Artificial Intelligence Methods in Control

Lars Grüne
Chair of Applied Mathematics
Mathematical Institute
University of Bayreuth
95440 Bayreuth, Germany
lars.gruene@uni-bayreuth.de
https://num.math.uni-bayreuth.de/en/team/lars-gruene/

Lecture Notes

Summer Semester 2021

Preface

These Lecture Notes were written to accompany a Master Course in Applied Mathematics that I gave in the Summer Semester 2021 at the University of Bayreuth, Germany. I would like to thank all the students of the course for their valuable feedback, which considerably helped to improve these notes. Apart from the literature that is cited throughout the text, the books *Reinforcement Learning: An Introduction* by Andrew Barto and Richard S. Sutton [1] and *Neuro Dynamic Programming* by Dimitri P. Bertsekas and John N. Tsitsiklis [2] have been very valuable sources for writing these notes.

Since the course was given in a virtual format, all lectures are recorded and I'd be happy to provide links to the respective recordings upon request.

Bayreuth, July 2021 Lars Grüne

Contents

Chapter 1

Introduction

In this lecture we will be concerned with nonlinear control systems, either in continuous time

$$\dot{x}(t) \quad = \quad f(x(t), u(t)) \tag{1.1}$$

or in discrete time

$$x(k+1) \quad = \quad g(x(k), u(k)). \tag{1.2}$$

Here $f : R^n \times \mathbb{R}^m \to \mathbb{R}^n$ is either a vector field in continuous time or $g : X \times U \to X$ is a transition function in discrete time, where X and U are arbitrary sets. In order to unify the notation, we define $X = \mathbb{R}^n$ and $U = \mathbb{R}^m$ in the continuous time case. As usual, x is the state and u is the control input of the system. In RL u is also called the *control action* and U is referred to as the *action space*. We denote the solution satisfying $x(0) = x_0$ by

$$x_u(t, x_0) \qquad \text{or} \qquad x_u(k, x_0),$$

respectively. The set of control functions in continuous time and the set of control sequences in discrete time are denoted by $\mathcal{U}$. In continuous time we assume measurability of the control functions in order to ensure solvability of (1.1) in the Caratheodory sense under the usual conditions on f. In discrete time we sometimes write (1.2) briefly as $x^+ = f(x, u)$.

The topics in this lecture center around a method called "reinforcement learning" (RL). In this method, a feedback control strategy is "learned" based on a so-called loss function $\ell(x, u)$, that assigns a loss to each state x and control u (the precise problem fornulation will be given in the next chapter). The goal is then to minimise the loss. Conceptually, this is nothing but an optimal control problem of a similar type as we have already discussed it in the Mathematical Control Theory lecture. Indeed, RL can be used as an alternative solution technique to the Riccati equation or to MPC. However, RL can also be used if f and ℓ are not known exactly or not known, at all, but can only be evaluated by means of measurements.

In the first half of this lecture we will discuss the foundations of RL for deterministic and non-deterministic discrete time problems with finitely many states and control inputs. We will in particular investigate conditions under which RL provably converges to the optimal strategy.

In the second half we will turn to deep RL — i.e., RL with deep neural networks as approximators — for problems with high-dimensional state space. Here we will in particular investigate the question when deep RL can overcome the so-called "curse of dimensionality", which describes the fact that typically the numerical effort grows exponentially with the dimension of the state space.

Chapter 2

Problem formulation

July 21, 2021

In discrete time, our goal is to find a control strategy such that

$$J(x_0, u(\cdot)) = \sum_{k=0}^{\infty} \gamma^k \ell(x_u(k, x_0), u(k)) \tag{2.1}$$

becomes minimal, where $\gamma \in (0, 1]$ is called the *discount factor*. In RL, "strategy" is usually understood as a control in feedback form and the corresponding fedback law is usually denoted by π. Hence, we are looking for a map $\pi : X \to U$, such that the solution $x_\pi(k, x_0)$ of

$$x(k+1) = g(x(k), \pi(x(k))), \qquad x(0) = x_0 \tag{2.2}$$

together with the corresponding control values $u(k) = \pi(x(k))$ minimises (2.1).

In continuous time, the problem is to find a control such that

$$J(x_0, u(\cdot)) = \int_0^{\infty} e^{-\delta t} \ell(x_u(t, x_0), u(t)) dt \tag{2.3}$$

becomes minimal, where $\delta \in (0, 1]$ is called the *discount rate*. Again, one would usually like to have the optimal strategy in feedback form. Again, we are looking for a map $\pi : X \to U$, such that the solution $x_\pi(k, x_0)$ of

$$\dot{x}(t) = f(x(t), \pi(x(t))), \qquad x(0) = x_0 \tag{2.4}$$

together with the corresponding control values $u(t) = \pi(x(t))$ minimises (2.3). We note that while equation (2.2) is always solvable without any additional conditions on π, equation (2.4) is a differential equation whose right hand side $x \mapsto f(x, \pi(x))$ needs to satisfy certain conditions in order to guarantee the existence and uniqueness of a solution. For this reason, the continuous-time problem is more difficult from a mathematical point of view. This is why in RL the discrete-time formulation is often preferred.

In order to avoid difficulties with the existence of the infinite sum and integral in (2.1) and (2.3), we make the following standing assumption throughout this lecture.

Assumption 2.1 One of the following two properties holds:

(i) $\ell(x,u) \geq 0$ for all $x \in X, u \in U$

(ii) $\sup_{x\in X, u\in U} |\ell(x,u)| < \infty$ and $\gamma < 1$ in discrete time or $\delta > 0$ in continuous time

□

This assumption implies that the infinite sum or integral always has a well defined value (which may be infinite). In addition, the assumption implies certain estimates for the *optimal value function*

$$V(x_0) := \inf_{u(\cdot)\in\mathcal{U}} J(x_0, u(\cdot)).$$

Lemma 2.2 Consider the optimal control problems of minimising (2.1) subject to (1.2) or of minimising (2.3) subject to (1.1). Let Assumption 2.1 hold. Then for all $u \in \mathcal{U}$ the limit

$$\lim_{K\to\infty} \sum_{k=0}^{K} \gamma^k \ell(x_u(k, x_0), u(k))$$

or

$$\lim_{T\to\infty} \int_0^T e^{-\delta t} \ell(x_u(t, x_0), u(t))dt$$

exists and has a finite value or diverges to $+\infty$. Moreover, for each trajectory $x(\cdot) = x_u(\cdot, x_0)$ we have the inequality

$$\liminf_{k\to\infty} \gamma^k V(x(k)) \geq 0 \quad \text{or} \quad \liminf_{t\to\infty} e^{-\delta t} V(x(t)) \geq 0.$$

In the particular case of Assumption 2.1(ii) we moreover have the inequalities

$$|V(x_0)| \leq \frac{M}{1-\gamma} \quad \text{and} \quad |J(x_0,u)| \leq \frac{M}{1-\gamma},$$

or

$$|V(x_0)| \leq \frac{M}{\delta} \quad \text{and} \quad |J(x_0,u)| \leq \frac{M}{\delta},$$

respectively, for all $x_0 \in X$ and any upper bound M for $|\ell(x,u)|$.

Proof: We show the assertion in discrete time; the continuous-time case follows similarly. If case (i) of Assumption (2.1) holds, then obviously

$$\sum_{k=0}^{K} \gamma^k \ell(x_u(k, x_0), u(k))$$

is nonnegative and strictly increasing in K. This shows the first claim and also implies that $V(x) \geq 0$ for all $x \in X$, which implies the second claim.

If case (ii) of Assumption (2.1) holds, then an upper bound $M \in \mathbb{R}$ with $|\ell(x,u)| \leq M$ for all $x \in X$, $u \in U$ holds. This implies that $|\gamma^k \ell(x_u(k, x_0), u(k))| \leq M\gamma^k$ and thus $\sum_{k=0}^{\infty} M\gamma^k$ is a convergent majorant series, implying absolute convergence and thus the first claim. Particularly, $|J(x_0,u)| \leq \frac{M}{1-\gamma}$ follows, which implies $|V(x)| \leq \frac{M}{1-\gamma}$ and thus the second claim since $\gamma^k \to 0$ as $k \to \infty$. □

We illustrate the definitions with three examples.

Example 2.3 Consider a system with six states as in Figure 2.1, denoted by $X = \{(i,j) \mid i = 1,2,\ j = 1,2,3\}$.

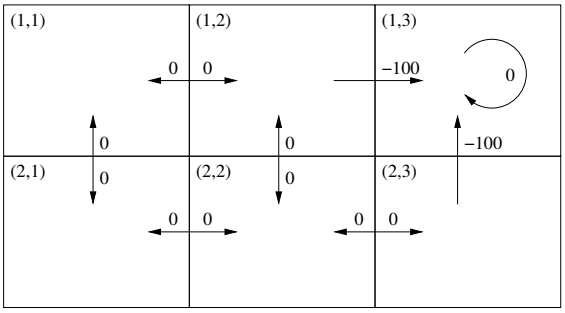

Figure 2.1: Sketch of Example 2.3

From each state $(i,j) \neq (1,3)$ it is possible to move to each neighbouring state. This can be formalised by setting $U = \{1,2,3,4\}$ and defining

$$g((i,j),u) := \begin{cases} (i, \min\{j+1,3\}), & \text{if } u = 1 \quad \text{(go right)} \\ (\min\{i+1,2\}, j), & \text{if } u = 2 \quad \text{(go down)} \\ (i, \max\{j-1,1\}), & \text{if } u = 3 \quad \text{(go left)} \\ (\max\{i-1,1\}, j), & \text{if } u = 4 \quad \text{(go up)} \end{cases}$$

for all $(i,j) \neq (1,3)$. Once the system is in state $(1,3)$, it cannot move anymore, i.e., we set

$$g((1,3),u) := (1,3) \quad \text{for all } u \in U.$$

Such a state is called an *absorbing state*.

The goal is to move the system to the absorbing state $(1,3)$. Hence we give a negative cost (i.e., a reward) to the transition to $(1,3)$ and a cost of 0 (i.e., no reward) to all other transitions. To this end, we set

$$\ell(x,u) = \begin{cases} -100, & \text{if } x \neq (1,3) \text{ and } g(x,u) = (1,3) \\ 0, & \text{else} \end{cases}$$

It is thus desirable to reach $(1,3)$ and if we use a discount factor $\gamma < 1$, then it is also desirable to do this as fast as possible, because the earlier $\ell(x,u) = -100$ occurs, the smaller $\gamma^k(-100)$ becomes. □

Example 2.4 Consider the second order differential equation $\ddot{x} = u$, which we can rewrite as the first order system

$$\begin{aligned} \dot{x}_1(t) &= x_2(t) \\ \dot{x}_2(t) &= u(t). \end{aligned}$$

This can be seen as a model of a car on a one-dimensional track with position x_1, velocity x_2 and acceleration u. A typical control task may be to bring the car to stop in a certain position (e.g., in $x = 0$). The cost function could therefore be chosen as $\ell(x,u) = \|x\|^2 + \lambda\|u\|^2$ with a parameter $\lambda \geq 0$. For $\lambda > 0$, this is a linear quadratic problem of the type we considered in Mathematical Control Theory.

If we keep the acceleration constant on the interval $[0,1]$, then we can explicitly calculate the solution

$$x_u(1,x) = \begin{pmatrix} x_1 + x_2 + u/2 \\ x_2 + u \end{pmatrix}.$$

The model

$$x^+ = g(x,u) = \begin{pmatrix} x_1 + x_2 + u/2 \\ x_2 + u \end{pmatrix}, \tag{2.5}$$

can thus be seen as a sampled-data model of the continuous-time model with sampling time $\tau = 1$. We can thus also define a discrete-time optimal control problem for this model. □

Example 2.5 A classical model in control theory is the inverted pendulum on a cart, also known as cart-pole system. This model consists of an inverted rigid pendulum fixed on a cart, cf. Figure 2.5.

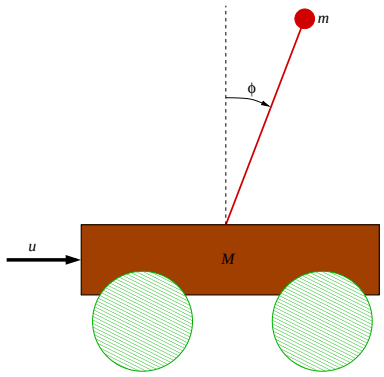

Figure 2.2: Schematic illustration of a pendulum on a cart

The control u here is the acceleration of the cart. By means of physical laws an "exact"[1] differential equation model can be derived.

$$\left.\begin{aligned} \dot{x}_1(t) &= x_2(t) \\ \dot{x}_2(t) &= -kx_2(t) + g\sin x_1(t) + u(t)\cos x_1(t) \\ \dot{x}_3(t) &= x_4(t) \\ \dot{x}_4(t) &= u \end{aligned}\right\} =: f(x(t),u(t)) \tag{2.6}$$

In this model the state vector $x \in \mathbb{R}^4$ consists of 4 components: x_1 represents the angle ϕ of the pendulum (cf. Fig. 2.5), which increases in counterclockwise direction, where $x_1 = 0$ corresponds to the upright pendulum. x_2 is the angular velocity, x_3 the position of the cart and x_4 its velocity. The constant k is a measure for the friction in the model (the larger k the more friction) and $g \approx 9.81 m/s^2$ is the gravitational constant.

[1] The model (2.6) is not really exact, since it is already simplified: We have assumed that the pendulum is so light that it does not influence the motion of the cart. Moreover, a number of constants was chosen such that they cancel each other.

We will use this model as a test problem in the exercises in the second half of this lecture. □

Optimal control problems often involve constraints on x and u. These constraints specify sets of admissible values of x and u and demand that no values outside these sets are used when minimising (2.1) or (2.3). In order to simplify the presentation we will not explicitly consider constraints in this lecture, but always consider them implicitly, by encoding them into the dynamics f or g or in the cost function ℓ.

For instance, in Example 2.3 there is the implicit state constraint that the system does not leave the rectangle depicted in Figure 2.1. This is realised by defining the dynamics g in such a way that leaving the rectangle is simply not possible.

In Example 2.4, it may be desirable to restrict acceleration and speed to physically meaningful quantities and it may also be desirable to restrict the position of the car. This can be done by defining a so-called *penalty function* ℓ_p, which yields large values $\ell_p(x,u)$ whenever the constraints are violated, and use $\ell + \ell_p$ as new cost function. This procedure is known as *soft constraints*. For instance, for a state constraint of the form $x_1 \in [-1,1]$, which may occur in Example 2.4, a possible penalty function ℓ_p might be

$$\ell_p(x,u) = \mu \begin{cases} (x-1)^2, & \text{if } x > 1 \\ 0, & \text{if } x \in [-1,1] \\ (x+1)^2, & \text{if } x < -1 \end{cases},$$

where $\mu > 0$ is a sufficiently large parameter. More generally, if control and state constraints are given in the form

$$\{(x,u) \,|\, g_i(x,u) \le 0 \text{ for all } i = 1,\dots,q\}$$

for functions $g_1,\dots,g_q$, then a penalty function could be defined as

$$\ell_p(x,u) = \sum_{i=1}^{q} \mu_i \max\{g_i(x,u), 0\}^2.$$

An alternative to penalty functions are *barrier functions*, whose value tends to ∞ as (x,u) approach the boundary of the constraint set. A typical barrier function is the logarithmic barrier

$$\ell_b(x,u) = \sum_{i=1}^{q} \mu_i(-\log(-g_i(x,u))).$$

Chapter 3

Dynamic Programming

July 21, 2021

Dynamic Programming is the name for an algorithm for solving optimal control problems that is very similar to RL. In fact, the basic principles behind dynamic programming, which we will present in this chapter, are very important for formulating and understanding the basic RL algorithm. We will present these in this chapter in the deterministic setting and will extend them to non-deterministic problems in Chapter 5.

3.1 Dynamic programming principle

Definition 3.1 Consider the optimal control problem of minimising (2.1) or (2.3) with initial value $x_0 \in X$

(i) The function
$$V(x_0) := \inf_{u(\cdot)\in\mathcal{U}} J(x_0, u(\cdot))$$
is called *optimal value function*.

(ii) A control sequence or function $u^\star(\cdot) \in \mathcal{U}$ is called *optimal* for initial value x_0 if
$$V(x_0) = J(x_0, u^\star(\cdot))$$
holds. The corresponding trajectory $x_{u^\star}(\cdot, x_0)$ is called *optimal trajectory*.

(iii) A strategy $\pi^\star : X \to U$ is called *optimal* if
$$V(x_0) = J(x_0, \pi^\star)$$
holds for all $x_0 \in X$, where, for an arbitrary feedback law $\pi : X \to U$,
$$J(x_0, \pi) := \sum_{k=0}^{\infty} \gamma^k \ell(x_\pi(k, x_0), \pi(x_\pi(k, x_0)))$$
in discrete time and
$$J(x_0, \pi) := \int_0^\infty e^{-\delta t} \ell(x_\pi(t, x_0), \pi(x_\pi(t, x_0)))dt$$
in continuous time, where $x_\pi(\cdot, x_0)$ solves (2.2) or (2.4), respectively. As in (ii), the corresponding trajectories $x_{\pi^\star}(\cdot, x_0)$ are called *optimal trajectories*.

□

We note that if $\pi^\star$ is an optimal feedback law, then $u^\star(\cdot) = \pi^\star(x_{\pi^\star}(\cdot, x_0))$ is an optimal control for initial value x_0.

The first result we state is the dynamic programming principle in discrete time.

Theorem 3.2 [Dynamic programming principle] Consider the optimal control problem (2.1) with $x_0 \in X$. Then for all $K \in \mathbb{N}$ the equation

$$V(x_0) = \inf_{u(\cdot)\in\mathcal{U}} \left\{ \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k, x_0), u(k)) + \gamma^K V(x_u(K, x_0)) \right\} \tag{3.1}$$

holds. If, in addition, an optimal control sequence $u^\star(\cdot)$ exists for x_0, then we get the equation

$$V(x_0) = \sum_{k=0}^{K-1} \gamma^k \ell(x_{u^\star}(k, x_0), u^\star(k)) + \gamma^K V(x_{u^\star}(K, x_0)). \tag{3.2}$$

In particular, in this case the "inf" in (3.1) is a "min".

Proof: From the definition of J for $u(\cdot) \in \mathcal{U}$ we immediately obtain

$$J(x_0, u(\cdot)) = \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k, x_0), u(k)) + \gamma^K J(x_u(K, x_0), u(\cdot + K)), \tag{3.3}$$

where $u(\cdot + K)$ denotes the shifted control sequence defined by $u(\cdot + K)(k) = u(k + K)$.

We now prove (3.1) by showing "≥" and "≤" separately: From (3.3) we obtain

$$\begin{aligned} J(x_0, u(\cdot)) &= \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k, x_0), u(k)) + \gamma^K J(x_u(K, x_0), u(\cdot + K)) \\ &\geq \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k, x_0), u(k)) + \gamma^K V(x_u(K, x_0)). \end{aligned}$$

Since this inequality holds for all $u(\cdot) \in \mathcal{U}$, it also holds when taking the infimum on both sides. Hence we get

$$\begin{aligned} V(x_0) &= \inf_{u(\cdot)\in\mathcal{U}} J(x_0, u(\cdot)) \\ &\geq \inf_{u(\cdot)\in\mathcal{U}} \left\{ \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k, x_0), u(k)) + \gamma^K V(x_u(K, x_0)) \right\}, \end{aligned}$$

i.e., (3.1) with "≥".

In order to prove "≤", fix $\varepsilon > 0$ and let $u^\varepsilon(\cdot) \in \mathcal{U}$ be an approximately optimal control sequence for the right hand side of (3.3), i.e.,

$$\sum_{k=0}^{K-1} \gamma^k \ell(x_{u^\varepsilon}(k, x_0), u^\varepsilon(k)) + \gamma^K J(x_{u^\varepsilon}(K, x_0), u^\varepsilon(\cdot + K))$$

$$\leq \inf_{u(\cdot)\in\mathcal{U}} \left\{ \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k,x_0),u(k)) + \gamma^K J(x_u(K,x_0),u(\cdot+K)) \right\} + \varepsilon.$$

Now, observing that the different terms either depend on $u(0),\dots,u(k-1)$ or on $\hat{u}(k) = u(k+K)$, $k\in\mathbb{N}$, we can rewrite this as

$$\begin{aligned}
&\inf_{u(\cdot)\in\mathcal{U}} \left\{ \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k,x_0),u(k)) + \gamma^K J(x_u(K,x_0),u(\cdot+K)) \right\} \\
&= \inf_{\substack{u(\cdot)\in\mathcal{U}\\ \hat{u}(\cdot)\in\mathcal{U}}} \left\{ \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k,x_0),u(k)) + \gamma^K J(x_u(x_0),\hat{u}(\cdot)) \right\} \\
&= \inf_{u(\cdot)\in\mathcal{U}} \left\{ \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k,x_0),u(k)) + \gamma^K V(x_u(x_0)) \right\}
\end{aligned}$$

Now (3.3) yields

$$\begin{aligned}
V(x_0) &\leq J(x_0,u^\varepsilon(\cdot)) \\
&= \sum_{k=0}^{K-1} \gamma^k \ell(x_{u^\varepsilon}(k,x_0),u^\varepsilon(k)) + \gamma^K J(x_{u^\varepsilon}(K,x_0),u^\varepsilon(\cdot+K)) \\
&\leq \inf_{u(\cdot)\in\mathcal{U}} \left\{ \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k,x_0),u(k)) + \gamma^K V(x_u(K,x_0)) \right\} + \varepsilon,
\end{aligned}$$

i.e.,

$$V(x_0) \leq \inf_{u(\cdot)\in\mathcal{U}} \left\{ \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k,x_0),u(k)) + \gamma^K V(x_u(K,x_0)) \right\} + \varepsilon.$$

Since $\varepsilon > 0$ was arbitrary and the expressions in this inequality are independent of ε, this inequality also holds for $\varepsilon = 0$, which shows (3.1) with "$\leq$" and thus (3.1).

In order to prove (3.2) we use (3.3) with $u(\cdot) = u^\star(\cdot)$. This yields

$$\begin{aligned}
V(x_0) &= J(x_0,u^\star(\cdot)) \\
&= \sum_{k=0}^{K-1} \gamma^k \ell(x_{u^\star}(k,x_0),u^\star(k)) + \gamma^K J(x_{u^\star}(K,x_0),u^\star(\cdot+K)) \\
&\geq \sum_{k=0}^{K-1} \gamma^k \ell(x_{u^\star}(k,x_0),u^\star(k)) + \gamma^K V(x_{u^\star}(K,x_0)) \\
&\geq \inf_{u(\cdot)\in\mathcal{U}} \left\{ \sum_{k=0}^{K-1} \gamma^k \ell(x_u(k,x_0),u(k)) + \gamma^K V(x_u(K,x_0)) \right\} = V(x_0),
\end{aligned}$$

where we used the (already proved) equality (3.1) in the last step. Hence, the two "$\geq$" in this chain are actually "=" which implies (3.2). □

In the special case $K = 1$ the dynamic programming principle becomes

$$V(x_0) = \inf_{u\in U} \{\ell(x_0,u) + \gamma V(g(x_0,u))\}. \tag{3.4}$$

This equation is known as the Bellman equation. In RL, the term in braces on the right hand side of (3.4) plays a particular important role, which is why it is commonly denoted with its own symbol

$$Q(x,u) := \ell(x,u) + \gamma V(g(x,u)). \tag{3.5}$$

Remark 3.3 In continuous time, an analogous proof shows that for all $T > 0$ the equation

$$V(x_0) = \inf_{u(\cdot)\in\mathcal{U}} \left\{ \int_0^T e^{-\delta t}\ell(x_u(t,x_0),u(t))dt + e^{-\delta T}V(x_u(T,x_0)) \right\} \tag{3.6}$$

and, for any optimal control function, the equation

$$V(x_0) = \int_0^T e^{-\delta t}\ell(x_{u^\star}(t,x_0),u^\star(t))dt + e^{-\delta T}V(x_{u^\star}(T,x_0)) \tag{3.7}$$

hold. □

The following corollary states an immediate consequence from the dynamic programming principle. It shows that tails of optimal controls are again optimal controls for suitably adjusted initial value and time.

Corollary 3.4 If $u^\star(\cdot)$ is an optimal control sequence minimising (2.1) with initial value x_0, then for each $K \in \mathbb{N}$ the sequence $u_K^\star(\cdot) = u^\star(\cdot + K)$, i.e.,

$$u_K^\star(k) = u^\star(k+K), \quad k = 0,1,\dots$$

is an optimal control sequence for initial value $x_{u^\star}(K,x_0)$. □

Proof: Inserting $V(x_0) = J(x_0,u^\star(\cdot))$ and the definition of $u_K^\star(\cdot)$ into (3.3) we obtain

$$V(x_0) = \sum_{k=0}^{K-1} \gamma^k \ell(x_{u^\star}(k,x_0),u^\star(k)) + \gamma^K J(K,x_{u^\star}(x_0),u_K^\star(\cdot))$$

Subtracting (3.2) from this equation yields

$$0 = \gamma^K J(x_{u^\star}(K,x_0),u_K^\star(\cdot)) - \gamma^K V(x_{u^\star}(K,x_0))$$

which shows the assertion. □

Remark 3.5 Analogously, in continuous time the control function $u_T^\star(\cdot) = u^\star(\cdot + T)$, i.e.,

$$u_T^\star(t) = u^\star(t+T), \quad t \ge 0$$

is an optimal control sequence for initial value $x_{u^\star}(T,x_0)$. □

In the next theorem by "argmin" we denote the set of minimisers of an expression.

Theorem 3.6 Consider the optimal control problem of minimising (2.1) and let Assumption 2.1 hold. Consider a feedback law $\pi^\star : X \to U$ satisfying

$$\pi^\star(x) \in \operatorname*{argmin}_{u\in U} \{\ell(x,u) + \gamma V(g(x,u))\} = \operatorname*{argmin}_{u\in U} Q(x,u) \tag{3.8}$$

for all $x \in X$. Then $\pi^\star$ is an optimal strategy in the sense of Definition 3.1(iii).

Proof: We pick an arbitrary $x_0 \in X$ and abbreviate $\hat{x}(k) = x_{\pi^\star}(k, x_0)$ and $\hat{u}(k) = \pi^\star(x_{\pi^\star}(k, x_0))$. Then $J(x_0, \pi^\star) = J(x_0, \hat{u})$ and we need to show that

$$J(x_0, \hat{u}) = V(x_0),$$

where it is enough to show "$\leq$" because the opposite inequality follows by definition of V. Using (3.8) and (3.4) with $x_0 = \hat{x}(k)$ we get

$$\gamma^k V(\hat{x}(k)) = \gamma^k \ell(\hat{x}(k), \hat{u}(k)) + \gamma^{k+1} V(\hat{x}(k+1))$$

for $k = 0, 1, \ldots$. Summing these equalities for $k = 0, \ldots, K-1$ for arbitrary $K \in \mathbb{N}$ and eliminating the identical terms $\gamma^k V(\hat{x}(k))$, $k = 1, \ldots, K-1$ on the left and on the right we obtain

$$V(x_0) = \sum_{k=0}^{K-1} \gamma^k \ell(\hat{x}(k), \hat{u}(k)) + \gamma^K V(\hat{x}(K)).$$

Now Lemma 2.2 implies $\liminf_{K\to\infty} \gamma^K V(\hat{x}(K)) \geq 0$. Hence we obtain for the limit

$$\lim_{K\to\infty} \sum_{k=0}^{K-1} \gamma^k \ell(\hat{x}(k), \hat{u}(k)) \leq V(x_0),$$

which implies

$$J(x_0, \hat{u}) = \sum_{k=0}^{\infty} \gamma^k \ell(\hat{x}(k), \hat{u}(k)) = \lim_{K\to\infty} \sum_{k=0}^{K-1} \gamma^k \ell(\hat{x}(k), \hat{u}(k)) \leq V(x_0),$$

i.e., the desired inequality. □

3.2 Value iteration

Value iteration is a first simple algorithmic method for computing V or an approximation thereof. We consider it here under case (ii) of Assumption 2.1, because in this setting its convergence is easier to proof. We moreover limit ourselves to the discrete-time case. The algorithm works as follows.

Algorithm 3.7 (Value iteration)

(0) Set $V_0 :\equiv 0$ and $k := 0$

(1) For $k = 0, 1, 2, \ldots$:

set

$$V_{k+1}(x) := \inf_{u \in U} Q_k(x, u) \quad \text{for all } x \in X,$$

with $Q_k(x, u) := \ell(x, u) + \gamma V_k(g(x, u))$

□

Of course, there are many questions related to this algorithm: How do we store V_k on a computer? How to compute the infimum over u for all x? These are exactly the questions that we will have to deal with when making RL a practical algorithm. However, if for the moment we simply assume that this is possible, then we can prove the following theorem.

Theorem 3.8 Consider the discrete-time problem of minimising (2.1) and let Assumption 2.1(ii) hold. Let $M > 0$ be a bound for $|\ell|$. Then the inequality

$$\sup_{x \in X} |V_k(x) - V(x)| \leq \frac{M\gamma^k}{1 - \gamma}$$

holds.

Proof: From (3.4) we know that

$$V(x) = \inf_{u \in U} \{\ell(x, u) + \gamma V(g(x, u))\}.$$

Fix $\varepsilon > 0$, let $x \in X$ be arbitrary and let u^ε and u_k^ε be control values satisfying

$$\ell(x, u^\varepsilon) + \gamma V(g(x, u^\varepsilon)) \leq \inf_{u \in U} \{\ell(x, u) + \gamma V(g(x, u))\} + \varepsilon$$

and

$$\ell(x, u_k^\varepsilon) + \gamma V_k(g(x, u_k^\varepsilon)) \leq \inf_{u \in U} \{\ell(x, u) + \gamma V_k(g(x, u))\} + \varepsilon.$$

Then we can estimate

$$\begin{aligned} V(x) - V_{k+1}(x) &\leq \ell(x, u_k^\varepsilon) + \gamma V(g(x, u_k^\varepsilon)) - \ell(x, u_k^\varepsilon) - \gamma V_k(g(x, u_k^\varepsilon)) + \varepsilon \\ &= \gamma V(g(x, u_k^\varepsilon)) - \gamma V_k(g(x, u_k^\varepsilon)) + \varepsilon \\ &\leq \gamma \sup_{x \in X} |V(x) - V_k(x)| + \varepsilon \end{aligned}$$

and

$$\begin{aligned} V_{k+1}(x) - V(x) &\leq \ell(x, u^\varepsilon) + \gamma V_k(g(x, u^\varepsilon)) - \ell(x, u^\varepsilon) - \gamma V(g(x, u^\varepsilon)) + \varepsilon \\ &= \gamma V_k(g(x, u_\varepsilon)) - \gamma V(g(x, u_\varepsilon)) + \varepsilon \\ &\leq \gamma \sup_{x \in X} |V(x) - V_k(x)| + \varepsilon, \end{aligned}$$

which yields

$$|V(x) - V_{k+1}(x)| \leq \gamma \sup_{x \in X} |V(x) - V_k(x)| + \varepsilon.$$

Since this inequality holds for all $\varepsilon > 0$ and all $x \in X$, it follows that

$$\sup_{x\in X} |V(x) - V_{k+1}(x)| \le \gamma \sup_{x\in X} |V(x) - V_k(x)|.$$

By induction we thus obtain

$$\sup_{x\in X} |V(x) - V_k(x)| \le \gamma^k \sup_{x\in X} |V(x) - V_0(x)|.$$

Since $V_0 \equiv 0$, Lemma 2.2 yields that

$$\sup_{x\in X} |V(x) - V_0(x)| = \sup_{x\in X} |V(x)| \le \frac{M}{1-\gamma}.$$

This shows the claim. □

We illustrate Algorithm 3.7 with two examples.

Example 3.9 Consider Example 2.3. We depict the values of V_k in the algorithm for $\gamma = 0.9$ schematically.

V_0 :

0	0	0
0	0	0

V_1 :

0	-100	0
0	0	-100

V_2 :

-90	-100	0
0	-90	-100

V_3 :

-90	-100	0
-81	-90	-100

After the third iteration the value function does not change anymore. This means that Algorithm 3.7 converges in finitely many steps for this example.

The optimal strategy π can now easily be computed from Formula (3.8). For the states $x = (1,1)$, $(1,2)$ and $(2,2)$ it is optimal to move to the right, i.e, $\pi^\star(x) = 2$. For state $x = (2,3)$ it is optimal to move up, i.e., $\pi^\star(x) = 1$. For state $x = (2,1)$ the argmin in (3.8) contains two elements, namely 1 ("go up") or 2 ("go right"), which means that both $\pi^\star(x) = 1$ and $\pi^\star(x) = 2$ are possible. Finally for the absorbing state $x = (1,3)$, all four controls are optimal, because they all lead to the same behavior and involve the same cost. □

Example 3.10 As the second example we consider Example 2.4 in the discrete-time version (2.5). For this example, the value iteration algorithm works even in the case that $\gamma = 1$. This can be proved similarly to the existence of the solution of the algebraic Riccati equation in Theorem 6.13 of Mathematical Control Theory. Here we carry out the first three steps of the iteration with $\ell(x,u) = x_1^2 + x_2^2 + u^2$ and $\gamma = 1$:

Starting with $V_0 \equiv 0$, it is easy to see that $V_1(x) = \|x\|^2$. This leads to

$$Q_1(x,u) = 2x_1^2 + 3x_2^2 + \frac{9}{4}u^2 + 2x_1x_2 + x_1u + 3x_2u.$$

Minimizing this expression with respect to u yields

$$V_2(x) = \frac{17}{9}x_1^2 + 2x_2^2 + \frac{4}{3}x_1x_2.$$

From this we can compute

$$Q_2(x,u) = \frac{26}{9}x_1^2 + \frac{19}{3}x_2^2 + \frac{11}{2}u^2 + \frac{46}{9}x_1x_2 + \frac{29}{9}x_1u + \frac{26}{3}x_2u.$$

Minimizing this expression yields

$$V_3(x) = \frac{4307}{1782}x_1^2 + \frac{289}{99}x_2^2 + \frac{764}{297}x_1x_2.$$

□

3.3 The Hamilton-Jacobi-Bellman equation

While Theorem 3.2 and Corollary 3.4 have direct continuous-time counterparts — as explained in the subsequent remarks — there is no such counterpart to Theorem 3.6. This is because while we can choose a minimal $K > 0$ in (3.1) in order to arrive at (3.4), we cannot choose a minimal $T > 0$ in (3.6), because this identity holds for all real numbers $T > 0$ and for the infimum $T = 0$ over all these T it becomes trivial.

The trick now lies in rewriting (3.6) before making T "minimal". This leads to the following theorem.

Theorem 3.11 (Hamilton-Jacobi-Bellman differential equation)
Let ℓ be continuous in x and u. Moreover, let $O \subseteq \mathbb{R}^n$ be open and such that $V|_O$ is finite.

(i) If V is continuously differentiable in $x_0 \in O$, then

$$-\delta V(x_0) + DV(x_0) \cdot f(x_0, u_0) + \ell(x_0, u_0) \geq 0$$

holds for all $u_0 \in \mathbb{R}^m$.

(ii) If u^* is an optimal control for initial value $x_0 \in O$, which is continuous in $t = 0$, and V is continuously differentiable in x_0, then

$$-\delta V(x_0) + \min_{u \in \mathbb{R}^m} \{DV(x_0) \cdot f(x_0, u) + \ell(x_0, u)\} = 0 \tag{3.9}$$

and the minimum is attained in $u = u^*(0)$. Equation (3.9) is called *Hamilton-Jacobi-Bellman equation*.

Proof: We first show the auxiliary identity

$$\lim_{\tau \searrow 0} \frac{1}{\tau} \int_0^\tau e^{-\delta\tau} \ell(x(t, x_0, u), u(t)) dt = \ell(x_0, u(0))$$

for each $u \in \mathcal{U}$ that is continuous in $t = 0$. Because of continuity of x and u in $t = 0$ and since ℓ is continuous, for any $\varepsilon > 0$ there is $t_1 > 0$ with

$$|e^{-\delta t} \ell(x_u(t, x_0), u(t)) - \ell(x_0, u(0))| \leq \varepsilon$$

for all $t \in [0, t_1)$. For $\tau \in (0, t_1]$ this yields

$$\begin{aligned} \left| \frac{1}{\tau} \int_0^\tau e^{-\delta t} \ell(x_u(t, x_0), u(t)) dt - \ell(x_0, u(0)) \right| &\leq \frac{1}{\tau} \int_0^\tau |\ell(x_u(t, x_0), u(t)) - \ell(x_0, u(0))| dt \\ &\leq \frac{1}{\tau} \int_0^\tau \varepsilon \;=\; \varepsilon \end{aligned}$$

and thus the statement for the limit, since $\varepsilon > 0$ was arbitrary.

Now both assertions follow:

(i) For $u(t) \equiv u_0 \in \mathbb{R}^m$, inequality (3.6) implies

$$V(x_0) \leq \int_0^\tau e^{-\delta t} \ell(x_u(t, x_0), u(t)) dt + e^{-\delta\tau} V(x(\tau, x_0, u))$$

and thus

$$\begin{aligned} -\delta V(x_0) + DV(x_0)f(x_0,u(0)) &= \frac{d}{dt}\Big|_{t=0} e^{-\delta t} V(x_u(t,x_0)) \\ &= \lim_{\tau \searrow 0} \frac{e^{-\delta\tau} V(x_u(\tau,x_0)) - V(x_0)}{\tau} \\ &\geq \lim_{\tau \searrow 0} -\frac{1}{\tau}\int_0^\tau \ell(x_u(t,x_0),u(t))dt = -\ell(x_0,u(0)), \end{aligned}$$

i.e., the first assertion.

(ii) From (i) we get

$$-\delta V(x_0) + \inf_{u\in\mathbb{R}^m}\{DV(x_0)\cdot f(x_0,u) + g(x_0,u)\} \geq 0.$$

Equation (3.7) moreover implies

$$V(x_0) = \int_0^\tau e^{-\delta t}\ell(x_{u^*}(t,x_0),u^*(t))dt + V(x_{u^*}(\tau,x_0)).$$

This yields

$$\begin{aligned} -\delta V(x_0) + DV(x_0)f(x_0,u^*(0)) &= \frac{d}{dt}\Big|_{t=0} e^{-\delta t} V(x_{u^*}(t,x_0)) \\ &= \lim_{\tau \searrow 0} \frac{e^{-\delta\tau} V(x_{u^*}(\tau,x_0)) - V(x_0)}{\tau} \\ &= \lim_{\tau \searrow 0} -\frac{1}{\tau}\int_0^\tau \ell(x_{u^*}(t,x_0),u^*(t))dt = -\ell(x_0,u^*(0)), \end{aligned}$$

which implies the existence of the minimum in $u = u^*(0)$ and the claimed identity. □

With the help of the Hamilton-Jacobi-Bellman equation we can now fornulate the counterpart of Theorem 3.6.

Theorem 3.12 Let V be continuously differentiable and let $\pi^\star : X \to U$ be a feedback law satisfying

$$\pi^\star(x) \in \operatorname*{argmin}_{u\in\mathbb{R}^m}\{DV(x)\cdot f(x,u) + \ell(x,u)\} \tag{3.10}$$

for all $x \in \mathbb{R}^n$ and such that the solutions $x_{\pi^\star}(t,x_0)$ of (2.4) exist and are continuous. Then $\pi^\star$ is an optimal strategy in the sense of Definition 3.1(iii)

Proof: We abbreviate $\hat{x}(t) = x_{\pi^\star}(t,x_0)$ and $\hat{u}(t) = \pi(x_{\pi^\star}(t,x_0))$. Then we get that $J(x_0,\pi^\star) = J(x_0,\hat{u})$ and equation (3.10) together with equation (3.9) evaluated in $x_0 = \hat{x}(t)$ yields

$$-\delta V(\hat{x}(t)) + DV(\hat{x}(t))\cdot f(\hat{x}(t),\hat{u}(t)) + \ell(\hat{x}(t),\hat{u}(t)) = 0$$

for all $x \in \mathbb{R}^n$. Using

$$e^{-\delta t}\big(-\delta V(\hat{x}(t)) + DV(\hat{x}(t))\cdot f(\hat{x}(t),\hat{u}(t))\big) = \frac{d}{dt}e^{-\delta t}V(\hat{x}(t))$$

yields

$$-\int_0^\tau \frac{d}{dt} e^{-\delta t} V(\hat{x}(t))dt = \int_0^\tau e^{-\delta t}\ell(\hat{x}(t),\hat{u}(t))dt.$$

Applying the fundamental theorem of calculus we then obtain

$$V(x_0) - e^{-\delta\tau}V(\hat{x}(\tau)) = \int_0^\tau e^{-\delta t}\ell(\hat{x}(t),\hat{u}(t))dt.$$

As in the proof of Theorem 3.6, Lemma 2.2 yields $\liminf_{\tau\to\infty} e^{-\delta\tau}V(\hat{x}(\tau)) \geq 0$. Thus, we obtain

$$V(x_0) \geq \lim_{\tau\to\infty}\left[V(x_0) - e^{-\delta\tau}V(\hat{x}(\tau))\right] = \int_0^\infty \ell(\hat{x}(t),\hat{u}(t))d\tau = J(x_0,\hat{u}).$$

This shows the claim since the converse inequality $V(x_0) \leq J(x_0,\hat{u})$ follows by definition of V. □

It should be noted that the assumptions for the continuous-time Theorem 3.12 are significantly more restrictive as those for its discrete-time counterpart Theorem 3.6. First of all, there are many optimal control problems in which the optimal value function V is not continuously differentiable. Fortunately, there is a remedy for this, because in this case, a generalised solution concept — the so-called viscosity solutions — can be used. However, then in general any feedback law $\pi^\star$ satisfying (3.10) is discontinuous, which makes the assumption that (2.4) has a unique solution very difficult to check; in fact, this may not even be true. All these difficulties motivate the fact that in RL often the discrete-time formulation is preferred.

Chapter 4

Reinforcement learning with finite state and action space

July 21, 2021

In this chapter we introduce the reinforcement learning algorithm and analyse its convergence behaviour. We restrict ourselves to discrete time problems (1.2), (2.1) for which the sets X and U are finite, i.e., they only contain finitely many elements. Obviously, Example 2.3 falls into this class, but we may also convert Example 2.4 into a model that satisfies this assumption. To this end, consider numbers $x_{1,\max}, x_{2,\max}, u_{\max} \in \mathbb{N}$ such that $u_{\max} \geq 2x_{2,\max}$. Define

$$X = \{(x_1, x_2) \in \mathbb{R}^2 \,|\, 2x_1 \in \mathbb{Z}, x_2 \in \mathbb{Z}, |x_1| \leq x_{1,\max}, |x_2| \leq x_{2,\max}\}, \quad U = \{u \in \mathbb{Z} \,|\, |u| \leq u_{\max}\}.$$

Then the structure of X and U and the inequality $u_{\max} \geq 2x_{2,\max}$ implies that for each $x \in X$ there is $u \in U$ with $g(x,u) \in X$. In what follows we assume that $g(x,u) \in X$ for all $x \in X$, $u \in U$. This can be achieved for this model by suitably modifying g for those x, u for which this condition does not hold.

4.1 Q-learning

The basic reinforcement learning algorithm works as follows. The algorithm "learns" the map $Q : X \times U \to \mathbb{R}$ from (3.5) and is thus called *Q-learning*. Since X and u are finite, Q can be represented by its finitely many values $Q(x,u)$, $x \in X$, $u \in U$. If we number the elements of X and U as $x^1, \dots, x^N$, $u^1, \dots, u^M$, then the map Q can be represented by the $N \times M$-matrix $(Q_{ij}) = (Q(x^i, u^j))$.

Algorithm 4.1 (Q-learning)

(0) Set $\widetilde{Q} \coloneqq 0$, pick a state $x \in X$

(1) Select $u \in U$, evaluate/observe $x' = g(x,u) \in X$ and evaluate $\ell(x,u)$

(2) Set $\widetilde{Q}(x,u) \coloneqq \ell(x,u) + \gamma \min_{u' \in U} \widetilde{Q}(x',u')$

(3) Set $x \coloneqq x'$ or select a new $x \in X$ and go to (1)

□

The choice between "set $x := x'$" and "select a new $x \in X$" depends on whether we have the formulas for g at hand and can choose x freely, or whether we observe a real process, where it may need additional effort to restart it with a different state than x'.

4.2 Convergence analysis

The following theorem gives a first convergence result for this algorithm.

Theorem 4.2 Consider the discrete-time problem of minimising (2.1) with finite X and U and let Assumption 2.1(ii) hold. Denote by $\widetilde{Q}_j$ the $\widetilde{Q}$-function after Step (2) of Algorithm 4.1 has been executed j times. Assume that each pair $(x,u) \in X \times U$ appears infinitely often in Step (1) of the algorithm. Then

$$\lim_{j\to\infty} \widetilde{Q}_j(x,u) = Q(x,u)$$

for all $x \in X$, $u \in U$.

Proof: Observe that the definition of Q from (3.5) and the Bellman equation (3.4) imply

$$\min_{u\in U} Q(x,u) = \min_{u\in U}\{\ell(x,u) + \gamma V(g(x,u))\} = V(x)$$

and thus Q satisfies the relation

$$Q(x,u) = \ell(x,u) + \gamma V(g(x,u)) = \ell(x,u) + \gamma \min_{u'\in U} Q(g(x,u),u'). \tag{4.1}$$

Moreover, from the inequality for $|J(x,u)|$ in Lemma 2.2 we obtain that $|Q(x,u)| \le \frac{M}{1-\gamma}$ for $M = \max_{x\in X, u\in U} |\ell(x,u)|$.

Since every state-action pair $(x,u) \in X \times U$ appears infinitely often in Step (1), for each $j \in \mathbb{N}$ there is $p(j) \in \mathbb{N}$ such that each pair $(x,u) \in X \times U$ appears at least once in Step (2) of the algorithm during the executions $j+1, j+2, \dots, p(j)$ in the algorithm. We define

$$\|\widetilde{Q}_j - Q\|_\infty := \max_{x\in X, u\in U} |\widetilde{Q}_j(x,u) - Q(x,u)|$$

and claim that

$$\|\widetilde{Q}_k - Q\|_\infty \le \gamma \|\widetilde{Q}_j - Q\|_\infty \tag{4.2}$$

for all $k \ge p(j)$. This shows the claim because if we define $p^1(j) = p(j)$, $p^{l+1}(j) = p(p^l(j))$, then applying (4.2) inductively implies that

$$\|\widetilde{Q}_k - Q\|_\infty \le \gamma^l \|\widetilde{Q}_0 - Q|_\infty \le \gamma^l \frac{M}{1-\gamma}$$

for all $k \ge p^l(0)$, which proves the claim since $\gamma^l \to 0$ as $l \to \infty$.

Now consider the j-th time that Step (2) of the algorithm is executed. Then, using the definition of $\widetilde{Q}_j$ and (4.1), for x and u from Step (2) we can compute

$$\begin{aligned}
|\widetilde{Q}_j(x,u) - Q(x,u)| &= \left|\left(\ell(x,u) + \gamma \min_{u' \in U} \widetilde{Q}_{j-1}(x',u')\right) - \left(\ell(x,u) + \gamma \min_{u' \in U} Q(x',u')\right)\right| \\
&= \gamma \left|\min_{u' \in U} \widetilde{Q}_{j-1}(x',u') - \min_{u' \in U} Q(x',u')\right| \\
&\leq \gamma \max u' \in U \left|\widetilde{Q}_{j-1}(x',u') - Q(x',u')\right| \\
&\leq \gamma \|\widetilde{Q}_{j-1} - Q\|_\infty .
\end{aligned}$$

For all other $x \in X$ and $u \in U$ we obtain that

$$|\widetilde{Q}_j(x,u) - Q(x,u)| = |\widetilde{Q}_{j-1}(x,u) - Q(x,u)|.$$

These inequalities in particular imply

$$\|\widetilde{Q}_j - Q\|_\infty \leq \|\widetilde{Q}_{j-1} - Q\|_\infty$$

and thus

$$\|\widetilde{Q}_{j'} - Q\|_\infty \leq \|\widetilde{Q}_j - Q\|_\infty$$

for all $j' \geq j$. Now for any pair (x,u) denote by $q_{x,u}$ the largest iteration number in $\{j+1, j+2, \ldots, p(j)\}$ for which x and u appear in Step (2) of the algorithm. Then we obtain

$$|\widetilde{Q}_{p(j)}(x,u) - Q(x,u)| \leq |\widetilde{Q}_{q_{x,u}}(x,u) - Q(x,u)| \leq \gamma \|\widetilde{Q}_{q(x,u)-1} - Q\|_\infty \leq \gamma \|\widetilde{Q}_j - Q\|_\infty$$

implying for each $k \geq p(j)$

$$\|\widetilde{Q}_k - Q\|_\infty \leq \|\widetilde{Q}_{p(j)} - Q\|_\infty = \max_{x \in X, u \in U} |\widetilde{Q}_{p(j)}(x,u) - Q(x,u)| \leq \gamma \|\widetilde{Q}_j - Q\|_\infty$$

and thus (4.2). □

We now turn to the analysis of part (i) of Assumption 2.1. Here, we will in particular look at the case $\gamma = 1$, because in the case $\gamma < 1$ and with finite states and action sets part (i) of Assumption 2.1 implies part (ii). Hence, this situation is readily covered by Theorem 4.2.

In order to prove convergence in this case, we need a preparatory lemma.

Lemma 4.3 Consider the discrete-time problem of minimising (2.1) with finite X and U and $\gamma = 1$, and let Assumption 2.1(i) hold. Assume that the optimal value function V satisfies $V(x) < \infty$ for all $x \in X$. Then for each control sequence $u(\cdot)$ the inequality

$$J(x,u) \geq Q(x,u(0)) - \limsup_{K \to \infty} Q(x(K), u(K))$$

holds.

Proof: We abbreviate $x(k) = x_u(k,x)$. Using (4.1) with $\gamma = 1$, $x = x(k)$ and $u = u(k)$, we obtain

$$\ell(x(k),u(k)) = Q(x(k),u(k)) - \min_{u' \in U} Q(x(k+1),u') \geq Q(x(k),u(k)) + Q(x(k+1),u(k+1)).$$

This implies

$$\begin{aligned}\sum_{k=0}^{K} \ell(x(k),u(k)) &\geq \sum_{k=0}^{K} (Q(x(k),u(k)) - Q(x(k+1),u(k+1))) \\ &= Q(x(0),u(0)) - Q(x(K+1),u(K+1)).\end{aligned}$$

Taking the limit inferior on both sides and using that due to non-negativity of ℓ it coincides with the (possible infinite) limit of the sum on the left side and that the identity $\liminf -a_k = -\limsup a_k$ holds on the right then shows the assertion. □

Now we can prove the theorem in case Assumption 2.1(i) holds.

Theorem 4.4 Consider the discrete-time problem of minimising (2.1) with finite X and U and $\gamma = 1$, let Assumption 2.1(i) hold and assume that the optimal value function V satisfies $V(x) < \infty$ for all $x \in X$. Denote by $\widetilde{Q}_j$ the Q-function after Step (2) of Algorithm 4.1 has been executed j times. Assume that each pair $(x,u) \in X \times U$ appears infinitely often in Step (1) of the algorithm. Then for all sufficiently large $j \in \mathbb{N}$ we have that

$$\widetilde{Q}_j(x,u) = Q(x,u)$$

for all $x \in X$, $u \in U$.

Proof: We first note that since $\ell \geq 0$ the values $\widetilde{Q}(x,u)$ during the algorithm are always ≥ 0. Next we prove that $\widetilde{Q}(x,u) \leq Q(x,u)$ holds at each time during the execution of the algorithm for all x and u. Clearly, since $Q(x,u) \geq 0$ this holds at the start of the algorithm. Now assume that this property holds before Step (2) of the algorithm. Then, using (4.1) after Step (2) we obtain

$$\widetilde{Q}(x,u) = \ell(x,u) + \min_{u' \in U} \widetilde{Q}(x',u') \leq \ell(x,u) + \min_{u' \in U} Q(x',u') = Q(x,u).$$

Hence, the inequality persists in each iteration of the algorithm and thus for all times.

Now we show that $\widetilde{Q}(x,u)$ is increasing when the algorithm proceeds. To this end, denote again by $\widetilde{Q}_j$ the function obtained after the j-th iteration. Clearly, since $\widetilde{Q}_0 \equiv 0$ and $\widetilde{Q}_1 \geq 0$, the statement is true in the first iteration. Now assume that $\widetilde{Q}_0, \ldots, \widetilde{Q}_j$ are increasing. Let (x,u) be the state-action pair in the $j+1$-st iteration. Then either $\widetilde{Q}(x,u)$ was not updated before, implying $\widetilde{Q}_j(x,u) = 0$, or it was updated before. In this case we let $j' \leq j$ be the last iteration where $\widetilde{Q}(x,u)$ was updated, implying that

$$\widetilde{Q}_j(x,u) = \ell(x,u) + \min_{u' \in U} \widetilde{Q}_{j'-1}(x',u').$$

Since Q is increasing until iteration j, we obtain $\widetilde{Q}_j(x',u') \geq \widetilde{Q}_{j'-1}(x',u')$. This implies

$$\widetilde{Q}_{j+1} = \ell(x,u) + \min_{u' \in U} \widetilde{Q}_j(x',u') \geq \ell(x,u) + \min_{u' \in U} \widetilde{Q}_{j'-1}(x',u').$$

Hence, $\widetilde{Q}_0, \ldots, \widetilde{Q}_{j+1}$ are increasing and by induction we can conclude that $\widetilde{Q}_j$ is increasing for all $j \in \mathbb{N}$.

From what we have shown wo far we can immediately conclude that if $\widetilde{Q}_j(x,u) = Q(x,u)$ for some $j \in \mathbb{N}$, then $\widetilde{Q}_{j'}(x,u) = Q(x,u)$ for all $j' > j$. Moreover, we have the inequality

$$\sum_{x\in X, u\in U} \widetilde{Q}_j(x,u) \leq \sum_{x\in X, u\in U} Q(x,u),$$

in which "=" holds if and only if $\widetilde{Q}_j = Q$. Moreover, the expression $\sum_{x\in X,u\in U} \widetilde{Q}_j(x,u)$ is increasing in j and can only attain finitely many different values, because ℓ can only attain finitely many different values.

We now use these properties to show the claim. To this end, using $p(j)$ as defined in the proof of Theorem 4.2, we prove that unless $\widetilde{Q}_j = Q$, for at least one $(x,u) \in X \times U$ the inequality $\widetilde{Q}_{p(j)}(x,u) > \widetilde{Q}_j(x,u)$ holds. This shows that $\sum_{x\in X,u\in U} \widetilde{Q}_j(x,u)$ increases and since this sum can only attain finitely many different values, after finitely many increases it will coincide with $\sum_{x\in X,u\in U} Q(x,u)$. Then, $\widetilde{Q}_j$ and Q also coincide.

In order to show $\widetilde{Q}_{p(j)}(x,u) > \widetilde{Q}_j(x,u)$ for at least one (x,u) we proceed by contradiction. We assume that $\widetilde{Q}_j(\hat{x},\hat{u}) \neq Q(\hat{x},\hat{u})$ for at least one $(\hat{x},\hat{u}) \in X \times U$ (implying $\widetilde{Q}_j(\hat{x},\hat{u}) < Q(\hat{x},\hat{u})$) and that $\widetilde{Q}_{j'}(x,u)$ does not grow for any $j' \in \{j+1,\dots,p(j)\}$ and any $(x,u) \in X\times U$. The latter implies that

$$\widetilde{Q}_j(x,u) = \ell(x,u) + \min_{u'\in U} \widetilde{Q}_j(x',u') \tag{4.3}$$

holds for all $(x,u) \in X \times U$.

Now for each $x \in X$ by $u_x \in U$ we denote denote the control value satisfying

$$\widetilde{Q}_j(x,u_x) = \min_{u'\in U} \widetilde{Q}_j(x,u').$$

Then we can inductively define a control sequence and a corresponding trajectory by setting $u(0) := \hat{u}$, $x(0) := \hat{x}$ and

$$x(i+1) := g(x(i),u(i)), \quad u(i+1) := u_{x(i+1)}.$$

Using (4.3) and the definition of u_x, this yields

$$\begin{aligned}
\sum_{k=0}^{K} \ell(x(k),u(k)) &= \sum_{k=0}^{K} \big(\widetilde{Q}_j(x(k),u(k)) - \widetilde{Q}_j(x(k+1),u(k+1))\big) \\
&= \widetilde{Q}_j(x(0),u(0)) - \widetilde{Q}_j(x(K+1),u(K+1)) \\
&\leq \widetilde{Q}_j(x(0),u(0)) = \widetilde{Q}_j(\hat{x},\hat{u}).
\end{aligned}$$

Since $\ell \geq 0$, this implies that the limit for $K \to \infty$ exists and we obtain

$$J(\hat{x},u) \leq \widetilde{Q}_j(\hat{x},\hat{u}) < Q(\hat{x},\hat{u}). \tag{4.4}$$

Since $J(\hat{x},u)$ is finite, $\ell(x(k),u(k))$ must converge to 0. Since ℓ can only attain finitely many values, this implies that there is $k' \in \mathbb{N}$ with $\ell(x(k),u(k)) = 0$ for all $k \geq k'$. This implies that $V(x(k)) = 0$ for all $k \geq k'$ and $Q(x(k),u(k)) = 0$ for all $k \geq k'$. Hence, Lemma 4.3 yields

$$J(\hat{x},u) \geq Q(\hat{x},u(0)) - \limsup_{K\to\infty} Q(x(K),u(K)) = Q(\hat{x},u(0)) = Q(\hat{x},\hat{u}),$$

which contradicts (4.4). □

Once the Q-Learning algorithm has computed a sufficiently accurate approximation $\widetilde{Q} \approx Q$, a policy can be defined by choosing a policy satisfying

$$\tilde{\pi}(x) \in \operatorname*{argmin}_{u \in U} \widetilde{Q}(x,u). \tag{4.5}$$

The following theorem shows that this is an approximately optimal policy. For brevity, we only formulate and prove it for the case $\gamma < 1$.

Theorem 4.5 Let the assumptions of Theorem 4.2 and Assumption 2.1(ii) hold and assume that

$$\sup_{x \in X, u \in U} |Q(x,u) - \widetilde{Q}(x,u)| \le \varepsilon$$

for some $\varepsilon > 0$. Then for all $x \in X$ the inequality

$$J(x,\tilde{\pi}) \le V(x) + \frac{2\varepsilon}{1-\gamma}$$

holds.

Proof: The assumption on $\widetilde{Q}$ and the definition of $\tilde{\pi}$ implies that

$$Q(x,\tilde{\pi}(x)) \le \widetilde{Q}(x,\tilde{\pi}(x)) + \varepsilon = \min_{u \in U} \widetilde{Q}(x,u) + \varepsilon \le \min_{u \in U} Q(x,u) + 2\varepsilon = V(x) + 2\varepsilon.$$

This yields the inequality

$$\ell(x,\tilde{\pi}(x)) + \gamma V(g(x,\tilde{\pi}(x))) = Q(x,\tilde{\pi}(x)) \le V(x) + 2\varepsilon$$

and thus, since $x_{\tilde{\pi}}(k+1,x_0) = g(x_{\tilde{\pi}}(k,x_0), \tilde{\pi}(x_{\tilde{\pi}}(k,x_0)))$,

$$\begin{aligned} J(x_0,\tilde{\pi}) &= \sum_{k=0}^{\infty} \gamma^k \ell(x_{\tilde{\pi}}(k,x_0), \tilde{\pi}(x_{\tilde{\pi}}(k,x_0))) \\ &\le \sum_{k=0}^{\infty} \gamma^k \Big(V(x_{\tilde{\pi}}(k,x_0)) - \gamma V(x_{\tilde{\pi}}(k+1,x_0)) + 2\varepsilon \Big) \\ &= \sum_{k=0}^{\infty} \gamma^k 2\varepsilon + V(x) - \lim_{K \to \infty} \gamma^k V(x_{\tilde{\pi}}(k,x_0)). \end{aligned}$$

By Lemma 2.2 the last term is ≥ 0 and since $\gamma < 1$ the sum over γ^k evaluates to $1/(1-\gamma)$. We can thus conclude the claimed inequality

$$J(x,\tilde{\pi}) \le V(x) + \frac{2\varepsilon}{1-\gamma}.$$

□

4.3 Choice of x and u in the algorithm

Choice of x The possible choices of x in Step (3) of the algorithm depend on whether we obtain the values of $g(x,u)$ and $\ell(x,u)$ by simulation or by experiment. In the second case, it may be more efficient to use $x = x'$ in most cases, as using $x \neq x'$ means that we have to restart the experiment with a new initial value, which may be costly. On the other hand, always using $x = x'$ may be inefficient, because then the algorithm only "sees" one particular solution any fails to see those parts of the state space X that are not visited by this solution. It is therefore common to reset x after a couple of steps. The time between two resets is usually called *episode* in RL.

A method that is often effective is to store the values $g(x,u)$ and $\ell(x,u)$ of an episode and reuse them. This can be done in the same order as they originally occured or in reverse order. This can be particularly efficient if one step of the experiment to evaluate $g(x,u)$ and $\ell(x,u)$ takes a long time.

In case that we know $g(x,u)$ and $\ell(x,u)$ and can efficiently evaluate them, many more efficient algorithms are possible. For instance, in the setting of Assumption 2.1(i), it is possible to order x and u "on the fly" in such a way that each value $\widetilde{Q}(x,u)$ can be computed correctly in one shot, i.e., without the need of an iteration. This approach is known as a *Dijkstra-like* algorithm. Even with the computational cost of the sorting taken into account, the computational complexity with such an algorithm can be brought down to $NM\log(N)$ which is much faster than the "brute force" trying of all x and u in a random order. In the setting of Assumption 2.1, so-called *policy iteration schemes* can be used, which also converge much faster in many situations.

Choice of u Clearly, if the set U is large, the number of iterations until all values $Q(x,u)$ are updated is very large and it will take a lot of time until the algorithm converges. Then, however, not all controls are really needed to be considered in order to arrive at a good solutions. It suffices to use the "good" controls, which actually realize the minimum of Q. The trouble, however, is, that we do not know in advance which controls are "good". In order to use only relevant u in Step (1), several selection strategies have been proposed.

An obvious strategy would be to always use the u that minimises $\widetilde{Q}(x,u)$. However, when the values of $\widetilde{Q}$ are still far from those of Q, this can lead to non-optimal choices and, more importantly, the algorithm will never be able to correct these non-optimal choices. Hence, a good strategy should try other control values, too, but is is still a good idea to use those that lead to a small value of $\widetilde{Q}(x,u)$ more often. This idea is realised by choosing a $k \geq 1$ and assigning to each control the value

$$P(u) = \frac{k^{-\widetilde{Q}(x,u)}}{\sum_{u' \epsilon U} k^{-\widetilde{Q}(x,u')}}.$$

Then the control u in Step (1) is chosen randomly with probability $P(u)$. In order to see how this works and how the choice of k affects the results, assume we have three controls u_1, u_2, u_2 with values $\widetilde{Q}(x,u_1) = 1$, $\widetilde{Q}(x,u_2) = 2$, and $\widetilde{Q}(x,u_3) = 3$. For $k = 2$, we then obtain $P(u_1) = 4/7$, $P(u_2) = 2/7$, and $P(u_3) = 1/7$. For $k = 3$ we obtain $P(u_1) = 9/13$, $P(u_2) = 3/13$, and $P(u_3) = 1/13$. In the opposite direction, for $k = 1$ we obtain $P(u_1) = P(u_2) = P(u_3) = 1/3$. This means, the larger k is, the more the control values with small $\widetilde{Q}$

are favoured and the closer k is to one, the more the probabilities are equal. There are also variants of the algorithm in which k is variied with the number of iterations, with $k \approx 1$ in the beginning (such that all control values are explored with equal probability) and larger k as the iteration progresses.

Another method for choosing u is the so-called *ε-greedy* choice. Here one fixes a small $\varepsilon > 0$ and always uses the u that minimises $\widetilde{Q}(x,u)$ with probability $1-\varepsilon$. With probability ε, an arbitrary u is chosen. This method is in particular interesting in the context of the so called *SARSA* algorithm. This is a variant of Q-learning in which the update step (2) is replaced by

(2') Set $\widetilde{Q}(x,u) := (1-\alpha)\widetilde{Q}(x,u) + \alpha\left(\ell(x,u) + \gamma\widetilde{Q}(x',u')\right)$

Here $\alpha \in (0,1]$ is a step size and $u' \in U$ a control value, which can be chosen by different rules. If one chooses $\alpha = 1$ and u' as the minimiser of $u \mapsto \widetilde{Q}(x',u)$, then we obtain the original Q-Learning algorithm. If we keep this choice of u but set $\alpha < 1$, then one can prove that the SARSA algorithm converges to the same Q and thus the same optimal policy as Q-Learning. If, however, u' is chosen according to the ε-greedy algorithm, then the algorithm may converge to a different solution. While this solution is not the optimal solution anymore, it may have other beneficial properties.

Chapter 5

Non-deterministic Reinforcement Learning

July 21, 2021

So far we have assumed that for each pair of state x and control action u there is a unique successor state $g(x,u)$. This, however, is not true in many practically relevant situations:

- When we obtain the value $x' = g(x,u)$ from experiments, it is most very that the measurements are subject to noise and thus if we use a pair (x,u) several times it may be likely that we do not always get the same successor state.
- When we have an infinite state space, e.g., $x \in \Omega \subset \mathbb{R}^2$, then a typical way to pass to a finite state space X is by quantization. This means that each state $x \in X$ represents a small region (e.g., e square or rectangle in $\mathbb{R}^2$). Even if the original dynamics is deterministic, the image of a region in $\mathbb{R}^2$ under the dynamics will usually cover several regions.
- Finally, RL has been used very successfully in games such as backgammon or chess, in which the next state depends also on the other player's action and, possibly, on chance (like the rolling of a dice).

5.1 Definitions

For these reasons, we now extend the setting to non-deterministic models. As in the previous chapter, we will stick to discrete time and finite state and control action sets X and U. However, for each pair (x,u) the expression $g(x,u)$ is now a random variable, which, depending on chance, can yield different successor states x' with different probabilities. These probabilities are modeles by the map

$$p : X \times U \times X \to [0,1]$$

with the convention that

$$\sum_{x' \in X} p(x,u,x') = 1$$

for all $(x,u) \in X \times U$. The interpretation of the map p is:

If we are in state x and use the control u, then $p(x,u,x')$ is the probability to be in state x' after one time step, i.e., the probability that $g(x,u) = x'$.

Such a is called a *finite-state Markov chain.* The deterministic setting of the last chapter is then recovered by defining $p(x,u,x') = 1$ if $x' = g(x,u)$ and $p(x,u,x') = 0$ else.

Example 5.1 We reconsider Example 2.3, but now for each transition from one state to another there is an uncertainly of 10% that the system moves to a different neighbouring state than intended. Transitions that do not change the state remain unchanged. To this end, we define, e.g.,

$$\begin{aligned}
p((1,1),1,(1,2)) &= 0.9\\
p((1,1),1,(2,1)) &= 0.1\\
p((1,1),2,(1,2)) &= 0.1\\
p((1,1),2,(2,1)) &= 0.9\\
p((1,1),3,(1,1)) &= 1.0\\
p((1,1),4,(1,1)) &= 1.0,
\end{aligned}$$

$$\begin{aligned}
p((1,2),1,(1,3)) &= 0.9\\
p((1,2),1,(1,1)) &= 0.1\\
p((1,2),2,(2,2)) &= 0.1\\
p((1,2),2,(1,1)) &= 0.9\\
p((1,2),3,(1,1)) &= 0.9\\
p((1,2),3,(1,2)) &= 0.1\\
p((1,2),4,(1,2)) &= 1.0,
\end{aligned}$$

Here all values with $p(x,u,x') = 0$ are omitted. Similarly, we can define $p(x,u,x')$ for all other states x. □

Due to the non-deterministic model, for each initial state x_0 and each control sequence $u \in \mathcal{U}$ there is not a single trajectory $x_u(k,x_0)$ but many of them, each with its own probability. In other words, $x_u(k,x_0)$ is now a random variable. We express this by using a capital "X" and by adding an additional argument $X_u(k,x_0,\omega)$, $\omega \in \Omega$, where (Ω,Σ,P) is a probability space. If we omit ω, the symbol $X_u(k,x_0)$ stands for the set if all possible trajectories. Likewise, we write $g(x,u,\omega)$ and $g(x,u)$. Note that for a control sequence with $u(0) = u_0$ we then obtain $X_u(1,x_0) = g(x_0,u_0)$.

The fact that there are now many trajectories raises the need to generalise the concept of control sequences. For instance, in Example 5.1, the goal is to reach the state $(1,3)$, as quickly as possible, as this is the only action that gives us reward (= negative cost). If we start in state $(1,1)$, then the best control action is to use "1" and then again "1", leading us first to $(1,2)$ with a probability of 90% and then further to the desired state $(1,3)$ with again 90%, so alltogether we reach $(2,3)$ with a probability of 81%. However, we may also reach the states $(2,1)$, $(2,2)$ or $(1,1)$ with a total probability of 19%. If we end up in

$(2,1)$ or $(2,2)$, then we need to make sure that we go up again in one of the next steps, while if we end up in $(1,1)$ then we should keep on going right. The next control actions should thus depend not only on time but also on the state we reached. In order to derive a dynamic programming principle (which we will do in the next section), we actually need even more flexibility in the choice of u. We allow that the value of u depends on time and on the whole history of states $X(0),\dots,X(k)$. At each time k we thus use controls from the set

$$\mathcal{U}_k := \{u_k : X^{k+1} \to U\}$$

and the overall set of control functions is defined as the set of infinite sequences

$$\mathcal{P} := \{u = (u_0, u_1, u_2, \dots) \,|\, u_k \in \mathcal{U}_k\}.$$

We refer to the elements of $\mathcal{P}$ as *control processes*. For each $u \in \mathcal{P}$ we then consider the random solutions $X(k) = X_u(k, x_0)$, $k \in \mathbb{N}$ satisfying

$$X(0) = x_0 \quad \text{and} \quad X(k+1) = g(X(k), u_k(X(0:k))),$$

where we assume that the random variables $g(X(k), u_k(X(0:k)))$ are stochastically independent for different k, i.e., that the values $X(0),\dots,X(k)$, which are known at time k, do not give any stochastic information about $X(k+1)$. Using the definition of p it moreover follows that

$$P(X(k+1) = x' \,|\, X(j) = x_j, j = 0,\dots,k)) = p(x_k, u_k(x_0,\dots,x_k), x').$$

We note that $X(1)$ only depends on $u_0(x_0)$, but not on $u_k(x_0,\dots,x_k)$ for $k \geq 1$. This means that we only need to specify $u_0(x_0) \in U$ in order define $X(1) = g(x_0, u_0(x_0))$. We also use the short notation

$$X(0:k) \quad \text{or} \quad X_u(0:k, x_0)$$

for the arguments $(X(0),\dots,X(k))$ or $(X_u(0,x_0),\dots,X_u(k,x_0))$ of u_k.

The optimisation criterion then takes the expected value of the cost along all these trajectories, i.e.,

$$J(x_0, u) = E\left(\sum_{k=0}^{\infty} \gamma^k \ell(X_u(k,x_0), u_k(X_u(0:k,x_0)))\right) \tag{5.1}$$

This non-deterministic optimal control problem is also called a *Markov Decision Problem* (MDP).

5.2 Dynamic programming

In this section we derive counterparts to some of the results from Chapter 3. To this end, we note that Definition 3.1 can be directly applied also in the non-deterministic setting by using control processes instead of control sequences everywhere. The following theorem then provides the counterpart of Theorem 3.2.

Theorem 5.2 Consider the optimal control problem of minimising (5.1) with respect to all control processes. Then for all $K \in \mathbb{N}$ and all $x_0 \in X$ the optimal value function satisfies

$$V(x_0) = \inf_{u \in \mathcal{P}} E\left\{ \sum_{k=0}^{K-1} \gamma^k \ell(X_u(k, x_0), u_k(X_u(0:k, x_0))) + \gamma^K V(X_u(K, x_0)) \right\}. \tag{5.2}$$

If, in addition, an optimal control process $u^\star \in \mathcal{P}$ exists, then the equation

$$V(x_0) = E\left\{ \sum_{k=0}^{K-1} \gamma^t \ell(X_{u^\star}(k, x_0), u_k(X_{u^\star}(0:k, x_0))) + \gamma^K V(X_{u^\star}(K, x_0)) \right\}. \tag{5.3}$$

holds and the "inf" in (5.2) is a "min".

Proof: In the following proof we use properties of conditional expectations that may not be common knowledge. We refer to [Stochastische Dynamische Optimierung] for an explanation.

We first prove (5.2) for $K = 1$. Throughout the proof we abbreviate $X(k) = X_u(k, x_0)$ and $u_0 = u_0(x_0)$.

"$\geq$": Let $x_0 \in \mathbb{R}^n$ and $u \in \mathcal{P}$ arbitrary. Then, for $X'(k) = X_{u'}(k, x_0')$ with $u_k'(x_0', \ldots, x_k') = u_{k+1}(x_0, x_0', \ldots, x_k')$ we get

$$\begin{aligned}
J(x_0, u) &= E\left\{ \sum_{k=0}^{\infty} \gamma^k \ell(X(k), u_k(X(0:k))) \right\} \\
&= E\left\{ \ell(x_0, u_0) + \sum_{k=1}^{\infty} \gamma^k \ell(X(k), u_k(X(0:k))) \right\} \\
&= \ell(x_0, u_0) + \gamma E\left\{ \sum_{k=0}^{\infty} \gamma^k \ell(X(k+1), u_{k+1}(X(0:k+1))) \right\} \\
&= \ell(x_0, u_0) + \gamma E\left\{ E\left(\sum_{k=0}^{\infty} \gamma^k \ell(X'(k), u'(k, X'(0:k))) \,\middle|\, X'(0) = X(1) \right) \right\} \\
&= \ell(x_0, u_0) + \gamma E\left\{ J(X(1), u') \right\} \\
&\geq E\left\{ \ell(x_0, u_0) + \gamma V(X(1)) \right\} \\
&\geq \inf_{u \in \mathcal{P}} E\left\{ \ell(x_0, u) + \gamma V(X(1)) \right\}.
\end{aligned}$$

Since this inequality holds for all $u \in \mathcal{P}$, it also holds for

$$V(x_0) = \inf_{u \in \mathcal{P}} J(x_0, u),$$

which implies "$\geq$".

"$\leq$": Let $\varepsilon > 0$. For any $x \in X$ we choose a control process $\bar{u}^x \in \mathcal{P}$ with

$$J(x, \bar{u}^x) \leq V(x) + \varepsilon$$

and abbreviate $\overline{X}(k) = X_{\bar{u}^{x_0}}(k, x_0)$. Moreover, for each $x \in X$ we choose a control value $\hat{u}_x \in U$ with

$$E\{\ell(x, \hat{u}_x) + \gamma V(g(x, \hat{u}_x))\} \leq \inf_{u \in U} E\{\ell(x, u) + \gamma V(g(x, u))\} + \varepsilon,$$

a define the control process $\tilde{u} \in \mathcal{P}$ as[1]

$$\tilde{u}_k(x_0, \dots, x_k) := \begin{cases} \hat{u}_{x_0}, & k = 0 \\ \bar{u}_{k-1}^{x_1}(x_1, \dots, x_k), & k \geq 1. \end{cases}$$

The corresponding solution is denoted by $\widetilde{X}(k) = X_{\tilde{u}}(k, x_0)$. Then $\overline{X}(k) = \widetilde{X}(k+1)$ holds if $\overline{X}(0) = \widetilde{X}(1)$. With these definitions we obtain

$$\begin{aligned}
V(x_0) &= \inf_{u \in \mathcal{P}} J(x_0, u) \\
&= \inf_{u \in \mathcal{P}} E\left\{ \sum_{k=0}^{\infty} \gamma^k \ell(X(k), u_k(X(0:k))) \right\} \\
&= \inf_{u \in \mathcal{P}} E\left\{ \ell(x_0, u_0) + \sum_{k=1}^{\infty} \gamma^k \ell(X(k), u_k(X(0:k))) \right\} \\
&\leq E\left\{ \ell(x_0, \tilde{u}_0(x_0)) + \gamma \sum_{k=0}^{\infty} \gamma^k \ell(\widetilde{X}(k+1), \tilde{u}_{k+1}(\widetilde{X}(0:k+1))) \right\} \\
&= \ell(x_0, \hat{u}_{x_0}) + \gamma E\left\{ E\left(\sum_{k=0}^{\infty} \gamma^k \ell(\overline{X}(k), \bar{u}_k^{\widetilde{X}(1)}(\overline{X}(0:k))) \middle| \overline{X}(0) = \widetilde{X}(1) \right) \right\} \\
&\leq \sup_{u \in U} E\left\{ \ell(x_0, u) + \gamma V(X(1)) \right\} + 2\varepsilon
\end{aligned}$$

where in the last step we used the properties of $\bar{u}^x$ and $\hat{u}_x$. Since $\varepsilon > 0$ was arbitrary, it follows that

$$V_\infty(x_0) \leq \inf_{u \in U} E\left\{ \ell(x_0, u) + \gamma V(X(1)) \right\},$$

i.e., the desired inequality.

For $K \geq 2$, equation (5.2) now follows by induction. For $K = 1$ there is nothing to show. For $K \to K+1$ we obtain

$$\begin{aligned}
V(x_0) &= \inf_{u \in \mathcal{P}} E\left\{ \sum_{k=0}^{K-1} \gamma^k \ell(X(t), u_k(X(0:k)) + \gamma^K V(X(K)) \right\} \\
&= \inf_{u \in \mathcal{P}} E\Bigg\{ \sum_{k=0}^{K-1} \gamma^k \ell(X(k), u_k(X(0:k))) \\
&\qquad + \gamma^K \inf_{\tilde{u} \in \mathcal{P}} E\{ \ell(X(K), \tilde{u}_0(X(K))) + \gamma V(\widetilde{X}(1)) \,|\, \widetilde{X}(0) = X(K) \} \Bigg\} \\
&= \inf_{\mathbf{u} \in \mathcal{P}} E\Bigg\{ \sum_{k=0}^{K-1} \gamma^k \ell(X(k), u_k(X(0:k))) \\
&\qquad + \gamma^K (\ell(X(K), u_K(X(0:K))) + \gamma V(X(K+1))) \Bigg\} \\
&= \inf_{u \in \mathcal{P}} E\left\{ \sum_{k=0}^{K} \gamma^t \ell(X(k), u_k(X(0:k))) + \beta^{K+1} V(X(K+1)) \right\},
\end{aligned}$$

[1]The definition of $\tilde{u}$ is the reason for allowing the control processes to depend on the whole history of states $x_0, \dots, x_k$. This is because even if each $\bar{u}_k^x$ only depended on x_k, the newly defined $\tilde{u}_k$ depends on x_1 for all $k \geq 1$, because it uses $u_{k-1}^{x_1}$.

where we used that we can set $u_K(X(0:K)) := \tilde{u}_0(X(K))$ in the second last step.

Equation (5.3) then follows as in proof of Theorem 3.2. □

As in the deterministic setting, in the case $K = 1$ the dynamic programming principle yields the Bellman equation

$$V(x_0) = \inf_{u \in U} E\left\{\ell(x_0, u) + \gamma V(g(x_0, u))\right\}. \tag{5.4}$$

We again define the quantity

$$Q(x, u) := E\{\ell(x, u) + \gamma V(g(x, u))\}. \tag{5.5}$$

The following theorem is the counterpart to Theorem 3.6. In the non-deterministic setting we can, however, only prove it under Assumption 2.1(ii). This is because we do not have a non-deterministic counterpart of Lemma 2.2.

Theorem 5.3 Consider the optimal control problem of minimising (5.1) with $x_0 \in X$ and let Assumption 2.1(ii) hold. Consider a feedback law $\pi^\star : X \to U$ satisfying

$$\pi^\star(x) \in \operatorname*{argmin}_{u \in U} E\left\{\ell(x, u) + \gamma V(g(x, u))\right\} = \operatorname*{argmin}_{u \in U} Q(x, u) \tag{5.6}$$

for all $x \in X$. Then $\pi^\star$ is an optimal strategy in the sense of Definition 3.1(iii).

Proof: We abbreviate $\widehat{X}(k) = X_{\pi^\star}(k, x_0)$. Then we need to show that

$$V(x_0) = J(x_0, \pi^\star).$$

Using (5.6) and (5.4) with $x_0 = \widehat{X}(k)$ we get

$$\gamma^k E\{V(\widehat{X}(k))\} = \gamma^k E\{\ell(\widehat{X}(k), \pi^\star(\widehat{X}(k))) + \gamma^{k+1} V(\widehat{X}(k+1))\}$$

for $k = 0, 1, \ldots$. Summing these equalities for $k = 0, \ldots, K-1$ for arbitrary $K \in \mathbb{N}$ and eliminating the identical terms $\gamma^k V(\hat{x}(k))$, $k = 1, \ldots, K-1$ on the left and on the right we obtain

$$V(x_0) = E\left\{\sum_{k=0}^{K-1} \gamma^k \ell(\widehat{X}(k), \pi^\star(\widehat{X}(k))) + \gamma^K V(\widehat{X}(K))\right\}.$$

Now Assumption 2.1(ii) implies that V is bounded, which yields $\lim_{K\to\infty} \gamma^K E\{V(\widehat{X}(K))\} = 0$ since $\gamma^K \to 0$ as $K \to \infty$. Hence we obtain

$$V(x_0) = E\left\{\lim_{K\to\infty} \sum_{k=0}^{K-1} \gamma^k \ell(\widehat{X}(k), \pi^\star(\widehat{X}(k))) + \gamma^K V(\widehat{X}(K))\right\} = E\left\{\sum_{k=0}^{\infty} \ell(\hat{x}(k), \pi^\star(\widehat{X}(k)))\right\}.$$

Note that here we can take the limit under the expectation because the series is absolutely convergent, as ℓ is bounded and $\gamma < 1$. □

Theorem 5.3 has a surprising consequence: while in the minimisation problem we took the minimum over all control processes, which may depend on time and on the whole history of the states $X(k)$, the optimal control can be expressed via strategies, which only depend on the current state and neither on time nor on the past states. It may thus seem unnecessary to introduce the complicate definition of control processes. However, without this detour it would not have been possible to prove that optimal controls can always expressed in form of strategies.

5.3 Q-Learning

The Q-Learning algorithm for the non-deterministic setting is quite similar to the algorithm in the deterministic setting, with two major changes.

First, the value x' obtained in Step (1) is now non-deterministic, i.e., for one and the same pair (x,u) different x' may occur. In case the values x' are obtained by observing a real process, then this does not require any changes in the algorithm. However, in case the evolution of the system is simulated, instead of evaluating $x' = g(x,u)$, we must perform a stochastic simulation based on the information from $p(x,u,x')$ in order to obtain x'. In the finite state case we discuss here, this can be done as follows.

Let $(x,u) \in X \times U$ be given. Let $x'_1,\dots,x'_r$ be the states for which $p(x,u,x'_j) \neq 0$. Define inductively

$$q_0 := 0, \quad q_j := q_{j-1} + p(x,u,x'_j) \text{ for } j = 1,\dots,r. \tag{5.7}$$

Generate a uniformly distributed random number $z \in [0,1]$ using a random number generator, let j be the smallest index with $z \in [q_{j-1}, q_j]$ and set $x' = x'_j$.

It should be noted that if the probabilities p are known, then there are more efficient ways to modify Q-Learning than the one discussed in the following using the simulation (5.7). The algorithm we present below is more suited for the case that we have a real process or a simulation tool for evaluating g but no explicit knowledge of p. Algorithm 5.8 presents a variant of Q-Learning that takes advantage of the knowledge of p.

The second change concerns the update of $\widetilde{Q}$ in Step (2). In order to motivate that this rule needs to be changed, consider the following very simple example.

Example 5.4 We consider a non-deterministic problem with exactly two states $X = \{x_1, x_2\}$ and only one control $U = \{u\}$. Regardless of in which state the system is, the (only) control u_1 always brings the system to x_1 with probability 0.5 and to x_2 with probability 0.5. This means that for both $x = x_1$ and $x = x_2$ the map p is defined as

$$p(x,u,x') = \begin{cases} 0.5, & \text{if } x' = x \\ 0.5, & \text{if } x' = x. \end{cases}$$

The cost is defined as $\ell(x_1,u) = 0$ and $\ell(x_2,u) = 1$.

It is easily seen that from time $k = 1$ on, the system is in state x_1 and x_2 with the same probability of 0.5. If we use, e.g., the discount factor $\gamma = 0.5$, this leads to the average cost

$$Q(x_1,u) = E\left\{\sum_{k=0}^{\infty} \gamma^k \ell(x(k),u(k))\right\} = 0 + E\left\{\sum_{k=1}^{\infty} \gamma^k \ell(x(k),u(k))\right\} = \sum_{k=1}^{\infty} 0.5^k 0.5 = 0.5$$

and

$$Q(x_2,u) = E\left\{\sum_{k=0}^{\infty} \gamma^k \ell(x(k),u(k))\right\} = 1 + E\left\{\sum_{k=1}^{\infty} \gamma^k \ell(x(k),u(k))\right\} = 1 + \sum_{k=1}^{\infty} 0.5^k 0.5 = 1.5.$$

Now assume that the random generator draws a sequence in which every once in a while $x' = x_1$ occurs twice in a row for two consecutive iterations j and $j+1$ (which is very likely to happen). Then, each time $x' = x_1$ appears for the second time, the update rule says that

$$\widetilde{Q}_{j+1}(x_1, u) = \ell(x_1, u) + \gamma \widetilde{Q}_j(x_1, u) = 0.5 \widetilde{Q}_j(x_1, u).$$

This means that $\widetilde{Q}_j(x_1, u)$ never converges, because it keeps changing its value (unless it converges to 0, but this would not be the correct limit). The same happens for $\widetilde{Q}_j(x_2, u)$. □

The behaviour in this example, which is typical for most other examples, shows that due to the random nature of the x' the $\widetilde{Q}$-values do not converge but rather jump randomly between different values if we use the update rule of the deterministic Q-Learning algorithm. For this reason, the update rule must be modified as follows.

Algorithm 5.5 (non-deterministic Q-learning)

(0) Set $\widetilde{Q} :\equiv 0$, fix a real sequence $(\alpha_j)_{j\in\mathbb{N}}$ with $\alpha_j \in [0,1]$, pick a state $x \in X$, set $j := 0$

(1) Select $u \in U$, evaluate/simulate $x' = g(x,u) \in X$ and evaluate $\ell(x,u)$

(2) Set $\widetilde{Q}(x,u) := (1-\alpha_j)\widetilde{Q}(x,u) + \alpha_j\big[\ell(x,u) + \gamma \min_{u'\in U} \widetilde{Q}(x',u')\big]$

(3) Set $x := x'$ or select a new $x \in X$, set $j := j+1$ and go to (1)

□

The new feature of the update rule is that the new value of $\widetilde{Q}$ is now a convex combination of its old value and the update value $\ell(x,u)+\gamma \min_{u'\in U} \widetilde{Q}(x',u')$. We recover the old update rule if we choose $\alpha_j = 1$ for all j.

5.4 Convergence analysis

The trick is now to let α_j tend to 0, such that the random jumps in the $\widetilde{Q}$-values become smaller and smaller as the iterations progress, but slowly enough such that the correct value can be learned before the α_j become too small. The following theorem shows how this sequence must be chosen in order to achieve this goal. We note that, e.g., $\alpha_j = 1/j$ satifies (5.8), while $\alpha_j = 1/j^2$ converges "too fast" and $\alpha_j = 1/\sqrt{j}$ converges "too slow".

Theorem 5.6 Consider the discrete-time problem of minimising (5.1) with finite X and U and let Assumption 2.1(ii) hold. Denote by $\widetilde{Q}_{j+1}$ the $\widetilde{Q}$-function after Step (2) of Algorithm 4.1, with j being the iteration counter in the algorithm. Assume that each pair $(x,u) \in X \times U$ appears infinitely often in Step (1) of the algorithm and let $j(i,x,u) \geq 1$ be the iteration number in which the pair (x,u) appears for the i-th time in Step (1). Assume that for each $(x,u) \in X \times U$ the sequence $(\alpha_j)_{j\in\mathbb{N}}$ satisfies

$$\lim_{j\to\infty} \alpha_j = 0, \quad \sum_{i=1}^{\infty} \alpha_{j(i,x,u)} = \infty, \quad \text{and} \quad \sum_{i=1}^{\infty} \alpha^2_{j(i,x,u)} < \infty. \tag{5.8}$$

Then

$$\lim_{j\to\infty} \widetilde{Q}_j(x,u) = Q(x,u)$$

for all $x \in X$, $u \in U$ with probability 1.

The proof of this theorem can be obtained by studying abstract iterations of the form

$$r_{j+1}(z) := (1-\alpha_j) r_j(z) + \alpha_j (\Phi(r_j)(z) + w_j(z)), \quad j = 0,1,2,\ldots \tag{5.9}$$

with the following ingredients of (5.9):

- For each j the term r_j is a map from Z to $\mathbb{R}$, where Z is a finite set. If we number the elements of Z as $z^1,\ldots,z^S$, then r_j can be identified with the vector $(r_j(z^1),\ldots,r_j(z^S))^T \in \mathbb{R}^S$.
- The $w_j(z)$ are random variables (possibly dependent on the current and past terms in (5.9)) with $E(w_j(z)) = 0$ and $E(w_j^2(z)) \le A + B\|r_j\|^2$ for constants A, B. Here the expected values are understood as conditioned on all information that is available in the j-th iteration of (5.9).
- The map $\Phi : \mathbb{R}^S \to \mathbb{R}^S$ is a *contraction*[2] for the ∞-norm, i.e. there is a constant $\beta \in [0,1)$ such that
$$\|\Phi(r_1) - \Phi(r_2)\|_\infty \le \beta \|r_1 - r_2\|_\infty$$
holds for all $r \in \mathbb{R}^S$.

From Banach's fixed point theorem one can then conclude that Φ has a unique fixed point $r^* \in \mathbb{R}^S$, i.e., a unique $r^* \in \mathbb{R}^S$ with $\Phi(r^*) = r^*$.

Proposition 5.7 Under the assumptions just listed and (5.8), if each z appears infinitely often in the iteration (5.9), then for each r_0 the iteration (5.9) converges to r^* with probability 1. □

A complete proof of this proposition can be found as Proposition 4.4 in [2]. We will not reproduce this proof here, but at least motivate why the condition (5.8) is needed.

To this end, let $S = 1$ and $\Phi(r) = 0$, which is clearly a contraction with $r^* = 0$. The result of the iteration can then be written explicitly as

$$r_{j+1} = \prod_{l=0}^{j} (1-\alpha_l) r_0 + \sum_{k=0}^{j} \alpha_k w_k \prod_{l=k+1}^{j} (1-\alpha_l).$$

Now the limit of this iteration should not depend on r_0, because this would mean that convergence would depend on the choice of the initial value. This means that

$$\lim_{j\to\infty} \prod_{l=0}^{j} (1-\alpha_l) = 0$$

[2]The proof of Proposition 5.7 in the mentioned reference only requires Φ to be a pseudo contraction in a more general norm, but the ∞-norm contraction condition given here is sufficient for this.

must hold. This is equivalent to

$$\lim_{j\to\infty}\sum_{l=0}^{j}\log(1-\alpha_l)=\lim_{j\to\infty}\log\left(\prod_{l=0}^{j}(1-\alpha_l)\right)=-\infty.$$

From the Taylor series for the logarithm it follows that $\log(1-\alpha_l)\le-\alpha_l$, so the first condition in (5.8) ensures this property.

Now consider the same setting with $r_0=0$. The explicit result of the iteration then reads

$$r_{j+1}=\sum_{k=0}^{j}\alpha_k w_k\prod_{l=k+1}^{j}(1-\alpha_l).$$

This implies that the expected value satisfies $E(r_{j+1})=0$. A necessary condition for $r_{j+1}\to 0$ with probability 1 is that the variance $E(r_{j+1}^2)$ also tends to 0. This is given by

$$E(r_{j+1}^2)=E\left(\left(\sum_{k=0}^{j}\alpha_k w_k\prod_{l=k+1}^{j}(1-\alpha_j)\right)^2\right).$$

Assuming that the w_k are identically distributed and stochastically independent, we obtain $E(w_kw_l)=E(w_k)E(w_l)=0$ for $k\neq l$ and $E(w_k^2)=E(w_0^2)$ for all $k\ge 0$. Thus, the expression simplifies to

$$E(r_{j+1}^2)=E\left(\sum_{k=0}^{j}\alpha_k^2w_k^2\prod_{l=k+1}^{j}(1-\alpha_l)^2\right)=E(w_0^2)\sum_{k=0}^{j}\alpha_k^2\left(\prod_{l=k+1}^{j}(1-\alpha_l)\right)^2.$$

Now, since $\prod_{l=k+1}^{j}(1-\alpha_l)\to 0$ as $j\to\infty$, one sees that the second condition in (5.8), i.e., $\lim_{j\to\infty}\sum_{k=0}^{j}\alpha_k^2<\infty$, ensures that the variance of r_{k+1} convergence to 0. Of course, these are only special cases, but they illustrate why the assumptions on the α_j are reasonable.

Proof of Theorem 5.6: We define $Z=X\times U$, $z=(x,u)$, $r(z):=\widetilde{Q}(x,u)$, and $\Phi:\mathbb{R}^S\to\mathbb{R}^S$ as

$$\Phi(\widetilde{Q})(z)=\sum_{x'\in X}p(x,u,x')\left(\ell(x,u)+\gamma\min_{u'\in U}\widetilde{Q}(x',u')\right).$$

Denoting by x_j the value x' from the j-th iteration of the algorithm, we can write the Q-Learning iteration as

$$\widetilde{Q}_{j+1}(x,u)=(1-\alpha_j)\widetilde{Q}_j(x,u)+\alpha_j(\Phi(\widetilde{Q})(x,u)+w_j(x,u)),$$

with

$$\begin{aligned}w_j(x,u) &= \ell(x,u)+\gamma\min_{u'\in U}\widetilde{Q}_j(x_j,u')-\sum_{x'\in X}p(x,u,x')\left(\ell(x,u)+\gamma\min_{u'\in U}\widetilde{Q}_j(x',u')\right)\\ &= \gamma\left(\min_{u'\in U}\widetilde{Q}_j(x_j,u')-\sum_{x'\in X}p(x,u,x')\min_{u'\in U}\widetilde{Q}_j(x',u')\right).\end{aligned}$$

Since x_j is distributed according to the probability p, it is easily checked that the expectation of w_j satisfies $E(w_j(x))=0$. Moreover,

$$E(w_j(x)^2)\le K(1+\max_{x\in X,u\in U}\widetilde{Q}_j(x,u)^2)$$

for a suitable constant K. The fact that Φ is a contraction follows immediately from the fact that (using the hint from Exercise 1 in Sheet 2)

$$\begin{aligned}|\Phi(\widetilde{Q}_1)(x,u) - \Phi(\widetilde{Q}_2)(x,u)| &= \sum_{x' \in X} p(x,u,x')\gamma |\min_{u' \in U} \widetilde{Q}_1(x',u') - \gamma \min_{u' \in U} \widetilde{Q}_2(x',u')| \\ &\leq \sum_{x' \in X} p(x,u,x')\gamma \max_{u' \in U} |\widetilde{Q}_1(x',u') - \widetilde{Q}_2(x',u')| \\ &\leq \gamma \|\widetilde{Q}_1 - \widetilde{Q}_2\|_\infty.\end{aligned}$$

Hence, the assertion follows from Proposition 5.7. □

The choices of x and u in the algorithm discussed in Section 4.3 can be adapted to the nondeterministic setting. Again for brevity we will not discuss details.

5.5 The case of known transition probabilities

We briefly state how one can improve Algorithm 5.5 if the probabilities p are known.

Algorithm 5.8 (non-deterministic Q-learning with known p)

(0) Set $\widetilde{Q} :\equiv 0$, fix a real sequence $(\alpha_j)_{j \in \mathbb{N}}$ with $\alpha_j \in [0,1]$, pick a state $x \in X$, set $j := 0$

(1) Select $u \in U$ and evaluate $\ell(x,u)$

(2) Set $\widetilde{Q}(x,u) := \ell(x,u) + \gamma \min_{u' \in U} \sum_{x' \in X} p(x,u,x')\widetilde{Q}(x',u')$

(3) Select a new $x \in X$, set $j := j+1$ and go to (1)

□

Instead of simulating x', this algorithm computes the exact expected value

$$E(\widetilde{Q}(x',u')) = \sum_{x' \in X} p(x,u,x')\widetilde{Q}(x',u')$$

in each step. Rather than "collecting" the stochastic information over many iterations, it thus uses the exact information in each step. For this reason, vanishing step sizes are not needed in this variant and convergence is typically much faster than for Algorithm 5.5.

In the RL-literature, the Q-Learning Algorithm 5.5 is called a *model-free* algorithm while Algorithm 5.8 is called a *model-based* algorithm. We note that model-based algorithms can also be also used when p is not known a priori. In this case, another learning scheme computes the probabilities p from the evaluations of g during Q-Learning, in the simplest case by using the empirical distribution, i.e., by counting the observed transitions and dividing by the number of overall transitions. Depending on the problem, Algorithm 5.8 with such a "learned" p can be faster and more reliable than Algorithm 5.5.

Chapter 6

Deep Neural Networks

July 21, 2021

The Q-Learning algorithms proposed so far were formulated under the assumption that the state and action space are finite. In practical problems this is most often not the case.

If X and U are compact subsets of $\mathbb{R}^d$ and $\mathbb{R}^m$ (which is a realistic assumption in many applications), a standard way to overcome this problem is to replace X and U with finite approximations by discretising the state and action space. For instance, for the unit cube

$$X = [0,1]^d \subset \mathbb{R}^d$$

one can select a step size $h = 1/J$, $J \in \mathbb{N}$, and use the finite set

$$X_h := \{(hq_1, \dots, hq_d)^T \,|\, q = (q_1, \dots, q_d)^T \in \{0, \dots, J\}^d\}.$$

Then, of course, the dynamics of the system on X must also be discretised in order to obtain a dynamics on X_h. This can be done by different techniques that we will not discuss in detail here; we just mention that quantization, which was briefly explained at the beginning of the last chapter, is one of these methods.

Regardless of how the dynamics is converted, this procedure leads to the situation that the exact function $Q : X \times U \to \mathbb{R}$ is approximated by a function $Q_h : X_h \times U_h \to \mathbb{R}$ for all $x \in X_h \times U_h$. In order to obtain an aproximation that is defined on $X \times U$ interpolation can be used. Piecewise constant or piecewise linear interpolation are the simplest choices, but much more sophisticated methods are possible.

Now, in order to be able to approximate Q by Q_h with a certain accuracy, the step size $h > 0$ must be sufficiently small, meaning that the value J must be sufficiently large. Then, however, one immediately sees that the number of elements in X_h grows rapidly — more precisely exponentially — when the dimension n grows. If in the above example we use $h = 1/9$, i.e., $J + 1 = 10$ states per coordinate direction, then for a two dimensional problem we need 100 states in X_h (which can be easily handled in a Q-Learning algorithm), but in a ten dimensional problem we need $10^{10} = 10$ billion (10 Milliarden) states, which already needs a very powerful computer, even though 10 states per coordinate direction is still a quite coarse discretization.

This phenomenon is known as the curse of dimensionality and leads to the fact that this way of discretising the state space is not suitable for high-dimensional problems. Deep

Neural Networks (DNNs) can bring a remedy here and we will discuss in this chapter under which conditions this is provably true.

Generally, a DNN of the type we consider in this lecture can be seen as a function from a set $Y \subset \mathbb{R}^d$ to $\mathbb{R}$, which in addition depends on a vector of parameters $\theta = (\theta_1, \dots, \theta_P)^T \in \mathbb{R}^P$. This means, a DNN represents a function

$$W : Y \times \mathbb{R}^P \to \mathbb{R}, \quad \text{or, if we fix a } \theta \in \mathbb{R}^P, \quad W(\cdot;\theta) : Y \to \mathbb{R}.$$

Now, given a function $z : Y \to \mathbb{R}$, the goal is to find a parameter vector $\theta^* \in \mathbb{R}^P$ such that

$$\|W(\cdot;\theta^*) - z(\cdot)\|$$

is small in some norm $\|\cdot\|$. In Q-Learning, z would typically be the function Q from (3.5), in which case $d = n + m$ and W would be an approximation of Q. Alternatively, one can define z to be the optimal value function V. Then $d = n$ and $\ell(x,u) + \gamma W(g(x,u))$ would be an approximation of Q. This approach has the advantage that the function to be approximated depends on a lower dimensional argument, but the disadvantage that g must be known and easy to evaluate in order to evaluate the approximation of Q.

Now, three questions need to be answered:

- What exactly is W and the underlying neural network?
- When is it possible to find $\theta^* \in \mathbb{R}^P$ such that $\|W(\cdot,\theta^*) - z(\cdot)\|$ becomes small?
- How do we compute this θ^*.

These will be clarified in the following four sections.

6.1 Definition of DNNs

In this section we describe the type of neural networks that we use in this lecture. A deep neural network is a computational architecture that has several inputs, which are processed through $\ell \geq 1$ hidden layers of neurons. The values in the neurons of the layer with the largest ℓ are used in order to compute the output of the network. In this lecture, we will only consider feedforward networks, in which the input is processed consecutively through the layers 1, 2, ..., ℓ. For our purpose of representing Q or V we will use networks with the input vector $x = (x_1, \dots, x_d)^T \in \mathbb{R}^d$ (where this x may contain x and u in case Q is represented) and a scalar output $W(x;\theta) \in \mathbb{R}$. Here, the vector $\theta \in \mathbb{R}^P$ represents the free parameters in the network that need to be tuned (or "learned") in order to obtain the desired output. Figure 6.1 shows generic neural networks with one and two hidden layers.

Here, the lowest layer is the input layer, followed by one or two hidden layers numbered with ℓ, and the output layer. The number $\ell_{\max}$ determines the number of hidden layers, here $\ell_{\max} = 1$ or 2. Each hidden layer consists of N_ℓ neurons and the overall number of neurons in the hidden layers is denoted by $N = \sum_{\ell=1}^{\ell_{\max}} N_\ell$. The neurons are indexed using

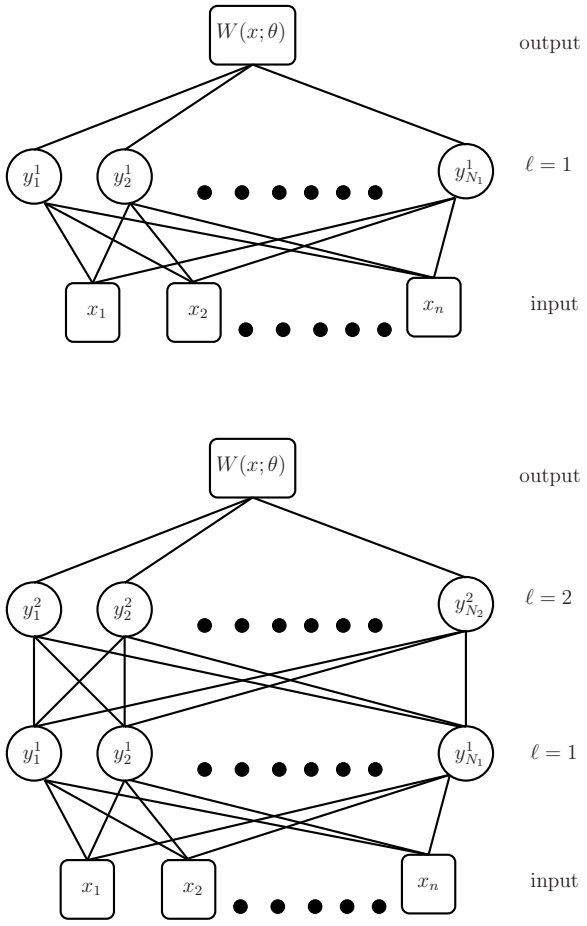

Figure 6.1: Neural network with 1 and 2 hidden layers

the number of their layer ℓ and their position in the layer k. Every neuron has a scalar value $y_k^\ell \in \mathbb{R}$ and for each layer these values are collected in the vector $y^\ell = (y_1^\ell, \dots, y_{N_\ell}^\ell)^T \in \mathbb{R}^{N_\ell}$. The values of the neurons at the lowest level are given by the inputs, i.e., $y^0 = x \in \mathbb{R}^d$. The values of the neurons in the hidden layers are determined by the formula

$$y_k^\ell = \sigma^\ell(w_k^\ell \cdot y^{\ell-1} + b_k^\ell),$$

for $k = 1, \dots N_\ell$. Here $x \cdot y$ denotes the Euclidean scalar product between two vectors $x, y \in \mathbb{R}^n$, $\sigma^\ell : \mathbb{R} \to \mathbb{R}$ is a so called activation function and $w_k^\ell \in \mathbb{R}^{N_{\ell-1}}$, $a_k^\ell, b_k^\ell \in \mathbb{R}$ are the

parameters of the layer. Popular activation functions include

$$\begin{aligned}
\sigma(r) &= r && \text{(linear)}\\
\sigma(r) &= \frac{1}{1+e^{-x}} && \text{(sigmoid)}\\
\sigma(r) &= \frac{2}{1+e^{-2x}} - 1 && \text{(hyperbolic tangent)}\\
\sigma(r) &= \max\{0, r\} && \text{(rectified linear unit, ReLU)}\\
\sigma(r) &= \ln(e^r + 1) && \text{(softplus).}
\end{aligned}$$

Among these functions, ReLU activation functions have become particularly popular for time-critical applications, because the evaluation of the function $x \mapsto W(x;\theta^*)$ can be implemented very efficiently. In contrast, for analytic considerations it is sometimes desirable that $x \mapsto W(x;\theta^*)$ is differentiable, which excludes the non-smooth ReLU function.

In the output layer, the values from the topmost hidden layer $\ell = \ell_{\max}$ are affine-linearly combined to deliver the output, i.e.,

$$W(x;\theta) = \sum_{k=1}^{N_{\ell_{\max}}} a_k y_k^{\ell_{\max}} + c = \sum_{k=1}^{N_{\ell_{\max}}} a_k \sigma^{\ell_{\max}}(w_k^{\ell_{\max}} \cdot y^{\ell_{\max}-1} + b_k^{\ell_{\max}}) + c. \tag{6.1}$$

The vector θ collects all parameters a_k, c, w_k^ℓ, b_k^ℓ of the network.

In case of one hidden layer, in which $\ell_{\max} = 1$ and thus $y^{\ell_{\max}-1} = y^0 = y$, we obtain the closed-form expression

$$W(x;\theta) = \sum_{k=1}^{N_1} a_k \sigma^1(w_k^1 \cdot x + b_k^1) + c.$$

For two hidden layers, the closed-form expression reads

$$W(x;\theta) = \sum_{k=1}^{N_2} a_k \sigma^2\left(w_k^2 \cdot \begin{pmatrix} \sigma^1(w_1^1 \cdot x + b_2^1) \\ \vdots \\ \sigma^1(w_{N_1}^1 \cdot x + b_{N_1}^2) \end{pmatrix} + b_k^1 \right) + c.$$

6.2 The universal approximation theorem

The universal approximation theorem states that a neural network with one hidden layer can approximate all smooth functions arbitrarily well. In its qualitative version, going back to [3, 6], it states that the set of functions that can be approximated by neural networks with one hidden layer is dense in the set of continuous functions. In Theorem 6.1, below, we state a quantitative version, given as Theorem 1 in [10], which is a reformulation of Theorem 2.1 in [9].

For its formulation consider compact sets $K_d \subset \mathbb{R}^d$ for which there exists a $C > 0$, $c_d \in \mathbb{R}^d$ satisfying

$$K_d \subseteq c_d + [-C, C]^d \quad \text{for all } d \in \mathbb{N}.$$

Note that C is assumed to be independent while c_d may depend on d. On these sets we want to perform our computation. For a continuous function $z : K_n \to \mathbb{R}$ we define the infinity-norm over K_d as

$$\|z\|_{\infty,K_d} := \max_{x\in K_d} |z(x)|.$$

We then define the set of functions

$$\mathcal{W}_m^d := \left\{ z \in C^m(K_d, \mathbb{R}) \;\middle|\; \sum_{0\le|\alpha|\le m} \|D_\alpha z\|_{\infty,K_d} \le 1 \right\}$$

where $C^m(K_d, \mathbb{R})$ denoted the functions from K_d to $\mathbb{R}$ that are m-times continuously differentiable, α are multiindices of length $|\alpha|$ with entries $\alpha_i \in \{1, \dots, d\}$, $i = 1, \dots, |\alpha|$ and

$$D_\alpha z = \frac{\partial^{|\alpha|} z}{\partial x_{\alpha_1} \dots \partial x_{\alpha_{|\alpha|}}}$$

denotes the m-th directional derivative with respect to α, with $D_\alpha z = z$ if $|\alpha| = 0$.

Theorem 6.1 Let $\sigma : \mathbb{R} \to \mathbb{R}$ be infinitely differentiable and not a polynomial[1]. Then, for any $\varepsilon > 0$, a neural network with one hidden layer provides an approximation

$$\inf_{\theta\in\mathbb{R}^P} \|W(x;\theta) - z(x)\|_{\infty,K_d} \le \varepsilon$$

for all $z \in \mathcal{W}_m^d$ with a number N of neurons satisfying

$$N = \mathcal{O}\left(\varepsilon^{-\frac{d}{m}}\right)$$

and this is the best possible.

Proof: See [10, Theorem 1] or [9, Theorem 2.1] for this result with $K_d = [-1, 1]^d$. The extension to $K_d \subset c_d + [-C, C]^d$ will be carried out in the exercises. □

We note that if $\theta \in \mathbb{R}^P$ realizing the infimum in the inequality in Theorem 6.1 exists, then in general it depends on g. Theorem 6.1 implies that one can readily use a network with one hidden layer for approximating Lyapunov functions. However, in general the number N of neurons needed for a fixed approximation accuracy $\varepsilon > 0$ grows exponentially in n, and so does the number of parameters in θ. This means that the storage requirement as well as the effort to determine θ easily exceeds all reasonable bounds already for moderate dimensions n. Hence, just like the simple discretisation approach sketched at the beginning of this chapter, this approach also suffers from the curse of dimensionality.

Remark 6.2 The differentiability requirement in Theorem 6.1 excludes the popular ReLU activation functions, which are obviously not differentiable. However, there are analogous results for ReLU activation functions, cf. [10, Section 4.1] and the references therein. □

[1] Polynomials are excluded because in the proof of this theorem it is needed that the derivatives $\sigma^{(k)}$ for all degrees $k \in \mathbb{N}$ do not vanish. See also the discussion after Theorem 1 in [10].

6.3 Improved results for compositional functions

In this section we will exploit the particular structure of compositional functions in order to obtain approximation results for DNNs with (asymptotically) much lower N. The following definition and theorem are inspired by [10] but significantly reformulated.

Definition 6.3 A function $z : \mathbb{R}^d \to \mathbb{R}$ is called a compositional function of degree $K \in \mathbb{N}$ and level $L \in \mathbb{N}$, if there are functions $h_{lj} : \mathbb{R}^K \to \mathbb{R}$, $l = 1, \dots, L$, $j = 1, \dots, d$, such that $z(y) = z(x_1, \dots, x_d)$ can be written in the form

$$z(x) = \sum_{j=1}^{d} \beta_j z_j^L,$$

where the values z_j^l are recursively defined as

$$z_j^l = h_{lj}(\alpha_{lj1} z_{i_{lj1}}^{l-1}, \dots, \alpha_{ljK} z_{i_{ljK}}^{l-1})$$

for $l = 1, \dots, L$, $z_i^0 = x_i$, $i_{ljk} \in \{1, \dots, d\}$ and $\alpha_{ljk}, \beta_j \in \mathbb{R}$ □

An example for a compositional function of degree 2 and level 1 from $\mathbb{R}^5$ to $\mathbb{R}$ is

$$z(x) = h_{11}(x_1, x_3) + 5h_{12}(x_5) + 0.5h_{13}(x_2, x_3)$$

and an example for a compositional function of degree 3 and level 2 from $\mathbb{R}^4$ to $\mathbb{R}$ is

$$z(x) = h_{11}(h_{21}(x_1, x_2, x_3), 2h_{22}(x_2, x_4), 7h_{23}(x_1, x_2, x_3)).$$

Note that although all functions h_{lj} are formally defined as function from $\mathbb{R}^K$, they may also have less than K arguments (since they do not depend on some of the arguments that are formally present). Likewise, it may possible that some of the h_{lj} are constantly equal to 0. In words, the degree K limits the maximal number of arguments of each function h_{lj} while the level L limits the number of nestings of the functions h_{lj} into each other.

For this class of functions we can now prove the following improved approximation result.

Theorem 6.4 Let $\sigma : \mathbb{R} \to \mathbb{R}$ be infinitely differentiable and not a polynomial. Let $\mathcal{C}$ be the set of compositional functions with fixed K and L but arbitrary d, where each function h_{lj} lies in $\mathcal{W}_m^K$ and the moduli $|\alpha_{ljk}|$ and $|\beta_j|$ as well as of the weights $|w_{klm}|$ needed for its approximation according to Theorem 6.1 with $z = h_{lj}$ are bounded by a constant M that is independent of d and of the desired accuracy.

Then, for any $\varepsilon > 0$, a neural network with L hidden layers provides an approximation

$$\inf_{\theta \in \mathbb{R}^P} \|W(\cdot; \theta) - z\|_{\infty, K_N} \leq \varepsilon$$

for all $z \in \mathcal{C}$ with a number N of neurons satisfying

$$N = \mathcal{O}\left(d^{\frac{K}{m}+1} \varepsilon^{-\frac{K}{m}}\right).$$

Proof: For simplicity of notation, throughout this proof we assume that the number of neurons N_ℓ in each level is identical and an integer multiple of d. We denote this number by $N_\ell = \widehat{N}d$, $\widehat{N} \in \mathbb{N}$. Then the overall number of neurons is $N = L\widehat{N}d$.

Now to each h_{lj} we assign the subset of neurons with values $y^l_{(j-1)\widehat{N}+1}, \dots, y^l_{j\widehat{N}}$. We refer to this subset of the overall DNN as a *sublevel*. Then, by Theorem 6.1, for any $\varepsilon_{lj} > 0$ we find weights[2] $\tilde{a}^l_{mj}$, $\tilde{b}^l_{mj}$, $\tilde{c}^l_j$, and $\tilde{w}^l_{kj}$ such that

$$\Big| \underbrace{h_{lj}(\alpha_{lj1} z^{l-1}_{i_{lj1}}, \dots, \alpha_{ljK} z^{l-1}_{i_{ljK}})}_{=z^l_j} - \sum_{m=1}^{\widehat{N}} \tilde{a}^l_{mj} \underbrace{\sigma^l\left(\sum_{k=1}^{K} \tilde{w}^l_{mjk} z^{l-1}_{i_{ljk}} + \tilde{b}^l_{mj} \right)}_{=y^l_{(j-1)\widehat{N}+m}} + \tilde{c}^l_j \Big| \le \varepsilon_{lj}.$$

Using the same approximation for z^{l-1}_j we can write

$$\begin{aligned} y^l_{(j-1)\widehat{N}+m} &= \sigma^l\left(\sum_{k=1}^{K} \tilde{w}^l_{mjk} z^{l-1}_{i_{ljk}} + \tilde{b}^l_{mj} \right) \\ &\approx \sigma^l\left(\sum_{k=1}^{K} \tilde{w}^l_{mjk} \left(\sum_{\tilde{m}=1}^{\widehat{N}} \tilde{a}^{l-1}_{\tilde{m} i_{ljk}} y^{l-1}_{(i_{ljk}-1)\widehat{N}+\tilde{m}} + \tilde{c}^{l-1}_{i_{ljk}} \right) + \tilde{b}^l_{mj} \right) \\ &= \sigma^l\left(\sum_{k=1}^{N_\ell} w^{l-1}_{mjk} y^{l-1}_k + b^{l-1}_{mj} \right), \end{aligned}$$

where the weights w^{l-1}_{mjk} and b^{l-1}_{mj} are obtained by expanding the sums in the second last line. This defines the weights for the neurons $y^l_{(j-1)\widehat{N}+1}, \dots, y^l_{j\widehat{N}}$ of this subnet and in the same way we can compute the weights for all neurons.

Since the partial derivatives of the functions h_{lj} are bounded by 1, we can conclude that each h_{lj} maps a set $K_d \subset c_d + [-C, C]^d$ to a set $\hat{K}_d \subset \hat{c}_d + [-C, C]^d$. Hence, if $y \in K_d \subset c_d + [-C, C]^d$, then the arguments of the functions h_{lj} lie in a set $K_{dlj} \subset c_{dlj} + [-CM^{l-1}, CM^{l-1}]^K$. If we moreover make sure that the approximation error ε_{lj} for each sublevel is bounded by 1, by induction we can guarantee that the internal values in the network are contained in the set $c_{dlj} + [-(CM^{l-1} - (l-1)M^{l-1}, CM^{l-1} + (l-1)M^{l-1}]^K$. We thus have to make sure that in each sublevel the approximation errors satisfy the tolerance ε_{lc} on this set. Finally, in each sublevel the error in the approximation of z^l_j is amplified by the weights $\tilde{w}^l_{kj}$. Hence, we have to make sure that the errors in the lower levels are choosen small enough that after this amplification they are still within the desired tolerance. This is possible since we assumed these values to be bounded independent of d.

Now, given a desired overall accuracy $\varepsilon > 0$, choose the values ε_{lj} such that the outermost functions h_{1k}, $k = 1, \dots, d$ are approximated with an accuracy $\varepsilon_{1k} = \varepsilon/(d\beta_k)$. Then, choosing the DNN weights of the top layer as $a_k = \beta_k$, the overall accuracy of the resulting weighted sum is ε.

Due to the fact that the individual accuracies ε_{lj} amplify multiplicatively when propagated through the network, there exists a constant $\gamma > 0$ (depending on L and M), such that

[2] More precisely, we first find weights for approximating $h_{lj}(z^{l-1}_{i_{lj1}}, \dots, z^{l-1}_{i_{ljK}})$ which by appropriate rescaling yield the weights for approximating $h_{lj}(\alpha_{lj1} z^{l-1}_{i_{lj1}}, \dots, \alpha_{ljK} z^{l-1}_{i_{ljK}})$.

$\varepsilon_{lj} \le \gamma\varepsilon/d$ ensures the desired bound on ε_{1k}. According to Theorem 6.1, each subnet must consist of

$$\mathcal{O}\left((\gamma\varepsilon/d)^{-\frac{K}{m}}\right) = \mathcal{O}\left(d^{\frac{K}{m}}\varepsilon^{-\frac{K}{m}}\right)$$

neurons, where the γ vanishes in the $\mathcal{O}$-term because it is independent of d. Since the number of subnets is bounded by dL, in which L is independent of d, the order of the overall number of neurons is obtained if we multiply the number, above, by d. This yields the desired estimate. □

Remark 6.5 The difference in the order of the number of neurons is best illustrated using some sample numbers. Assume we want an approximation accuracy $\varepsilon = 0.1$ and have functions with $d = 10$, $K = 5$ and $m = 1$. Then Theorem 6.1 requires an order of 10^{10} neurons while Theorem 6.4 requires only 10^7 neurons. For $d = 20$ the first number increases to 10^{20} (i.e., by a factor of 10^{10}), while the second only increases to $1.6 \cdot 10^8$ (i.e., by a factor of 16).

If the functions to be approximated by the DNN are sufficiently smooth (and their derivatives bounded), such that we can set $m = 3$, then for $d = 100$ we get only the order of 10^5 neurons from Theorem 6.4, but the order of 10^{33} neurons from Theorem 6.1. □

6.4 Training the DNN

The process of finding a parameter vector θ such that the DNN approximates the function it should approximate is commonly called *training*. To this end, we define a so-called *loss function* L, which penalises the pointwise deviation of $W(x;\theta)$ from a desired value. In the simplest case, one may want to minimise the expression

$$(W(y;\theta) - z(x))^2$$

for a given function z. Then $L : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ could be defined as

$$L(W, y) = (W - z(y))^2, \tag{6.2}$$

such that $L(W(x;\theta), x) = (W(x;\theta) - z(x))^2$. We note that this problem does not yet fit the typical task in RL, because there the desired function $z = Q$ or $z = V$ is not known. We will explain below how this problem can be solved.

Now we would not only want to approximate z in a single point $y \in \mathbb{R}^n$, but for all points $y \in K_n$. Ideally, we would like to minimize

$$\|W(\cdot;\theta) - z\|_\infty \quad \text{or} \quad \|W(\cdot;\theta) - z\|_2.$$

In order to obtain this at least approximately, we pick a large number of test points $y^1_{test}, \dots, y^J_{test}$, which are typically chosen randomly in K_n using a random number generator. Then we minimise the sum

$$F(\theta) = \frac{1}{J}\sum_{j=1}^{J} L(W(y^j_{test};\theta), y^j_{test})$$

with respect to θ. Since the number of test points in DNN training is usually very large, this minimisation is not performed at once, but by means of a so called *stochastic gradient method*. To this end, we define an iteration counter j, which is set to $j = 0$ at the beginning and pick an initial guess θ_0 for θ.

Then the set $Y_{test} = \{y^1_{test}, \dots, y^J_{test}\}$ is divided into M randomly generated subsets Y^1_{test}, ..., Y^M_{test}, the so called *batches*, each containing B elements. Then for $m = 1, 2, \dots, M$, the gradient ∇F_m of the function

$$F_m(\theta) := \frac{1}{B} \sum_{y_{test} \in Y^m_{test}} L(W(y_{test}; \theta), y_{test})$$

is computed in $\theta = \theta_j$ and a gradient step

$$\theta_{j+1} := \theta_j - \alpha_j \nabla F_m(\theta_j)$$

is performed for a step size $\alpha_j > 0$ and j is set to $j + 1$. When this is done for all $m = 1, \dots, M$, one says that the first *epoch* of the optimization is completed. Then new batches $X^1_{test}, \dots X^M_{test}$ are created from X_{test} and the next epoch of the optimization is started. This procedure is repeated until no further progress for the value of $F(\theta_j)$ can be achieved. Note that j is not reset to 0 after an epoch is finished but keeps on counting the overall iterations, i.e., we have $j = (k-1)M, \dots, kM - 1$ during the k-th epoch.

The good thing about the neural network structure is that the gradient ∇_m can be computed very efficiently. Practical algorithms that implement this basic idea come in many different variants. Particularly, the choice of the step size α_j differs in these variants. It may also be beneficial to add a regularising term to F, e.g., $\lambda \|\theta\|_2^2$ for a small parameter $\lambda > 0$. This prevents the algorithm from choosing extreme values for θ. The lecture notes "Optimization for Machine Learning" by Prof. Anton Schiela, available via the e-Learning course for his lecture in the summer semester 2020, discuss these kind of algorithms in great detail and also provide convergence statements under appropriate assumptions.

6.5 Deep reinforcement learning

The learning algorithm we discussed so far is called *supervised learning*, because the values of the function to be approximated are known and can be used in order to "teach" the neural network via the loss function.

In RL, the loss function must be defined in a different way, because the desired function $z = Q$ or $z = V$ is not known; we only know an equation it should satisfy. This is called *unsupervised learning*. For instance, in the case that we want to approximate the function Q, it is known that this function satisfies the dynamic programming principle

$$Q(x, u) = \ell(x, u) + \gamma \inf_{u' \in U} Q(g(x, u), u').$$

Hence, for $y = (x, u)$ and $x' = g(x, u)$ we can define

$$L(W, y) = \left(W - \ell(x, u) - \gamma \inf_{u \in U} W(x'; \theta_j) \right)^2.$$

The difference to (6.2) is that the loss function now depends on θ_j and thus changes when the iteration progresses. In the form defined here, the parameter θ_j appearing in the loss function is updated after each batch, but one could think of other ways of doing this.

As in conventional RL, $x' = g(x,u)$ can be obtained by evaluating $g(x,u)$ if this function is known or from simulated or measured data. Also, it is common not to generate the test points in X_{test} entirely randomly but rather use the test points generated along certain trajectories, i.e., defining the x-component of the test point $y^+ = (x^+, u^+)$ in the next time step as $x^+ = x'$. The u-component can, e.g., be obtained by the ε-greedy choice described in Section 4.3. If these trajectories are obtained by simulation or from measured data, then one can store and reuse them in later epochs. However, it may still be advisable to update the test point set X_{test} during the process, because the better the approximation of V or Q is, the better the controls generated by the ε-greedy choice are, which may provide better training progress. Obviously, there are a lot of different ways to implement this, involving quite a number of parameters to be tuned.

It is worth to summarise the main differences between classical RL and deep RL:

Classical Reinforcement Learning	Deep Reinforcement Learning
$\widetilde{Q}$ is updated for each (x,u)	θ_j is updated after each batch
the update only changes $\widetilde{Q}(x,u)$	the update changes $W(\cdot;\theta)$ everywhere
the new value for (x,u) is exactly assigned to $\widetilde{Q}(x,u)$	the new values determine $W(\cdot;\theta_{j+1})$ only indirectly via the optimisation
each (x,u) is visited many times	only a selection of (x,u) is visited

The last two points make it difficult to obtain rigorous convergence results for deep RL. The analysis is further complicated by the fact that, compared to simple deep learning problems, the cost function in deep RL depends on θ_j and thus changes as θ_j is updated.

Interestingly, this last complication vanishes if we consider continuous-time problems. In this case, the loss function is not derived from the Bellman equation (3.4) but from the Hamilton-Jacobi-Bellman equation (3.9). In continuous-time, it is more reasonable to approximate V instead of Q. This leads to the loss function

$$L(W, DW, x) = \left(-\delta W + \min_{u\in\mathbb{R}^m}\{DW \cdot f(x,u) + \ell(x,u)\}\right)^2,$$

which is obviously not depending on θ_j.

The price to pay is that now we also need the derivative $DW(x;\theta) = d/dx W(x;\theta)$ as an argument of L, and we also need to compute its derivative with respect to θ for computing the gradient ∇F_m with respect to θ. This, however, is not too difficult to implement with state-of-the-art software for neural networks.

Chapter 7

Compositional Lyapunov functions

July 21, 2021

Although compositionality of the optimal value function V or of the Q-function seems to be a promising way to ensure that Deep RL works in high dimensions, so far there are no general results that would ensure this property. However, there is a first result for a simplified problem, namely for computing Lyapunov functions. We consider this problem in continuous time and for ordinary differential equations without control input, i.e.,

$$\dot{x}(t) = f(x(t)). \tag{7.1}$$

We assume that $f : \mathbb{R}^d \to \mathbb{R}^d$ is Lipschitz continuous and that $x = 0$ is an equilibrium, i.e., $f(0) = 0$.

7.1 Lyapunov functions

Definition 7.1 A continuously differentiable function $V : O \to \mathbb{R}$ defined on an open set O with $0 \in O$ is a Lyapunov function if it satisfies the following properties: $V(0) = 0$, $V(x) > 0$ for all $x \neq 0$, and the orbital derivative $DV(x)f(x)$, i.e., the derivative DV of V multiplied with the direction of the vector field f, satisfies

$$DV(x)f(x) \leq -h(x) \tag{7.2}$$

for a function $h : \mathbb{R}^d \to \mathbb{R}$ with $h(x) > 0$ for all $x \in O \setminus \{0\}$. If $O = \mathbb{R}^d$ and $V(x) \to \infty$ as $\|x\| \to \infty$, then V is called a global Lyapunov function. □

It is known that a Lyapunov function exists if and only if the equilibrium 0 is asymptotically stable. If V is a Lyapunov function, then any connected component of a sublevel set of V containing 0 is part of the domain of attraction of $x = 0$, which is the set of all initial conditions for which $x(t; x_0) \to 0$ as $t \to \infty$. Hence, a given set $K_d = [-C, C]^d$ is part of the domain of attraction of $x = 0$ if it is contained in such a sublevel set. In this case we call V a Lyapunov function on K_d.

Under suitable regularity conditions on h, there exists a Lyapunov function that solves the partial differential equation

$$DV(x)f(x) + h(x) = 0.$$

This was proved by Zubov in the 1960s. This equation, called the *Zubov equation* is nothing but a particular simple version of a Hamilton-Jacobi-Bellman PDE (3.9). Here h plays the role of the cost function while V plays the role of the optimal value function (the close relation between optimal value functions and Lyapunov functions was already observed several times in Mathematical Control Theory). Hence, the computation of Lyapunov functions can be seen as a particular special case of computing optimal value functions.

7.2 Separable Lyapunov functions

For Lyapunov functions, we will consider compositional functions with level $L = 1$. Such functions are also called *separable*. In order to define this structure, the system (7.1) is divided into s subsystems Σ_i of dimensions d_i, $i = 1, \dots, s$. To this end, the state vector $x = (x_1, \dots, x_d)^T$ and the vector field f are split up as

$$x = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_s \end{pmatrix} \quad \text{and} \quad f(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_s(x) \end{pmatrix},$$

with $z_i = (x_{\hat{d}_{i-1}+1}, \dots, x_{\hat{d}_i}) \in \mathbb{R}^{d_i}$ and $f_i : \mathbb{R}^d \to \mathbb{R}^{d_i}$ denoting the state and dynamics of each Σ_i, $i = 1, \dots, s$, with state dimension $d_i \in \mathbb{N}$ and $\hat{d}_i = \sum_{j=1}^{i} d_j$. With

$$z_{-i} := \begin{pmatrix} z_1 \\ \vdots \\ z_{i-1} \\ z_{i+1} \\ \vdots \\ z_s \end{pmatrix}$$

and by rearranging the arguments of the f_i, the dynamics of each Σ_i can then be written as

$$\dot{z}_i(t) = f_i(z_i(t), z_{-i}(t)), \quad i = 1, \dots, s.$$

Using this decomposition, we can define the following Lyapunov function structure[1].

Definition 7.2 A Lyapunov function V for (7.1) is called *separable*, if there exist C^1-functions $\widehat{V}_i : \mathbb{R}^{d_i} \to \mathbb{R}$ such that V is of the form

$$V(x) = \sum_{i=1}^{s} \widehat{V}_i(z_i). \tag{7.3}$$

□

In the remainder of this section we discuss conditions on f under which a Lyapunov function of the form (7.3) exists.

[1] In order to avoid an overly technical presentation, the exposition in this section is limited to global Lyapunov functions.

One situation in which a system (7.1) admits a separable Lyapunov function is when the f_i do not depend on z_{-i}, i.e., if $f_i(z_i, z_{-i}) = f_i(z_i)$. This means that the subsystems are completely decoupled. In this case, consider Lyapunov functions $\hat{V}_i$ of $\dot{x}_i = f_i(x_i)$ on compact sets $\widehat{K}_i \subset \mathbb{R}^{d_i}$, and V from (7.3). Then, clearly $V(x) \geq 0$ and $V(x) = 0$ if and only if $x = 0$. Moreover,

$$DV(x)f(x) = \sum_{i=1}^{s} DV_i(z_i)f_i(z_i) < 0$$

for all $x \in K_d = \prod_{i=1}^{s} \widehat{K}_i$ with $x \neq 0$.

Assuming that f decomposes into s completely decoupled subsystems is relatively restrictive. Fortunately, a similar construction can also be made if the f are coupled, provided the coupling is such that it does not affect the stability of the overall subsystem. The systems theoretic tool for this approach is nonlinear small-gain theory, which relies on the input-to-state stability (ISS) property introduced in [12]. It goes back to [7, 8] and in the form for large-scale systems we require here it was developed in the thesis [11] and in a series of papers around 2010, see, e.g., [4, 5] and the references therein. ISS small-gain conditions can be based on trajectories or Lyapunov functions and exist in different variants. Here, we use a variant from [4], which is most convenient for obtaining approximation results because it yields a smooth Lyapunov function.

For formulating the small gain condition, we assume that for the subsystems Σ_i there exist C^1 ISS-Lyapunov functions $V_i : \mathbb{R}^{d_i} \to \mathbb{R}$, satisfying for all $z_i \in \mathbb{R}^{d_i}$ $z_{-i} \in \mathbb{R}^{n-d_i}$

$$DV_i(z_i)f_i(z_i, z_{-i}) \leq -\alpha_i(V_i(z_i)) + \sum_{j \neq i} \gamma_{ij}(V_j(z_j))$$

with rates $\alpha_i \in \mathcal{K}_\infty$ and gains $\gamma_{ij} \in \mathcal{K}_\infty$,[2] $i, j = 1 \ldots, s$, $i \neq j$. Here, the states z_{-i} of the other subsystems are interpreted as the input to the i-th subsystem and the term $\sum_{j \neq i} \gamma_{ij}(V_j(z_j))$ in the ISS inequality quantifies how much this input affects the stability of the i-th subsystem. Particularly, the larger the ISS-gains γ_{ij} are, the more the other systems' influence can affect the decrease of the Lyapunov function V_i. Setting $\gamma_{ii} := 0$, we define the map $\Gamma : [0, \infty)^s \to [0, \infty)^s$ by

$$\Gamma(r) := \begin{pmatrix} \sum_{j=1}^{s} \gamma_{1j}(r_j), \ldots, \sum_{j=1}^{s} \gamma_{sj}(r_j) \end{pmatrix}^T$$

and the diagonal operator $A : [0, \infty)^s \to [0, \infty)^s$ by

$$A(r) := (\alpha_1(r_1), \ldots, \alpha_s(r_s))^T .$$

Definition 7.3 We say that (7.1) satisfies the small-gain condition, if there is a decomposition into subsystems Σ_i, $i = 1, \ldots, s$, with ISS Lyapunov functions V_i satisfying the following condition: there are positive definite[3] functions η_i, $i = 1, \ldots, s$, satisfying $\int_0^\infty \eta_i(\alpha_i(r))dr = \infty$ and such that for $\eta = (\eta_1, \ldots, \eta_s)^T$ the inequality

$$\eta(r)^T \Gamma \circ A^{-1}(r) < \eta(r)^T r \tag{7.4}$$

[2] As usual, we define $\mathcal{K}_\infty$ to be the space of continuous functions $\alpha : [0, \infty) \to [0, \infty)$ with $\alpha(0) = 0$ and α is strictly increasing to ∞.

[3] A continuous function $\rho : [0, \infty) \to [0, \infty)$ is called positive definite if $\rho(0) = 0$ and $\rho(r) > 0$ for all $r > 0$.

holds for all $r \in [0,\infty)^s$ with $r \neq 0$. Here A^{-1} denotes the operator consisting of the inverse functions α_i^{-1}. □

It is easily seen that this inequality is satisfied whenever the gains γ_{ij} are sufficiently small, which explains the name small-gain condition. However, it is not necessary that *all* γ_{ij} are small, as the following example shows.

Example 7.4 Consider the 2d ordinary differential equation

$$\begin{aligned} \dot{x}_1 &= -x_1 + \rho x_2 \\ \dot{x}_2 &= \varepsilon x_1 - x_2 \end{aligned}$$

for parameters $\varepsilon, \rho > 0$. We decompose the system with $z_1 = x_1$ and $z_2 = x_2$. A suitable way for obtaining the Lyapunov functions V_i is often to set z_{-i} to 0 and try to find Lyapunov functions for $\dot{z}_i = f_i(z_i, 0)$. In this example, this is very easy and one sees that the simple choice $V_i(z_i) = z_i^2$ does the job (this is typically a good choice in the 1d case). We obtain

$$DV_1(z_1) f_1(z_1, 0) = 2z_1(-z_1) = -2z_1^2$$

as well as

$$DV_2(z_2) f_2(z_2, 0) = 2z_2(-z_2) = -2z_2^2.$$

Including the coupling we obtain for the first equation

$$DV_1(z_1) f_1(z_1, z_2) = -2z_1^2 + 2z_1\rho z_2$$

Using $a^2 - 2ab + b^2 = (a-b)^2 \geq 0$ with $a = z_1$ and $b = \rho z_2$ we obtain $2z_1\rho z_2 \leq z_1^2 + \rho^2 z_2^2$. Thus we can conclude

$$DV_1(z_1) f_1(z_1, z_2) \leq -2z_1^2 + z_1^2 + \rho^2 z_2^2 \leq -z_1^2 + \rho^2 z_2^2 = -V_1(z_1) + \rho^2 V_2(z_2).$$

Similarly, for the second equation we obtain

$$DV_2(z_2) f_2(z_2, 0) = -2z_1^2 + 2z_2\varepsilon z_1 \leq -z_2^2 + \varepsilon^2 z_1^2 = -V_2(z_2) + \varepsilon^2 V_1(z_1).$$

This yields $\alpha_1(w) = \alpha_2(w) = w$, $\gamma_{12}(w) = \rho^2 w$, and $\gamma_{21}(w) = \varepsilon^2 w$. This means that the size of the gains depend on ρ and ε. The larger these parameters, the larger values the respective gain functions have.

The operators Γ and A then evaluate to

$$\Gamma(r) = \left(\rho^2 r_2,\, \varepsilon^2 r_1\right)^T \quad \text{and} \quad A^{-1}(r) = (r_1, r_2)^T,$$

implying

$$\Gamma \circ A^{-1}(r) = \left(\rho^2 r_2, \varepsilon^2 r_1\right)^T.$$

The small-gain condition (7.4) then reads

$$\eta_1(r_1)\rho^2 r_2 + \eta_2(r_2)\varepsilon^2 r_1 < \eta_1(r_1) r_1 + \eta_2(r_2) r_2.$$

With the choice $\eta_1(r_1) = r_1/\rho^2$, $\eta_2(r_2) = r_2/\varepsilon^2$ this inequality becomes

$$2r_1r_2 < \frac{r_1^2}{\rho^2} + \frac{r_2^2}{\varepsilon^2}.$$

If $r_1r_2 \le 0$ and $r_1, r_2 \neq 0$, then this inequality is always satisfied. For $r_1r_2 > 0$, if we assume that $\rho\varepsilon < 1$, then using $2ab \le a^2 + b^2$ with $a = r_2/\varepsilon$, $b = r_1/\rho$, yields

$$2r_1r_2 < 2\frac{r_1}{\rho}\frac{r_2}{\varepsilon} \le \frac{r_1^2}{\rho^2} + \frac{r_2^2}{\varepsilon^2}$$

and this the desired inequality. The small-gain condition (7.4) thus reduces to the requirement that $\rho\varepsilon < 1$. This shows that a single gain can be large—even vary large—if the other gain is sufficiently small. It does not matter whether a single gain is large. Rather, the combination of the gains matters. □

It should be mentioned that there are small-gain conditions in the literature that are easier to check and to interpret. However, these conditions lead to Lyapunov functions with less regularity. The advantage of the condition we use here is that the following theorem from [4] yields a differentiable Lyapunov function.

Theorem 7.5 Assume that the small-gain conditions from Definition 7.3 hold. Then V from (7.3) is a Lyapunov function for the C^1-functions $\widehat{V}_i : \mathbb{R}^{d_i} \to \mathbb{R}$ given by

$$\widehat{V}_i(z_i) := \int_0^{V_i(z_i)} \lambda_i(\tau)d\tau$$

where $\lambda_i(\tau) = \eta_i(\alpha_i(\tau))$.

Proof: We use the notation

$$\vec{V}(x) = \begin{pmatrix} V_1(z_1) \\ \vdots \\ V_s(z_s) \end{pmatrix}, \quad D\vec{V}(x)f(x) = \begin{pmatrix} DV_1(z_1)f_1(z_1, z_{-1}) \\ \vdots \\ DV_2(z_s)f_s(z_s, z_{-s}) \end{pmatrix} \quad \text{and} \quad \lambda(\vec{V})(x) = \begin{pmatrix} \lambda_1(V_1(z_1)) \\ \vdots \\ \lambda_s(V_s(z_s)) \end{pmatrix}.$$

Similarly we define $\eta(\vec{V})$. Moreover, we write $(-A + \Gamma)(r) = -A(r) + \Gamma(r)$. With this notation, it holds that

$$DV(x)f(x) = \lambda(\vec{V})(x)^T D\vec{V}(x)f(x) \le \lambda(\vec{V})(x)^T(-A + \Gamma)(\vec{V}(x)).$$

The small-gain condition then implies for all $x \neq 0$ that

$$\eta(A(\vec{V}(x)))^T\Gamma \circ A^{-1} \circ A(\vec{V}(x)) < \eta(A(\vec{V}(x)))^T A(\vec{V}(x)).$$

This implies

$$\underbrace{-\lambda(\vec{V})(x)^T A(\vec{V}(x)) + \lambda(\vec{V}(x))^T\Gamma(\vec{V}(x))}_{=:h(x)} < 0$$

for all $x \neq 0$. This shows (7.2). The fact that $V(0) = 0$, $V(x) > 0$ if $x \neq 0$ follow from the construction of V and the requirement $V(x) \to \infty$ as $\|x\| \to \infty$ follows from $\int_0^\infty \eta_i(\alpha_i(r))dr = \infty$. □

We note that for various reasons small-gain conditions are not easy to check and to apply: the gains γ_{ij} may be difficult to estimate, the functions η_i may be hard to find and, above all, appropriate Lyapunov functions V_i for the subsystems may be nontrivial to construct. However, none of these ingredients need to be known for our approach. Moreover, not even the number and the dimension of the subsystems needs to be known and we will also be able to define the z_i in a more general way than we did in this section. All that needs to be assumed in what follows is that a separable Lyapunov function V of the form (7.3) exists. In summary, the small-gain theory just presented only serves to show that it is realistic to assume the existence of such a V, but the particular subsystem structure does not need to be known for constructing it. Rather, provided that an upper bound for the dimension of the subsystems is known, the resulting compositional form of V will be detected by the training algorithm of the neural network.

7.3 Approximation results

The separable Lyapunov function (7.2) is nothing but a compositional function with degree $K = \max_i d_i$ and level $L = 1$. Provided that this K is independent of d, Theorem 6.4 readily implies that separable Lyapunov functions can be approximated with a number of neurons that grows only polynomially in the dimension d, using a DNN with one hidden layer.

However, we can even go one step further. It may be that one cannot find the Lyapunov functions $\hat{V}$ in (7.2) depending on subvectors of x, however, it may be possible that a Lyapunov function of the form (7.2) exists if we allow the z_i to be linear combinations of subvectors of x. Formally, this means that there is a coordinate transformation $T : \mathbb{R}^d \to \mathbb{R}^d$, such that $\tilde{f}(x) = Tf(T^{-1}x)$ admits a separable Lyapunov function, i.e., a Lyapunov function of the form (7.2). In this case, one easily sees that if $\widetilde{V}$ is a separable Lyapunov function for $\tilde{f}$, then $V(x) = \widetilde{V}(Tx)$ is a Lyapunov function for f. This is because $DV(x) = D\widetilde{V}(Tx)T$ and $f(x) = T^{-1}\tilde{f}(Tx)$, and thus

$$DV(x)f(x) = D\widetilde{V}(Tx)TT^{-1}\tilde{f}(Tx) = D\widetilde{V}(Tx)\tilde{f}(Tx) \le -\widetilde{h}(Tx) =: -h(x).$$

In order to approximate $V(x)$ with a DNN, it suffices to add a second layer to the neural network and if we limit ourselves to linear coordinate transformations, it is sufficient to use linear activation functions in this second layer. It is easily seen that each linear coordinate transformation T can be exactly represented by this layer.

If we know (or conjecture) a certain separable structure, i.e., if we know upper bounds for K and s, then we can use this information for setting up the network. We can then remove some of the connections in the network in order to represent the fact that $\widehat{V}(z_1)$ will only depend on the first subset components of Tx, $\widehat{V}(z_2)$ depends on the second subset of components and so on. Figure 7.1 shows such a network. Using this network structure allows us to get an indication whether a separable Lyapunov exists also in low dimensions. While the computational advantage of separable functions only becomes visible for sufficiently large d, this particular network structure will only be able to represent Lyapunov functions with a certain separability structure, but not general ones. Hence, if the network fails to find a Lyapunov function in the training process, then this is an indication (though, of course, not a proof), that no separable Lyapunov function exists.

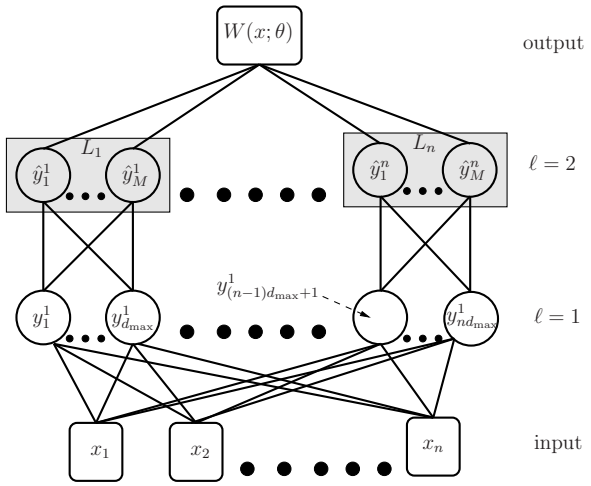

Figure 7.1: Neural network for Lyapunov functions $V = \widetilde{V} \circ T$ with known upper bounds $d_{\max}$ and n for K and s, respectively

7.4 Training the network

For training the network we need to specify a loss function $L : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. As explained in the previous chapter, training then consists of finding parameters θ such that

$$\frac{1}{m} \sum_{i=1}^{M} L\left(W(x_{test}^{(i)};\theta), DW(x_{test}^{(i)};\theta), x_{test}^{(i)}\right) \tag{7.5}$$

becomes minimal, where $x_{test}^{(i)} \in K_n$ are the elements of a test dataset, which we refer to as test points. In our numerical results in the next section we always use $K_n = [-1,1]^n$ and the test points $x^{(i)}$ are chosen randomly and uniformly distributed from K_n. Note that as in the case of general Hamilton-Jacobi-Bellman equations, the loss function L also depends on the values of the derivative of W with respect to x in the test points, which we denote by $DW(x^{(i)},\theta)$.

The main work is now to design the loss function such that minimizing (7.5) w.r.t. θ yields a Lyapunov function. To this end, a straightforward idea is to express the Lyapunov function property as a partial differential equation (PDE) and follow the approaches in the literature for solving PDEs with neural networks mentioned in the introduction. A simple PDE that is suitable for this purpose is the Zubov equation with $h(x) = \|x\|^2$, i.e.,

$$DW(x;\theta)f(x) = -\|x\|^2. \tag{7.6}$$

This PDE needs to be complemented by suitable boundary conditions, which in the PDE setting (with $x = 0$ being the equilibrium of interest) are of the form

$$W(0,\theta) = 0 \quad \text{and} \quad W(x;\theta) > 0 \text{ for all } x \in K_n \setminus \{0\}.$$

However, in this form the boundary conditions are difficult to be implemented numerically: the equality condition $W(0,\theta) = 0$ is difficult because it is only given in a single point, while

the strict ">" condition is difficult because numerically only "≥" can be enforced directly. To resolve these problems, we replace the boundary conditions above by the stronger conditions

$$\alpha_1(\|x\|) \le W(x;\theta) \le \alpha_2(\|x\|) \text{ for all } x \in K_n, \tag{7.7}$$

with $\alpha_1, \alpha_2 \in \mathcal{K}$. Of course, the functions α_i have to be chosen appropriately in order to allow for the existence of a solution of (7.6) that satisfies (7.7). However, it is known that if a Lyapunov function on K_n exists, then it is always possible to find such α_i. Loosely speaking, α_1 must be sufficiently flat while α_2 must be sufficiently steep. In case $x = 0$ is exponentially stable and f is continuously differentiable, one can choose the α_i as quadratic functions $\alpha_i(r) = c_i r^2$ with $c_1 > 0$ sufficiently small and $c_2 > 0$ sufficiently large.

Given the vector field f from (7.1), the loss function L is now defined as

$$L(w,p,x) := \big(pf(x) + \|x\|^2\big)^2 + \nu\left(([w-\alpha_1(\|x\|)]_-)^2 + ([w-\alpha_2(\|x\|)]_+)^2\right), \tag{7.8}$$

where $[a]_- := \min\{a,0\}$, $[a]_+ := \max\{a,0\}$, and $\nu > 0$ is a weighting parameter (chosen as $\nu = 1$ in all our numerical examples in the next section). One easily checks that for this L the expression (7.5) is always ≥ 0 and equals 0 if and only if (7.6) and (7.7) are satisfied for all test points $x^{(i)}$. Conversely, if a Lyapunov function exists for which the bounds (7.7) are feasible, and if this Lyapunov function can be represented by neural network under consideration, the minimizing (7.5) w.r.t. θ will result in the optimal value of (7.5) being 0.

Unfortunately, while this approach works in principle, it is not necessarily compatible with the complexity analysis from the previous section. The reason is that when a Lyapunov function with the particular small gain structure (7.3) exists, it may not be a solution of (7.6), (7.7). As a consequence, while a solution of (7.6), (7.7) may exist, it may not be representable by the neural network structure from Figure 7.1. Hence, with the choice of L from (7.8), it may not be possible to exploit the low computational complexity provided by this particular network structure. The result depicted in Figure 7.3, below, shows that this indeed happens.

Hence, we need to provide more flexibility to our approach, which we can do by enlarging the set of minima of the loss function. To this end, note that (7.6) is actually a much too strong condition. Requiring the partial differential inequality (PDI)

$$DW(x;\theta)f(x) \le -\|x\|^2, \tag{7.9}$$

instead of (7.6), will also yield a Lyapunov function. While one may argue that the bound "$-\|x\|^2$" on the derivative is somewhat arbitrary here, it is easily seen that by appropriate rescaling any Lyapunov function can be modified such that this bound holds. Hence, modifying the right hand side of (7.9) does not provide more flexibility (but, of course, it affects the set of α_i for which (7.9) and (7.7) together are feasible).

Incorporating (7.9) instead of (7.6) in the loss function L leads to the expression

$$L(w,p,x) := \left(\left[pf(x) + \|x\|^2\right]_+\right)^2 + \nu\left(([w-\alpha_1(\|x\|)]_-)^2 + ([w-\alpha_2(\|x\|)]_+)^2\right). \tag{7.10}$$

One easily checks that for this L the expression (7.5) is again always ≥ 0, but now it equals 0 if and only if (7.9) and (7.7) are satisfied for all test points $x^{(i)}$. As Example 7.11 and Figure 7.2, below, show, this indeed allows to use the network structure from the previous section and it also allows for solving higher dimensional problems, see Example 7.12.

7.5 Numerical examples

We illustrate the proposed method with two examples, a low-dimensional one that shows that the the loss function (7.10) is in general preferable over (7.8) and a larger one that shows the ability of the method to work in find Lyapunov functions in higher dimensions. All computations were performed with Python 3.7.0 and TensorFlow 2.1.0 on a MacBook Pro (2017, 2.3 GHz Intel Core i5) running macOS Mojave (10.14.6). The python code and the trained networks are available from numerik.mathematik.uni-bayreuth.de/~lgruene/DeepLyapunov/.

Our first example considers a two-dimensional example that has a compositional Lyapunov function consisting of two one-dimensional functions. It is given by

$$\begin{aligned} \dot{x}_1 &= -x_1 - 10x_2^2 \\ \dot{x}_2 &= -2x_2. \end{aligned} \tag{7.11}$$

Using the Lyapunov-function candidate $V(x) = x_1^2 + x_2^2 + 13x_2^4$, one computes

$$DV(x)f(x) = -2x_1^2 - 20x_1x_2^2 - 4x_2^2 - 104x_2^4.$$

Since

$$-x_1^2 - 20x_1x_2^2 - 104x_2^4 \le -x_1^2 - 20x_1x_2^2 - 100x_2^4 = -(x_1 + 10x_2^2)^2 \le 0,$$

we obtain $DV(x)f(x) \le -x_1^2 - 4x_2^2 \le -\|x\|^2$. Hence, V is a Lyapunov function and it is obviously of the compositional form (7.3) with $z_1 = x_1$ and $z_2 = x_2$.

It should thus be possible to compute a Lyapunov function with the neural network from Figure 7.1. It turns out that using the loss function (7.10) (with $\alpha_1(r) = 0.1r^2$ and $\alpha_2(r) = 10r^2$) this is indeed possible. Here we used the network structure from Figure 7.1 with $n = 2$ and $d_{\max} = 1$, with the layers L_1 and L_2 consisting of 128 neurons, each, and softplus activation functions $\sigma^2(r) = \ln(e^r + 1)$, resulting in 775 trainable parameters. The training was performed with 200 000 test points[4], optimizing with batch size 32 using the Adam optimizer implemented in TensorFlow. The optimization was terminated when the accuracy for the final function $W(\cdot, \theta^*)$ satisfied[5]

$$err_1 := \frac{1}{m}\sum_{i=1}^{m} L\left(W(x^{(i)}, \theta^*), DW(x^{(i)}, \theta^*), x^{(i)}\right) < 10^{-6}$$

and

$$err_\infty := \max_{i=1,\dots,m} L\left(W(x^{(i)}, \theta^*), DW(x^{(i)}; \theta^*), x^{(i)}\right) < 10^{-6},$$

which was reached after 6 epochs in the run documented here.[6] The time needed for the optimization was 48s. Figure 7.2 shows the computed approximate Lyapunov function $W(\cdot, \theta^*)$ as a solid surface along with its derivative along the vector field $DW(x; \theta^*)f(x)$ as a wireframe, shown from two different angles. The graphs illustrate that the method was successful.

[4] In all examples, the number of test points was increased until the results produced satisfactory Lyapunov functions.

[5] Since L consists of squared penalization terms, err_1 is effectively the squared weighted $\|\cdot\|_2$-norm of the penalization terms.

[6] As the test points are random, the results of the training optimization are random, too. The error tolerance 10^{-6} was sometimes reached already after 4 epochs and sometimes it was not reached until epoch 20. In all successful runs, the resulting Lyapunov was very similar to the one depicted here.

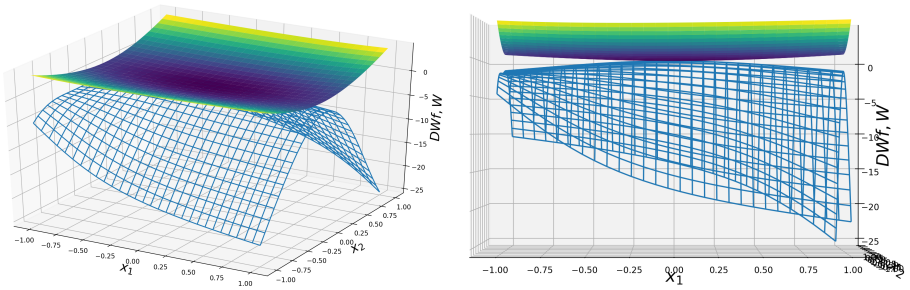

Figure 7.2: Approximate Lyapunov function $W(\cdot;\theta^*)$ (solid) and its orbital derivative $DW(\cdot;\theta^*)f$ (mesh) for Example (7.11) computed with loss function (7.10)

In contrast to this, performing the computation with the same parameters but with loss function (7.8) fails. As Figure 7.3 shows, the derivative $DW(x;\theta^*)f(x)$ (shown as a wire-frame) obviously not satisfy the equation $DW(x;\theta^*)f(x) = -\|x\|^2$. This is also visible in the values

$$err_1 = 1.363842 \quad \text{and} \quad err_\infty = 3.110839$$

that were reached after 20 epochs[7]. While this alone would not be a problem (as long as $DW(x;\theta^*)f(x)$ is still negative definite), the inability to meet this equation has the side effect that the optimization also does not enforce the inequalities (7.7). As a consequence, the minimum of the computed function is not located in the equilibrium at the origin, as the lateral view on the right of Figure 7.3 shows. This is because it is more difficult to represent a Lyapunov function satisfying $DV(x)f(x) = -\|x\|^2$ with the network structure from Figure 7.1. While this example does, of course, not exclude that the loss function (7.8) works for other parameters, it provides evidence that the advantage in computational complexity offered by our approach is more easily exploited using the loss function (7.10). Moreover, it illustrates the effect when the parameter $d_{\max}$ underestimates the maximal dimension of the subsystems.

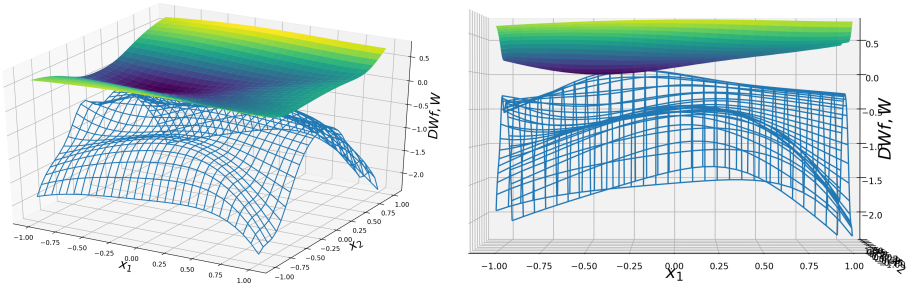

Figure 7.3: Attempt to compute a Lyapunov function $W(\cdot;\theta^*)$ (solid) with its orbital derivative $DW(\cdot;\theta^*)f$ (mesh) for Ex. (7.11) with loss function (7.8)

[7] In all runs these error values did not change significantly anymore after epoch 15. In some runs the resulting function had a different shape, but in all cases it visibly violated the required inequalities.

In our second example we illustrate the capability of our approach to handle higher dimensional systems and to determine the subspaces for the compositional representation of V. To this end we consider a 10-dimensional example of the form

$$\dot{x} = f(x) := T^{-1}\hat{f}(Tx). \tag{7.12}$$

with vector field $\hat{f} : \mathbb{R}^{10} \to \mathbb{R}^{10}$ given by

$$\hat{f}(x) = \begin{pmatrix} -x_1 + 0.5x_2 - 0.1x_9^2 \\ -0.5x_1 - x_2 \\ -x_3 + 0.5x_4 - 0.1x_1^2 \\ -0.5x_3 - x_4 \\ -x_5 + 0.5x_6 + 0.1x_7^2 \\ -0.5x_5 - x_6 \\ -x_7 + 0.5x_8 \\ -0.5x_7 - x_8 \\ -x_9 + 0.5x_{10} \\ -0.5x_9 - x_{10} + 0.1x_2^2 \end{pmatrix}$$

One easily sees that this system consists of five two-dimensional asymptotically stable linear subsystems that are coupled by four nonlinearities with small gains. It is thus to be expected that on $K_{10} = [-1, 1]^{10}$ the system is asymptotically stable and a Lyapunov function can be computed using the network from Figure 7.1 five two-dimensional sublayers $L_1, \dots, L_5$. The coordinate transformation $T \in \mathbb{R}^{10\times 10}$ is given by the (randomly generated) matrix

$$T = \begin{pmatrix} -\frac{1}{5} & -\frac{3}{10} & \frac{1}{2} & -\frac{4}{5} & \frac{4}{5} & \frac{2}{5} & \frac{7}{10} & \frac{7}{10} & -1 & \frac{4}{5} \\ \frac{1}{5} & 1 & \frac{9}{10} & \frac{4}{5} & -\frac{1}{10} & \frac{3}{5} & -\frac{3}{10} & \frac{1}{2} & \frac{4}{5} & -\frac{3}{10} \\ -\frac{3}{10} & \frac{3}{10} & \frac{2}{5} & -\frac{2}{5} & 0 & -\frac{3}{5} & \frac{3}{10} & \frac{3}{5} & 1 & -\frac{1}{2} \\ -\frac{7}{10} & -\frac{1}{10} & -\frac{3}{5} & -\frac{1}{5} & -\frac{3}{5} & \frac{2}{5} & \frac{1}{10} & -\frac{1}{10} & \frac{1}{10} & -\frac{3}{5} \\ \frac{1}{10} & -\frac{3}{5} & -\frac{9}{10} & -\frac{7}{10} & -\frac{1}{5} & -\frac{1}{10} & \frac{1}{10} & \frac{1}{5} & 0 & -\frac{4}{5} \\ \frac{3}{5} & \frac{9}{10} & -\frac{1}{5} & 1 & \frac{2}{5} & \frac{1}{2} & 0 & -\frac{1}{10} & -\frac{2}{5} & 0 \\ -1 & 1 & \frac{7}{10} & \frac{3}{5} & -\frac{4}{5} & -\frac{4}{5} & 0 & -\frac{1}{5} & -\frac{1}{5} & \frac{7}{10} \\ -\frac{9}{10} & \frac{4}{5} & \frac{1}{5} & 1 & -\frac{4}{5} & \frac{2}{5} & -\frac{3}{10} & \frac{7}{10} & \frac{1}{5} & -\frac{4}{5} \\ \frac{3}{5} & -\frac{1}{10} & -\frac{2}{5} & -\frac{1}{2} & -\frac{3}{10} & -\frac{1}{10} & -\frac{7}{10} & 1 & \frac{4}{5} & -\frac{3}{10} \\ 0 & -1 & -\frac{1}{10} & \frac{2}{5} & -\frac{3}{10} & -\frac{1}{10} & -\frac{1}{5} & \frac{7}{10} & -\frac{1}{10} & \frac{4}{5} \end{pmatrix}.$$

We have computed a Lyapunov function for this system for the loss function (7.10) with $\alpha_1(r) = 0.2r^2$ and $\alpha_2(r) = 10r^2$. We used the network structure from Figure 7.1 with $n = 5$ and $d_{\max} = 2$, with the layers $L_1, \dots, L_5$ consisting of 128 neurons, each, leading to 2671 trainable parameters. The training was performed with 400 000 test points, optimizing over 13 epochs. As for the 2d example, we used batch size 32, the Adam optimizer implemented in TensorFlow, and softplus activation functions σ^2. The time needed for the training was

266s[8] and the resulting function satisfies the inequalities

$$err_1 < 10^{-6}, \quad err_\infty < 10^{-6}.$$

Figures 7.4 and 7.5 show the resulting function $W(\cdot;\theta^*)$ (solid) and its derivative along f (wireframe) on the (x_2, x_8)-plane and the (x_9, x_{10})-plane, respectively. The remaining components of x were set to 0 in both figures. Figure 7.6 shows the value of $W(\cdot;\theta^*)$ along three trajectories of (7.12) (computed numerically using the ode45-routine from matlab). It shows the strict decrease that is expected from a Lyapunov function.

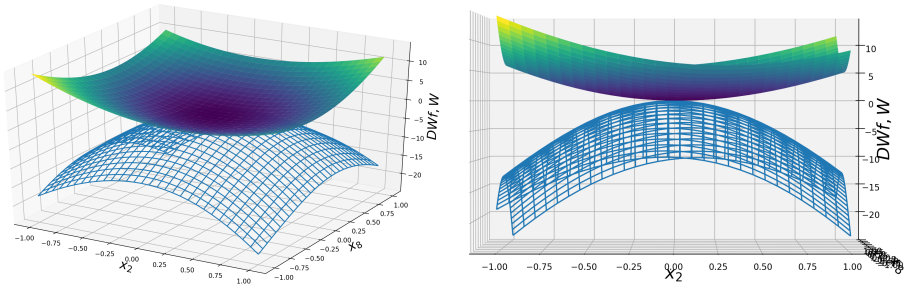

Figure 7.4: Approximate Lyapunov function $W(\cdot;\theta^*)$ (solid) and its orbital derivative $DW(\cdot;\theta^*)f$ (mesh) for Example (7.12) on (x_2, x_8)-plane

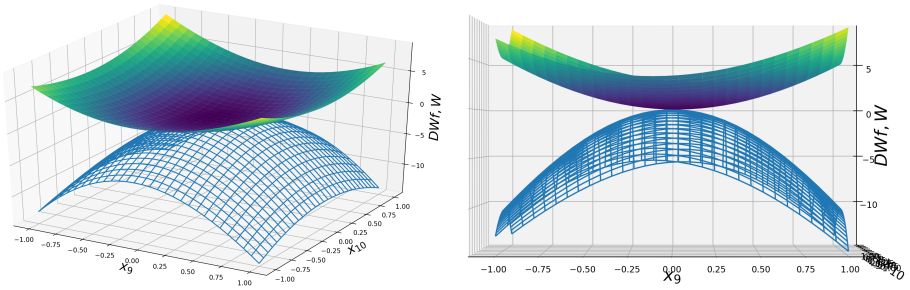

Figure 7.5: Approximate Lyapunov function $W(\cdot;\theta^*)$ (solid) and its orbital derivative $DW(\cdot;\theta^*)f$ (mesh) for Example (7.12) on (x_9, x_{10})-plane

[8]The time for the evaluation of $W(x;\theta^*)$ in 10 000 test points takes 0.3s, while the evaluation of the derivative $DW(x;\theta^*)$ in 10 000 test points takes 0.1s.

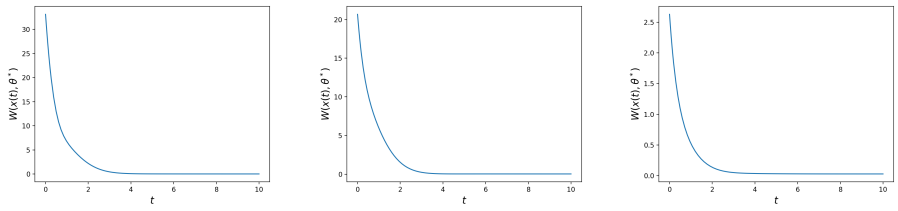

Figure 7.6: Value of approximate Lyapunov function $W(x(t);\theta^*)$ along trajectories for initial values $x_0 = (1,1,1,1,1,1,1,1,1,1)^T$, $(0,1,0,1,0,1,0,1,0,1)^T$, $(1,0,0,0,0,0,0,0,0,0)^T$ (left to right)

Bibliography

[1] A. Barto and R. S. Sutton, *Reinforcement Learning: An Introduction*, MIT Press, 2nd ed., 2018.

[2] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, 2nd ed., 1996.

[3] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Math. Control Signals Systems, 2 (1989), pp. 303–314.

[4] S. Dashkovskiy, H. Ito, and F. Wirth, *On a small gain theorem for ISS networks in dissipative Lyapunov form*, Eur. J. Control, 17 (2011), pp. 357–365.

[5] S. N. Dashkovskiy, B. S. Rüffer, and F. R. Wirth, *Small gain theorems for large scale systems and construction of ISS Lyapunov functions*, SIAM J. Control Optim., 48 (2010), pp. 4089–4118.

[6] K. Hornik, M. Stinchcombe, and H. White, *Multilayer feedforward networks are universal approximators*, Neural Networks, 3 (1989), pp. 551–560.

[7] Z.-P. Jiang, I. M. Y. Mareels, and Y. Wang, *A Lyapunov formulation of the nonlinear small-gain theorem for interconnected ISS systems*, Automatica, 32 (1996), pp. 1211–1215.

[8] Z. P. Jiang, A. R. Teel, and L. Praly, *Small-gain theorem for ISS systems and applications*, Math. Control Signals Syst., 7 (1994), pp. 95–120.

[9] H. N. Mhaskar, *Neural networks for optimal approximation of smooth and analytic functions*, Neural Computations, 8 (1996), pp. 164–177.

[10] T. Poggio, H. Mhaskar, L. Rosaco, M. Brando, and Q. Liao, *Why and when can deep – but not shallow – networks avoid the curse of dimensionality: a review*, Int. J Automat. Computing, 14 (2017), pp. 503–519.

[11] B. S. Rüffer, *Monotone Systems, Graphs, and Stability of Large-Scale Interconnected Systems*. Dissertation, Fachbereich 3, Mathematik und Informatik, Universität Bremen, Germany, 2007.

[12] E. D. Sontag, *Smooth stabilization implies coprime factorization*, IEEE Trans. Autom. Control, 34 (1989), pp. 435–443.