Reinforcement Learning and Learning-Based Control

Hubert Cumberdale

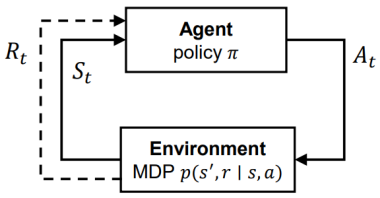
August 25, 2023

Contents

---

1 Introduction I: Reinforcement Learning (RL)

In the basic framework, an **agent** interacts with its **environment**, where the state captures everything relevant for the agent's decision-making using its **policy**, described using discrete random variables (DRVs) state $S_t \in \mathcal{S}$, action $A_t \in \mathcal{A}(S_t)$, and reward $R_t \in \mathcal{R}$ for time t.



The decision process the agent is facing is usually modelled by a *Markov Decision Process*. The interaction between agent and environment yields a *sequence* or *trajectory* $[S_t A_t R_{t+1}]^*$ for $t \in \mathbb{N}_0$.

1.1 Environment Interaction: Markov Decision Process (MDP)

The **dynamics** of an environment are captured as follows:

$$p(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r \,|\, S_t = s, A_t = a]$$

Definition: Markov

A stochastic process is **Markov**, if the distribution of the state depends only on the state at the previous time step:

$$\mathbb{P}[S_t|S_{t-1}] = \mathbb{P}[S_t|S_{t-1}, S_{t-2}, \dots]$$

Definition: MDP

A (finite) **Markov Decision Process** (MDP) is given by

$$\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p(\cdot|\cdot) \rangle \text{ or } \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p(\cdot|\cdot), r(\cdot) \rangle$$

with

- $\mathcal{S}, \mathcal{A}, \mathcal{R}$ finite sets of states, actions, and rewards
- p **dynamics**/transition model
- $r \in \mathcal{R}$ **reward function**

Variations of the MDP are

- *Markov Process* (MP) or *Markov Chain*: sequence of random states given by a (finite) set of states $\mathcal{S}$ and dynamics P
- *Markov Reward Process* (MRP): MP with an additional reward function $r \in \mathcal{R}$ given, e.g., as $r(s) = \mathbb{E}[R_{t+1}|S_t = s]$, and a discount factor $\gamma \in [0, 1]$

The dynamics P specify $\mathbb{P}[S_{t+1} = s'|S_t = s]$, and can be given as a *transition matrix*

$$P = (P_{ij})_{1 \le i,j \le n} = (\mathbb{P}[s_j|s_i])_{1 \le i,j \le n}$$

- $\sum_{j=1}^{n} P_{ij} = 1$ (sum over rows)
- $\sum_{j=1}^{n} \mathbb{P}[s_{ij}] = 1$ (sum over vector column of some state S)

The **state distribution** for n states at a given time t for given a transition matrix P is computed as

$$S_{t+1} = P^T S_t, \; S_t \in \mathbb{R}^n$$

A finite Markov chain has well-defined limiting behavior, if it is *irreducible* and *aperiodic*. Then, the *stationary* state distribution S_∞ has a unique solution

$$S_\infty = P^T S_\infty$$

If S_∞ exists, then it does not depend on the initial state, and can be interpreted to be an eigenvector of P^T with an eigenvalue of 1.

1.2 Policy, Reward, and Goal

The **policy** $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$ is the mapping from state to probabilities of selecting each of the possible actions, and is implemented by the agent. π is usually a *stochastic* policy, but we also can consider *deterministic* policies, in which $\pi(s) = a$. Note, that the reward does not directly affect the policy, but *indirectly* through the learning process.

Having the policy, one can compute the probabilities in a transition matrix using

$$p(s'|s) = \sum_{a \in \mathcal{A}} p(s'|s, a)\pi(a|s)$$

Given the reward function and a **discount factor** $\gamma \in [0, 1]$, we can compute the **return** G_t as the cumulative (discounted) reward for a state sequence starting at time t:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = R_{t+1} + \gamma G_{t+1}$$

The task is

- *episodic*, if $\gamma = 1$ and there is an absorbing state S_T, which, once reached, maps to itself and incurs 0 reward,
- *continuing*, otherwise.

Definition: Goal of RL

Starting at $S_0 = s_0$, we want to find a policy π that maximizes the expected return:

$$\max_\pi \mathbb{E}[G_0 | S_0 = \bar{s}_0]$$

The *reward hypothesis* assumes, that everything we care about is captured in the reward signal. Reward signals are design choices and require *reward shaping*.

---

2 Value Functions

Value Functions, given as *Bellman equations*, capture estimated future rewards depending on future actions and the policy.

Definition: State-Value Function

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) q_\pi(s,a) = q_\pi(s, \pi(s)) \end{aligned}$$

Definition: Action-Value Function

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r | s, a)(r + \gamma v_\pi(s')) \end{aligned}$$

2.1 Dynamical Programming (DP)

In the following, we assume *knowledge of the dynamics* p (model).

Definition: Bellman (Optimality) Equation for v_π

$$\begin{aligned} v_*(s) &= \max_a \sum_{s',r} p(s', r | s, a)(r + \gamma v_*(s')) \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a q_*(s,a) \end{aligned}$$

Definition: Bellman (Optimality) Equation for q_π

$$\begin{aligned} q_*(s,a) &= \max_a \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | \\ &\qquad S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned}$$

The **Policy Iteration** algorithm (as a *DP* task) iteratively optimizes π using **Bellman (optimality) equations** as update rules:

1. Evaluate the current policy by approximating $V \approx v_\pi$. (*Policy Evaluation*)
2. Using v_π, predict action maximizing q_π (*greedy*), resulting in an improved policy π'. (*Policy Improvement*)

We use the term **Generalized Policy Iteration** (GPI) to refer to the general idea of letting policy-evaluation and policy-improvement processes interact.

Algorithm: Policy Iteration

1. $V(s) \in \mathbb{R}, \pi(s) \in \mathcal{A}(s)$ arbitrary $\forall s \in \mathcal{S}$, $V(terminal) = 0$
2. **Policy Evaluation:**
   For each $s \in \mathcal{S}$:
   $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))$
   Until changes are small.
3. **Policy Improvement:**
   For each $s \in \mathcal{S}$:
   $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   If no more changes, stop and return $V \approx v_*$, $\pi \approx \pi_*$; else go to 2.

For policy improvement, we utilize the *policy improvement theorem*:

Theorem: Policy Improvement Theorem

If for deterministic policies π, π'

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \quad \forall_{s \in \mathcal{S}}$$

then

$$v_{\pi'}(s) \geq v_\pi(s) \quad \forall_{s \in \mathcal{S}}$$

To speed up the policy evaluation, we can utilize *in-place* updates, where in $V_{k+1}(s)$, we replace $V_k(s')$ by $V_{k+1}(s')$, if it is available.

Multiple sweeps through the state set may be computationally prohibitive. **Value Iteration** combines one sweep of policy evaluation with one sweep of policy improvement. The idea is to spend less effort on learning value functions of suboptimal policies and therefore speed up convergence.

One can show, that all of these algorithms converge to an optimal policy for *discounted finite MDPs*.

Algorithm: Value Iteration

1. $V(s) \in \mathbb{R}, \pi(s) \in \mathcal{A}(s)$ arbitrary $\forall s \in \mathcal{S}$, $V(terminal) = 0$
2. For each $s \in \mathcal{S}$:
   $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))$
   Until changes are small.
3. Output $\pi \approx \pi_*$ s.t.
   $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

2.2 On-Policy and Off-Policy Methods

We can divide the principal objective for an RL agent into two parts, namely *exploiting* current knowledge and *exploring* new states. This results in two conceptual approaches:

1. **On-policy methods**: Only one policy to be learned and used to generate data (*behavior policy* and *target policy* are the same)
2. **Off-policy methods**: One policy is learned (*target policy*), another is used to generate data (*behavior policy*)

---

3 Monte Carlo (MC)

If we want to improve our policy *without* a model p, we can use the statistical **Monte Carlo** approach. In essence, we replace the dynamics model with *samples*, i.e., data from interaction with the environment. MC then solves the problem by averaging sample returns. To ensure that well-defined returns are available, here we define MC only for *episodic* tasks.

3.1 Predictions (Policy Evaluation)

As before, the first step of the Monte Carlo method is to estimate the value function v_π by evaluating the currently given policy π.

3.1.1 First-Visit and Every-Visit MC

Using the returns $G_{t,i}$ for trajectory i and time t, we can estimate v_π by the average of these returns

$$v_\pi(s) := \mathbb{E}[G_t | S_t = s] \approx \hat{V}(s) = \frac{1}{M} \sum_{i=1}^{M} G_{t,i}$$

which is used in the **First-Visit MC**, where we consider only the first visit of a state within the trajectory for the average, and in **Every-Visit MC**, where we consider every visit.

Algorithm: First-Visit MC/Every-Visit MC

For each episode:
  Create trajectory $S_0, A_0, R_1, \ldots, R_T, S_T$ following π
  $G \leftarrow 0$
  For $t = T-1, ..., 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    ~~If $S_t \notin (S_i)_{i=0}^{t-1}$:~~ (Every-Visit)
      $Returns(S_t)$.append(G)
      $V(S_t) \leftarrow$ avg$(Returns(S_t))$

Definition: Law of Large Numbers

For random variables X_i:

$$\frac{1}{n} \sum_{i=1}^{n} X_i \approx \mathbb{E}[X]$$

Assuming

- i.i.d. variables
- integrability (expected value w.r.t. the Lebesgue measure exists)
- bounded variance

First-Visit MC creates no bias, but high variance, since only first-visits are considered. The *Law of Large Numbers* ensures convergence for First-Visit MC. Every-Visit MC, however, is biased. For smaller amounts of data, Every-Visit MC has a better average MSE.

3.2 Control (Policy Improvement)

Estimate $q_\pi \approx Q$, s.t.

$$\pi(S_t) \leftarrow \underset{a}{\text{argmax}}\, Q(S_t, a)$$

In order to do so, we need to make the following assumptions:

- **Exploring starts**: Every state-action pair has a *non-zero probability* of being selected at the start
- **Infinite number of episodes**: Number of episodes goes to infinity

Algorithm: MC Exploring Starts ($\pi \approx \pi_*$)

For each episode:
  Sample (S_0, A_0) randomly
  Create trajectory $S_0, A_0, R_1, \ldots, R_T, S_T$ using π
  $G \leftarrow 0$
  For $t = T-1, ..., 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    If $(S_t, A_t) \notin ((S_i, A_i))_{i=0}^{t-1}$:
      $Returns(S_t, A_t)$.append(G)
      $Q(S_t, A_t) \leftarrow$ avg$(Returns(S_t, A_t))$
      $\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$

3.2.1 With Exploring Starts

For the **Monte Carlo Exploring Starts** algorithm, we weaken the *infinite number of episode* assumption by bounding the amount of episodes used. MC ES cannot converge to suboptimal fix points.

3.2.2 Without Exploring Starts and ε-Greedy Methods

We remove the *exploring starts* assumption. The idea of **ε-Greedy Methods** is to add random *exploration* behavior to the greedy policy improvement:

Choose uniformly random action with small probability $\varepsilon > 0$ (**exploration**), and greedy action with probability $1 - \varepsilon$ (**exploitation**).

Replacing $\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ in MC ES by

$$A^* \leftarrow \underset{a}{\text{argmax}}\, Q(S_t, a)$$

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} \varepsilon/|\mathcal{A}(S_t)| & \text{, if } a \neq A^* \\ (1-\varepsilon) + \varepsilon/|\mathcal{A}(S_t)| & \text{, if } a = A^* \end{cases}$$

Definition: ε-Soft Policy

π is a *ε-soft policy*, if for all states and actions:

$$\pi(a, s) \geq \varepsilon/|\mathcal{A}(s)|$$

results in the *on-policy* **First-Visit MC Control** (for ε-soft policies) algorithm.

ε is not necessarily fixed over time. For $\varepsilon > 0$ we have no guarantee for an optimal policy, but for any *ε-soft* policy π, any *ε-greedy* policy regarding q_π is guaranteed to be better or equal to π.

3.2.3 Off-Policy MC Control and Importance Sampling

The idea of *Off-policy* methods is to use *all* the available data, especially from previous policies.

Definition: Importance Sampling

For a random variable X with density f, we estimate the expected value of Y, which is in the same space as X, with density g using

$$\mathbb{E}[Y] \approx \frac{1}{n} \sum_i \frac{g(X_i)}{f(X_i)} X_i$$

We use *importance sampling* to average current policy π with the behavior policy $b \neq \pi$ to estimate the value function:

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

$$\rightarrow \quad \mathbb{E}[\rho_{t:T-1} G_t | S_t = s] = v_\pi(s)$$

Algorithm: Off-Policy MC Control ($\pi \approx \pi_*$)

Init
- $Q(s, a) \in \mathbb{R}$
- $C(s, a) \leftarrow 0$
- $\pi(s) \leftarrow \text{argmax}_a\, Q(s, a)$

For each episode:
- $b \leftarrow$ any soft policy
- Create episode using b
- $G \leftarrow 0$
- $W \leftarrow 1$
- For $t = T-1, ..., 0$:
  - $G \leftarrow \gamma G + R_{t+1}$
  - $C(S_t, A_t)$ += W
  - $Q(S_t, A_t)$ += $\frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
  - $\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$
  - If $A_t \neq \pi(S_t)$, exit inner loop
  - W *= $\frac{1}{b(A_t|S_t)}$

Intuitively, if we do not trust the samples drawn from the behavior policy b, meaning, it is farther away from the target policy π, then we want to weight the corresponding Q-updates less.

---

4 Temporal Difference Learning

The idea of **TD** learning is that we only consider the change in the value function between two connected states. In particular, we do not need to wait for the end of an episode to update v (like in

MC), nor search over all possible next states to update v (like in DP). The approach doesn't require a dynamics model.

4.1 Prediction

4.1.1 TD(0)

Gently update $V(S_t) \leftarrow G_t$ by using the update rule for each step t in a sampled episode

$$\begin{aligned} V(S_t) &\leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \\ &= (1-\alpha)V(S_t) + \alpha \underbrace{[R_{t+1} + \gamma V(S_{t+1})]}_{\text{estimate of } G_t \text{ (TD target)}} \end{aligned}$$

where α is the step size and $\delta_t := R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ the **TD error** between $V_k(S_t)$ and estimate $R_{t+1} + \gamma V(S_{t+1})$. $V(terminal)$ is initialized with 0.

The TD value function estimation is a *biased* estimator with *low* variance, and is guaranteed to asymptotically converge to some fix point solution provided, that $\alpha < 1$.

TD vs. MC

- TD and MC both converge, but there is no proof, whether one of the methods converges faster. Usually, TD converges faster on stochastic tasks.
- TD uses less memory and less peak computation requirements.
- *The Markov property is critical for TD!* MC doesn't rely on it.
- TD has an advantage when using very long episodes, or continuing tasks without episodes at all. MC waits until the end of an episode.

4.2 Control

4.2.1 SARSA: On-Policy TD control

From the TD(0) update rule, we can directly deduce the action-value function update using the sequence $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ (hence, **SARSA**), assuming $Q(\text{terminal}, \cdot) = 0$

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \\ &= (1-\alpha)Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})] \end{aligned}$$

which is *dependent* from the policy.

Here, we assume, that the sample for the update rule comes from the same policy, that generated the trajectory (thus, *on-policy*). The *SARSA algorithm* is then simply a combination of a method, that samples A from S using policy derived from Q (e.g., ε-greedy) and this update rule.

4.2.2 Q-Learning: Off-Policy TD control

For **Q-learning**, we get the update rule,

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \\ &= (1-\alpha)Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a)] \end{aligned}$$

which is *independent* from the policy.

Here, we assume, that the sample updating the Q function comes from a greedy policy (thus, *off-policy*). The Q-learning algorithm is then simply a combination of a method, that samples A from S using policy derived from Q (e.g., ε-greedy) and this update rule.

Q-learning is usually biased towards unlikely high rewards. The usage of a function approximation introduces noise due to approximation errors, resulting in the max operator having an **overestimation bias**. This leads to suboptimal policies or even catastrophic failure.

---

5 Value Function Approximations (VFA)

In order to mitigate the issue of *large* MDPs having too many states and/or actions to store, we estimate the value function using **function approximation**, such that it *generalizes* from seen states to unseen states.

5.1 Parametric Approximators

Consider parametric models with parameters w. Then, we get *two levels* of approximations:

1. **Estimating value function**: $v_\pi(s) \approx \hat{V}(s)$
2. **Function approximation**: $v_\pi(s) \approx \hat{V}(s, w)$ (using 1.)

Perform a feature map on state s using *feature vector* $x(s) = (x_1(s), \dots, x_n(s))^T$ with $x_i : \mathcal{S} \to \mathbb{R}$. Features can also be a table look-up $x^{table}(s)$, which is essentially a one-hot-encoded vector with 1 exactly at the entry for state s.

5.1.1 Linear VFA

The **Parametric Function Approximator** is trained using *gradient descent*. Assuming, that the true value function $v_\pi(s)$ is *known*, find parameters w, such that it minimizes MSE between approximation $\hat{V}(s, w)$ and $v_\pi(s)$. We can either average MSE over the entire set of samples (large), or randomize smaller batches over training data, resulting in *stochastic* gradient descent (SGD).

We obtain the general update for $w \leftarrow w + \Delta w$ with step size α and loss function J using

$$\Delta w = -\frac{1}{2}\alpha \nabla J(w)$$

Algorithm: Linear VFA with Oracle v_π ($\hat{V} \approx v_\pi$)

For each episode, for each step in episode:
$A \sim \pi(\cdot|S)$, observe S', R
$w \leftarrow w + \alpha[v_\pi(S) - \hat{V}(S, w)]\nabla \hat{V}(S, w)$
$S \leftarrow S'$

For feature transformed $\hat{V}(s, w) = x(s)^T w$ and $J(w) = \mathbb{E}_\pi[(v_\pi(s) - \hat{V}(s, w))^2]$, we get the update rule for each step in a trajectory:

$$\Delta w = \alpha(v_\pi(s) - \hat{V}(s_t, w))\nabla \hat{V}(s_t, w) = \alpha(v_\pi(s) - \hat{V}(s, w))x(s)$$

An *unknown* $v_\pi(s)$ is estimated by a **target** according to the method:

- MC: Target is G_t (unbiased, but *noisy* estimate)

$$\Delta w = \alpha(G_t - \hat{V}(s_t, w))\nabla\hat{V}(s_t, w)$$

- Linear TD(0): Target is $r + \gamma\hat{V}(s_{t+1}, w)$ (*biased* estimate = offset of hyperplane)

$$\Delta w = \alpha(r + \gamma\hat{V}(s_{t+1}, w) - \hat{V}(s_t, w))\nabla\hat{V}(s_t, w)$$

Algorithm: Gradient MC ($\hat{V} \approx v_\pi$)

For each episode:
  Generate trajectory $S_0, A_0, R_1, \ldots, R_T, S_T$ using π
  For $t = T-1, \ldots, 0$:
    $w \leftarrow w + \alpha[G_t - \hat{V}(S, w)]\nabla\hat{V}(S, w)$

Algorithm: (Semi-)Gradient TD(0) ($\hat{V} \approx v_\pi$)

For each episode, for each step in episode:
  $A \sim \pi(\cdot|S)$, observe S', R
  $w \leftarrow w + \alpha[R + \gamma\hat{V}(S', w) - \hat{V}(S, w)]\nabla\hat{V}(S, w)$
  $S \leftarrow S'$
  until S terminal

When we additionally utilize importance sampling, we introduce the factor ρ for the fraction of target policy π, and the behavior policy b collecting data:

$$\Delta w = \alpha \underbrace{\frac{\pi(a|s)}{b(a|s)}}_{\rho} \left(r + \gamma\hat{V}(s_{t+1}, w) - \hat{V}(s_t, w)\right) \nabla_w \hat{V}(s_t, w)$$

Note, that since the function approximated is *linear*, we can only learn an (unbiased) hyperplane, which is, to some extent, restrictive.

5.1.2 Theoretical Properties

MC with VFA converges to weights, that have minimum MSE possible with respect to distribution μ of probabilities of visiting state s under policy π:

$$MSVE_\mu(w_{MC}) = \min_w \sum_{s\in\mathcal{S}} \mu(s)(v_\pi(s) - \hat{V}_\pi(s, w))^2$$

TD(0) on VFA converges to weights, that are within a constant factor of the possible optimum given the stationary distribution $d(s)$ over states of π:

$$MSVE_d(w_{TD}) \leq \frac{1}{1-\gamma} \min_w \sum_{s\in\mathcal{S}} d(s)(v_\pi(s) - \hat{V}_\pi(s, w))^2$$

5.1.3 The Deadly Triad

Combining the following three elements leads to *instability* and *divergence*, while only having two of them is fine:

- **(Linear) Function approximation**: Necessary to represent large state and action spaces.
- **Bootstrapping** (TD-learning): Update targets, that include existing estimates, rather than relying exclusively on actual rewards and complete returns (as in MC). Still, empirical results show, that TD methods often work better than MC methods.
- **Off-policy training**: Training on a distribution of transitions other than that produced by the target policy. The policy can be updates on data from any behavior policy, which is beneficial with respect to data efficiency.

5.1.4 Policy Control with VFA

Generalize ideas of state-value function approximations to action-value functions. The update for *linear* **Action-Value Function Approximation** is

$$\Delta w = -\frac{1}{2}\alpha\nabla\mathbb{E}_\pi[(q_\pi(s,a) - \hat{Q}(s,a,w))^2] = \alpha\mathbb{E}_\pi[(q_\pi(s,a) - \hat{Q}(s,a,w))\nabla\hat{Q}(s,a,w)]$$

where $q_\pi(s,a)$ is *unknown* and needs to be substituted through *targets*. Similarly, we can transform features using $x(s,a)$, such that $\hat{Q}(s,a,w) = x(s,a)^T w$.

Update rules for other algorithms:

- **MC**: Target is return G_t

$$\Delta w = \alpha(G_t - \hat{Q}(s_t,a_t,w))\nabla\hat{Q}(s_t,a_t,w)$$

- **SARSA**: TD target is $r + \gamma\hat{Q}(s',a',w)$ using current estimate $\hat{Q}(s',a',w)$

$$\Delta w = \alpha(r + \gamma\hat{Q}(s_{t+1},a_{t+1},w) - \hat{Q}(s_t,a_t,w))\nabla\hat{Q}(s_t,a_t,w)$$

- **Q-Learning**: TD target is $r + \gamma\max_{a'}\hat{Q}(s',a',w)$ (not suitable for linear VFA)

$$\Delta w = \alpha(r + \gamma\max_{a'}\hat{Q}(s_{t+1},a',w) - \hat{Q}(s_t,a_t,w))\nabla\hat{Q}(s_t,a_t,w)$$

5.2 Batch Methods

Since (direct) incremental methods are noisy, use *batch methods* to find the best fitting value function for previous experience as well.

For the *state-value function*, built an **experience dataset** $\mathcal{D} = \{(s_1,\hat{v}_1^\pi),\dots,(s_T,\hat{v}_T^\pi)\}$. Then, we can use *least squares algorithms* to find parameters minimizing the sum-squared error between VFA $\hat{V}(s,w)$ and (*off-policy*, like MC or TD) target $\hat{v}^\pi(s)$:

$$LS(w) = \sum_{t=1}^{T}(\hat{v}^\pi(s_t) - \hat{V}(s_t,w))^2$$

For the *action-value function*, we obtain a similar least squares formulation with **experience dataset** $\mathcal{D} = \{((s_1,a_1),\hat{v}_1^\pi),\dots,((s_T,a_T),\hat{v}_T^\pi)\}$, which can be either explicitly solved or approximated using SGD:

$$LS(w) = \sum_{t=1}^{T}(\hat{q}^\pi(s_t,a_t) - \hat{Q}(s_t,a_t,w))^2$$

Usually, old data of the experience dataset is eventually evicted during the training process. Creating an experience dataset leads to the idea of **Experience Replay**, which is crucial for modern RL methods.

Since direct LS solutions may be too costly in practice, it is advisable to consider *mini-batches* for updates and to apply SGD instead.

5.2.1 Batch TD(0) VFA

Reformulating the update step

$$\Delta w = \alpha(r_{t+1} + \gamma \hat{V}(s_{t+1}, w) - \hat{V}(s_t, w))\nabla \hat{V}(s_t, w) = \alpha(r_{t+1}x_t - x_t(x_t - \gamma x_{t+1})^T w_t)$$

results in

$$b = \mathbb{E}[r_{t+1}x_t], \quad A = \mathbb{E}[x_t(x_t - \gamma x_{t+1})^T] \quad \Rightarrow \quad 0 = b - Aw \quad \Rightarrow \quad w_{TD} = A^{-1}b$$

obtained at convergence.

5.2.2 Batch MC VFA

By minimizing the error

$$\underset{w}{\operatorname{argmin}} \sum_{i=1}^{N} (G(s_i) - x(s_i)^T w)^2$$

where $G(s_i)$ is an *unbiased* sample of expected return $v_\pi(s_i)$, we directly obtain the ordinary least squares estimator, when we set its derivative to zero

$$w^\pi = (X^T X)^{-1} X^T G$$

where G is the vector with all returns, and X is the matrix of all features. The direct solution is obtained in $\mathcal{O}(N^3)$, with Shermann-Morrison in $\mathcal{O}(N^2)$.

5.3 Deep Q-Learning

Creating the right features for linear VFA is usually tedious, such that we can fall back to *deep neural networks*, in particular *CNNs*, to do the feature extraction. From these features, we can also simply learn to predict the Q value for each action given an input state (*Q-learning*).

Use the same update rules as in policy control with VFA. Two main issues causing *Q-learning* to diverge:

1. **Autocorrelations**: Correlation of subsequent observations violates independent data assumption for NN training ($\rightarrow$ **Experience Replay**)
2. **Non-stationary targets**: Possible instability due to correlation between Q-values and their targets ($\rightarrow$ **Fixed Q-Targets**)
3. Small changes in Q-estimate lead to big distributional shifts of collected data, which violates i.i.d. data assumption for NN training ($\rightarrow$ average over a variety of state distributions)

Then, in the **DQN algorithm**, we store transition (S, A, S', R) in replay buffer D, from which we sample a random minibatch to train $\hat{Q}$ using Q-targets, which hold on to old w for c update steps.

Similarly to Q-Learning, DQL also faces the problem of an *overestimation bias*.

5.3.1 Fixed Q-Targets

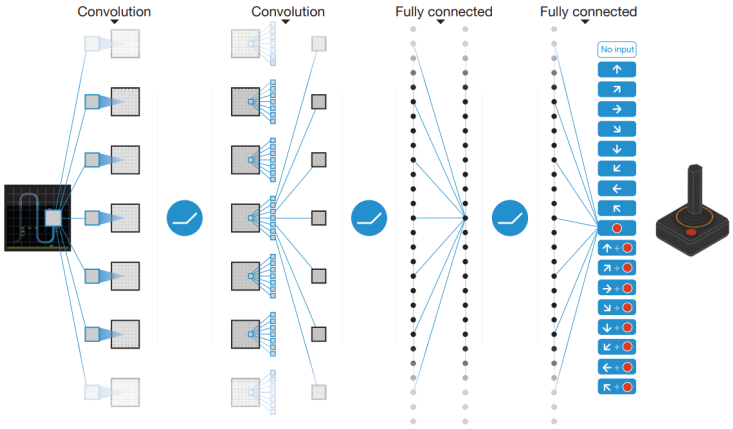
To improve stability, fix the weights used in the target calculation for a certain amount of updates, resulting in the update rule for SGD:

$$\Delta w = \alpha(r + \gamma \max_{a'} \hat{Q}(s', a', w^-) - \hat{Q}(s, a, w))\nabla \hat{Q}(s, a, w)$$

where weights w^- hold old w and are overwritten by new w after c steps.

5.3.2 Deep Q-Network (DQN) Example: Atari

In the "Human-level control through deep reinforcement learning" paper, researchers were able to train an agent playing all Atari games beyond human level. The input state s is represented as a stack of raw pixels from the last 4 frames. From these frames, we extract features using convolutions, process these features using an MLP, which predicts the $\hat{Q}(s,a)$ value for each action a.



5.3.3 Double DQN

In **Double DQN** (DDQN), the action selection and evaluation is done with different Q functions, having different weights:

$$\Delta w = \alpha(r + \gamma \hat{Q}(s', \underset{a'}{\operatorname{argmax}}\, \hat{Q}(s', a', w), w^-) - \hat{Q}(s,a,w))\nabla \hat{Q}(s,a,w)$$

w and w^- are switched, when the other Q function has to be updated. The main difference is, that DDQN mitigates the issue of DQN having an *overestimation bias*.

5.3.4 Q-Ensembles

In **Q-Ensembles**, we have N evaluation networks $\hat{Q}_{\theta'_i}(s,a)$, from which we choose $M < N$ for bootstrapping. More Q-networks lead to *lower expected variance of the bias*, and enabled to learn more efficiently from data, since the mini-batches for each Q-network are distinct.

---

6 Policy Gradient Methods

Before, the value function has been the central object for learning and acting and the policy only existed *implicitly*. Now, we want to directly learn the *stochastic* policy, in particular, the **policy parameters** θ in the *explicit* policy representation $\pi(a|s,\theta) = \pi_\theta(a|s)$. Examples for policy parametrization are

- **Soft-max**: $\pi_\theta(a|s) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}$
  where $h(s,a,\theta)$ can be a linear mapping of features, or a neural network;
  for discrete action spaces
- **Gaussian**: $\pi_\theta(a|s) = \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2}\right)$
  for continuous action spaces
- **Linear**: $\pi_\theta(a|s) = \theta^T s$
- **Neural Network**

6.1 Policy Search

To learn the policy parameters, we want to maximize some scalar performance measure $J(\theta)$, capturing the RL objective

$$\max_{\theta} J(\theta)$$

which is usually resolved with a *local* optimization (since non-convex). For **episodic problems**, we define

$$J(\theta) := \mathbb{E}_{\pi_\theta}[G_0|S_0 = s_0] = v_{\pi_\theta}(s_0) = V(s_0, \theta)$$

For **continuing problems**, we usually use some form of average reward:

$$J_{avV}(\theta) = \sum_s \mu(s) V(s, \theta), \qquad J_{avR}(\theta) = \sum_s \mu(s) \sum_a \pi_\theta(a|s) r(s, a) \tag{1}$$

where $\mu(s)$ is the stationary state distribution of the MDP under π_θ.

We consider **stochastic policies**, meaning, we have some *randomness* involved in the choice of the next action. It

- *avoids failures in adversarial settings*, where the opponent might learn the deterministic behavior of the agent,
- *is beneficial in partial observations problems*, where we might get stuck with a deterministic approach,
- and *takes advantages of exploration*.

This setting can converge to a deterministic policy and may be simpler to approximate. On the other hand, it typically converges to a *local* minimum, and its evaluation can be inefficient and have a *high variance*.

6.2 Policy Gradient Methods

In **gradient-based optimization**, we want to approximate *gradient ascent* in J (again, $J(\theta) = V(s_0, \theta)$):

$$\theta^{(i+1)} = \theta^{(i)} + \alpha \cdot \widehat{\nabla J(\theta^{(i)})}$$

where $\widehat{\nabla J(\theta^{(i)})}$ is the *estimate of the policy gradient*. For fast learning, an unbiased and low variance approximation will be of interest.

A simple, noisy, inefficient method to obtain a gradient estimate is using **finite differences**, where we perturb θ by a small amount ε in jth dimension:

$$\frac{\partial J(\theta)}{\partial \theta_j} \approx \frac{J(\theta + \varepsilon u_j) - J(\theta)}{\varepsilon}$$

where u_j is the unit vector in jth direction.

For potential better estimates, we compute the gradient *analytically*. Assume policy is *differentiable*. We can express V in two ways (considering *episodic case* $J(\theta) = V(s_0, \theta)$):

$$V(s_0, \theta) = v_{\pi_\theta}(s_0) = \sum_a \pi_\theta(a|s_0) q_{\pi_\theta}(s_0, a) \tag{2}$$

$$V(s_0, \theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{k=0}^{T} r(s_k, a_k) \middle| s_0\right] = \sum_r R(\tau) P(\tau|\theta) \tag{3}$$

where

- τ is the state-action trajectory,
- $P(\tau|\theta)$ is the probability for trajectory τ when executing π_θ and starting at s_0,
- $R(\tau) = \sum_{k=0}^{T} r(s_k, a_k) = \sum_{k=0}^{T} r_k$.

By smartly utilizing log and applying the chain rule for probabilities several times on Eq. (3), we obtain

$$\nabla_\theta V(s_0, \theta) = \mathbb{E}_\tau \left[R(\tau) \nabla_\theta \log P(\tau|\theta) \right] \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)}) \sum_{t=0}^{T-1} \nabla_\theta \underbrace{\log \pi_\theta(a_t^{(i)}|s_t^{(i)})}_{\text{score function}} \tag{4}$$

where i indicates the time step. Starting from s_0, we execute current policy π_θ m times, record the sample trajectories $\tau^{(i)}$, and estimate $\nabla_\theta V(s_0, \theta)$ using the above equation.

A more general derivation using similar tricks can be done for Equation (2) for a more general case of $J(\theta)$, which leads to the **Policy Gradient Theorem**, with the following approximation:

$$\nabla_\theta J(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) G_t^{(i)}$$

Definition: Policy Gradient Theorem

For any differentiable policy $\pi_\theta(a|s)$ and $J(\theta)$ in the *episodic* case, it holds:

$$\begin{aligned} \nabla_\theta J(\theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi_\theta(a|s) \\ &= \mathbb{E}_\pi \left[\nabla_\theta \log \pi_\theta(A_t|S_t) q_\pi(S_t|A_t) \right] \\ &= \mathbb{E}_\pi \left[\nabla_\theta \log \pi_\theta(A_t|S_t) G_t \right] \end{aligned}$$

The main difference to Equation (4) is, that we now sum over accumulative returns $G_t^{(i)}$, instead of the reward observed for each trajectory $R(\tau^{(i)})$, enabling us to update the policy in a step-by-step manner.

In the exercise, we derive the following gradients of the score function for the exemplary policy parametrizations:

- *Soft-max:* $\nabla_\theta \log \pi_\theta(s,a) = \phi(s,a) - \sum_{a'} \pi_\theta(s,a')^T \phi(s,a') = \phi(s,a) - \mathbb{E}_{\pi_\theta}[\phi(s,\cdot)]$
  where $\phi = h$ in the previous notation.
- *Gaussian:* $\nabla_\theta \log \pi_\theta(s,a) = \frac{a - \phi(s)^T\theta}{\sigma^2} \phi(s)$
  where variance $\mu(s,\theta) =: \phi(s)^T\theta$ is linear.

6.3 REINFORCE Algorithms

Algorithm: REINFORCE: MC Policy-Gradient Control (*episodic*)

Input: $\pi_\theta(a|s)$
Parameter: step size $\alpha > 0$
Loop forever:
  Generate episode τ following π_θ
  For $t = 0, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-(t+1)} R_k$
    $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \log \pi_\theta(A_t|S_t)$

Reformulation of the obtained gradient, such that we exploit the temporal structure of the MDP, yields to the **REINFORCE** algorithm with the following approximated gradient (*MC-style estimate*):

$$\nabla_\theta V(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) G_t^{(i)}$$

This expression involves fewer terms and tends to have *lower variance*, than the gradient from Eq. (4).

The algorithm requires a *complete* run from t to T, is therefore an MC algorithm.

Algorithm: REINFORCE with Baseline (*episodic*)

Parameter: step sizes $\alpha_w > 0, \alpha_\theta > 0$
Loop forever:
  Generate episode τ following π_θ
  For $t = 0, \ldots, T-1$
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-(t+1)} R_k$
    $\delta \leftarrow G - \hat{v}(S_t, w)$ **(advantage)**
    $w \leftarrow w + \alpha_w \delta \nabla_w \hat{v}(S_t, w)$
    $\theta \rightarrow \theta + \alpha_\theta \gamma^t \delta \nabla \log \pi_\theta(A_t|S_t)$

REINFORCE is quite *noisy*. One common approach to mitigate this is to introduce a **baseline** against which the return is compared:

$$\nabla_\theta V(\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - b(s_t))\right] \quad (5)$$

One can show, that the baseline doesn't introduce a bias, and it often helps to reduce the variance.

A common choice for the baseline is a *differentiable* state-value function approximation $b(s_t) = \hat{v}(s_t, w)$ with parameters w, which solves a regression problem fitting $s_t \rightarrow G_t$ using gradient descent: $\min_w (G_t - \hat{v}(s_t, w))^2$.

In practice, we usually don't perform the gradient step by hand (as it is done in the algorithms), but use more advanced gradient descent/ascent algorithms, such as Adam.

6.4 Continuing and Continuous Problems

Continuing problems involve an *infinite horizon*. As there is no defined start or end, the performance measure uses the average rate of reward per time step ($J_{avR}(\theta)$ as in Eq. (1)):

$$J(\theta) = \sum_s \mu(s) \sum_a \pi_\theta(a|s) r(s, a)$$

In this case, the policy gradient theorem can also be proven, such that policy gradient algorithms can be formulated in essentially the same way.

Algorithm: "Vanilla" Policy Gradient Algorithm

Init θ, baseline b
For iteration/episode $= 1, 2, \ldots$:
  Collect set of trajectories using π.
  For each t in each $\tau^{(i)}$, compute
    $G_t^{(i)} = \sum_{t'=t}^{T-1} r_t^{(i)}$ **(return)**
    $\hat{A}_t^{(i)} = G_t^{(i)} - b(s_t)$ **(advantage)**
  Refit b: min $\leftarrow \sum_{i,t} ||b(s_t) - G_t^{(i)}||^2$
  Update π using *policy gradient estimate* $\hat{g}$
  (sum of terms $\nabla_\theta \log \pi(a_t|s_t, \theta)\hat{A}_t$)

Continuous problems involve *continuous* action or state spaces. The learned policy is then a *PDF* (e.g., Gaussian).

---

7 Actor-Critic Methods

The idea of actor-critic methods is to combine *value-based* learning and *explicit* policy parametrization.

7.1 Policy Gradient-Focused Actor-Critic

In Eq. (5), we 1) replace the return with the Q-value function, 2) set the value function as baseline: $b(s_t) \rightarrow \hat{V}(s_t, w_2)$, giving rise to the (on-policy) **Actor-Critic Methods**, with the (policy) gradient

$$\nabla_\theta V(\theta) = \mathbb{E}_\pi\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(\hat{Q}(s_t, a_t, w_1) - \hat{V}(s_t, w_2))\right]$$

The **actor** is the policy approximator with parameters θ. The **critic** is the value function approximator with parameters w, used to assess actions and to suggest update direction. Both are often represented as NNs.

The function $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$ is called **advantage function**, capturing the "advantage" of an action over the current value of that policy.

We combine the two value functions into one network by estimating the advantage function:

$$\hat{A}_t = r_t + \gamma \hat{V}(s_{t+1}, w) - \hat{V}(s_t, w)$$

which is essentially the TD error δ_t, as in Section 4.1.1. This result in the **One-Step Actor-Critic** algorithm. The TD error is an *unbiased* estimate of the advantage.

The main difference to previous policy gradient algorithms is, that we don't need to first execute the full episode, but *we update the policy in every step* (hence, *TD-style estimate*).

Algorithm: One-Step Actor-critic (episodic)

Parameter: step sizes $\alpha_w > 0, \alpha_\theta > 0$
Loop forever:
  Init first state of episode S
  $I \leftarrow 1$
  While S not terminal:
    $A \sim \pi(\cdot|S,\theta)$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{V}(S', w) - \hat{V}(S, w)$
    $w \leftarrow w + \alpha_w \delta \nabla_w \hat{V}(S, w)$
    $\theta \leftarrow \theta + \alpha_\theta I \delta \nabla_\theta \log \pi(A|S,\theta)$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$
(if S' terminal, then $\hat{V}(S', w) = 0$)

To generalize to k-step TD-style estimates, we simply replace the advance $\hat{A}_t =: \hat{A}_t^{(1)}$ with

$$\hat{A}_t^{(k)} = \hat{G}_t^{(k)} - \hat{V}(s_t) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k \hat{V}(s_{t+k}, w) - \hat{V}(s_t)$$

where $\hat{A}_t^{(1)}$ tends to have high bias and low variance, and $\hat{A}_t^{(\infty)}$ vice versa. A popular actor-critic algorithm using this adjustment is **A3C**.

In summary, the policy gradient can have many forms (blue for MC-style estimates, green for TD-style estimates):

- REINFORCE: $\nabla_\theta J(\theta) = \mathbb{E}_\pi[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t]$
- Q Actor-Critic: $\nabla_\theta J(\theta) = \mathbb{E}_\pi[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \hat{Q}(s_t, a_t, w)]$
- TD / One-step Actor-Critic: $\nabla_\theta J(\theta) = \mathbb{E}_\pi[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \delta_t]$
- Advantage Actor-Critic: $\nabla_\theta J(\theta) = \mathbb{E}_\pi[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t]$

7.1.1 Choosing the Step Size: Trust Region

Choosing a *fixed* learning rate can be difficult, since the optimization landscape may vary locally / globally. Here, we introduce the following options:

- **Linear search**: Make evaluations along the direction of the gradient, searching for the optimum. Simple, but expensive, and ignores possible 2nd order information.
- **Trust Region**: Allow for a maximal change within a region. We want to keep policy updates from one set of policy parameters to another *bounded*, such that parameter updates stay within a region where we *trust* the objective function.
- Powerful optimization libraries with sophisticated step size control, e.g, Adam.

The main goal is to automatically ensure that the updated policy $\tilde{\pi}$ has a value greater or equal to the previous policy π, thus $J(\tilde{\theta}) \geq J(\theta)$. Consider the discounted value function:

$$J(\theta) = V^{\pi}(\theta) = \mathbb{E}_{\pi}\left[\sum_{t=1}^{\infty} \gamma^t R(s_t, a_s) \middle| s_0\right]$$

We need to capture the difference between the two value functions using the advantage function, where we want to optimize the *surrogate objective* $\max_{\tilde{\theta}} L_{\pi}(\tilde{\pi})$ instead of $\max_{\tilde{\theta}} V^{\tilde{\pi}}(\tilde{\theta})$:

$$V^{\tilde{\pi}}(\tilde{\theta}) = V^{\pi}(\theta) + \sum_s \mu_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a|s) A^{\pi}(s,a) \overset{\text{don't know } \mu_{\tilde{\pi}}}{\Longrightarrow} L_{\pi}(\tilde{\pi}) = V^{\pi}(\theta) + \sum_s \mu_{\pi}(s) \sum_a \tilde{\pi}(a|s) A^{\pi}(s,a)$$

with

- $A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$ advantage
- $\mu_{\tilde{\pi}}(s) = \sum_{t=0}^{\infty} \gamma^t \mathbb{P}[s_t = s|\tilde{\pi}]$ discounted state distribution under the new policy ("how often are future states visited under the new policy")
- $L_{\pi_\theta}(\pi_\theta) = V^{\pi}(\theta)$ and $\nabla_\theta L_{\pi_\theta}(\pi_\theta) = \nabla_\theta V^{\pi}(\theta)$

Hence, we expect the optimization to do something useful, if we are not moving away too far from the current policy. We can show, how good the solution obtained from optimizing $L_{\tilde{\pi}}$ vs. the real objective $V^{\tilde{\pi}}$ is:

$$\boxed{V^{\tilde{\pi}} \geq L_{\pi}(\tilde{\pi}) - C \cdot D_{KL}^{\max}(\pi, \tilde{\pi}) =: M(\tilde{\pi})}$$

where

- $D_{KL}^{\max} = \max_s D_{KL}(\pi_1(\cdot|s), \pi_2(\cdot|s))$ with **Kullback-Leibler divergence** D_{KL}, which is a *non-symmetric* distance measure for distributions, where π_2 is the reference distribution, and π_1 the actual distribution

$$D_{KL}(P||Q) = \sum_x p(x) \log \frac{p(x)}{q(x)}, \quad D_{KL}(P||Q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx$$

- $C = \frac{4\varepsilon\gamma}{(1-\gamma)^2}$ with $\varepsilon = \max_{s,a} |A^{\pi}(s,a)|$ comes with guaranteed improvement, but slow convergence and hard to compute

Now, that we have a lower bound $M(\tilde{\pi})$ on $V^{\tilde{\pi}}$, we can estimate this lower bound and maximize it ($\max_\theta M(\tilde{\pi}_{\tilde{\theta}})$). As long as the new policy π_{i+1} is an improvement over π_i, with respect to the lower bound M_i, then the value function is *guaranteed to monotonically improve* (**Minorization-Maximization (MM) algorithms**):

$$V^{\pi_{i+1}} - V^{\pi_i} \geq M_i(\pi_{i+1}) - M_i(\pi_i)$$

where

- $M_i(\pi) := L_{\pi_i}(\pi) - C \cdot D_{KL}^{max}(\pi_i, \pi)$

Additional ideas involve using the KL-term as a *constraint* instead of a penalty, using the *average KL* instead of the maximum, and using constraint δ as a hyperparameter (defining a **trust region** for expected difference between old and new policy):

$$\boxed{\max_\theta L_{\pi_{old}}(\pi_\theta) \text{ subject to } \mathbb{E}_{s\sim\mu_{\pi_{old}}}[D_{KL}(\pi_{old}, \pi_\theta)] < \delta}$$

Example: Trust Region Policy Optimization (TRPO)

TRPO builds on the previously described surrogate optimization and trust region formulation. It automatically constrains the weight update to a trusted region to approximate, where the surrogate objective is valid.

Example: Proximal Policy Optimization (PPO)

PPO aims to simplify TRPO by

- introducing a *dynamically adapted penalty* β for the expected difference between new and old policy, adapted depending on distance to the target distance, and
- clipping the advance by the ratio of new and old policy $\left(\frac{\tilde{\pi}(a|s)}{\pi(a|s)}\right)$, and choose the minimal such advance (*pessimistic bound*)

7.2 Value Function-Focused Actor-Critic

Combining (deep) Q-learning and policy gradient yields a new type of (deep) RL algorithms, that are *off-policy*, have a *continuous control policy/action space*, and a *deterministic* policy.

Consider an explicit policy parametrization π_θ. For the given Q-function $q^\pi(s,a)$ and the deterministic policy $\mu_\theta(s) = a$, ideally, the policy improvement step would be $\mu_\theta = \text{argmax}_a\, q^\pi(s,a)$, but we want μ_θ to *learn to approximate the max operator* via gradient ascent:

$$\theta^{k+1} = \theta^k + \alpha \mathbb{E}_{s\sim\mu_{\theta^k}}[\nabla_\theta q^{\mu_{\theta^k}}(s, \mu_{\theta^k}(s))]$$

$$\overset{\text{chain rule}}{\Longrightarrow} \theta^{k+1} = \theta^k + \alpha \mathbb{E}_{s\sim\mu_{\theta^k}}[\underbrace{\nabla_\theta \mu_{\theta^k}(s)}_{\substack{\text{change of policy/actions} \\ \text{wrt. parameters}}} \cdot \underbrace{\nabla_a q^{\mu_{\theta^k}}(s,a)}_{\substack{\text{change of Q-function} \\ \text{wrt. actions}}} |_{a=\mu_{\theta^k}(s)}]$$

where $s \sim \mu$ denotes states that are visited according to the state distribution induced by policy μ.

The standard objective function for continuous state/action states is

$$\begin{aligned} J(\mu_\theta) &= \mathbb{E}_\mu[G_1] = \int_{\mathcal{S}} \rho^\mu(s) r(s, \mu_\theta(s)) ds \\ &= \mathbb{E}_{s\sim\rho^\mu}[r(s, \mu_\theta(s))] \end{aligned}$$

where $\rho^\mu(s)$ is the discounted state distribution.

Definition: Deterministic Policy Gradient (DPG)

Assuming all derivatives exists and functions are continuous and bounded:

$$\begin{aligned} \nabla_\theta J(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s) \cdot \nabla_\theta \mu_\theta(s) \cdot \nabla_a q^\mu(s,a)|_{a=\mu_\theta(s)} ds \\ &= \mathbb{E}_{s\sim\rho^\mu}\left[\nabla_\theta \mu_\theta(s) \cdot \nabla_a q^\mu(s,a)|_{a=\mu_\theta(s)}\right] \end{aligned}$$

Examples show, that there are situations, where a deterministic policy performs better than a stochastic one.

7.2.1 Deep Deterministic Policy Gradient (DDPG)

For $\hat{Q}(s_t, a_t, w)$ and for the deterministic policy $\mu_\theta(s_t)$, use deep NNs. **DDPG** is now summarized into two steps:

- **Q-learning objective**: Use policy $\mu_\theta(s_t)$ in the target instead of the greedy policy ("max")

$$L(w) = (r + \gamma \hat{Q}(s_{t+1}, \mu_\theta(s_{t+1}), w^-) - \hat{Q}(s_t, a_t, w))^2$$

- **Policy improvement**: Instead of "argmax", update policy parameters θ using deterministic policy gradient (DPG):

$$\begin{aligned}\nabla_\theta J(\mu_\theta) &= \mathbb{E}_{s\sim\rho^\mu}\left[\nabla_\theta \mu_\theta(s)\cdot\nabla_a q^\mu(s,a)|_{a=\mu_\theta(s)}\right]\\ &\approx \mathbb{E}_{s\sim\rho^\pi}\left[\nabla_\theta \mu_\theta(s)\cdot\nabla_a \hat{Q}(s,a,w)|_{a=\mu_\theta(s)}\right]\end{aligned}$$

  where ρ^π is the behavioral policy. We can use data from stochastic policy π to learn q^μ (off-policy).
- Combined update rules for so-called *Actor-Critic Deep Q-Learning*:

$$\begin{aligned}\theta &\leftarrow \theta + \alpha_1[r+\gamma\hat{Q}_{\theta'}(s',\mu_{\phi'}(s)) - \hat{Q}_\theta(s,a)]\nabla_\theta\hat{Q}_\theta(s,a)\\ \phi &\leftarrow \phi + \alpha_2\nabla_a\hat{Q}_\theta(s,a)|_{s=s_i,a=\mu_\phi(s)}\nabla_\phi\mu_\phi(s)|_{s=s_i}\end{aligned}$$

Algorithm: Deep Deterministic Policy Gradient (DDPG)

Input: $\hat{q}$ Q-net; μ policy net; $\hat{q}', \mu'$ target nets; random process $\mathcal{N}$
Repeat for each episode:
  $s \leftarrow$ from environment
  For each step of episode:
    Take action $a = \mu(s;\theta) + \mathcal{N}_t$, observe r, s' (**behavioral policy**)
    Store $(s,a,r,s') \rightarrow \mathcal{D}$
    Sample minibatch $D \subset \mathcal{D}$
    Optimize critic $\hat{q}$ with respect to $L(w)$: $L(w) = \frac{1}{N}\sum_i^{|D|}[(r_i + \gamma\hat{q}'(s_i', \mu'(s_i';\theta');w') - \hat{q}(s_i,a_i;w))^2]$
    Optimize policy μ using $\nabla_\theta J(\mu_\theta)$: $\nabla_\theta J(\mu_\theta) \approx \frac{1}{N}\sum_i^{|D|}[\nabla_\theta\mu(s_i;\theta)\nabla_a\hat{q}(s_i,a;w)|_{a=\mu(s_i;\theta)}]$
    $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$
    $w' \leftarrow \tau w + (1-\tau)w'$

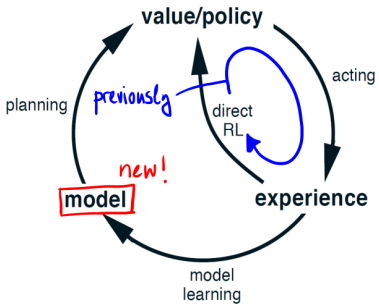
DDPG has been successful, but also has some problems, such as a *value overestimation bias*. Therefore, several extensions have been proposed, such as the **Twin Delayed Deep Deterministic** (**TD3**) policy gradient algorithm. The main ideas of TD3 are:

- **Clipped double Q-learning**: Use two *separate* value estimators and two policies, take the minimum of these two as *pessimistic* estimation. Tends to result in underestimation, which is preferable over overestimation.
- **Delayed policy update**: Only update policy after several updates to the critic.
- **Target policy "smoothing"**: Add noise to target policy.

---

8 Model-based Reinforcement Learning (MBRL)

Model-based methods use and maintain an *explicit transition/reward model of the environment* ($p(s',r|s,a)$), that is either known (like in DP) or learned. Essentially, we want to learn a *model* to simulate experience, where we "squeeze" more information out of provided samples from the environment, making these methods more sample efficient. We differentiate between *direct RL*, where we directly improve the value function and policy and *indirect RL*, where we improve the model via model learning.



Model

In the context of MBRL, **model** refers to a description of the environment, consisting of prediction of how the environment will respond to actions. Usually, the environment and thus the model are *stochastic*.

- *Distributional model*: Provides full distribution of the next state $p(s'|s,a)$.
- *Sample model*: Provides a sample from the possible outcomes according to their probabilities (e.g., samples of trajectories).

Planning

Planning refers to the use of a model to come up with actions or to improve a policy. The difference to learning is, that learning uses *real* experience generated by the *environment*, while planning used *simulated* experience generated by a *model*.

- **Background planning** *(during learning)*: Planning to generate simulated experience, which is used in the value or policy function. (model $\rightarrow$ simulated experience $\rightarrow$ values or policy)
- **Decision-time planning** *(during decision-time)*: Planning involved in the computation, whose output is the selection of a single action. (model + state $\rightarrow$ simulated experience $\rightarrow$ action)

Often planning in RL is *"online"*, i.e., where the model is interacting with an environment and gathering real experience.

8.1 Tabular Methods

Q-Planning

The "planning" version of the standard (tabular) Q-learning called **Q-Planning** consists of the following steps, repeated forever:

1. Randomly select $S \in \mathcal{S}, A \in \mathcal{A}(S)$.
2. Send S, A to a **sample model**, obtaining a sample next reward, and a sample next state S'.
3. *One-step Q-learning* on S, A, R, S': $\quad Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(s',a) - Q(S,A)]$

Planning is used in *"offline"* fashion, i.e., the model is used for data generation without actual interaction with the environment.

Dyna-Q

Dyna-Q utilizes the **Dyna architecture**, where we run *direct* and *indirect* RL in parallel to use simulated and real data to the full extent. This is a classical algorithm for MBRL with *background planning*. The Dyna architecture can be used in addition to any other model-free RL methods.

When the environment changes, the model becomes wrong, i.e. the samples from the model may do more harm than good.

Algorithm: Tabular Dyna-Q

Loop forever:
  $R, S' \leftarrow$ *one-step Q-learning* on S
  $Model(S,A) \leftarrow R, S'$
  Repeat n times:
    $S \leftarrow$ random previously observed state S
    $A \leftarrow$ random action previously taken in S
    $R, S' \leftarrow Model(S,A)$ (assuming deterministic env)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

Dyna-Q+ mitigates this issue by giving extra reward to states, that we have not seen for a while, which encourages *exploration*. We change the reward to $r + \kappa\sqrt{\tau}$, where τ is the last time (a,s) was

visited.

In general, the model used in the tabular Dyna-Q framework is not able to generalize (between similar state and action pairs), model stochastic transitions, and doesn't scale well.

8.2 Dyna-Style MBRL (Background Planning)

Typically, background planning gives better performance than decision time planning. Ensembles of Probabilistic Neural Networks, **Probabilistic Ensembles (PEs)**, are an often used type of model for background planning, having the following properties:

- high degree of freedom to model dynamics
- relatively low computational effort to train on a lot of data
- notion of uncertainty to avoid exploitation of model-bias

Probabilistic Neural Networks (PNN)

The state transition probability is expressed with a NN parametrizing a multivariate Gaussian

$$p(s'|s,a) = \mathcal{N}(\mu_\theta(s,a), \mathbf{\Sigma}_\theta(s,a))$$

where $\mu_\theta(s,a), \mathbf{\Sigma}_\theta(s,a)$ are *learned* by a network with input (s,a).

Since uncertainty due to lack of data is not captured by a single probabilistic NN (hence, NN overfits), we combine B probabilistic NNs. Such a **Probabilistic Ensemble (PE)** allows one to distinguish two types of uncertainty; Assume input (s_t, a_t):

- **Aleatoric uncertainty**: Uncertainty due to *inherent stochasticity of the system*. Estimation is mean of the individual PNN variances:

$$u_a = \frac{1}{B}\sum_{b=1}^{B} \mathbf{\Sigma}_{\theta_b}(s_t, a_t)$$

- **Epistemic uncertainty**: Uncertainty due to *lack of data*. Estimation is the variance of the individual PNN mean predictions:

$$u_e = \frac{1}{B}\sum_{b=1}^{B} (\mu_{\theta_b}(s_t, a_t) - \bar{\mu}_{\theta_b}(s_t, a_t))^2$$

  with $\bar{\mu}_{\theta_b}(s_t, a_t) := \frac{1}{B}\sum_{b=1}^{B} \mu_{\theta_b}(s_t, a_t)$.

Adding up those uncertainties gives us a notion of *how certain the model is about its prediction*.

These ensembles have additional advantages over the vanilla Dyna-Q framework:

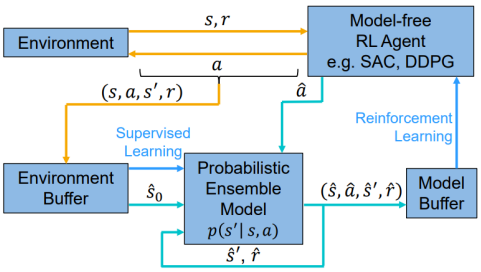
+ performs well in the low as well as in the high data regime
+ can generalize between similar state and action pairs
+ can handle stochastic state transitions

Model-based Policy Optimization (MBPO)

These observations of PEs lead to the **Model-based Policy Optimization (MBPO)** algorithm.

Algorithm: MBPO (Dyna-style Deep RL)

For N epochs:
  Train p_θ (PE) on $\mathcal{D}_{env}$ via maximum likelihood
  For E steps:
    Take action in env using π_ϕ; add to $\mathcal{D}_{env}$
    For M model rollouts:
      $s_t \sim \mathcal{D}_{env}$ (uniform)
      k-step model rollout starting from s_t using π_ϕ; add to $\mathcal{D}_{model}$
    For G gradient updates:
      $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi, \mathcal{D}_{model})$
      (e.g. Soft-Actor-Critic, DDPG, TD3)



This combination of a model-free learner and a probabilistic dynamics model results in *asymptotic performance* and a *reduction of the required environment interaction*. It can be shown, that the model gets better after having seen more data, thus, we can use the model more over time.

8.3 MPC-Style MBRL (Decision-Time Planning)

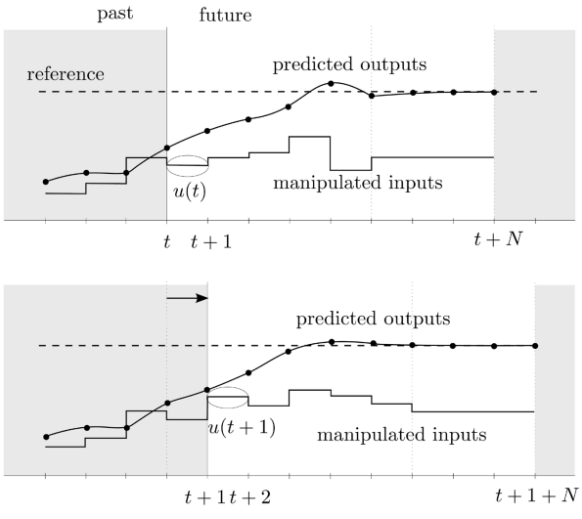
Decision-time planning can be considered as planning in a stricter sense: Consider *different* actions, predict their outcome, pick the best based on the prediction.

- *Choosing an action*: random, tree search, optimization (by some criterion), parametrized policy
- *Model class*: NN, Gaussian process, ensemble, ...
- *Prediction horizon*: to the end (episodes may be too long), just one step (often too shortsighted), receding horizon over multiple steps (plan for a horizon, execute one step, re-plan)

Model-Predictive Control (MPC)

Model-predictive control (MPC) formalizes the idea of optimal planning with a receding horizon. The model is assumed *given*. An example formulation of the optimization problem can be expressed as

$$\min_{U_N} J(x_0, U_N) = \min_{U_N} \sum_{t=0}^{N} Q(x_t, u_t) + \bar{Q}(x_N)$$
$$\text{with} \quad U_N := (u_0, \ldots, u_N)$$



subject to

- $x_{t+1} = f(x_t, u_t)$ (dynamics model: solution has to be compatible with the physical system)
- $x_t \in \mathcal{X}$ (constraints on the states)
- $u_t \in \mathcal{U}$ (constraints on the control input)
- $x_N \in \mathcal{X}_f$ (terminal state conditions)

where control input u_t corresponds to action a_t, and state x_t corresponds to state s_t.

We re-solve the optimization problem at every time step t, where we only apply the first u_0 for sequence U_N, observe the next state x_{t+1}, and restart the optimization with this new state as the initial state. This is also known as **receding horizon control**.

Probabilistic Ensemble Trajectory Sampling (PETS)

Algorithm: Cross Entropy Method

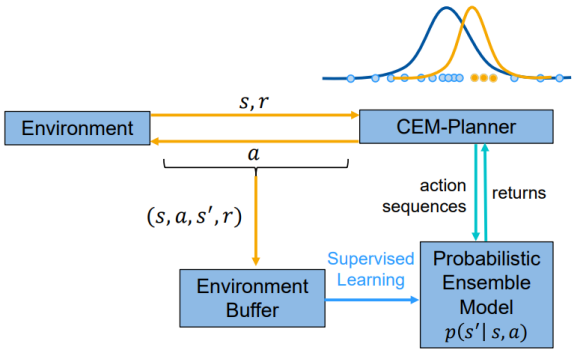
Init $\mu \leftarrow 0, \sigma \leftarrow 0.5$
For N iterations:
  $a_{k,i} \sim \mathcal{N}(\mu_i, \sigma_i^2)$
    $i = 1, \ldots, H$ (horizon)
    $k = 1, \ldots, K$ (samples)
  Simulate a_k and take m best
  Fit μ, σ to best action sequence

PETS maintains a "population" of candidate solutions (here, a Gaussian distribution of actions). The **Cross-Entropy Method (CEM)** optimizer generates action sequences by sampling actions from a Gaussian distribution for each time step in the prediction horizon H. These distributions are updated towards high-rewarding action sequences.

PETS can be sketched as follows:

- While planning:
  - Sample action sequences from a sample distribution ($\mathcal{N}$).
  - Starting from current state, simulate trajectories sampled from the dynamic model, and evaluate average return of resulting trajectories.
  - Update sampling distribution towards best performing ε-percentile of the population.
- Apply first action of best performing action sequence to the environment.
- Observe transition (s, a, r, s') and store it in buffer.
- After a while, train dynamics model (PE) from buffer data using supervised learning.

The rollout mechanism in the PEs works as follows: Given an action sequence $(a_0, \ldots, a_{H-1})$ within the horizon H and the current state s_0, P particles (same action sequence, possibly different state predictions) are propagated through a randomly chosen sequences of models in the ensemble. That is, s_i, a_i are fed through a randomly chosen model, predicting a Gaussian, from which we sample the next state s_{i+1} and compute the resulting reward r_{i+1}.



The return of the action sequence is defined by the average of the corresponding P particle returns over horizon H:

$$G = \frac{1}{P} \sum_{p=1}^{P} \sum_{t=1}^{H} \gamma^{t-1} r_t^{(p)}$$

PETS: "long rollouts from the same start state"; MBPO: "short rollouts from different start states"

---

9 Introduction II: Learning-based Control (LBC)

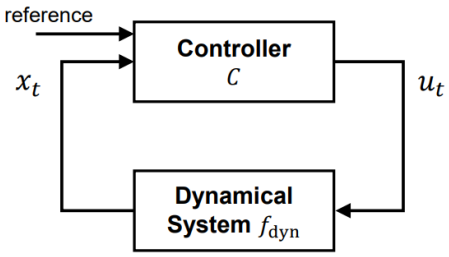
The components of a control systems are

- a *dynamical system* $x_{t+1} = f_{dyn}(x_t, u_t,$ 'noise'$)$ (x_t: state, u_t: input), that evolves over time,
- and a *controller* algorithm, that computes control inputs u_i to be applied to the dynamic system to achieve some objective

For **closed-loop control**, the control input is based on the reference r_t, the current measurement x_t (or y_t) and a **model** of f_{dyn}:

$$u_t = C(r_t, x_t)$$

For an **open-loop control**, u_t only depends on the reference without a feedback loop.



The main difference to RL is that the control design is typically based on a *model*, whereas RL *learns from interacting* with the environment ("RL is optimal control without a model"). Also, RL focuses on solving a task using lots of compute and data, whereas in control, we want to emphasize some guarantees (e.g., stability) for a rather specific problem class with precise assumptions.

Learning-based control denotes the intersection or combination of (machine) learning and control, where we can either learn the controller, or learn the model $\hat{f}_{dyn}$ within the controller.

Comparison (Linear Quadratic Regulation) Control Loop and RL

	LQR Control Loop	*RL*
model of controlled system	$x_{t+1} = Ax_t + Bu_t$	$s_{t+1} \sim p(s_{t+1}\|s_t, a_t)$
model of controller	$u_t = Kx_t$	$a_t \sim \pi(a_t\|s_t)$
differences	linear deterministic continuous **dynamics known**	non-linear stochastic continuous or discrete **dynamics unknown**

Common attributes are, that they are both Markovian and time invariant.

9.1 Model Learning and System Identification

System Identification describes a wide class of methods, that typically assume some *prior knowledge* about the system. Thereby, dynamical (time dependent) system equations (models) for time index t are given as

Nonlinear System	**Linear System**
$x(t+1) = f_t(x(t), u(t), w(t))$ $y(t) = h_t(x(t), u(t), v(t))$	$x(t+1) = A_t x(t) + B_t u(t) + w(t)$ $y(t) = C_t x(t) + D_t u(t) + v(t)$

where

- $x(t)$ state, $u(t)$ control, $w(t)$ process noise, $f(\cdot)$ dynamics function
- $y(t)$ measurement, $v(t)$ measurement noise, $h(\cdot)$ measurement function

For linear and time invariant systems, we can analytically compute the optimal controller, whereas changing and nonlinear problems are much harder.

9.1.1 Linear Systems

The goal of **Linear System Identification** is to determine the unknown matrices A, B, C, D.

In a linear setting without any noise, we can simply determine the unknown matrices A, B using *least squares estimation* (assuming, pseudo-inverse $H^\dagger$ exists):

$$X_{1:N} := \begin{pmatrix} x(1) & \cdots & x(N) \end{pmatrix}, \quad H := \begin{pmatrix} x(0) & \cdots & x(N-1) \\ u(0) & \cdots & u(N-1) \end{pmatrix}$$

$$X_{1:N} = \begin{bmatrix} A & B \end{bmatrix} H \quad \Longrightarrow \quad X_{1:N} H^T = \begin{bmatrix} A & B \end{bmatrix} H H^T \quad \Longrightarrow \quad \begin{bmatrix} A & B \end{bmatrix} = X_{1:N} H^T (H H^T)^{-1} = X_{1:N} H^\dagger$$

where $(x(0), u(0)), \dots (x(N), u(N))$ is a trajectory, and $H^\dagger$ the pseudo-inverse of H. This results in the *exact* solution up to tiny numerical errors.

Measurement Noise

When we have measurements $y(t)$ with *iid* noise $v(t) \sim \mathcal{N}(0, \Sigma)$, the matrices A, B can be *estimated* similarly:

$$\begin{bmatrix} \hat{A} & \hat{B} \end{bmatrix} = Y_{1:N} H^\dagger$$

where we use trajectories of type $(y(0), u(0)), \dots, (y(N), u(N))$.

Process Noise

If we add process noise $w(t)$ to our linear dynamics, the estimation can be done as previously, but the quality of data heavily influences the accuracy of the estimation. For stabilizing control $u(t) = Kx(t)$, we observe, that correlated process noise is propagated through the dynamics:

$$x(t+1) = (A + BK)^{t+1} x(0) + \sum_{i=0}^{t} (A + BK)^i w(t-1)$$

Approaches, such as open loop identification, mitigate the issue of such a non-iid problem.

Full Problem

The full problem of a linear system can be solved with sophisticated solutions such as subspace identification methods. The solving of the problem is, in theory, quite data efficient. But the problem is hard and has infinitely many solutions, and it becomes harder to understand, what actually happens.

9.1.2 Nonlinear Systems

In **Nonlinear System Identification**, we need to find an arbitrary function f (as opposed to matrix entries in linear systems), which makes the problem much harder.

It is often a good approach to work with linear systems and estimate bounds for the nonlinear part:

$$x(t+1) = Ax(t) + Bu(t) + f(x(t), u(t)) + w(t)$$

There exist robust control approaches for such linear systems, that incorporate the effect of nonlinearity.

When process noise is propagated through the nonlinearity, it is important to ensure a good signal to noise ratio by applying excitation to the system. Non-iid learning and closed loop control amplifies the challenging effects. Similarly to linear systems, observation noise is per se okay, but partial observations are bad, since they have similar implications as with linear systems.

Therefore, for nonlinear system identification we parametrize and learn the dynamics function $\hat{f}_\theta(x(t), u(t), w(t))$, and/or using (expensive) physical models.

Nonlinear State Space Models (NLSS)

$$x(t+1) = f(x(t), u(t)) + w(t)$$
$$y(t) = h(x(t), u(t), v(t))$$

Nonlinear Autoregressive Exogenous Model (NARX)

In **NARX** models, observations are modeled and learned through MSE or maximum likelihood:

$$y(t) = F(\phi(t)) = F(y(t-1), y(t-2), \dots, u(t-1), u(t-2), \dots)$$
$$\hat{y}_\theta(t) = F_\theta(\phi(t))$$
$$\Rightarrow \min_\theta \sum_{i=1}^{N} ||y(t) - \hat{y}(t, \theta)||^2$$

NARX only defines the input-output structure, not what F is. F_θ can be modeled using Fourier basis, radial basis functions, or deep neural networks. Possible NN architectures include feedforward NN, (temporal) CNNs, and also NN for sequences, such as *Recurrent Neural Networks (RNNs)*, *Long Short-Term Memory (LSTMs)*, *Large Language Models (LLMs)*.

9.1.3 Time Varying systems

Assume linear dynamics. For **windowing approaches**, we get time shifted trajectories of type $(x(1), u(1)), \dots, (x(N+1), u(N+1))$, resulting in time shifted matrices $A_{1:N+1}, B_{1:N+1}$:

$$X_{2:n+1} = \begin{bmatrix} A_{1:N+1} & B_{1:N+1} \end{bmatrix} \begin{pmatrix} x(1) & \cdots & x(N) \\ u(1) & \cdots & u(N) \end{pmatrix}$$

If the data distribution changes rarely and suddenly, we can simply have an accurate non-changing model, which is updated on demand, whenever a change in the dynamics is detected.

9.2 Learning-Based Model-Predictive Control (MPC)

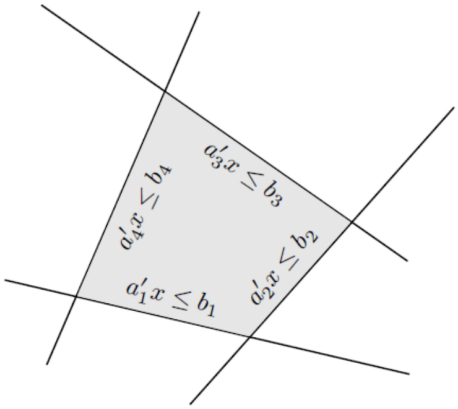
The main idea behind **Model Predictive Control (MPC)** is to minimize the cost of the next N states by unrolling the dynamics or shooting trajectories of control inputs $U_N := (u_0, \dots, u_N)$ for state x_0

$$\min_{U_N} J(x(0), U_N) = \sum_{k=0}^{N} p(x_k, u_k) + q(x_N)$$

with the same conditions and constraints as in MPC. Since the control inputs are only given implicitly (no explicit *control law*/policy $u_k = \phi(x_k)$), we want to obtain them through prediction and optimization. In order to mitigate the modelling errors, optimization is then

only applied onto the first input u_0, before moving on to the next (predicted) control input, where we unroll again.

The constraints of the optimization problem are usually **polytopes**, which are bounded **polyhedrons**. Having polyhedra constraints yields a polyhedron, resulting in *linear* constraints, making the optimization tractable with theoretical guarantees.



Definition: Polyhedron

A polyhedron $\mathcal{P}$ in $\mathbb{R}^n$ denotes an intersection of a finite set of closed halfspaces in $\mathbb{R}^n$:

$$\mathcal{P} = \{x \in \mathbb{R}^n : Ax \leq b\}$$

A common choice of costs are based on weighted L^p-norms $||\cdot||_p$:

$$\min_{U_N} J(x(0), U_N) = \sum_{k=0}^{N} ||Qx_k||_p + ||Ru_k||_p + ||Px_N||_p$$

subject to $x_{t+1} = f(x_k, u_k)$, with constraints $x_k \in \mathcal{P}_x$, $u_k \in \mathcal{P}_u$, $x_N \in \mathcal{X}_f$ for $k = 0, \ldots, N$.

Historically, MPC was heavily applied in process engineering with *slow* systems, where the is enough time for computations. For fast systems, however, this is actually a problem. Thus, the MPC can be evaluated *offline* and a NN can be trained to emulate the control actions. The main advantage here is, that the during inference, the evaluation of a NN is much faster than the evaluation of the MPC.

Algorithm: Agent-Based MPC (AMPC)

1. Choose $\bar{w}_d$ (uncertainty bound), determine $\bar{w}_{d,model}$ and calculate $\bar{w}_{d,approx}$.
2. Design the *RMPC* (Robust MPC) (offline calculations).
3. Learn $\pi_{approx} \approx \pi_{MPC}$ using NN.
4. Validate π_{approx}, if it is behaving similarly to π_{MPC}. If validation fails, repeat learning.

We differentiate between the following types of learning-based MPC:

- Approximate the controller via deep learning or other methods: $\pi(x, \theta) \approx \min_{U_N} J(x(0), U_N)$
- Learn the dynamics function to improve prediction accuracy: $x_{k+1} = \hat{f}_\theta(x_k, u_k)$
- Change controller design to achieve different behavior: $\sum_{k=0}^{N} p(x_k, u_k, \theta_p) + q(x_N, \theta_q)$
- Use MPC to enforce safety in combination with other control and learning algorithms.