dynamics is represented in the general form

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

with $M(q)$: Inertia/Mass matrix, $c(q, \dot{q})$: Coriolis forces and centripetal forces and $g(q)$: Gravity

which is an inverse dynamics model! The general form can easily be inverted to get the joint accelerations as the **mass matrix is always positive definite** and hence invertible:

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

There are two central methods to compute the rigid body forces. Other forces than rigid body forces e.g. friction are a lot harder to model as there is no general recipe for modeling them!

- **Newton-Euler Method:** Force dissection-based approach.
- **Lagrangian Method:** Energy based approach.

We can already build a robot simulator using the general form of the dynamics which is the **inverse dynamics model**:

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

This equation can easily be inverted to get the joint accelerations (forward dynamics model) since the mass matrix is always positive definite:

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

The joints velocities are obtained by integrating the joint accelerations w.r.t time:

$$\dot{q}(t) = \int_0^t \ddot{q}(\tau) d\tau$$

The joint positions are obtained by integrating the joint velocities w.r.t time:

$$q(t) = \int_0^t \dot{q}(\tau) d\tau$$

But this is not possible in closed form so numerical integrating techniques are required. For example, the symplectic Euler method can be used.

3. Representing Trajectories

Looking at the block diagram of the complete system above the desired trajectory is: $q_d(t), \dot{q}_d(t), \ddot{q}_d(t)$. The trajectory specifies the joint positions, velocities, and accelerations for each instant of time t . It's used to specify the desired movement plan and inherently includes velocities and accelerations.

If we look at the mathematical model of the robot, we can see that the motor commands can only influence the acceleration! Any trajectory must be twice differentiable! Positions and velocities cannot jump! For that reason, we are using polynomials or **splines**!

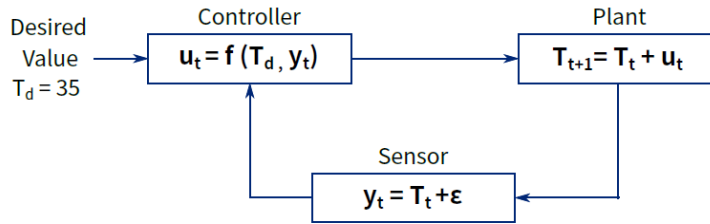
Quintic splines are the way to go since we have 6 free parameters meanwhile in cubic splines, we still have jumps in acceleration because we only constrained position and velocity!

Other methods:

- Linear Segments with Parabolic Blends
- Trapezoidal Minimum Time Trajectories
- Nonlinear Dynamical Systems: $\ddot{q} = f(q, \dot{q}, \theta)$
- Potential Fields: $V(q), \dot{q} = \frac{dV(q)}{dq}$

4. Control in Joint Space

Given the desired trajectory $q_d(t), \dot{q}_d(t), \ddot{q}_d(t)$ we still need to find the control u to follow this trajectory! 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 below 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.



4.1. Linear 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

Based on position error: $u_t = K_p(q_d - q_t)$. It causes high oscillations in the position, making it not suitable for real control.

PD-Controller

Based on position and velocity errors: $u_t = K_p(q_d - q_t) + K_D(\dot{q}_d - \dot{q}_t)$. There is still a steady state error 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 law. This approach is the most used in industrial robots!

$$u_t = K_p(q_d - q_t) + K_D(\dot{q}_d - \dot{q}_t) + g(q)$$

PID-Controller

Alternatively, to doing gravity compensation we can estimate the motor command to compensate for the steady state error by integrating the error:

$$u_t = K_p(q_d - q_t) + K_D(\dot{q}_d - \dot{q}_t) + K_I \int_{-\infty}^T (q_d - q) d\tau$$

For tracking control, it may create havoc and disaster!

4.2. Model-Based Control

PD with gravity compensation is not a good choice. We need an error to generate a control signal. To be accurate, we need to magnify a small error, i.e. we have huge gains. Huge gains are costly, make the robot very stiff and dangerous. Mechanical systems are second order systems, i.e. we can only change the acceleration by inserting torques! 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:

Forward Model: $\ddot{q} = M^{-1}(q)(u - c(\dot{q}, q) - g(q))$

Inverse Model: $u = M(q)\ddot{q}_d + c(\dot{q}, q) + g(q)$

Exploiting inverse Model as control policy, we achieve desired behavior: $\ddot{q} = \ddot{q}_d$

For errors adapt only reference acceleration and insert it into our inverse model:

$$\ddot{q}_{\text{ref}} = \ddot{q}_d + K_D(\dot{q}_d - \dot{q}) + K_P(q_d - q)$$

$$u = M(q)\ddot{q}_{\text{ref}} + c(\dot{q}, q) + g(q)$$

As with this model-based controller, $\ddot{q} = \ddot{q}_{\text{ref}}$ the system behaves a linear decoupled system. I.e. it is a decoupled double integrator!

Feedforward Control

Feedforward control assumes $q \approx q_d$ and $\dot{q} \approx \dot{q}_d$. Hence, we have

$$u = u_{\text{FF}}(q_d, \dot{q}_d, \ddot{q}_d) + u_{\text{FB}}$$

with feedforward torque prediction using an inverse dynamics model

$$u_{\text{FF}} = M(q_d)\ddot{q}_d + c(q_d, \dot{q}_d) + g(q_d)$$

and a linear PD control law for feedback

$$u_{\text{FB}} = K_P(q_d - q) + K_D(\dot{q}_d - \dot{q})$$

Key points:

- FF can be done with less real-time computation as feedforward terms can often be pre-computed.
- FF is generally more stable – even with bad models or approximate models.
- Only when you have a very good model, you should prefer Model-based Feedback Control.
- In practice, FF is often more important.

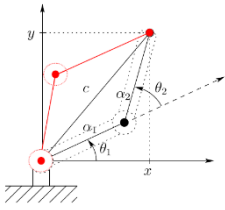
5. Control in Task Space

5.1. Inverse Kinematics

Control in task space: We want the end-effector to follow a desired trajectory $x_d(t)$. The inverse kinematics maps the task-space to joint space!

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

We can solve the task space control geometrically:



BUT we have a problem.

- Multiple solutions even for non-redundant robots (example above)
- **Redundancy results in infinitely many solutions.**
- Often only numerical solutions are possible!

5.2. Differential Inverse Kinematics

Given current joint positions, compute joint velocities that minimize the task space error. This is computable! Assuming the robot is non-redundant $n=r$ where r is the dimensionality of the task space the Jacobian becomes a square. Assuming not to be in a singularity the joint velocities can therefore be obtained by inverting $J(q)$:

$$\dot{q} = h(x_d, q_t) = J^{-1}(q)\dot{x}$$

1. Numerically Integrate \dot{q}_d and directly use it for joint space control.
2. Iterate differential IK algorithm to find q_d :

$$q_{k+1} = q_k + h(x_d, q_k)$$

and plan trajectory to reach q_d .

We can solve the task space control numerically:

- Jacobian transpose
- Jacobian Pseudo Inverse
- Damped Pseudo Inverse
- Task prioritization with Null Space Movements
- Advanced solutions like operational space control laws

Jacobian transpose

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

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

has to be minimized (where $f(q)$ 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

$$\frac{dE}{dq} = -\frac{df(q)}{dq}^T (\mathbf{x}_d - f(\mathbf{q})) = -J^T(q)(\mathbf{x}_d - f(\mathbf{q}))$$

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

$$\dot{q} = -\gamma \frac{dE}{dq} = \gamma J^T(q)(\mathbf{x}_d - f(\mathbf{q})) = \gamma J^T(q)\mathbf{e}$$

This is called the **Jacobian transpose method**. As already said, the desired joint position q_d can then be recovered **by numerically integrating \dot{q}_d** . All of this can then be fed into a 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

Assume that we are not so far from our solution manifold. It is possible to also set desired task space velocities whilst finding the smallest \dot{q} that has desired task space velocity. This can be formulated as an optimization problem:

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

Solution (Right pseudo inverse):

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

However, the inversion in the pseudo-inverse can be problematic. In the case of singularities, $J J^T$ can not be inverted!

Singularities and Damped Pseudo Inverse

Find a tradeoff between minimizing the error and keeping the joint movement small. λ is the regularization constant.

$$\min_{\dot{q}} (\dot{x}_d - J(q)\dot{q})^T (\dot{x}_d - J(q)\dot{q}) + \lambda \dot{q}^T \dot{q}$$

Damped Pseudo Inverse Solution **works much better for singularities**:

$$\dot{q} = J^\dagger (J J^T + \lambda I)^{-1} \dot{x}_d = J^{\dagger(\lambda)} \dot{x}_d$$

Task-Prioritization with Null-Space-Movements

What's Null-Space? Now, in robotics, the null space is the subspace within the joint configuration space where additional joint motions do not affect the end-effector's position or the robot's primary task. In other words, the null space allows the robot to perform secondary or auxiliary motions, such as avoiding obstacles or adjusting joint angles for comfort, without interfering with the main task.

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**. This can be, for example, to push the robot into a rest position q_{rest} where it does not consume energy. This base task can be formulated with a P-controller:

$$\dot{q}_0 = K_p (q_{rest} - q)$$

The optimization problem then is as following:

$$\min_{\dot{\mathbf{q}}} (\dot{\mathbf{q}} - \dot{\mathbf{q}}_0)^T (\dot{\mathbf{q}} - \dot{\mathbf{q}}_0), \quad \text{s.t. } J(\mathbf{q})\dot{\mathbf{q}} = \dot{\mathbf{x}}_d$$

The result of this optimization problem is:

$$\dot{\mathbf{q}} = J^\dagger \dot{\mathbf{x}}_d + (I - J^\dagger J) \dot{\mathbf{q}}_0 = \dot{\mathbf{q}}_{\text{nominal}} + \dot{\mathbf{q}}_{\text{null}}$$

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

More advanced solutions

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

$$\ddot{\mathbf{q}} = J^\dagger (\ddot{\mathbf{x}}_d - \dot{J}\dot{\mathbf{q}}) + (I - J^\dagger J) \ddot{\mathbf{q}}_0$$

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:

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

Discrete State-Action Optimal Control

1. Objectives

- Manual programming all scenarios is too costly!
- Similarly *supervised learning* (see later in lecture 14/15) is not enough for acquiring behaviours
 - Imperfect demonstrations
 - Correspondence problem
 - We cannot demonstrate everything
- Hence, we need *(self-) improvement*!
 - The robot explores by trial and error
 - We give evaluative feedback => reward
- Today, we are going to look at the problem of how to take *optimal decisions that maximize the reward*
- Note: reward = - cost, max(reward)=min(cost)

2. Introduction

2.1. MDP

A (non-stationary) Markov Decision Process is defined by:

- State space $s \in S$
- Action space $a \in A$
- Transition dynamics $p(s_{t+1}|s_t, a_t)$ depend only on the current time step!
- Reward $r_t(s, a)$
- Initial state properties $\mu_0(s)$

Crucial Assumption Markov property:

The Markov property states that the future state of the system depends only on the current state and action, and not on the sequence of states and actions that led to the current state.

$$p_t(s_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, \dots) = p_t(s_{t+1} | s_t, a_t)$$

2.2. Policy

A policy π in an MDP describes the action to take given a state. It either be **deterministic** $a = \pi(s)$, $\pi: S \rightarrow A$ or **stochastic** $a \sim \pi(\cdot|s)$, $\pi: S \times A \rightarrow R^+$. 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.

3. Value Iteration – Finite Horizon Problems

3.1. Value Iteration Algorithm

Algorithm 1: Value Iteration for Finite-Horizon Problems

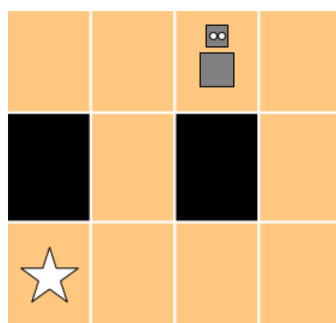
```

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

```

- $\max_a Q_t^*(s, a)$ is the maximum value (if it exists) of $Q_t^*(s, a)$ as we have different actions we can take, while $\arg \max_a Q_t^*(s, a)$ is the value of a (meaning which action) at which this maximum is attained.
- However, there could be more than one action for which we get $\max_a Q_t^*(s, a)$, in which case $\arg \max_a Q_t^*(s, a)$ would be this set of values of actions a .
- This also means the optimal policy can have multiple solutions as you can see in the example down below!

3.2. Example



Navigation Task

■ States:

- Robot s_R
- Goal s_G
- Obstacles s_{o_1}, s_{o_2}

■ Reward:

$$r_t(s) = \begin{cases} 0 & s = s_G \\ -5 & s \in \{s_{o_1}, s_{o_2}\} \\ -1 & \text{else} \end{cases}$$

Start first with the last time step!

$$V_T^*(s) = r_T(s)$$

t = T = 6:

-1	-1	-1	-1
	-1		-1
0	-1	-1	-1

Compute V-function for time step t (iterations $T-t+1$) for all $s \in S$.

$$V_t^*(s) \leftarrow \max_a (r_t(s, a) + \sum_{s'} p_t(s'|s, a) V_{t+1}^*(s'))$$

t = 5:

-2	-2	-2	-2
	-2		-2
0	-1	-2	-2

t = 4:

-3	-3	-3	-3
	-2		-3
0	-1	-2	-3

t = 3:

-4	-3	-4	-4
	-2		-4
0	-1	-2	-3

t = 2:

-4	-3	-4	-5
	-2		-4
0	-1	-2	-3

4. Finite and Infinite Horizon Objectives

4.1. Finite Horizon Optimal Control

The goal is to find an optimal policy $\pi_t^*(a|s)$ that maximizes its expected return J_π for a finite time horizon! The accumulated expected reward for T steps is:

$$\pi_t^* = \arg \max_{\pi} J_\pi$$

$$J_\pi = \mathbb{E}_{\mu_0, p, \pi} \left[r_T(s_T) + \sum_{t=1}^{T-1} r_t(s_t, a_t) \right]$$

$r_T(s_T)$ is the final reward;
 $\mu_0(\cdot)$ is the initial distribution;
 $p(\cdot|s_t, a_t)$ is the transition probability function;

For finite-horizon problems, all of the transition dynamics, reward function and state-action value function and value function and therefore also the policy are time-dependent.

Also, there is a last reward $r_T(s_T)$ that is independent of the action. This can be interpreted as that it is relevant how much steps are left to make decisions.

Robotic Example: Ball-in-cup, where once the ball is in the cup the problem is no longer interesting.

Value Function

For some policy π the value function (How good is it to be in a state s) is assessing the quality of a state!

$$V_t^\pi(s) = E_{p,\pi} \left[\sum_{\tau=t}^T r_\tau(s_\tau, a_\tau) | s_t = s \right]$$

State-Action Value Function (Q-Function)

For some policy π the **action value function** (how good is it to take action a in state s) is assessing the quality of a state!

$$Q_t^\pi(s, a) = E_{p,\pi} \left[\sum_{\tau=t}^T r_\tau(s_\tau, a_\tau) | s_t = s \right]$$

4.2. Infinite Horizon Optimal Control

The goal is to find an optimal policy $\pi^*(a|s)$ that maximizes its expected long-term return J_π :

$$\pi^* = \arg \max_{\pi} J_\pi, \quad J_\pi = \mathbb{E}_{\mu_0, p, \pi} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right]$$

$0 \leq \gamma < 1$ is the discount factor
Discount factor: Trades off long term vs. immediate reward
Time horizon: infinite

For infinite-horizon objectives, time is no longer relevant, and the time-dependencies is dropped for all components.

Robotic Example: Balancing an inverted pendulum where more reward is gained the longer the pendulum can be held upright!

Value Function

For some policy π the value function (How good is it to be in a state s) is assessing the quality of a state!

$$V^\pi(s) = E_{p,\pi} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s \right]$$

State-Action Value Function (Q-Function)

For some policy π the action value function (how good is it to take action a in state s) is assessing the quality of a state!

$$Q^\pi(s, a) = E_{p, \pi} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s, a_0 = a \right]$$

Relationship of V and Q for an optimal and suboptimal policy

Relationship between Value and Q-Function for the Infinite Horizon Optimal Control (same applies for Finite Horizon Optimal Control)

Suboptimal policy:

$$V^\pi(s) = E_\pi[Q^\pi(s, a)|a] = \sum_a \pi(a|s) * Q^\pi(s, a)$$

$$Q^\pi(s, a) = r(s, a) + \gamma E_{s' \sim p(\cdot|s, a)}[V^\pi(s')] = r(s, a) + \gamma \sum_{s'} p(s'|s, a) V^\pi(s')$$

Optimal policy:

$$V^*(s) = \max_a Q^*(s, a)$$

$$Q^*(s, a) = r(s, a) + \gamma E_{s' \sim p(\cdot|s, a)}[V^*(s')] = r(s, a) + \gamma \sum_{s'} p(s'|s, a) V^*(s')$$

Value function and state-action value function

A value function tells us how good it is to be in a state s . The state-action value function tells us how good it is to take action in a state s . The first is estimating the quality of a state and the second the quality of a state-action pair!

5. Finite-Horizon Optimal Control

5.1. Value Iteration

```

Init:  $V_T^*(s) \leftarrow r_T(s)$ ,  $t = T$ 
for  $t = T, T-1, \dots, 3, 2, 1$  do
    Compute Q-Function for time step  $t$  (for each state action pair):
         $Q_t^*(s, a) \leftarrow r_t(s, a) + \sum_{s'} p_t(s' | s, a) V_{t+1}^*(s')$ 

    Compute V-Function for time step  $t$  (for each state):
         $V_t^*(s) \leftarrow \max_a Q_t^*(s, a)$ 
end for

return Optimal policy for each time step:
     $\pi_t^*(s) = \arg \max_a Q_t^*(s, a)$ 

```

6. Infinite Horizon Value Iteration

```

Init:  $V^*(s) \leftarrow r(s)$ 
repeat
    Compute  $Q$ -Function (for each state action pair):
         $Q^*(s, a) \leftarrow r(s, a) + \gamma \sum_{s'} p(s' | s, a) V^*(s')$ 

    Compute  $V$ -Function (for each state):
         $V^*(s) \leftarrow \max_a Q^*(s, a)$ 
until  $V^*(s)$  converges

return Optimal policy:
     $\pi^*(s) = \arg \max_a Q^*(s, a)$ 

```

7. Infinite Horizon Policy Iteration

```

Init:  $V_0^\pi(s) \leftarrow 0, \pi \leftarrow \text{uniform}$ 
repeat
    repeat
         $k \leftarrow k + 1$ 
        Compute  $Q$ -Function (for each state action pair):
             $Q_{k+1}^\pi(s, a) \leftarrow r(s, a) + \gamma \sum_{s'} p(s' | s, a) V_k^\pi(s')$ 

        Compute  $V$ -Function (for each state):
             $V_{k+1}^\pi(s) \leftarrow \sum_a \pi(a | s) Q_{k+1}^\pi(s, a)$ 
    until convergence of  $V$ 

     $\pi(a | s) \leftarrow \begin{cases} 1 & \text{if } a = \arg \max_{a'} Q^\pi(s, a') \\ 0 & \text{otherwise} \end{cases}$ 
until convergence of policy

```

Policy Iteration is done in 2 separate steps:

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.

So, we are assuming we already have a policy given!

Policy iteration iterates policy evaluation and improvement until convergence to find the optimal policy. Value Iteration also finds the optimal policy but iterates the Bellman equation directly!

Difference between policy iteration vs value iteration

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.

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:

$$V_t^*(\mathbf{s}) = \max_a \left(r_t(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_p [V_{t+1}^*(\mathbf{s}_{t+1}) | \mathbf{s}_t, \mathbf{a}_t] \right)$$

8. Wrap-Up

Unfortunately, we can apply Dynamic Programming only in this in two cases:

- Discrete Systems:
 - Easy: Integrals turn into sums, but the world is not discrete!
 - Separating the world in buckets, but this would cause an exponential explosion in memory. The number of discrete actions and states grows exponentially with the system's dimensions. This issue is known as the curse of dimensionality.
- Linear Systems, Quadratic Reward, Gaussian Noise (LQR), but the world is also not linear!

Continuous State-Action Optimal Control

1. Objectives

- Today we will investigate a class of continuous decision-making problems which we can solve efficiently
 - Linear system, Quadratic Reward, Gaussian Noise. The optimal policy is called *Linear Quadratic Regulator* (LQR)
- For problems outside of this class we will derive an algorithm based on LQR to solve them approximately
- Optimal decision making for continuous dynamical systems is also called Optimal Control

2. Optimal Control

2.1. Intro

A continuous action space allows the agent to select an action from a range of values for each state. Just as with a discrete action space, this means for every incrementally different environmental situation, the agent's neural network selects a speed and direction for the car based on input from its camera(s) and (optional) LiDAR sensor. However, in a continuous action space, you can define the range of options the agent picks its action from. In this example, the AWS DeepRacer car in a continuous action space approaching a turn can choose a speed from 0.75 m/s to 4 m/s and turn left, right, or go straight by choosing a steering angle from -20 to 20 degrees. The transition dynamics is a probability distribution! But how does this even build? Imagine we have 3D coordinate system. Each axis represents a , s , s' . These state and action spaces are e.g. Gaussian Distributions. For the conditional you just slice along the conditions! This is complicated!

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 x for the state and u for the action. 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).

2.2. LQR

An LQR system is define as its

- state space $x \in R^n$
- action space $u \in R^m$
- (possible time dependent) linear transition dynamics with gaussian noise:

$$p_t(x_{t+1}|x_t, u_t) = N(x_{t+1}|A_t x + B_t u_t + b_t, \Sigma_t)$$

A_t : System matrix
 B_t : Control matrix
 b_t : drift term
 Σ_t : system noise

- Quadratic reward function:

$$r_t(x, u) = -(x - x_d)^T R_t (x - x_d) - u_t^T H_t u_t$$
$$r_T(x) = -(x - x_d)^T R_T (x - x_d) = r_t(x, 0)$$

- Initial state density: $\mu_0(x) = N(x|\mu_0, \Sigma_0)$

3. Solving the Optimal Control for LQR Systems

The objective is to maximize the expected long-term reward with a finite time horizon:

$$J_\pi = E_{\mu_0, p, \pi} \left[r_T(s_T) + \sum_{t=1}^{T-1} r_t(s_t, a_t) \right], \quad r_T(s_T) \text{ is the final reward}$$

Derive the LQR value function and optimal policy for the basic case. Applying the principle of optimal control now says to start with the last time step T by setting $V_T^*(x) = r_T(x)$ for all states. Subsequently, iterate over $t = T - 1, \dots, 1$ and compute $V_t^*(x) = \max_u (r_t(x_t, u_t) + E_p[V_{t+1}^*(x_{t+1})|x_t, u_t])$. The optimal value function/policy for time step t is obtained after T - t + 1 iterations: $V_1^*(x_1)$. 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 which is called the Bellman Recipe:

1. Compute the value function for the last time step for all states: $V_T^*(x) = r_T(x)$

Solution:

$$V_T^*(x) = r_T(x) = -(x - x_d)^T R_T (x - x_d) = -(x - x_d)^T V_T (x - x_d)$$

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

$$Q_t^*(x_t, u_t) = r_t(x_t, u_t) + E_p[V_{t+1}^*(x_{t+1})|x_t, u_t]$$

Solution:

- Firstly, the expectation has to be computed by assuming a quadratic structure for the value function of time step t+1:

$$V_{t+1}^*(x) = -(x - x_d)^T V_{t+1} (x - x_d)$$

- Then the expectation becomes:

$$\begin{aligned}
\mathbb{E}_{x' \sim p(\cdot | x, u)} [V_{t+1}^*(x') | x, u] &= \int V_{t+1}^*(x') p(x' | x, u) dx' \\
&= - \int (x - x_d)^T V_{t+1} (x - x_d) \mathcal{N}(x' | A_t x + B_t u_t + b_t, \Sigma_t) dx' \\
&= -(A_t x + B_t u_t + b_t - x_d)^T V_{t+1} (A_t x + B_t u_t + b_t - x_d) - \text{tr}(V_{t+1} \Sigma_t)
\end{aligned}$$

- Plugging in the reward function yields the Q-function:

$$\begin{aligned}
Q_t^*(x, u) &= -(x - x_d)^T R_t (x - x_d) - u^T H_t u \\
&\quad - (A_t x + B_t u_t + b_t - x_d)^T V_{t+1} (A_t x + B_t u_t + b_t - x_d) \\
&\quad - \text{tr}(V_{t+1} \Sigma_t)
\end{aligned}$$

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

$$\pi_t^*(x) = \arg \max_u Q_t^*(x, u)$$

Solution: Computing the policy is done by maximizing the Q-function w.r.t the action u. This means taking the derivative of it and setting it to zero:

$$\begin{aligned}
\frac{\partial}{\partial u} Q_t^*(x, u) &= -2H_t u - 2B_t^T V_{t+1} (A_t x + B_t u_t + b_t - x_d) \\
&= -2H_t u - 2B_t^T V_{t+1} A_t x - 2B_t^T V_{t+1} B_t u_t - 2B_t^T V_{t+1} b_t + 2B_t^T V_{t+1} x_d \\
&= -2(H_t + B_t^T V_{t+1} B_t) u - 2B_t^T V_{t+1} (A_t x + b_t - x_d) \stackrel{!}{=} 0
\end{aligned}$$

$$\pi_t^* = u^* = -(H_t + B_t^T V_{t+1} B_t)^{-1} B_t^T V_{t+1} (A_t x + b_t - x_d)$$

The optimal policy is a time-dependent linear feedback controller with time-dependent offset!

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

$$V_t^*(x) = Q_t^*(x, \pi_t^*(x))$$

Solution:

- To compute the value function, plug the obtained optimal policy into the Q-function.
- The optimal value function has a quadratic and linear form!
- The optimal value function is of the form $V_t(x_t) = x_t^T V_t x_t + x_t^T v_t$ and it is quadratic-linear.
- The optimal control input (optimal policy) is of the form $u_t^* = K_t(x_t - x_d) + k_t$, and is a time-varying P-controller!

What is the format of the solution (value function and policy) for the LQR? What is their interpretation (qualitatively)? The optimal value function is of the form $V_t(x_t) = x_t^T V_t x_t + x_t^T v_t$ and it is quadratic-linear. The optimal control input is of the form $u_t^* = K_t(x_t - x_d) + k_t$, and is a time-varying P-controller!

4. Approximating Non-Linear Systems

What to do when dynamics and/or costs are not linear/quadratic? What are the pitfalls? We can approximate the solution locally around an initial trajectory by linearizing the dynamics and/or the **rewards** using a first order and/or **second order** Taylor expansion around a point $(\tilde{x}_t, \tilde{a}_t)$. 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. We can construct a local LQR problem about the optimal trajectory but we need the local LQR parameters to calculate the optimal trajectory. Therefore, for nonlinear problems, our optimal solver is iterative, updating the approximately optimal state-action trajectory until it converges to a locally optimal solution. This algorithm is known as iterative Linear Quadratic Regulator (iLQR).

5. Differential Dynamic Programming

iLQR approximates the dynamics $f(x_t, u_t)$ with a first order Taylor expansion.

$$\mathbf{x}_{t+1} \approx \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t) + \begin{bmatrix} \mathbf{f}_x^\top & \mathbf{f}_u^\top \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \mathbf{u} \end{bmatrix}$$

It Approximates the cost with a second order Taylor approximation:

$$C(\mathbf{x} + \Delta \mathbf{x}, \mathbf{u} + \Delta \mathbf{u}) \approx C_0 + \begin{bmatrix} \Delta \mathbf{x}^\top & \Delta \mathbf{u}^\top \end{bmatrix} \begin{bmatrix} C_x \\ C_u \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \Delta \mathbf{x}^\top & \Delta \mathbf{u}^\top \end{bmatrix} \begin{bmatrix} C_{xx} & C_{xu} \\ C_{ux} & C_{uu} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \mathbf{u} \end{bmatrix}$$

Why? Because it made the value function a quadratic function that we can compute in closed form!

DDP Algorithm: But with exact dynamics $f(x_t, u_t)$ in iLQR, the Q-function would not be quadratic:

$$Q(\mathbf{x}_t, \mathbf{u}_t) = C(\mathbf{x}_t, \mathbf{u}_t) + V(\mathbf{f}(\mathbf{x}_t, \mathbf{u}_t))$$

However, instead of approximating the dynamics $f(x_t, u_t)$ linearly in iLQR, we could also directly approximate $Q(x_t, u_t)$ with a quadratic function. This yields the Differential Dynamic Programming (DDP) Algorithm

1. Execute u_0, u_1, \dots, u_{T-1} forward in time over the planning horizon, linearize dynamics f and quadratize cost function C .
2. Update the local Q and value function backwards in time via dynamic programming. Compute time-varying control law.
3. Update the new action sequence with each u_t using line search.

6. Wrap-Up

- You have solved an (stochastic) optimal control problem today!
- Only two cases are solvable: linear and discrete!
- The optimal policy for a LQR system is a time-varying linear feedback controller
- For non-linear systems we can use Taylor approximations and solve the problem approximately (iLQR and DDP)
- Works well if the system is not too nonlinear
- These methods require you to know the model! We will see how models can be learned from data (stay tuned!)

Machine Learning

All statistical methods in machine learning share the fundamental assumption that the data generation process is governed by the rules of probability.

The data is understood to be a set of random samples from some underlying probability distribution

Keep in mind for future lectures: Even if we do not explicitly mention the existence of an underlying probability distribution the basic assumption about how the data is generated is always there!

Objective

- Refresh your probability & statistics skills
- Quick overview
 - All topics of machine learning
 - More on foundations than applications
- Focus on ML basics important to robot learning
 - Core problems in ML
 - Regression
 - Feature construction

2. Introduction

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 sheer 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 have to be made like all training data is independent and identically distributed, queries

(Once a model is trained on a dataset, it can be used to make predictions or classify new data points, which are referred to as queries. These queries are typically drawn from the same distribution as the training data, assuming that the underlying data distribution remains consistent.)

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. Six Machine Learning Choices

Choices 1-3 describe the problem, 4-6 describe the solution

1. Problem Class
 - **What is the nature of the training data and what kinds of queries will be made at testing time?**
2. Assumptions
 - **What do we know about the source of the data or the form of the solution?**
3. Evaluation criteria
 - **What is the goal of the system?**
 - **How will the answers to individual queries be evaluated?**
 - **How will the overall performance of the system be measured?**
4. Model type
 - Will an intermediate model be made?
 - What aspects of the data will be modeled?
 - How will the model be used to make predictions?
5. Model Class
 - **What particular parametric class of models will be used?**
 - **What criterion will we use to pick a particular model from the model class?**
6. Algorithm
 - **What computational process will be used to fit the model to the data and/or to make predictions?**
 - **Without making some assumptions about the nature of the process generating the data, we cannot perform generalization.**

3.1 Problem Class

Main Machine Learning Problems can be categorized in 3 different domains:

- **Descriptions** which are **Unsupervised Learning** ($D = \{x_i\}$) Methods such as **Clustering** $x_i \rightarrow z_i \in \{0, 1, \dots, n\}$, **Density Estimation** ($x_i \rightarrow p(x_i) \in R^+$) and **Dimensionality reduction** ($x_i \rightarrow z_i \in R^n, \dim x_i > \dim z_i$)
- **Predictions** which are **Supervised Learning** ($D = \{(x_i, y_i)\}$) methods such as **Classification** ($x_i \rightarrow y_i \in \{0, 1, \dots, n\}$) and **Regression** ($x_i \rightarrow y_i \in R^n$)
- If Supervised and Unsupervised Learning are **merged** we will get **Semi supervised Learning**
 - Two data sets $D_1 = \{(x_i)\}$ and $D_2 = \{(x_j, y_j)\}$ simplify supervised learning if $|D_1| \gg |D_2|$
- Supervised Learning is a simplification of **Active Learning**
 - Sample expensive labels when needed ($\subseteq \text{Contextual Bandits} \subseteq \text{RL}$)
- **Decisions** which are **Reinforcement Learning** methods.
 - **Active Learning** is a simplification of Reinforcement Learning.
 - Sample expensive labels when needed ($\subseteq \text{Contextual Bandits} \subseteq \text{RL}$)

Tabular comparison

	Supervised Learning	Unsupervised Learning
Discrete Labels	Classification	Clustering
Continuous Labels	Regression	Dimensionality Reduction

- **Supervised Learning:** Training Data points (x, y) contain known label y
- **Unsupervised Learning:** Training Data points (x, y) contain hidden/latent label y

What kind of reductions are there?

- **Density estimation** $p(x, y)$ is the most general problem
- Can be used for **Regression** $\bar{y} = E_y[y|x]$ and **Classification** $y^* = \operatorname{argmax}_y p(y|x)$ for $y \in \{-1, 1\}$
- Natural candidates for clusters are the modes of $p(x, y)$
- Local variance yields lower-dimensional representation

Other ML problems

- **Sequence learning:** Sequence prediction $x_1, x_2, \dots, x_n \rightarrow y_1, y_2, \dots, y_n$, filtering, smoothing (not i.i.d., structured output)
- **Transfer learning/Meta learning:** Generalize solutions from different related distributions to new ones

3.2 Problem Assumptions

- Learning is possible (e.g., existence of causal structure)
- Smoothness, e.g., in regression, or finitely many switches
- Stationary data generating process
- Process assumptions: i.i.d., Markovian
- Observability
- Frequentist assumption: Existence of a true model

3.3 Evaluation

How can we evaluate an arbitrary method on an ML problem?

1. Evaluate individual predictions

Classification e.g. by $y_p(x) = \operatorname{sign}(f(x))$

- **0-1 loss:** $L_f(x, y) = h(y - f(x)) = 1 - yy_p(x)$
- **Hinge loss:** $L_f(x, y) = \rho(y - f(x))$
- **Soft plus loss:** $L_f(x, y) = \tau \log(1 + \exp(-\tau(y - f(x))))$

Regression e.g. by $y_p = f(x)$

- **Square Loss:** $L_f(x, y) = (y - f(x))^2$

$$\text{SVR loss: } L_f(\mathbf{x}, y) = \begin{cases} 0 & \text{if } |y - f(\mathbf{x})| \leq \epsilon \\ \rho(|y - f(\mathbf{x})| - \epsilon) & \text{else} \end{cases}$$

$$\text{Huber loss: } L_f(\mathbf{x}, y) = \begin{cases} \frac{1}{2}(y - f(\mathbf{x}))^2 & \text{if } |y - f(\mathbf{x})| \leq \delta \\ \delta(|y - f(\mathbf{x})| - \frac{1}{2}\delta) & \text{else} \end{cases}$$

2. Evaluate overall performance

- Minimize expected loss (= risk): $\min_{f(x)} E_{x,y}[L_f(x, y)]$
- Minimize maximum loss
- Minimize regret
- Asymptotic behavior for infinite training data
- Probably approximately correct (PAC)

Important Conclusions:

- Starting point for algorithm design: Performance optimization!
- Evaluation loss: Use and abuse
 - Rationality
 - Minimizing expected loss is considered rational
 - Methodological
 - Evaluation loss as starting point for algorithm design?
 - In practice: Optimizable surrogate objective
 - Criticism. Real-world loss functions are
 - highly asymmetric
 - nonlinear
 - discontinuous

3.4 Model Type

Two General Model Types with Many Subtypes:

- Nonparametric models
 - Use training data as model
 - Data points become parameters
 - Examples: Histograms, Nearest Neighbor, Kernel Density
 - Estimation, LWR, Kernel Methods (SVM, GP, ...)
- Parametric models
 - Collapse training data onto parameters θ of a parameterized model class M_θ
 - Use parameter values for predictions/description/decision making
 - Examples: Mixture Models, LWPR, (Deep) Neural Networks, ...

3.5 Model Class Selection

Nonparametric Example: Classification $x \rightarrow y$ on $D = \{(x_j, y_j)\}_{j=1:N}$

- Model type “k-nearest neighbors”, model class determined by parameter $k \in \{1, 2, \dots, N\}$
- For a query point x , select a set of k data points $S \subseteq D$ such:

$$\forall x_i \in S: \|x - x_i\| \leq \min_{x_j \in D \setminus S} \|x - x_j\|$$

- Obtain a k-Nearest neighbor prediction

$$y_p(x) = \text{sign} \left(\frac{1}{k} \sum_{i=1}^k y_i \right)$$

Parametric Example: Binary Classification $x \rightarrow y$ on $D = \{(x_j, y_j)\}_{j=1:N}$ with labels $y_j \in \{-1, 1\}$

- **Model type:** Linear discriminant model (Perceptron)

$$y_\theta(x) = f(\theta^T \phi(x))$$

$$f(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

with activation function $f(x)$, feature vector $\phi(x) \in R^n$ and model parameters $\theta \in R^n$

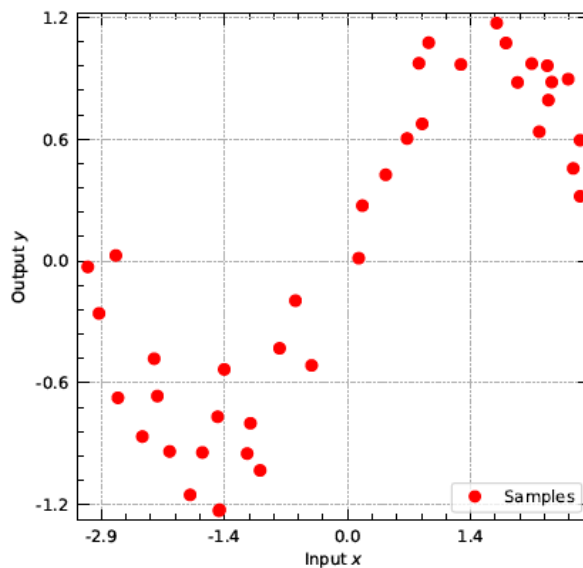
- **Model class:** Feature structure $\phi(x)$ is crucial!

3.6 Algorithmic Realization

All Algorithmic Aspects

- Assume that we have described our problem well, found the right model type and class, how do we make the model work?
- That means:
 - Obtains good results on evaluation metric, e.g., accurate prediction
 - Efficient to obtain (computation/memory)
 - Interpretable solution
- For the nonparametric approach: How can we extract the k nearest neighbors
 - For large data sets?
 - For high-dimensional data?
- For the parametric approach: What optimization algorithm works with the current model class?

3.7 Toy Example



3.7.1 Problem Class

Regression: $y \approx f(x)$

3.7.2 Problem Assumptions

Smoothness in outputs

Stationary independent and identical distribution data generation process

No outliers

→ We can use standard methods

3.7.3 Evaluation

Common in real world: forgive small errors punish large ones drastically

$$\rightarrow \text{Expected quadratic loss } L_f = E_{x,y} \left[\frac{1}{2} (y - f(x))^2 \right]$$

3.7.4 Model Type

Parametric model: $y = f_\theta(x) + \epsilon$ (We are searching for a function that can model the points!)

Equivalent probabilistic model (joint distribution): $p(y|x; \theta) = N(y|f_\theta(x), \sigma^2)$

What type of noise do we have?: Gaussian $\epsilon \sim N(0, \sigma^2)$ → noise follows a normal distribution with a mean of **0** and a variance of σ^2

3.7.5 Model Class Selection

Linear in parameters and features: $f_\theta(x) = \varphi(x)^T \theta$

Assume polynomial features: $\varphi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^n \end{bmatrix}$

Questions:

- Number of parameters a.k.a. degree of polynomial n ? We will answer this later! (Learn also the features with Neural Networks)
- Is the model class sufficiently rich? Too rich? (-> A rich model class typically has a large number of parameters or degrees of freedom, allowing it to capture intricate structures and make fine-grained predictions.)

3.7.6 Algorithm Realization

Minimize risk (=expected loss) by optimizing loss on training data: $L(f_\theta) = E_{x,y}[L_{f_\theta}(x, y)] =$

$$E_{x,y} \left[\frac{1}{2} (y - f_\theta(x))^2 \right] \approx \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (y_i - f_\theta(x_i))^2$$

with $(x_i, y_i) \sim p(x, y)$ where $p(x, y)$ is the joint distribution of the data!

Approximation of the true solution with Gauss principle of Least squares: $\theta^* = \operatorname{argmin}_\theta \sum_{i=1}^N \frac{1}{2} (y_i - \varphi(x_i)^T \theta)^2$

Analytical solution: $\theta^* = (\sum_{i=1}^N \varphi(x_i) \varphi(x_i)^T)^{-1} \sum_{i=1}^N \varphi(x_i) y_i$

Re-arrange features into a design matrix: $\varphi = \begin{pmatrix} \varphi_0(x_1) & \cdots & \varphi_{n-1}(x_1) \\ \vdots & \ddots & \vdots \\ \varphi_0(x_N) & \cdots & \varphi_{n-1}(x_N) \end{pmatrix} \in \mathbb{R}^{N \times n}$

Outputs into a target vector: $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \in \mathbb{R}^N$

End result: $\theta^* = \operatorname{argmin}_\theta \frac{1}{2} (\mathbf{y} - \varphi \theta)^T (\mathbf{y} - \varphi \theta)$

Model Fitting: $\theta^* = (\varphi^T \varphi)^{-1} \varphi^T * \mathbf{y}$

Left pseudo inverse: $\varphi^+ = (\varphi^T \varphi)^{-1} \varphi^T$

Model Prediction: $y_p = \varphi(x_q)^T \theta$

Interpretation of the solution: $n \in \{0, 1\} \rightarrow \text{underfitting}$, $n \in \{3, 6\} \rightarrow \text{good model}$, $n \in \{15, 18\} \rightarrow \text{overfitting}$

Next step: Evaluating our Model Class!

4. Evaluating our Model Class

A small training error does not mean that we have a good model since **overfitting** can occur which leads to a high test error. We need to rethink the model class selection process! Going back to the model selection questions we still need to find the number of parameters a.k.a. degree of polynomial n ! If the model is not sufficiently rich we will get underfitting if it's too rich this will lead to overfitting. According to Occam's Razor one always chooses the simplest model that fits the data. The simplest model leads to the smallest model complexity!

4.1 Bias and Variance trade off (Regression Example)

Estimator \hat{f}_D from training data D , data generated by $y(x_q) = f(x_q) + \epsilon$
 Expected squared error for query x_q estimated from all possible datasets D :

$$L_{\hat{f}}(x) = \mathbb{E}_D [(y(x) - \hat{f}_D(x))^2] = \sigma_\epsilon^2 + \text{Bias}^2 [\hat{f}_D(x)] + \text{Var} [\hat{f}_D(x)]$$

$$\text{bias}^2 [\hat{f}_D(x_q)] = E_{x_q} [(f(x_q) - E_D [\hat{f}_D(x_q)])^2]$$

- Structure error
- Model cannot do better
- So it's about how well we model our data! High bias -> shit, low bias -> good
- Bias is the difference between the average **prediction of our model** and the **correct value which we are trying to predict**. Model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to high error on training and test data.

$$\text{var} [\hat{f}_D(x_q)] = E_{x_q, D} [(\hat{f}_D(x_q) - E_{\bar{D}} [\hat{f}_{\bar{D}}(x_q)])^2]$$

- Estimation error
- Finite data sets will always have errors
- So it's about how well does our model on unseen data: High Variance -> Shit, Low variance -> good
- Variance is the **variability of model prediction** for a given data point or a value which tells us spread of our data. Model with high variance pays a lot of attention to training data and **does not generalize on the data which it hasn't seen before**. As a result, such models perform very well on training data but has high error rates on test data.

Expected Total Error $\propto \text{Bias}^2 + \text{Variance}$

- Typically cannot minimize both -> Trade off
- Low Bias (good model of the data) and High Variance (bad prediction) -> Overfitting
- High Bias (bad model of the data) and low variance (good prediction) -> Underfitting

4.2 How to choose the model

Split data in a training, validation test set!

4.3 Cross validation

Partition data into K sets D_k , use $K - 1$ for Training and 1 for Validation and compute:

$$\theta_k(\mathcal{M}_j) = \arg \min_{\theta \in \mathcal{M}_j} \sum_{\kappa \neq k} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{D}_\kappa} L_{f_\theta}(\mathbf{x}_i, y_i)$$

$$L_k(\mathcal{M}_j) = \sum_{(\mathbf{x}_i, y_i) \in \mathcal{D}_k} L_{f_{\theta_k}}(\mathbf{x}_i, y_i)$$

Exhaustive Cross Validation: Try all partitioning possibilities -> Computationally expensive

Bootstrap: Randomly sample non-overlapping training/validation sets

4.4 K-fold cross validation

Compute the validation loss and choose Model with smallest average validation loss:

$$\mathcal{M}^* = \arg \min_{\mathcal{M}} \frac{1}{K} \sum_{k=1}^K L_k(\mathcal{M})$$

Leave-one-out cross-validation (LOOCV): $K = N - 1 \rightarrow$ Validation set size 1

5. Frequentist vs Bayesian Assumptions (Probability Density Estimation)

This section deals with Probability Density Estimation (see SML). The question is how do we get the probability distributions from a bunch of data $D = \{(x_i, y_i)\}_{i=1}^N$ so we can classify them or predict them?

There are many techniques for solving this problem, although two common approaches are:

- Maximum a Posteriori (MAP), a Bayesian method.
- Maximum Likelihood Estimation (MLE), frequentist method.

Maximum likelihood is a **point-wise estimate** of the model parameters. We Bayesians like posterior distributions.

Maximum likelihood assumes **no prior distribution**. We Bayesians need our priors, it could be informative or uninformative, but it needs to exist.

5.1 Maximum Likelihood Estimator (MLE)

The frequentist assumptions state that probabilities are frequencies of a repeated experiment. **The true parameters of the experiment are in our model class.** They reveal themselves by the frequency (i.e. likelihood) at which we can repeat the outcome of the experiment. We can obtain good parameters by maximizing likelihood of the outcome!

Goal: Find the parameters θ of a distribution $p(x, y; \theta)$ that best describe the data $D = \{(x_i, y_i)\}_{i=1}^N$ (Note: This conditional probability is often stated using the semicolon (;) notation instead of the bar notation (|) because θ is not a random variable, but instead an unknown parameter)

Let's look at the simple case of using a Gaussian Distribution $p(x, y) = N(x, y | \mu, \sigma)$.
 $D \sim p(D | \theta)$. The parameter for the Gaussian distribution is $\theta = (\mu, \sigma)$ which leads to
 $x, y \sim p(x, y | \mu, \sigma)$

Learning: Estimating the parameters θ given the training data D ! This means μ should have the value which represents the most data which is where the data concentrates the most!

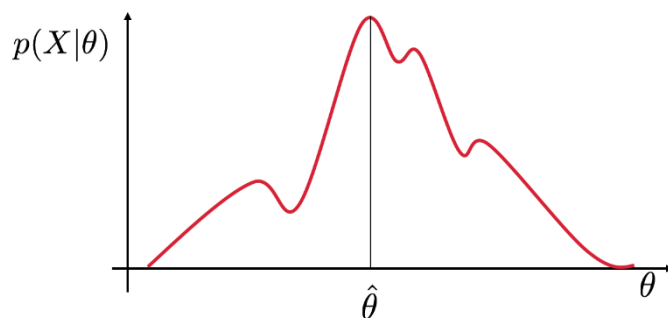
Likelihood of θ is defined as the probability that the data D was generated from the probability density function with parameters θ :

$$L(\theta) = p(D|\theta) = p((x_1, y_1) \dots, (x_n, y_n); \theta) = p(x_1, y_1; \theta) * \dots * p(x_n, y_n; \theta) = \prod_{n=1}^N p(x_n, y_n; \theta)$$

$$= \prod_{n=1}^N p(y_n|x_n; \theta) * p(x_i)$$

Assumes that the data is **independent and identically distributed (i.i.d)**

Maximum Likelihood Estimation: $\hat{\theta}_{ML} = \operatorname{argmax}_{\theta} p(D|\theta)$ seeks for the parameter which best explains the data D



Maximum log-likelihood Estimator Method: Its more convenient and numerical stable to maximize the log-likelihood w.r.t θ . For $N \gg 10$ is numerically impossible! Maximizing a sum of terms is always easier than maximizing a product. The difficulty of expressing the derivative of a long product of terms!

$$J_{ML}(\theta) = \log p(D|\theta)$$

$$= \sum_{i=1}^N \log p(y_i|x_i; \theta)$$

+ const with Gaussina distributed errors proportional to $-\frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - f_{\theta}(x_i))^2$

$$\hat{\theta}_{ML} = \operatorname{argmax}_{\theta} J_{ML}(\theta) = \operatorname{argmax}_{\theta} \log p(D|\theta)$$

Maximizing the likelihood is the same as minimizing the KL divergence. Suppose we want to fit a parameterized distribution $q(x; \theta)$ to the data from $p(x)$:

$$\begin{aligned} & \min_{\theta} D_{KL}[p(x)||q(x; \theta)] \\ &= \min_{\theta} E_{p(x)} [-\log q(x; \theta)] - E_{p(x)} [-\log p(x)] \\ &= \min_{\theta} E_{p(x)} [-\log q(x, \theta)] \\ &= \min_{\theta} - \int p(x) \log q(x; \theta) dx \\ &= \max_{\theta} \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N \log q(x_i; \theta), \text{ with } x_i \sim p(x) \end{aligned}$$

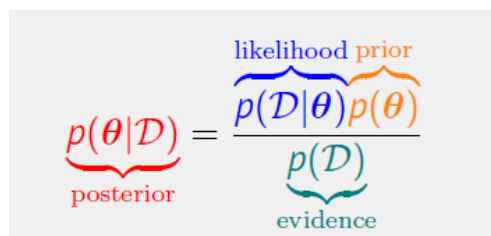
5.2 Maximum A Posteriori (MAP)

Maximizing $p(D|\theta) = p(y|X, \theta)p(X)$ implies maximizing the accuracy of the reproduction of the outcomes.

What if there is no true θ^* ?: θ becomes a random variable! Quantity of interest becomes $p(\theta|D) = p(\theta|X, y)$

But how can we obtain this quantity?: Bayesian Thinking

Bayesians: Parameters are just random variables, encode your subjective belief into a prior. The Bayes rule states:



The diagram shows Bayes' theorem with labels for each part of the equation. The posterior $p(\theta|D)$ is labeled 'posterior' in red. The numerator consists of the likelihood $p(D|\theta)$ labeled 'likelihood' in blue and the prior $p(\theta)$ labeled 'prior' in orange. The denominator is the evidence $p(D)$ labeled 'evidence' in green. Brackets group the terms accordingly.

$$\underbrace{p(\theta|D)}_{\text{posterior}} = \frac{\overbrace{p(D|\theta)}^{\text{likelihood}} \overbrace{p(\theta)}^{\text{prior}}}{\underbrace{p(D)}_{\text{evidence}}}$$

If you assign each parameter estimator a “probability of being right,” you can choose better among imperfect estimators or even do “smarter combinations”. The average of these estimators will be better than a single parameter. This is how it works:

1. **Specify prior:** Begin with a prior probability distribution, denoted as $p(\theta)$, that represents your initial beliefs or knowledge about the parameter θ before observing any data. This can be informative or not!
2. **Calculate likelihood function:** The likelihood function, denoted as $p(D|\theta)$, which represents the probability of modelling/observing the data we want to model can be modeled by a parameter θ
3. **Calculate Posterior:** Use Bayes' theorem to calculate the posterior distribution, denoted as $p(\theta|D)$, which represents the updated probability that our parameter θ can model the given data D we want to model?
4. **In the next iteration use Posterior as Prior:** In the next iteration, use the posterior obtained from the previous step as the new prior. The posterior becomes the updated belief about θ after observing the current data.
5. **Repeat Iteratively:** Repeat steps 2 to 4 for each subsequent iteration, updating the prior with the new posterior obtained from the previous step. This iterative process allows the Bayesian analysis to adapt and incorporate more information as new data becomes available. As the number of iterations increases, the posterior distribution converges to the most appropriate representation of our knowledge about the parameter θ , considering both the prior information and the data.

Encode Beliefs: You can add more assumptions into your prior e.g., θ should be small $p(\theta) = N(\theta|0, W^{-1})$

Goal: Find parameters that maximize the posterior! For that we can use the log-trick again:

$$J_{MAP}(\theta) = \log p(\theta|D) \propto \log p(D|\theta) + \log p(\theta) = \log \prod_{n=1}^N p(y_i|x_i; \theta) p(x_i) + \log p(\theta)$$

$$\hat{\theta}_{MAP} = \operatorname{argmax}_{\theta} J_{MAP}(\theta) = \operatorname{argmax}_{\theta} \log p(\theta|D)$$

5.3 Ridge Regression

Ridge Regression is used as a regulator and prevents overfitting. It penalizes the slope of the function. So it adds bias to gain less variance, because a smaller slope leads to predictions that are less sensitive to changes in the input. It can make a huge difference in comparison to normal regression!

5.4 Bayesian Prediction

Assumption: We have good parameters θ^* (e.g. from ML or MAP). We can do prediction with:

$$p(\underbrace{\mathbf{y}_p}_{\text{prediction}} \mid \underbrace{\mathbf{x}_q}_{\text{query point } \mathcal{D}}; \underbrace{\theta^*}_{\text{parameters}}) = \mathcal{N}(\mathbf{y}_p \mid \mu(\mathbf{x}_q), \sigma^2(\mathbf{x}_q))$$

$$\begin{aligned} \mu(\mathbf{x}_q) &= \phi^T(\mathbf{x}_q)\theta^* && \text{Predictive mean} \\ \sigma^2(\mathbf{x}_q) &= \sigma^2 && \text{Predictive variance} \end{aligned}$$

Can we estimate our uncertainty in θ ? Our parameter estimate is also noisy that's why!
 Yes by computing the probability of θ given the data! Remember the posterior! We can get the posterior with
 By applying the bayes theorem for gaussians we can the posterior with:

$$\begin{aligned} p(\theta \mid \mathbf{y}, \Phi) &= \mathcal{N}(\theta \mid \mu_N, \sigma^2 \Sigma_N) \\ \Sigma_N &= (\Phi^T \Phi + \sigma^2 \mathbf{W}^{-1})^{-1} \\ \mu_N &= \Sigma_N \Phi^T \mathbf{y} \end{aligned}$$

We can sample from it to estimate uncertainty $\theta \sim p(\theta \mid D) = (\theta \mid \mu_N, \Sigma_N)$

Full Bayesian Prediction:

We don't care about parameter θ , but predictions $p(y_p \mid x_q, D)$

- for query point x_q
- given data set D

If you assign each parameter estimator a "probability of being right", the average of these estimators will be better than a single one

Similar to Wisdom of the Crowds!

5.5 Bayesian Linear Regression

Prediction in closed form

$$p(\underbrace{\mathbf{y}_p}_{\text{prediction}} | \underbrace{\mathbf{x}_q}_{\text{query point } \mathcal{D}}, \underbrace{\Phi, \mathbf{y}}_{\text{training data } \mathcal{D}}) = \int p(\mathbf{y}_p | \mathbf{x}_q, \theta) p(\theta | \Phi, \mathbf{y}) d\theta$$

Predictive distribution is Gaussian again

$$\begin{aligned} p(\mathbf{y}_p | \mathbf{x}_q, \Phi, \mathbf{y}) &= \mathcal{N}(\mathbf{y}_p | \mu(\mathbf{x}_q), \sigma^2(\mathbf{x}_q)) \\ \mu(\mathbf{x}_q) &= \phi^T(\mathbf{x}_q) \left(\Phi^T \Phi + \sigma^2 \mathbf{W}^{-1} \right)^{-1} \Phi^T \mathbf{y} \\ \sigma^2(\mathbf{x}_q) &= \underbrace{\sigma^2 (1 + \phi^T(\mathbf{x}_q) \Sigma_N \phi(\mathbf{x}_q))}_{\text{state dependent variance}} \end{aligned}$$

The variance depends on the information in the data!

6. Feature Construction Basics

Feature construction is a powerful technique used in machine learning to introduce non-linearity into the model. By creating new features or transforming existing ones in a way that captures non-linear relationships, the model can better represent complex patterns present in the data. Non-linear relationships are common in real-world data, and feature construction allows the model to better capture those relationships, leading to improved predictive performance.

6.1 XOR Problem

The XOR problem is a binary classification task based on the exclusive OR (XOR) logical operator. It involves finding a decision boundary to separate data points into two classes (0 and 1) in a way that no straight line can achieve due to the non-linearity of the XOR function. The XOR problem can be solved using non-linear models like multi-layer neural networks, which can learn complex patterns and capture the non-linear relationships between the input features to correctly classify the data. One way to also construct new features for the XOR problem is to introduce polynomial features.

6.2 Handcrafted Features

Discrete Inputs

- **One-Hot Code:** There can only be one

$$\phi_i(x) = \begin{cases} 1 & \text{if } \delta(x - x_i) \\ 0 & \text{otherwise} \end{cases}$$

Continuous inputs

Polynomial features $\phi_i(\mathbf{x}) = \prod_{j=1}^k x_j^{c_{ij}}$ with constants c_{ij}

Fourier basis $\phi_i(\mathbf{x}) = \cos(\pi \mathbf{x}^\top \mathbf{c}_i)$ with constants \mathbf{c}_i

Tabular: One-hot discretization (often called binning)

$$\phi_i = h \left(1 - \left\| \frac{\mathbf{x} - \mathbf{x}_i}{2\delta x} \right\|_1 \right)$$

Radial basis functions (RBF): Soft tile coding – more later!

7. Automatic Linear Feature Construction

- Hand coding is certainly not enough. . .
- Automatic feature generation is needed!
- We will treat one example:
 - Automatic adaptation of RBF Networks
- There is way more, e.g.:
 - Regression trees
 - Random projections
 - Clustering, e.g., clusters become features
 - Density estimation, e.g., mixture components become features

So how do we learn good RBFs? Learning θ was easy (=linear!) But learning μ_i, Σ_i is difficult because of nonlinearity!

But we can use numerical optimization by computing gradient w.r.t. validation error

$$\nabla_{\theta} J(\theta) = -\frac{1}{2N_{\text{validation}}} \sum_{i=1}^{N_{\text{validation}}} (y_i - \phi(x_i)^\top \theta) \nabla_{\theta} \phi(x_i)^\top \theta$$

Go down steepest direction ($-\nabla_{\theta} J(\theta)$) to a (local) minimum with learning rate α_k :

$$\theta_{k+1} = \theta_k - \alpha_k \nabla_{\theta} J(\theta)$$

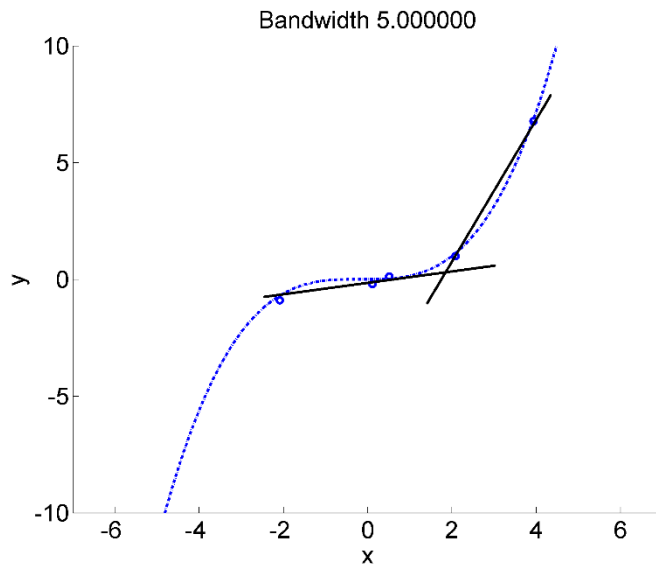
8. Non parametric Approaches

If you choose to have one feature/basis function per sample, you have a nonparametric method.

Don't need to select the number of basis. Nonparametric means infinitely many parameters, and not zero

parameters! Expressiveness of the model depends on the number of data points. No predetermined parametric form necessary (e.g. 5th-degree polynomial). One of them is **locally-weighted linear regression**.

Locally all data is linear so we can take the neighboring data points to predict a solution.



We use a higher importance or weighting of neighboring data points determined by bandwidth / length scale l .

For each query point x_q , weight training points x_i by:

$$w_i(x) = \exp\left(-\frac{\|x_q - x_i\|}{2l^2}\right)$$

8.1 Weighted Linear Regression

Weighted cost function:

$$J = \frac{1}{2} \sum_{i=1}^N w_i(\mathbf{x}_q) (y_i - f_{\theta}(\mathbf{x}_i))^2, \quad w_i(\mathbf{x}_q) = \exp\left(-\frac{\|\mathbf{x}_q - \mathbf{x}_i\|^2}{2l^2}\right)$$

The function is linear in \mathbf{x} :

$$f_{\theta} = \theta^T \begin{bmatrix} 1 & \mathbf{x} \end{bmatrix} = \theta^T \tilde{\mathbf{x}}$$

In matrix form with $\mathbf{W} = \text{diag}(w_1, w_2, \dots, w_n)$:

$$J = \frac{1}{2} (\tilde{\mathbf{X}}\theta - \mathbf{y})^T \mathbf{W} (\tilde{\mathbf{X}}\theta - \mathbf{y})$$

The solution to this problem: *weighted pseudo inverse*

$$y_q = x_q^T \theta = x_q^T (\tilde{X}^T W \tilde{X})^{-1} \tilde{X}^T W y$$

⇒ **W** can be large – don't implement it like this...

⇒ Dismiss data points with small weights

Local ridge regression:

$$y_q = x_q^T \theta = x_q^T (\tilde{X}^T W \tilde{X} + \sigma^2 I)^{-1} \tilde{X}^T W y$$

Advantages: Fast (real-time capable), Scales (lots of data),

Interpolates linearly (useful in control)

Disadvantages: Tuning is not easy

Frequent method of choice for control problems!

9. Wrap-Up

- Basic problem types of machine learning
- The six machine learning choices:
 - Problem class & assumptions, and evaluation
 - Model type & class, and algorithmic realization
- **Evaluation:** Overfitting is bad, model selection
- ML algorithms usually are derived by (1) minimize cost functions like least squares and/or (2) using probability calculus
- **Frequentists:** Find the “true” parameters of a model in our model class.
- **Bayesians:** Parameters are random variables.
 - **MAP:** Most likely parameters given our assumptions
 - **Full Bayesian:** Best prediction over all models!

Neural Networks

What neural networks are and how they relate to the brain?

Recap: We know how to solve linear models of the form: $\hat{y} = f_w(x) = w^T \phi(x)$. Either in closed-form with Linear Algebra or via Gradient descent (e.g., with a convex loss). The question that still arises is how should we choose $\phi(x)$?

The basic idea of NN is to also learn the features $\phi_\theta(x)$, parameterized by θ !

So the model is $\hat{y} = f_\theta(x) = w^T \phi_\theta(x)$

Example: Suppose we have a regression problem and want to minimize the mean-squared error:

$$\begin{aligned} L_\theta &= \mathbb{E}_{\mathcal{D}} \left[\frac{1}{2} (y - \hat{y})^2 \right] \\ &= \mathbb{E}_{\mathcal{D}} \left[\frac{1}{2} (y - \mathbf{w}^T \phi_\theta(\mathbf{x}))^2 \right] \end{aligned}$$

We now want to find:

$$\begin{aligned} \theta^* &= \arg \min_{\theta} L_\theta \\ &= \arg \min_{\theta} \mathbb{E}_{\mathcal{D}} \left[\frac{1}{2} (y - \mathbf{w}^T \phi_\theta(\mathbf{x}))^2 \right] \end{aligned}$$

If $\phi_\theta(x)$ is non-linear in θ , also L_θ is non-linear in θ . Contrary to the linear model, L_θ is no longer a convex objective

And difficult to optimize! In fact, L_θ can be highly non-convex!

Abstract neuron model: $y = f(w^T x + b)$

- Input x
- weights w
- Bias b
- Activation function f

For simplicity we include the bias in the weight matrix

$w = [w^T, b]^T$ and extend x with a 1 $x = [x^T, 1]^T \rightarrow \mathbf{y} = \mathbf{f}(\mathbf{w}^T \mathbf{x})$

Neural networks in the brains are often determined by the sheets of tissue

- Sheets = Vectors of Neurons
- For simplicity in synthesis and analysis

We pool neurons together in layers of m inputs and n outputs, where each layer has

- Weight matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$ (with bias included)

- Input vector $x \in R^{m \times 1}$
- Pre-activation vector $z = Wx \in R^{n \times 1}$
- Output vector $y = f(z)$ with $f: R^{n \times 1} \rightarrow R^{n \times 1}$ (f is the activation function)

How do neural networks build stacks of feature representations?

Using multiple n hidden layers with different sizes and activation functions.
These are called Multi-Layer Perceptron (MLP).

Each layer can be seen as a feature transformation, producing features for the next layer. Complex representations/features are possible! We can see a Multi-Layer network as a stack that builds features (of features (of features (of features (of features (...))))

$$\begin{aligned}
 y &= W_n \left(\dots f_3 \left(W_3 f_2 \left(\underbrace{W_2 f_1 (W_1 x)}_{\phi(x; W_2)} \right) \right) \right) \\
 &\quad \underbrace{\hspace{10em}}_{\phi(x; W_1, W_2)} \\
 &\quad \underbrace{\hspace{15em}}_{\phi(x; W_1, \dots, W_{n-1})} \\
 &= W_n \phi(x; W_1, \dots, W_{n-1})
 \end{aligned}$$

What kind of neural networks are there?

- **Feedforward Neural networks:** Fully connected Multi-Layered Perceptron's (MLPs), Convolution Neural Networks (CNNs)
- Recurrent neural networks (RNNs)

What activation functions are there?

- **Sigmoid:** E.g. for classification

$$\begin{aligned}
 f(z) &= \sigma(z) = \frac{1}{1 + \exp(-z)} \\
 f'(z) &= \sigma(z)(1 - \sigma(z))
 \end{aligned}$$

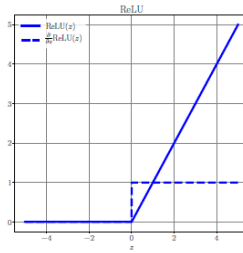
- **Hyperbolic Tangent (tanh)**

$$\begin{aligned}
 f(z) &= \tanh(z) \\
 f'(z) &= 1 - \tanh^2(z)
 \end{aligned}$$

- **Rectified Linear Unit (ReLU):** e.g. for regression

$$f(z) = \max(0, z)$$

$$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$



If a network of one layer is enough, but in practice not a good idea?

Universal Function Approximation Theorem states that one hidden layer can represent every function arbitrarily accurate! Even though true, we would need an exponential number of units. Instead, multiple layers allow for a similar effect with less units!

$$\# \text{regions} = O\left(\left(\frac{n}{d}\right)^{d(l-1)} n^d\right) \quad \text{with} \quad \begin{cases} n & \text{Number of neurons per layer} \\ l & \text{Number of hidden layers} \\ d & \text{Number of inputs} \end{cases}$$

Exponential growth in regions

$d = 1$	$l = 1$	$l = 2$	\dots	$l = k$
Regions	$O(n)$	$O(n^2)$	\dots	$O(n^k)$

How to do forward and backpropagation?

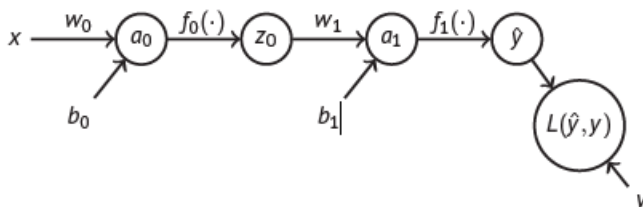
The algorithm to optimize NNs is commonly known as Forward and Backward Propagation.

Forward propagation

In forward propagation compute

- Pre-activations and activations at each hidden layer
- Output at the output layer
- (Scalar) loss function

Example: It's pretty easy. Just don't forget to add the bias since we don't include it in the weight matrix:



$$\begin{aligned}
a_0 &= w_0 x + b_0 \\
z_0 &= f(a_0) \\
a_1 &= w_1 z_0 + b_1 \\
\hat{y} &= f_1(a_1) \\
L &:= L(\hat{y}, y)
\end{aligned}$$

Backward propagation

1. Compute the contribution of each parameter to the loss (=gradient)

$$\nabla_{\theta} L(\theta) = \left[\frac{\partial L}{\partial \theta_1}, \dots, \frac{\partial L}{\partial \theta_n} \right]^T$$

Use the chain rule:

$$\begin{aligned}
\frac{\partial f(g(x))}{\partial x} &= \frac{\partial f}{\partial g} \frac{\partial g}{\partial x} \\
\frac{\partial f(g(x), h(x))}{\partial x} &= \frac{\partial f}{\partial g} \frac{\partial g}{\partial x} + \frac{\partial f}{\partial h} \frac{\partial h}{\partial x}
\end{aligned}$$

For example (if the next stage does not depend on the derivative skip it!)

$$\frac{\partial L(\hat{y}(a_1(w_1)))}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1} \frac{\partial a_1}{\partial w_1}$$

2. Update each parameter with gradient descent

$$\theta^{k+1} \leftarrow \theta^k - \alpha \nabla_{\theta} L(\theta)$$

Learning rate α

Gradient from Backpropagation $\nabla_{\theta} L$

Automatic Differentiation

For simple networks and loss functions you can compute the gradients by hand - symbolic differentiation. But for larger and more complicated ones it is easy to make mistakes! In case you implement gradient methods, test them with Finite Differences!

$$\frac{\partial L}{\partial \theta_j} \approx \frac{L(\theta + \epsilon \mathbf{u}_j) - L(\theta - \epsilon \mathbf{u}_j)}{2\epsilon}$$

Different ways of doing fast gradient descent? Full, stochastic, mini-batch? When to update θ ?

Full Gradient Descent: Use the full training set $\{(x_i, y_i)\}_{i=1, \dots, n}$. But this computationally **expensive** for large n !

$$\nabla_{\theta} J = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L_f(x_i, y_i, \theta)$$

Stochastic Gradient Descent: Use one data point of the training set. But this needs adaptive learning rate

η_t with $\sum_{t=1}^{\infty} \eta_t = \infty$ and $\sum_{t=1}^{\infty} \eta_t^2 < \infty$. High variance gradient estimation!

$$\nabla_{\theta} J \approx \nabla_{\theta} L_f(x_i, y_i, \theta)$$

Mini-batch Gradient Descent: Nowadays also called Stochastic Gradient Descent. Use a batch of the training set:

$$\nabla_{\theta} J = \frac{1}{k} \sum_{i=1}^k \nabla_{\theta} L_f(x_i, y_i, \theta) \text{ with } k < n$$

In practice: balance mini-batches approximately by random shuffling of the training data

Speedup training via learning rate adaptation?

We can use optimizer **Adam** which combines momentum term with Adadelta to speed up training.

Momentum and **Adadelta** are optimizers too.

For small networks **hessian approaches**, **Conjugate gradient** or **Levenberg-Marquart** are also possibilities to speedup training!

How to initialize the parameters?

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 neural networks overfit and what you can do to about it?

Neural Networks are prone to overfit because they can contain hundreds, thousands and (sometimes) even millions of parameters. In most cases we do not have datasets with millions of datapoints!

Fight overfitting with an algorithmic realization of a prior:

- Regularization
- Early stopping
- Input noise augmentation
- Dropout

Covariate shift can make training hard. Batch normalization can solve this!



Questions

1. Robotics in a Nutshell

What is task-space control?

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.

Given the joint state of a robot, which model is used to compute the end-effector position?

Using the forward kinematics model.

Given the joint state of a robot, which model is used to compute the torques/forces applied by the physics?

Using the inverse dynamics model.

Given the desired end-effector state of a robot, which model is used to compute the joint positions to achieve it?

Using the inverse kinematics model.

How to compute the forward kinematics?

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.

What are the limitations of the P-controller?

It oscillates around the desired position and does not include velocity-control.

How can model-based control deal with mismatches between the real system and the model?

Using feedforward control, a model-based controller is combined with a “standard” PD-controller to eradicate modeling errors.

How to compute the analytical solution for inverse kinematics?

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.

What are a few examples in which null-space control would make sense.

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 joints are located around the center.

Assume that your robot has two different control settings, depending on its working environment. The first setting is designed for factory environments, while the second setting is designed for home operation. Both control settings are based on a PD controller.

a) Explain which gains (i.e. high or low gains) you would use for each of the settings and why.

Factory environment: The robot would be in a structured environment with no obstacles like humans etc. and repetitive as well as precise tasks! It's not dangerous for humans. High gains for fast response to error and stable operation!

Home operation: Less structured setting where safety is more important. Low gains for gentle control and not overreaction by disturbances. Goal is to not hurt any humans or things!

b) Assuming you want to ensure that the robot precisely follows the desired trajectory, which additional term should you add to the PD controller? Explain your choice.

I would add a model to compensate for the gravity. So, a PD Controller with gravity compensation. The reason for that is.

Given a certain desired cartesian position for the end-effector of our robot, we wish to calculate the robot's joint configuration that guarantees the desired cartesian position.

a) What is this problem called in robotics?

Inverse Kinematics

b) Let us assume that our robot has multiple joint configurations that guarantees the tip position to be in the desired cartesian position. We now have the problem of deciding which one we should choose. State two variations that can be used to solve this problem!

Differential Inverse kinematics: Jacobian Transpose

Task-Prioritization with Null-Space Movements

c) Write down the optimization problem each method is trying to solve.

Jacobian Transpose: The task-space error needs to be minimized.

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

Task-Prioritization with Null-Space Movements: This base task can be formulated with a P-controller.

$$\dot{\mathbf{q}}_0 = K_p(\mathbf{q}_{rest} - \mathbf{q})$$

The optimization problem then is as following:

$$\min_{\dot{\mathbf{q}}} (\dot{\mathbf{q}} - \dot{\mathbf{q}}_0)^T(\dot{\mathbf{q}} - \dot{\mathbf{q}}_0), \quad \text{s.t. } \mathbf{J}(\mathbf{q})\dot{\mathbf{q}} = \dot{\mathbf{x}}_d$$

What is a null space? What is task-prioritization with null space? Give a minimal example.

Now, in robotics, the null space is the subspace within the joint configuration space where additional joint motions do not affect the end-effector's position or the robot's primary task. In other words, the null space allows the robot to perform secondary or auxiliary motions, such as avoiding obstacles or adjusting joint angles for comfort, without interfering with the main task.

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**. This can be, for example, to push the robot into a rest position q_{rest} where it does not consume energy. This base task can be formulated with a P-controller:

$$\dot{q}_0 = K_p(q_{rest} - q)$$

The optimization problem then is as following:

$$\min_{\dot{q}} (\dot{q} - \dot{q}_0)^T (\dot{q} - \dot{q}_0), \quad \text{s.t. } J(q)\dot{q} = \dot{x}_d$$

The result of this optimization problem is:

$$\dot{q} = J^\dagger \dot{x}_d + (I - J^\dagger J) \dot{q}_0 = \dot{q}_{\text{nominal}} + \dot{q}_{\text{null}}$$

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

Write down and explain the equations for: a) P controller b) PD controller c) PD + gravity compensation. Intuitively explain what the limitations / problems of each of these controllers are.

P-Controller: Based on position error $u_t = K_p(q_d - q_t)$. It causes high oscillations in the position, making it not suitable for real control.

PD-Controller: Based on position and velocity errors $u_t = K_p(q_d - q_t) + K_D(\dot{q}_d - \dot{q}_t)$. There is still a steady state error 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 law. This approach is the most used in industrial robots!

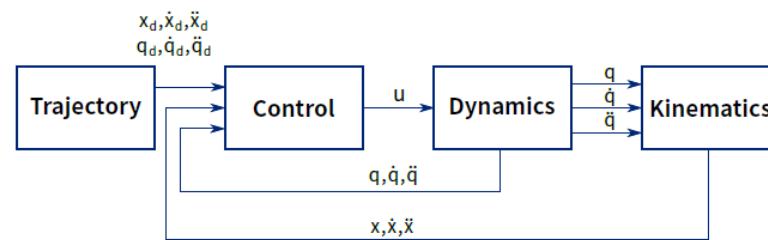
$$u_t = K_p(q_d - q_t) + K_D(\dot{q}_d - \dot{q}_t) + g(q)$$

PID-Controller: Alternatively, to doing gravity compensation we can estimate the motor command to compensate for the steady state error by integrating the error:

$$u_t = K_p(q_d - q_t) + K_D(\dot{q}_d - \dot{q}_t) + K_I \int_{-\infty}^T (q_d - q) d\tau$$

For tracking control, it may create havoc and disaster!

Make an illustrative sketch that shows and connects the following terms: dynamics, control, trajectory, kinematics, desired states, and actual states. Label all connections.



2. Discrete State-Action Optimal Control

A friend provides you with optimal V and Q function. Which one you want to choose for control where you get optimal actions for state?

We must choose the optimal Q-function

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q^*(s, a)$$

Define MDP, what is a crucial assumption?

A (non-stationary) Markov Decision Process is defined by:

- State space $s \in S$
- Action space $a \in A$
- Transition dynamics $p(s_{t+1}|s_t, a_t)$ depend only on the current time step!
- Reward $r_t(s, a)$
- Initial state properties $\mu_0(s)$

Crucial Assumption Markov property: The Markov property states that the future state of the system depends only on the current state and action, and not on the sequence of states and actions that led to the current state.

$$p_t(s_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, \dots) = p_t(s_{t+1} | s_t, a_t)$$

When are we using policy iteration instead of Value Iteration?

Policy iteration iterates policy evaluation and improvement until convergence to find the optimal policy. So, we need a good initial policy. Value Iteration also finds the optimal policy but iterates the Bellman equation directly! Value iteration is inefficient due to redundant max operations. Its good we don't have a good initial policy.

3. Continuous State-Action Optimal Control

What's the problem of discretizing?

The real world is not discrete but continuous! Separating the world in buckets, but this would cause an exponential explosion in memory. The number of discrete actions and states grows exponentially with the system's dimensions. This issue is known as the curse of dimensionality. We can use approximation like iLQR and DDP.

iLQR for inverse pendulum?

Approximates the dynamics $f(x_t, u_t)$ with a first order Taylor expansion.

$$x_{t+1} \approx f(x_t, u_t) + \begin{bmatrix} f_x^\top & f_u^\top \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix}$$

Approximates the cost with a second order Taylor approximation:

$$C(x + \Delta x, u + \Delta u) \approx C_0 + \begin{bmatrix} \Delta x^\top & \Delta u^\top \end{bmatrix} \begin{bmatrix} C_x \\ C_u \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \Delta x^\top & \Delta u^\top \end{bmatrix} \begin{bmatrix} C_{xx} & C_{xu} \\ C_{ux} & C_{uu} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix}$$

Why? Because it made the value function a quadratic function that we can compute in closed form!

What is the formula for transition dynamics und reward function in LQR? How does the reward function look for the Final step T? How do we get the optimal policy?

- (possible time dependent) linear transition dynamics with gaussian noise:
 $p_t(x_{t+1}|x_t, u_t) = N(x_{t+1}|A_t x + B_t u_t + b_t, \Sigma_t)$
- Quadratic reward function:

$$\begin{aligned} r_t(x, u) &= -(x - x_d)^T R_t (x - x_d) - u_t^T H_t u_t \\ r_T(x) &= -(x - x_d)^T R_T (x - x_d) = r_t(x, 0) \end{aligned}$$

5. Compute the value function for the last time step for all states: $V_T^*(x) = r_T(x)$
Solution:

$$V_T^*(x) = r_T(x) = -(x - x_d)^T R_T (x - x_d) = -(x - x_d)^T V_T (x - x_d)$$

6. To get from t+1 to t, first compute the Q function for all states and actions:
 $Q_t^*(x_t, u_t) = r_t(x_t, u_t) + E_p[V_{t+1}^*(x_{t+1})|x_t, u_t]$

Solution:

- Firstly, the expectation has to be computed by assuming a quadratic structure for the value function of time step t+1:

$$V_{t+1}^*(x) = -(x - x_d)^T V_{t+1} (x - x_d)$$

- Then the expectation becomes:

$$\begin{aligned} \mathbb{E}_{x' \sim p(\cdot|x, u)} [V_{t+1}^*(x') | x, u] &= \int V_{t+1}^*(x') p(x' | x, u) dx' \\ &= - \int (x - x_d)^T V_{t+1} (x - x_d) \mathcal{N}(x' | A_t x + B_t u_t + b_t, \Sigma_t) dx' \\ &= -(A_t x + B_t u_t + b_t - x_d)^T V_{t+1} (A_t x + B_t u_t + b_t - x_d) - \text{tr}(V_{t+1} \Sigma_t) \end{aligned}$$

- Plugging in the reward function yields the Q-function:

$$\begin{aligned} Q_t^*(x, u) &= -(x - x_d)^T R_t (x - x_d) - u^T H_t u \\ &\quad - (A_t x + B_t u_t + b_t - x_d)^T V_{t+1} (A_t x + B_t u_t + b_t - x_d) \\ &\quad - \text{tr}(V_{t+1} \Sigma_t) \end{aligned}$$

7. Compute the optimal policy for time step t for all states.
 $\pi_t^*(x) = \arg \max_u Q_t^*(x, u)$

Solution: Computing the policy is done by maximizing the Q-function w.r.t the action u . This means taking the derivative of it and setting it to zero:

$$\begin{aligned}\frac{\partial}{\partial u} Q_t^*(x, u) &= -2H_t u - 2B_t^T V_{t+1} (A_t x + B_t u_t + b_t - x_d) \\ &= -2H_t u - 2B_t^T V_{t+1} A_t x - 2B_t^T V_{t+1} B_t u_t - 2B_t^T V_{t+1} b_t + 2B_t^T V_{t+1} x_d \\ &= -2(H_t + B_t^T V_{t+1} B_t) u - 2B_t^T V_{t+1} (A_t x + b_t - x_d) \stackrel{!}{=} 0 \\ \pi_t^* = u^* &= -(H_t + B_t^T V_{t+1} B_t)^{-1} B_t^T V_{t+1} (A_t x + b_t - x_d)\end{aligned}$$

The optimal policy is a time-dependent linear feedback controller with time-dependent offset!

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

$$V_t^*(x) = Q_t^*(x, \pi_t^*(x))$$

Solution:

- To compute the value function, plug the obtained optimal policy into the Q-function.
- The optimal value function has a quadratic and linear form!
- The optimal value function is of the form $V_t(x_t) = x_t^T V_t x_t + x_t^T v_t$ and it is quadratic-linear.
- The optimal control input (optimal policy) is of the form $u_t^* = K_t(x_t - x_d) + k_t$, and is a time-varying P-controller!

iLQR uses A_{hut} and B_{hut} and $f(s_{t+1}, a_{t+1})$. How are these related and in which step are they calculated?

We can approximate the solution locally around an initial trajectory by linearizing the dynamics and/or the **rewards** using a first order and/or **second order** Taylor expansion around a point $(\tilde{x}_t, \tilde{u}_t)$.

Using the first order Taylor expansion for the dynamics yields:

$$\begin{aligned}x_{t+1} &= f_t(x_t, u_t) \approx f(\tilde{x}_t, \tilde{u}_t) + \frac{\partial f}{\partial x} (x_t - \tilde{x}_t) + \frac{\partial f}{\partial u} (u_t - \tilde{u}_t) \\ &= A_t x_t + B_t u_t + b_t \\ \text{with } A_t &= \left. \frac{\partial f}{\partial x} \right|_{x=\tilde{x}_t, u=\tilde{u}_t} \quad \text{and } B_t = \left. \frac{\partial f}{\partial u} \right|_{x=\tilde{x}_t, u=\tilde{u}_t}\end{aligned}$$

Using second order Taylor for the reward function yields

$$\begin{aligned}r_t(x_t, u_t) &\approx r(\tilde{x}_t, \tilde{u}_t) + \frac{\partial r}{\partial x} (x_t - \tilde{x}_t) + (x_t - \tilde{x}_t)^T \frac{\partial^2 r}{\partial x^2} (x_t - \tilde{x}_t) - u_t^T H_t u_t \\ &= -x^T R_t x + 2r_t^T x - u^T H_t u + \text{const} \\ \text{with } R_t &= -\frac{\partial^2 r}{\partial x^2} \quad \text{and } r_t = \frac{1}{2} \frac{\partial r}{\partial x} - \frac{\partial^2 r}{\partial x^2} \tilde{x}_t\end{aligned}$$

Given LQR-Maximizationproblem, calculate: optimal policy so that $dQ/da \neq 0$

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

$$\pi_t^*(x) = \arg \max_u Q_t^*(x, u)$$

Solution:

Computing the policy is done by maximizing the Q-function w.r.t the action u . This means taking the derivative of it and setting it to zero:

$$\begin{aligned} \frac{\partial}{\partial u} Q_t^*(x, u) &= -2H_t u - 2B_t^T V_{t+1} (A_t x + B_t u_t + b_t - x_d) \\ &= -2H_t u - 2B_t^T V_{t+1} A_t x - 2B_t^T V_{t+1} B_t u_t - 2B_t^T V_{t+1} b_t + 2B_t^T V_{t+1} x_d \\ &= -2(H_t + B_t^T V_{t+1} B_t) u - 2B_t^T V_{t+1} (A_t x + b_t - x_d) \stackrel{!}{=} 0 \\ \pi_t^* = u^* &= -(H_t + B_t^T V_{t+1} B_t)^{-1} B_t^T V_{t+1} (A_t x + b_t - x_d) \end{aligned}$$

The optimal policy is a time-dependent linear feedback controller with time-dependent offset!

Name one difference between DDP and LQR and one similarity

DDP approximates the Q function with a second order Taylor expansion while iLQR approximates the dynamics with a first order Taylor expansion. DDP uses the Hessian of the dynamics, which makes its steps more accurate but also more expensive. Due to the higher computational demand, iLQR is more commonly used nowadays. These methods require you to know the model!

What is one similarity and one difference between the dynamic programming solution to LQR and the DDP-based algorithms?

In iLQR, the second order dynamics terms get dropped in the Q-function approximation. Otherwise, the equations are exactly the same.

4. Machine Learning

Why does statistics matter to machine learning?

Solution:

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?

Solution:

How can I derive linear regression? What is ridge regression?

Solution:

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. Ridge Regression is used as a regulator and prevents overfitting. It penalizes the slope of the function. So it adds bias to gain less variance, because a smaller slope leads to predictions that are less sensitive to changes in the input

How do priors change our solution?

Solution:

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.

Priors also allow us to add an initial belief

What is maximum a priori? What is maximum likelihood? What is MAP?

Maximum a priori doesn't exist!

For a maximum likelihood estimator, the likelihood $p(D|\theta)$ is maximized w.r.t. to the parameters θ .

For a maximum a-posteriori estimator (MAP), the posterior $p(\theta|D)$ is maximized. This requires placing a prior $p(\theta)$

What is overfitting and how does it relate to the Bias-Variance Tradeoff?

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

$$L_{\hat{f}}(x) = \mathbb{E}_{\mathcal{D}}[(y(x) - \hat{f}_{\mathcal{D}}(x))^2] = \sigma_{\epsilon}^2 + \text{Bias}^2[\hat{f}_{\mathcal{D}}(x)] + \text{Var}[\hat{f}_{\mathcal{D}}(x)]$$

where $y(x) = f(x) + \epsilon$ 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?

For Bayesians the parameters are random variables meanwhile for frequentists the true parameters of a model are in the model class or there is a true model / there are true parameters!

What does parametric and non-parametric mean?

Nonparametric means infinitely many parameters, and not zero parameters!

- Use training data as model
- Data points become parameters
- Examples: Histograms, Nearest Neighbor, Kernel Density Estimation, LWR, Kernel Methods (SVM, GP, . . .)

Parametric models

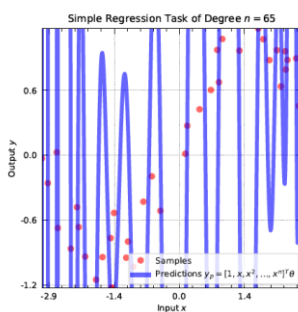
- Collapse training data onto parameters θ of a parameterized model class M_θ
- Use parameter values for predictions/description/decision making
- Examples: Mixture Models, LWPR, (Deep) Neural Networks, . . .

Exam: Bayes Theorem equation + description (like in mock exam)

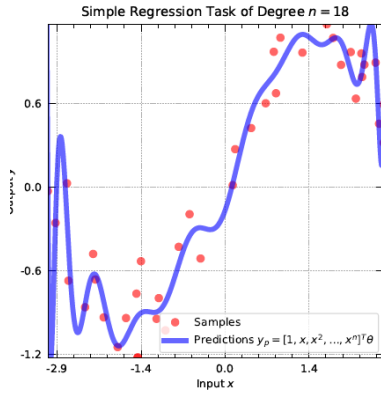
$$\underbrace{p(\theta|\mathcal{D})}_{\text{posterior}} = \frac{\overbrace{p(\mathcal{D}|\theta)}^{\text{likelihood}} \overbrace{p(\theta)}^{\text{prior}}}{\underbrace{p(\mathcal{D})}_{\text{evidence}}}$$

Exam: Plot of function fit with regression. Tell which plot has high degree of parameters, intermediate and low number of parameters.

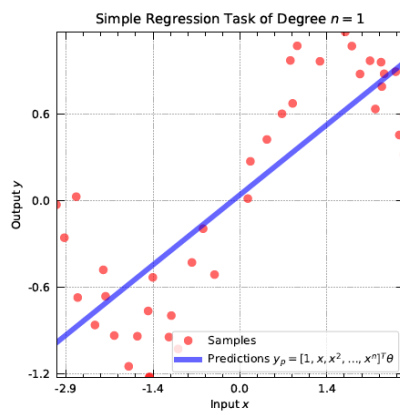
High



Intermediate



Low



Let $p(x)$ be the true data density function with $x \in \mathbb{R}^d$.

In maximum likelihood estimation we are interested in finding the optimal parameters θ^* of a family of parameterized distributions $q(x; \theta)$, that best match the true distribution p . We typically solve this problem by minimizing the forward KL divergence, $\theta^* = \min_{\theta} D_{\text{KL}}(p \| q_{\theta})$.

In this question, we want to minimize the KL divergence in the other "direction", i.e., $\min_{\theta} J(\theta)$, where $J(\theta) = D_{\text{KL}}(q_{\theta} \| p)$. To find the minimum, we are going to use gradient descent and therefore need to compute and **unbiased** gradient estimator for $\nabla_{\theta} J(\theta)$.

Give an expression, in the form of an expectation that can be solved by sampling, that computes an **unbiased** estimate of $\nabla_{\theta} J(\theta)$.

Assume:

- You know the mathematical expression of $q(x; \theta)$, you can sample from and evaluate it. You can differentiate q w.r.t. the input and parameters.
- You can evaluate the density $p(x)$, but cannot sample from it. $p(x)$ is **not differentiable** w.r.t. x .

$$f(x) = p(x) \text{ and } p(x; \theta) = q(x; \theta)$$

$$\begin{aligned}
\nabla_{\theta} \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x}; \theta)} [f(\mathbf{x})] &= \int_{\mathcal{X}} \nabla_{\theta} p(\mathbf{x}; \theta) f(\mathbf{x}) d\mathbf{x} \\
&= \int_{\mathcal{X}} p(\mathbf{x}; \theta) \frac{\nabla_{\theta} p(\mathbf{x}; \theta)}{p(\mathbf{x}; \theta)} f(\mathbf{x}) d\mathbf{x} \\
&= \int_{\mathcal{X}} p(\mathbf{x}; \theta) \nabla_{\theta} \log p(\mathbf{x}; \theta) f(\mathbf{x}) d\mathbf{x} \\
&= \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x}; \theta)} [\nabla_{\theta} \log p(\mathbf{x}; \theta) f(\mathbf{x})] \\
&\approx \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \log p(\mathbf{x}_i; \theta) f(\mathbf{x}_i), \quad \mathbf{x}_i \sim p(\mathbf{x}; \theta)
\end{aligned}$$

Describe the main idea behind Bayesian linear regression. Define all the relevant distributions. How do we perform predictions? Why is it useful?

Instead of assuming there are true parameters \mathbf{w} , Bayesian linear regression puts a prior $p(\mathbf{w})$ 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 \mathbf{w} , the input variable \mathbf{x} , and the predictive variable y , the posterior predictive distribution is

Prediction in closed form

$$p(\underbrace{\mathbf{y}_p}_{\text{prediction}} | \underbrace{\mathbf{x}_q}_{\text{query point } \mathcal{D}}, \underbrace{\Phi, \mathbf{y}}_{\text{training data } \mathcal{D}}) = \int p(\mathbf{y}_p | \mathbf{x}_q, \theta) p(\theta | \Phi, \mathbf{y}) d\theta$$

Predictive distribution is Gaussian again

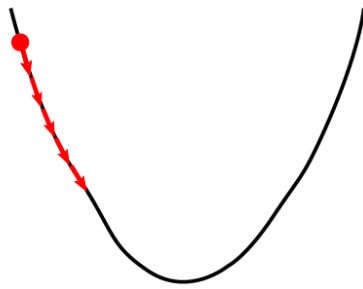
$$\begin{aligned}
p(\mathbf{y}_p | \mathbf{x}_q, \Phi, \mathbf{y}) &= \mathcal{N}(\mathbf{y}_p | \mu(\mathbf{x}_q), \sigma^2(\mathbf{x}_q)) \\
\mu(\mathbf{x}_q) &= \phi^T(\mathbf{x}_q) \left(\Phi^T \Phi + \sigma^2 \mathbf{W}^{-1} \right)^{-1} \Phi^T \mathbf{y} \\
\sigma^2(\mathbf{x}_q) &= \underbrace{\sigma^2 (1 + \phi^T(\mathbf{x}_q) \Sigma_N \phi(\mathbf{x}_q))}_{\text{state dependent variance}}
\end{aligned}$$

This results in \mathbf{x} -dependent variance in mean which can be computed in closed form for linear Gaussian models.

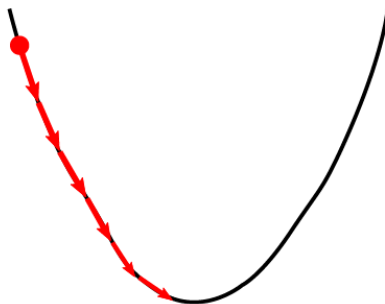
5. Neural Networks

Ein Diagramm mit 4 verschiedenen Verläufen. Zuordnen, welche learning rate aus 1.0, 0.1, 0.01, 0.001 zu welchem Verlauf gehört ?.

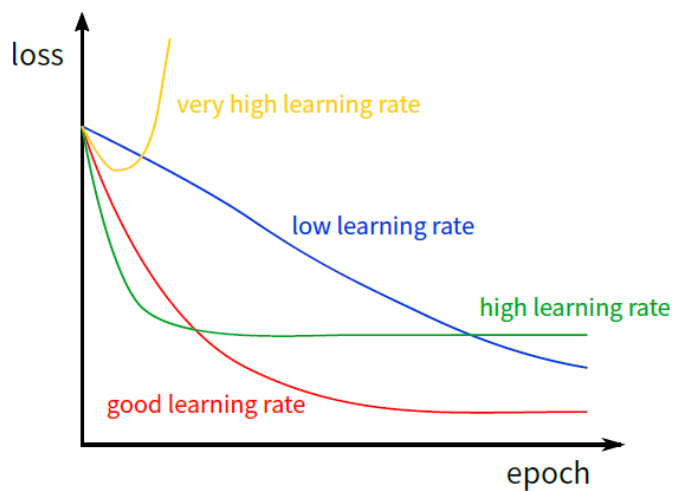
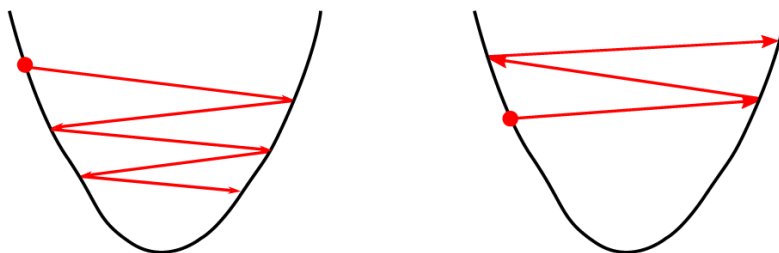
Very small learning rate



Good learning rate



■ **Large** learning rate (can converge slowly or even diverge)



Why NN are more complicated to optimize then quadratic optimization problems?

NNs are trained by using backpropagation. The problem is the loss function L_θ can be highly non-convex with many local optima! So we can get stuck in poor local optima!

Why even use NNs?

Example: Suppose we have a regression problem and want to minimize the mean-squared error:

The model is $\hat{y} = f_\theta(x) = w^T \phi_\theta(x)$

$$\begin{aligned} L_\theta &= \mathbb{E}_{\mathcal{D}} \left[\frac{1}{2} (y - \hat{y})^2 \right] \\ &= \mathbb{E}_{\mathcal{D}} \left[\frac{1}{2} (y - \mathbf{w}^T \phi_\theta(\mathbf{x}))^2 \right] \end{aligned}$$

We now want to find:

$$\begin{aligned} \theta^* &= \arg \min_{\theta} L_\theta \\ &= \arg \min_{\theta} \mathbb{E}_{\mathcal{D}} \left[\frac{1}{2} (y - \mathbf{w}^T \phi_\theta(\mathbf{x}))^2 \right] \end{aligned}$$

If $\phi_\theta(x)$ is non-linear in θ , also L_θ is non-linear in θ . Contrary to the linear model, L_θ is no longer a convex objective

And difficult to optimize! In fact, L_θ can be highly non-convex! That's why we use Neural networks!

Question 9

In neural networks, we typically deal with very high-dimensional parameters spaces. If θ are the parameters of a neural network and \mathcal{D} the dataset containing the training data, assume $L(\theta, \mathcal{D})$ is the loss function resulting from comparing the neural network predictions with the ground truth. We want to find the optimal parameters θ^* that minimize $L(\theta, \mathcal{D})$. Convex optimization solvers are known to work extremely well, even in very high-dimensions. Instead of doing gradient descent, why don't we use convex optimizers to train neural networks?

Example: Suppose we have a regression problem and want to minimize the mean-squared error:

The model is $\hat{y} = f_\theta(x) = w^T \phi_\theta(x)$

$$\begin{aligned} L_\theta &= \mathbb{E}_{\mathcal{D}} \left[\frac{1}{2} (y - \hat{y})^2 \right] \\ &= \mathbb{E}_{\mathcal{D}} \left[\frac{1}{2} (y - \mathbf{w}^T \phi_\theta(\mathbf{x}))^2 \right] \end{aligned}$$

We now want to find:

$$\begin{aligned}\theta^* &= \arg \min_{\theta} L_{\theta} \\ &= \arg \min_{\theta} \mathbb{E}_{\mathcal{D}} \left[\frac{1}{2} (y - \mathbf{w}^T \phi_{\theta}(\mathbf{x}))^2 \right]\end{aligned}$$

If $\phi_{\theta}(x)$ is non-linear in θ , also L_{θ} is non-linear in θ . Contrary to the linear model, L_{θ} is no longer a convex objective

And difficult to optimize! In fact, L_{θ} can be highly non-convex! That's why we use Neural networks!