Approximate Dynamic Programming applied to a wind energy storage problem

Mia Lu

Born 8th July 1996 in Bochum, Germany

13th August 2018

Master's Thesis Mathematics

Advisor: Prof. Dr. Jochen Garcke

Second Advisor: Prof. Dr. Ira Neitzel

Mathematical Institute

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Contents

1 Introduction

In recent years, the use of renewable energy sources has rapidly increased: The worldwide wind electricity generation rose from 104 GWh in the year 2005 to 838 GWh in the year 2015 [IEAb]. This accounts for 3.7% of the world's total electricity generation in the year 2015 [GWE]. Among renewable energy sources wind energy plays a significant role: Electricity generated from wind turbines both off-and on-shore amounted to 15.7% of the global total of electricity generated from renewable sources in the year 2015 [IEAa].

However, a drawback of using wind energy is that the energy source (wind) is inherently intermittent and hard to predict. This makes it difficult to integrate wind energy into the electricity grid: Power producers have to commit to delivering a certain amount of electricity, the size of which is determined in an auction well before delivery (for example the day before). Unlike for traditional and controllable energy sources like coal, power producers using renewable and stochastic energy sources like wind face the risk of not being able to deliver the required amount.

A possible solution is to use energy storage optimally to smoothen fluctuations in wind energy. Excess energy can be stored for later use when demand exceeds currently available wind energy. The goal is then to optimize this storage usage. "Optimizing" means from the viewpoint of a wind farm operator to maximize profit while satisfying demand. This constitutes the *(wind) energy storage problem*.

The canonical approach for solving general stochastic optimal control problems like this is *Dynamic Programming (DP)*, but it is computationally very expensive. It breaks a complex optimization problem consecutively into smaller subproblems which are then solved exactly, and was pioneered by Bellman in the 1950s [Bel57]. However, dynamic programming is only applicable in restricted settings (for instance, it can not handle continuous state spaces) and becomes computationally prohibitively expensive for large state spaces (it suffers from the "curse of dimensionality": the computational costs grow exponentially with the number of states).

As a generalization of dynamic programming which aims to circumvent these shortcomings by approximate computations, *Approximate Dynamic Programming (ADP)* (also called *Reinforcement Learning*, depending on the community) has gained significant popularity over the last few decades [Pow07, SB17]. Spectacular successes such as the victories of the program AlphaGo over a Go world champion in 2016 have fuelled the increasing interest in the application of ADP methods.

DP and ADP approaches have been applied to energy storage problems with mixed success [KHH03, CBJK08, HD11, SS16, JPP$^+$14]. In particular, Jiang et al. [JPP$^+$14] compared various ADP methods for a simple benchmark energy storage problem consisting of a single energy storage device which interacts with the grid and wind to satisfy demand. They considered both general-purpose methods (among them *approximate policy iteration* using *Gaussian Process Regression (GPR)*) and approaches specifically designed for the problem (*lookup table methods* exploiting properties like monotonicity [JP15] or concavity [SP17]). Their result was sobering – they found that none of these methods worked well: the general-purpose methods performed poorly, whereas the problem-specific approaches performed well but were not scalable and computationally too expensive.

The contributions of this thesis are the following: Our numerical results suggest that general-

purpose ADP methods might be a better method to solve energy storage problems than previously found. Specifically, we take one of the approaches from Jiang et al. [JPP+14] (approximate policy iteration using Gaussian Process Regression) and examine why it did not work well. We find that with a suitable choice of parameters (in particular, with a suitable choice of initial policy and a suitable choice of hyperparameters for the GPR) the approach does in fact perform significantly better than found by Jiang et al. The results are encouraging and suggest that, with further work on the optimal choice of parameters, even better results can be achieved. Furthermore, we give an overview over the theory underlying the approach and provide a practical analysis of the algorithm.

The rest of the thesis is organised as follows: In Chapter 2 we introduce the theoretical framework of discrete time stochastic optimal control. In particular, we explain the Dynamic Programming Principle and the policy iteration algorithm. In Chapter 3 we motivate and describe the model of the energy storage problem in detail. In Chapter 4 we present important concepts of Approximate Dynamic Programming and use them in Chapter 5 to characterize the approach in Jiang et al. [JPP+14]. Furthermore, we outline alternative approaches to the specific benchmark problem in [JPP+14] and provide a detailed introduction to Gaussian Process Regression. Finally, we discuss our implementation and the numerical results in Chapter 6. We analyse effectiveness and efficiency of the algorithm and suggest questions for further research.

Acknowledgements. First of all, I would like to express my deepest gratitude to my advisor Prof. Jochen Garcke who suggested the topic of my thesis. I am indebted to Prof. Garcke for his patient guidance throughout the last year as well as his insightful questions and excellent suggestions. I also thank Prof. Ira Neitzel for agreeing to be my second examiner. Many thanks go to Jannik Schürg for his open door and his invaluable comments from which this thesis has benefited tremendously.

My gratitude also goes to my family and my friends for supporting me throughout the thesis. Special thanks go to Anna for proof-reading and her helpful comments.

Finally, I thank Rahul; for proof-reading large parts of this thesis, for his motivating encouragement, and for his support in every possible way.

2 Theoretical framework

The energy storage problem we will consider is an optimal control problem that is continuous in state space, discrete in time, finite horizon and undiscounted. This choice of modelling will be motivated in the next chapter.

In this chapter we present the necessary theoretical groundwork to model the problem and to design an algorithm solving this problem. The chapter is split into three parts: In Section 2.1 we introduce the fundamental components of discrete time stochastic optimal control theory and present the Dynamic Programming Principle, which forms the basis for the algorithmic framework necessary for the following chapters. This algorithmic framework is then presented in Section 2.2, in particular we use Markov Decision Processes (which are a subclass of discrete time stochastic optimal control problems). This is the classical framework for the central Dynamic Programming algorithms. We will explain how they work and discuss important properties. In Section 2.3 we briefly introduce the concept of post-decision states, which we will need later on for the algorithm.

2.1 Discrete Time Stochastic Optimal Control Theory

In the following section we loosely follow the presentation in [Ber17] (especially Chapter 1 and 5 in Vol. I and Chapter 1 in Vol. II) which deals with both finite and infinite horizon stochastic optimal control problems, as well as discounted and undiscounted problems. In [Ber17], the author offers a technically precise approach and covers a breadth of algorithms.

In optimal control we aim to find a control policy for a given system such that a given optimality criterion is achieved. Optimality is often defined as minimizing or maximizing a certain reward function. The central components in the optimal control framework are the state dynamics, the admissible control policies, the rewards and the value function.

The first step is to model the underlying system and to understand how it behaves under a given control policy. We denote the set of potential *states* at time t by $\mathcal{S}_t$ and the set of potential *actions* at time t by $\mathcal{A}_t$. However, at each state s_t in $\mathcal{S}_t$ only a subset of actions is available. We call it the set of *admissible actions* or *admissible decisions* $\mathcal{A}_t(s_t) \subset \mathcal{A}_t$. As an example, later on in our energy storage problem, the state will consist among other things of the given energy storage and the available wind energy. For states with differing storage or wind energy, the admissible action set will look different: We can only sell what is either available in the storage or currently available in wind energy. Our admissible action sets will be constrained and of the form

$$\mathcal{A}_t(s_t) := \left\{ a_t \in \mathbb{R}^m \;\middle|\; \begin{array}{l} g_i(s_t, a_t) = 0, \;\; \forall i = 1, \dots, I \\ h_j(s_t, a_t) \geqslant 0, \;\; \forall j = 1, \dots, J \end{array} \right\},$$

where g_i and h_j are scalar real functions respectively representing the equality or inequality constraints which define admissibility.

Control policies (later on just called control or policy) are rules determining for every potential state what action is to be taken; thus a policy is a sequence of functions mapping each state to a decision. Control policies can either be a function of time (*open-loop control*) or a function of space (*feedback control*). In this case we consider feedback controls, that is, a function

of the state s_t. A *admissible control policy* maps for every time step each possible state to an admissible action.

We write a policy as $\pi = (\pi_t)_{0 \leqslant t \leqslant T}$, where $0 < T \leqslant \infty$ is our finite time horizon, and

$$\pi_t \in \Pi_t := \{q : \mathcal{S}_t \to \mathcal{A}_t \mid \forall s_t \in \mathcal{S}_t : q(s_t) \in \mathcal{A}_t(s_t)\}.$$

The set of admissible policies is then defined as $\Pi := \Pi_0 \times \ldots \times \Pi_T$. [1] A policy with time-invariant mapping is called *stationary policy* and is of the form $\pi = (\mu)_{0 \leqslant t \leqslant T}$. We identify it with μ and will denote the induced cost function by V^μ.

In most applications (ranging from engineering to economics), the behaviour of the system is governed by partial differential equations. In the discrete time case we consider difference equations. As we consider the stochastic case, the transition consists of two parts: the deterministic one determined by our control and a stochastic one determined by some random disturbance. The system dynamics then are

$$s_{t+1} = \underbrace{s_t + f(s_t, a_t) + \xi_{t+1}}_{:= g(s_t, a_t, \xi_{t+1}))}, \tag{2.1}$$

where f is a deterministic function, and ξ_{t+1} is a random variable containing noisy information ("disturbance") arriving between time t and $t+1$. The disturbance ξ_{t+1} takes on values in some space Ξ_t and is characterized by a probability distribution $P_t(\cdot|s_t, a_t)$, where $a_t \in \mathcal{A}_t$ is the action at time t (if we use a policy, then $a_t = \pi_t(s_t)$). Thus, both the states s_t and the disturbances ξ_t are random variables, their distributions are governed by (2.1).

Remark 2.1. We adapt the convention here that any variable indexed by t is known at time t. Hence, the stochastic information has the subscript $t+1$ instead of t.

We will also sometimes use the notation of a general transition function g, which combines the stochastic and the deterministic transition. Later on, we will sometimes use the expected value of some function h where the expectation is over the possible future states. For ease of notation we will often write $\mathbb{E}[h(s_{t+1})|s_t]$ instead of $\mathbb{E}_{\xi_{t+1}}[h(g(s_t, a_t, \xi_{t+1}))]$.

In optimal control, "optimal" is defined by satisfying a reward criterion, or more specifically, by optimizing a certain reward function.

Remark 2.2. **(Notation)**

(i) *Notation in and connection between different communities:* There are several different communities doing similar work, but with differing notations. These are mainly three communities: the optimal control community (coming from engineering, physics or economics), the reinforcement learning community using Markov Decision Process (MDP) models (coming from computer science), and operations research. For example, in the control theory context we typically speak of "costs" and aim to minimize the total cost.

[1] We will use Π to refer to the space of admissible policies in general, in particular also when we speak of truncated policies starting from time t on, where the space is $\Pi_t \times \ldots \times \Pi_T$. We use this abuse of notation to avoid an excessive use of indices, the meaning of each Π will be evident in context.

In reinforcement learning, we typically speak of "rewards" which we aim to maximize. In operations research, both is used, depending on the context. Similarly, we speak of an action a in the reinforcement learning context or a decision x in operations research, while x typically denotes the state in control theory and the state is denoted as S in operations research.

What is more: While it might seem as if only the notation differs, this actually represents the fact that the different communities look at different problems and algorithmic strategies. In operations research the problems tend to be very high-dimensional (up to the thousands) but discrete, typically considering resource allocation problems (such as managing inventory, or people). In reinforcement learning the state and the action space are typically small, famous examples are board games. The system dynamics and the reward function are often not known. In control theory, the control is typically low-dimensional (under ten), but the state and action spaces are continuous.

(ii) *Notation in this thesis:* In this case, we use a hybrid of notation: We mainly use the notation of discrete time stochastic optimal control, but also use terms from reinforcement learning and operations research, in order to have a suitable and unified notation. For example, since we will later on maximize the profit in our energy storage problem, we already speak of "rewards" here instead of "costs". The goal is to have a consistent notation throughout the different parts of this thesis and to not change notation between chapters.

As a reward function we consider

$$C_t : \mathcal{S}_t \times \mathcal{A}_t \times \Xi_t \to \mathbb{R}.$$

We model the rewards as additive.

Remark 2.3. We will assume in the following that we have *perfect state information*. It means that we observe all necessary information we need to compute transition or rewards.

Given a policy π, the sequence of events at time t is thus as follows:

1. The decision-maker observes the state s_t and chooses the action a_t (potentially based on some policy).
2. The disturbance ξ_{t+1} is generated according to the probability distribution $P(\cdot|s_t, a_t)$.
3. The decision-maker collects the reward $C_t(s_t, a_t, \xi_{t+1})$ and adds it to the previous rewards.
4. The next state s_{t+1} is generated according to the system dynamics (2.1).
5. If $t < T$, go to time $t + 1$. Otherwise, the process terminates.

We aim to optimize the total reward over all time periods. However, since the reward function is influenced by the random variable ξ, it is a random variable itself. We can therefore only optimize *in expectation*.

For a *given policy* π we denote the *expected reward from time t on* as

$$V_t^\pi(s_t) := \mathop{\mathbb{E}}_{\xi_1,\dots,\xi_{T+1}} \left[\sum_{\tau=t}^{T} C_\tau(s_\tau, \pi_\tau(s_\tau)) \middle| s_t \right], \tag{2.2}$$

where $s_t \in \mathcal{S}_t$ is an initial state at time t and $\pi = (\pi_\tau)_{t \leqslant \tau \leqslant T}$ as before. For $t = 0$ we omit the subscript zero and just write $V^\pi(s_0) := V_0^\pi(s_0)$. In (2.2), s_t depends on ξ_{t+1} via the transition function (2.1). We can consider V_t^π as a measure of how good a policy is.

Then we can define the *optimal expected reward from time t* on as

$$V_t(s_t) = \sup_{\pi \in \Pi} V_t^\pi(s_t),$$

where $s_t \in \mathcal{S}_t$ is an initial state at time t. Our objective function is then V_0, as above we will just use $V := V_0$. [2]

An *optimal policy* π^*, if it exists, is a policy that maximizes the expected reward:

$$V_t^{\pi^*}(s_t) = \sup_{\pi \in \Pi} V_t^\pi(s_t) = V_t(s_t).$$

This policy may depend on the state s_t, but we find that often the same policy is optimal for all states. We will call V the *value function*[3] and V_t the *value function from time t on*. In words, $V(s)$ denotes the reward we would obtain starting in state s if we chose an optimal policy from then on. In this way, it denotes the value of a state. (We can consider V_t, for $0 \leqslant t < T$, as a kind of truncated value function.)

Discount. A more general form of (2.2) would be

$$V_t^\pi(s_t) := \mathop{\mathbb{E}}_{\xi_1,\dots,\xi_{T+1}} \left[\sum_{\tau=t}^{T} \alpha^\tau C_\tau(s_\tau, \pi_\tau(s_\tau)) \middle| s_t \right],$$

where $0 < \alpha \leqslant 1$ is a *discount factor*. It represents the case where, for a fixed reward R, the decision-maker values obtaining R in the present more than obtaining R in the future. A typical example is for example the depreciation of money: Let r be the rate of interest. A dollar at time 0 is worth $(1+r)^t$ dollars at time t. Reversely, a dollar at time t is worth $\left(\frac{1}{1+r}\right)^t$ dollars at time 0. In this case we then have $\alpha = \frac{1}{1+r}$.

If $\alpha < 1$, we speak of a *discounted problem*, if $\alpha = 1$ we speak of an *undiscounted problem*. In the finite horizon literature, we normally have $\alpha = 1$. In the infinite horizon case, we often have $\alpha < 1$. This is to ensure (along with other assumptions) that the limit in the definition of the value function is finite.

[2] We assume here that the expectation is well-defined, see the following paragraph on measure theoretic issues.

[3] In the literature the value function is sometimes also called *cost-to-go-function*.

The Dynamic Programming Algorithm. We will now describe the *Dynamic Programming Principle*, also called *Bellman equation* or *Principle of Optimality*. It is a necessary condition for optimality.

In Bellman's own words, the Dynamic Programming Principle signifies that:

> "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." [Bel57, p.83]

It states that the value function V can be computed via the value functions V_t:

Theorem 2.4. *(Dynamic Programming Principle/ Bellman equation)*

$$V_t(s_t) = \sup_{a_t \in \mathcal{A}_t(s_t)} \left[C_t(s_t, a_t) + \mathbb{E}\left[V_{t+1}(s_{t+1}) | s_t, a_t \right] \right], \tag{2.3}$$

where $V_T(s_T) = 0$ for all s_t and s_{t+1} implicitly depends on s_t via the transition function (2.1).

The intuition behind Dynamic Programming (DP) is simple: Let $\pi^* = (\pi_0^*, \dots, \pi_T^*)$ be an optimal policy. Recall that V_{t+1} is the expected value of an optimal policy from time $t+1$ on. The Bellman equation (2.3) then expresses that, if the policy $(\pi_t^*, \dots, \pi_T^*)$ is optimal for the truncated problem from time t on, then for the truncated problem from time $t+1$ on the policy $(\pi_{t+1}^*, \dots, \pi_T^*)$ must still be optimal. The original optimization problem is then consecutively broken down in a sequence of smaller subproblems. Consider a shortest path example: If the shortest path between some points A and B, denoted by P, goes through C, then the part of $P|_{\overline{BC}}$ must also be the shortest path between B and C.

This directly translates into an algorithm - we just need to follow this logic and starting from the last stage T go backwards in time:

The Dynamic Programming Algorithm

(1) Set $v_T(s_T) := 0$, for all $s_T \in \mathcal{S}_T$.

(2) Repeat for all $t = T-1, \dots, 0$:

$$v_t(s_t) := \sup_{a_t \in \mathcal{A}_t(s_t)} \left[C_t(s_t, a_t) + \mathbb{E}\left[v_{t+1}(s_{t+1}) | s_t, a_t \right] \right], \text{ for all } s_t \in \mathcal{S}_t \tag{2.4}$$

We will in general denote by v_t the approximation of the value function V_t, where v_t is obtained in an algorithm or through some iteration (in some special cases, v_t may be accurate and coincide with V_t).

Then we can reformulate the Dynamic Programming Principle:

Proposition 2.5. *For every initial state s_0 we have $V_0(s_0) = v_0(s_0)$, where v_0 is obtained after the last step of the algorithm above.*

Proof: See [Ber17, pp.25-26, Vol.I], for a slightly informal proof. A rigorous mathematical proof can be found on [Ber17, pp.46-48] for the case where the stochastic factor ξ only takes on a countable number of values and where all expected values are finite.

The Dynamic Programming Principle is despite its conceptual simplicity immensely powerful. It has been applied in various disciplines, a prominent example is economics. There it is used for example in consumption theory, growth theory or asset pricing [4].

Remark 2.6. In the continuous time deterministic case the equivalent of the Bellman equation is the *Hamilton-Jacobi-Bellman equation*. It is a partial differential equation characterizing the value function as its unique viscosity solution (viscosity solutions are a generalization of the classical solution concepts for partial differential equations). An introduction to Hamilton-Jacobi-Bellman equations can be found in [ES98], for a more detailed presentation on this as well as approximation schemes we refer to [FF13].

This can be extended to the stochastic optimal control case where we now consider stochastic processes instead of functions. A main application for stochastic Hamilton-Jacobi-Bellman equations is in finance. A thorough presentation of stochastic differential equation as well as a discussion of stochastic optimal control can be found in [Øk00].

For finite horizon problems, we can theoretically always use the Dynamic Programming Principle to find the exact solution. In many applications however, exact computations are impossible or too expensive. The natural next step is to build concrete algorithms build on the Dynamic Programming Principle for these cases and to examine their properties. The two most popular of these algorithms (*value iteration* and *policy iteration*) will be presented and discussed in detail.

In order to do that we will focus on a subcategory of discrete time stochastic optimal control problems which are called *Markov Decision Processes*. They provide the original and intuitive framework in which the classical DP algorithms value iteration and policy iteration were developed. In themselves, the pure form of both value iteration and policy iteration are of limited utility for our application. This is because they require a perfect model and are computationally expensive. However, they are of fundamental theoretical importance and are the building blocks for the algorithms to come.

In the following section we will focus on *finite state* and finite horizon problems. In the later chapter on Approximate Dynamic Programming (ADP) we will deal with the case when the state space is continuous or too big for exact computation.

Before we proceed, however, some notes on technical precision:

Measure theoretic issues. For general state spaces we have to deal with several technical issues to ensure that all terms are well-defined and that the DP Algorithm is still valid. For that reason we need to introduce additional (measurability) assumptions.

[4] An good introduction to DP methods in the context of economics can be found in [Rus96].

The two main problems that we face when developing DP theory for general state spaces are:

(1) The value function (2.2) needs to be well-defined for all policies π. The expected value is an integral and this integral needs to be defined. Not only this, we also need to have sufficient assumptions, such that the DP results below also hold.

(2) The optimal expected total reward is $V^*(s) = \sup_{\pi \in \Pi} V^\pi(s)$. We want to ensure that V^* is actually attained at the end of the DP Algorithm.

For (1) there are two options to ensure that the integrals are meaningful: One option is to expand the notion of integrals, that is, use outer integrals. The basic DP results still hold, however, this framework has some limitations inherent to outer integrals. The second option is to place measurability constraints on the reward function C, the probability distribution of the disturbances ξ and the policies. Specifically, for the policies we need a class of measurable functions which is closed under some key operations, particularly under partial optimization (since the value function is obtained by partial minimization; we only optimize over the policies/actions, not over the possible states). This property is not satisfied by Borel measurable functions. Instead, one can use *universally measurable functions*, which are a related class of lower semianalytic functions.

In order to ensure (2) it suffices to be able to approximate the supremum arbitrarily close. Stated explicitly: It suffices to have ϵ-*optimal* policies for every $\epsilon > 0$, where a policy π is ϵ-optimal if

$$V^\pi(s) \geqslant V(s) + \epsilon \quad \forall s \in \mathcal{S}.$$

One can show that this holds for the class of universally measurable policies. With these assumptions the essential DP results carry over: The DP Algorithm generates the value function, and also an ϵ-optimal policy. If the supremum is attained for all states, then there exists an optimal policy.

However, these technical issues do not contribute significantly to the understanding of the central concepts. Apart from that, we will also typically deal with finite spaces in practice. We will therefore not go further into the technical details and implicitly assume that everything is well-defined. For a detailed analysis and treatment of the measure theoretic issues mentioned above, we refer to [BS07, Chapter 7-8]. We also note that in the finite horizon finite spaces case as in the section on MDPs below, additional assumptions are unnecessary since the expectation can be simply computed as a finite sum and thus no measurability framework is necessary.

2.2 Markov Decision Processes

Markov Decision Processes (MDPs) developed out of the study of sequential decision processes (the study of sequential decisions under uncertainty), where each decision possibly depends on the previous decisions and their outcomes. They have been considered to be first studied systematically as a tool for optimization by Bellman in the late 1950s. In his book "Dynamic programming" [Bel57] Bellman famously coined the term "Principle of Optimality", which is the basis of value iteration and which is formalized in the Bellman equation. Actually a couple

of years earlier, Shapley [Sha53] developed value iteration as a special case in his paper on stochastic games, but that was only recognized some decades later (an MDP can be seen as a stochastic game with only one player). Some also consider Howard [How60] to have started the systematic study of MDPs. In his seminal work, "Dynamic Programming and Markov Processes", he introduced the concept of policy iteration (which is therefore sometimes also called *Howard's algorithm*).

Since then, MDPs have become increasingly popular and have been studied extensively in countless books and articles. At this point the existence theory is fairly complete and numerous algorithms for computing optimal policies with respect to different objectives (maximizing for instance expected discounted reward, limiting expected average reward, or total reward) are established. MDPs have found many applications in operations research, economics, engineering and computer science. They are extensively used in reinforcement learning. From the 1950s to the 1980s, MDP mainly dealt with optimality equations and how to solve them, particularly using value iteration and policy iteration algorithms.From the 1980s on, a major methodological advance was the development of approximations of DP for large-scale problems. A classic on MDPs is also [Put94], for many more references we refer to [FS02].

The "Markov" in Markov Decision Processes derives from the fact that for a fixed policy, the MDP behaves like a Markov chain. MDPs can thus be considered as an extension of Markov Chains. What distinguishes them is the addition of actions and rewards (that is, allowing control over the process, and an explicit measure of how "good" an outcome is). Then Markov Chains are the special case when the rewards are constant and there is only one possible action for each state.

We focus on finite state MDPs. The theory does not require state or action spaces to be finite, but exact algorithms require this. The main goal of this chapter is to illustrate how policy iteration and value iteration work because they form the foundation of ADP and the algorithms to come. It is therefore not a significant restriction to focus on finite state MDPs [5]. For an in-depth treatment of MDPs for general state spaces we refer to [BR11].

Since Markov Decision Processes are a subclass of discrete time stochastic control problems, they are characterized by the same components, albeit with slightly different names in the literature. Traditionally, in MDPs the system is stationary: The system dynamics, the set of states, the set of actions, and the reward function stay the same over time. Thus we modify the setup from before slightly, we now consider:

- a finite set of states, written as $\mathcal{S} = \{1, \ldots, n\}$
- a finite set of actions $\mathcal{A}$
- a reward $C(i, a)$ for transitioning from i via a, for all states $i \in \mathcal{S}$ and all actions $a \in \mathcal{A}$
- a (time independent) transition probability $p_{ij}(a) = Prob(s_{k+1} = j | s_k = i, a_k = a)$ for the probability of transitioning from i to j via a, for all states $i, j \in \mathcal{S}$ and all actions $a \in \mathcal{A}$ (These are now the system dynamics.) Per definition it must hold that $\sum_{j \in \mathcal{S}} p_{ij}(a) = 1$.

[5] As implied, there are many other types of MDPs: continuous-time MDPs, MDPs with incomplete information (that is, some model parameters, such as transition probabilities, are not known),...

The definition of the expected total reward for a given policy and the definition of the value function stay the same.

Considering that we used the transition function with a stochastic disturbance (2.1) before, it might seem like a different theoretical approach to use transition probabilities. However, the notations are equivalent. The only reason for the different notation is that it is sometimes simply more convenient or common to use one instead of the other.

We can quickly see the equivalence [Ber17, Vol.I, p.8]: Let $p_{ij}(a)$ be the transition probability that the next state is j, when i is the current state and a the selected action. Then we can alternatively describe the system as

$$s_{t+1} = w_t,$$

where w_t is a random parameter distributed according to

$$P(w_t = j | s_t = i, a_t = a) = p_{ij}(a).$$

On the other hand, let $s_{t+1} = g(s_t, a_t, w_t)$ be the system equation, where w_t is distributed according to $P(w_t|s_t, a_t)$. Then the transition probability is simply

$$p_{ij}(a) = P(\{w | j = g(i, a, w)\} | s_t = i, a_t = a).$$

Remark 2.7. (i) We see the Markov property: The transition to a new state only depends on the current state and action.

(ii) The most general form of the reward function includes the next state, as in $C(i, a, j)$. However, in practice this is rarely needed. We will omit this here, the results can easily be transferred to the more general case. We also note that in the case where the reward function does depend on the successor state j, using $C(i, a)$ amounts to using the expectation of the reward over all possible states.

Note that if an action a is not available in state i ($a \notin \mathcal{A}(i)$), then we simply have $p_{ij}(a) = 0$ for all $j \in S \setminus \{i\}$. Thus, we omit the state dependence and always use the whole state space $\mathcal{A}$.

Type of policies. In this paragraph we follow the presentation in [Ber17, Vol.II, Section 1.1.4]. So far we have only considered *Markov policies*, that is, policies that at time t only depend on the current state s_t and not on past states. Furthermore, we have so far only considered *deterministic policies*, where the policy always yields the same action for the same state. However, one might wonder if we can obtain better results by considering a bigger set of policies. In this paragraph, we consider two extensions:

Firstly, we include policies which depend on the past history. We call a policy *history-dependent* if at time t it depends on the history h_t up to that point, where $h_t := \{s_0, \dots, s_t\}$.

Secondly, we include randomized policies where instead of one action we choose a probability distribution over the action set. We will thus consider randomized, history-dependent policies $\pi = \{\mu_0, \dots, \mu_t\}$, where μ_t is a function that maps a history h_t onto a probability distribution $\mu_t(a_t|h_t)$ over $\mathcal{A}$. For simplicity, we will assume that the action space $\mathcal{A}$ and the disturbance space Ξ are countable. (As we have seen before, the notation using a disturbance space and the notation with transition probabilities are equivalent.)

Proposition 2.8. *(Adequacy of Markov Policies) [Ber17, Vol.II, Proposition 1.1.1] Assume that the action space is countable, and consider an initial state distribution that takes values over a countable set. The probability distribution of each pair (s_t, a_t) and the expected reward of each stage corresponding to a randomized history-dependent policy can also be obtained with a randomized Markov policy.*

Moreover, in many cases we can even restrict ourselves to *deterministic* Markov policies (that is, in those cases we can obtain the same optimal cost using deterministic Markov policies as when using randomized Markov policies). This holds for example for finite space stochastic shortest path problems. What is more, as we will see, in the finite horizon version of the finite space stochastic shortest path problem we can even restrict our attention to *stationary* deterministic Markov policies.

Dynamic Programming Algorithms: Policy iteration and value iteration. The key idea of Dynamic Programming algorithms is to use value functions to find good policies. This is achieved by turning the Bellman equation into an update rule. We will now prepare the introduction of the policy iteration and the value iteration algorithm.

For that purpose, we will now consider *stochastic shortest path problems* of which our finite horizon problem is a special case of.

Stochastic shortest path problems are undiscounted, finite state, infinite horizon problems. In a sense they are a hybrid of finite and infinite horizon problems: There are effectively only finitely many time steps, but the length is random. Particularly, there is a unique cost-free *termination state* $t \in \mathcal{S}$. We will denote all other states as $1, \dots, n$, so that $\mathcal{S} = \{1, \dots, n, t\}$. Whenever the termination state t is reached, the process stops. Put formally,

$$p_{tt}(a) = 1 \text{ and } C(t, a) = 0 \text{ for all } a \in \mathcal{A}.$$

Every (finite state) finite horizon problem can be cast as a stochastic shortest path problem: We just need to include the time index in the state formulation. Let $\mathcal{S}$ be our original state space, and let T be the time horizon. Then we consider as our new state space

$$\mathcal{S}' = \left(\bigcup_{t=0}^{T-1} S_t \right) \cup \{t\}, \quad \text{where } S_t := \{(s, t) \mid s \in \mathcal{S}\} \text{ and } t := \bigcup_{s \in \mathcal{S}} (s, T). \tag{2.5}$$

That is, we consider all state-time pairs for the last time step together as our termination state, termination is therefore guaranteed after $t = T$.

In the general infinite horizon case, we need an additional assumption to ensure that termination eventually occurs for all policies:

Assumption 1. ([Ber17], Vol. I, Assumption 5.2.1) There exists an integer m such that regardless of the policy used and the initial state, there is positive probability that the termination state will be reached after no more than m stages; i.e., for all admissible policies π we have

$$\rho_\pi = \max_{i=1,\dots,n} P(s_m \neq t \mid s_0 = i, \pi) < 1$$

(For instance, this precludes cycles with only negative costs.) A stationary policy with this property is called *proper*. For the stochastic shortest path problems induced by finite horizon problems as above, this trivially holds, in fact $\rho_\pi = 0$.

Proposition 2.9. *([Ber17], Proposition 5.2.1)* [6]
Under Assumption 1, the following hold for the stochastic shortest path problem:

(a) Given any initial conditions $v^0(1), \dots, v^0(n)$, the sequence $v^k(i)$ generated by the iteration

$$v^{k+1}(i) = \max_{a \in \mathcal{A}} \left[C(i,a) + \sum_{j=1}^{n} p_{ij}(a) v^k(j) \right], \quad i = 1, \dots, n \tag{2.6}$$

converges to the value function (the optimal expected total reward) $V(i)$ for each i.

(b) The value function $V(1), \dots, V(n)$ satisfies the Bellman equation

$$V(i) = \max_{a \in \mathcal{A}} \left[C(i,a) + \sum_{j=1}^{n} p_{ij}(a) V(j), \right] \quad i = 1, \dots, n \tag{2.7}$$

and is the unique solution of this equation.

(c) For any stationary policy $\pi = (\mu)_{0 \leqslant t \leqslant T}$ the expected total rewards $V^\mu(1), \dots, V^\mu(n)$ are the unique solution of the equation

$$V^\mu(i) = C(i, \mu(i)) + \sum_{j=1}^{n} p_{ij}(\mu(i)) V^\mu(j), \quad i = 1, \dots, n \tag{2.8}$$

Furthermore, given any initial conditions $v^0(1), \dots, v^0(n)$, the sequence $v^k(i)$ generated by the DP iteration

$$v^{k+1}(i) = C(i, \mu(i)) + \sum_{j=1}^{n} p_{ij}(\mu(i)) v^k(j), \quad i = 1, \dots, n \tag{2.9}$$

converges to $V^\mu(i)$ for each i.

(d) A stationary policy μ is optimal if and only if for every state i, $\mu(i)$ attains the maximum in the Bellman equation (2.7).

Here, the superscript k only denotes the iteration number and is not to be confused with a time index. Note that in this section we can write max and min instead of sup and inf because the state space is finite. There may be more than just one optimizing policy (that is, it may not be unique), then we simply arbitrarily choose one of the optimizing policies.

[6] Bertsekas originally considers a minimization problem. The results from the original proposition can be transferred to the maximization formulation as above by reversing the sign of the costs (minimizing $-C$ instead of C). The sign reversal preserves Assumption 1 for the stochastic shortest path problem 2.5 induced by the original maximization problem of finding the value function.

All these results hold for our finite horizon MDP problem: As noted above we can consider it as a stochastic shortest path problem. Note also that in the Dynamic Programming Algorithm (adjusted for the MDP setting) we can just reverse the time indices of the approximations v in (2.4):

$$v^{k+1}(i) := \max_{a \in \mathcal{A}} \left[C(i,a) + \mathbb{E}\left[v^k(j) \mid i, a \right] \right], \text{ for all } i \in \mathcal{S}$$

Then $v^T(i) = V(i)$ for all states i.

Thus, we see that the iteration in (2.6) is the same as the Dynamic Programming Algorithm in the finite horizon case and that the iteration will terminate if and only if $t = T$. With this indexation we also see that $v^k(i)$ is the optimal expected total reward for k-stage problem starting from the state i.

Remark 2.10. Many algorithms in the area of DP, in particular also in the area of ADP, are not necessarily backed up by theoretical results. Their popularity often stems from their performance in practice, but not rigorous mathematical arguments.

Proposition 2.9 above is the basis for the two most fundamental classical DP algorithms: Value iteration and policy iteration. A central difference to the DP algorithm (2.4) is that we now step *forward* in time instead of backward. Note that as a consequence we now do not work anymore with the exact ("real") value function but only with some approximation.

Value iteration. The iteration (2.6)

$$v^{k+1}(i) = \max_{a \in \mathcal{A}} \left[C(i,a) + \sum_{j=1}^{n} p_{ij}(a) v^k(j) \right], \quad i = 1, \ldots, n$$

is also called value iteration. Value iteration is known under many names; it is sometimes also called successive approximation, backward induction, or sometimes even dynamic programming.

In the DP algorithm (2.4), we computed the value function exactly by computing successively all possible trajectories and by taking the reward-maximizing trajectory for each initial state. In essence, the value iteration extends the Dynamic Programming Algorithm by applying it to cases where exact computation is not feasible, in particular infinite horizon problems. The difference is that we now do not compute the value function exactly, but *iteratively*. Additionally, we do not explicitly compute the value of a policy for every state, but only calculate it within the value function whenever it is needed (whenever we come upon a state). Again the Bellman equation is crucial, here it is used as an update rule.

The hope is now that with enough number of iterations, the right-hand side converges against the Bellman operator. In general, this requires infinitely many steps. Convergence towards the true value function is only guaranteed in some cases, such as in this finite horizon finite state case.

In the case of stochastic shortest path problems as above, Bertsekas [Ber17, Vol. II, Section 3.4] showed that the value iteration method will end after $|\mathcal{S}|$ iterations, if the transition probability graph corresponding to some optimal stationary policy is acyclic. In the discounted case,

monotonic error bounds for value iteration can be proved, using contraction properties due to the discount factor [Ber17, pp. 86 ff.].

Policy iteration. An alternative to value iteration is the closely related policy iteration. Instead of improving the value function approximation directly, we focus on iteratively producing improved policies. The iteration consists of two steps, which are

$$V^{\mu_k}(i) = C(i,\pi(i)) + \sum_{j=1}^{n} p_{ij}(\mu_k(i))V^{\mu_k}(j), \quad i = 1,\dots,n, \quad \textit{(policy evaluation)} \tag{2.10}$$

and

$$\mu_{k+1}(i) = \max_{a\in\mathcal{A}} \left[C(i,a) + \sum_{j=1}^{n} p_{ij}(a)V^{\mu_k}(j) \right], \quad i = 1,\dots,n, \quad \textit{(policy improvement)}. \tag{2.11}$$

The algorithm works as follows: We start with a stationary policy μ_0 and then iteratively produce new stationary policies μ_1, μ_2 and so on, in the following way: Given a policy μ_k, we find the corresponding true value function V^{μ_k} for μ_k in the policy evaluation step (2.10) (= solving an equation) based on (2.8). That is why we can write V^{μ_k} in (2.10) and (2.11) instead of v^k. Then, using that updated value function, we obtain a new policy μ_{k+1} in the policy improvement step (2.11) (= optimization, for each state we choose an action that yields the best expected reward) and return to the policy evaluation step. The algorithm terminates when the policy improvement step does not yield a new policy anymore, that is, when

$$\mu_k = \mu_{k+1}.$$

In fact, in our case (and in many other cases, too) policy iteration is guaranteed to terminate with an optimal policy. We adapt the theorem in [Ber17, Vol. I, pp.246-247] to our case. We will prove the modified version by adapting the original proof as it illustrates well how policy iteration works and represents a typical proof in dynamic programming:

Proposition 2.11. *(Convergence of policy iteration) In a finite horizon finite state problem, the policy iteration algorithm generates an improving sequence of policies μ_k, that is,*

$$V^{\mu_k}(i) \leqslant V^{\mu_{k+1}}(i), \quad i = 1,\dots,n,$$

and terminates with an optimal policy.

Proof: Let the iteration step k be fixed. First, we want to show:

$$V^{\mu_{k+1}} \geqslant V^{\mu_k} \text{ (pointwise)} \tag{2.12}$$

In order to compare successive V^{μ_k}, we express $V^{\mu_{k+1}}$ differently, using the proposition from before: Consider the sequence from (2.9),

$$v^{l+1}(i) = C(i,\mu_{k+1}(i)) + \sum_{j=1}^{n} p_{ij}(\mu_{k+1}(i))v^{l}(j), \quad i = 1,\dots,n. \tag{2.13}$$

We know that for any initial conditions $v^0(1), \ldots, v^0(n)$ the sequence generated in (2.13) converges to $V^{\mu_{k+1}}$. We set $v^0(i) = V^{\mu_k}(i)$ for all i. The idea is to infer (2.12) by inequalities between the elements of the sequence v^l.

By definition of the policy evaluation (2.10) we have for all i, since $v^0 = V^{\mu_k}$,

$$\begin{aligned} v^0(i) &= C(i, \mu_k(i)) + \sum_{j=1}^{n} p_{ij}(\mu_k(i)) v^0(j) \\ &\leqslant C(i, \mu_{k+1}(i)) + \sum_{j=1}^{n} p_{ij}(\mu_{k+1}(i)) v^0(j) = v^1(i). \end{aligned} \tag{2.14}$$

The inequality holds by definition of the policy improvement step (2.11).

Using the inequality $v^0(i) \leqslant v^1(i)$ from (2.14) we see that

$$v^1(i) = C(i, \mu_{k+1}(i)) + \sum_{j=1}^{n} p_{ij}(\mu_{k+1}(i)) \underbrace{v^0(j)}_{\leqslant v^1(j)} \leqslant C(i, \mu_{k+1}(i)) + \sum_{j=1}^{n} p_{ij}(\mu_{k+1}(i)) v^1(j) = v^2(i)$$

By iterating this we obtain for all i

$$v^0(i) \leqslant v^1(i) \leqslant v^2(i) \leqslant \ldots \leqslant v^l(i) \leqslant \ldots. \tag{2.15}$$

Thus we have

$$V^{\mu_k}(i) = v^0(i) \leqslant v^l(i) \quad \forall l \quad \Rightarrow \quad V^{\mu_k}(i) \leqslant \lim_{l \to \infty} v^l(i) = V^{\mu_{k+1}}(i).$$

The last equation follows from (c) in Proposition 2.9. Thus the policy iteration algorithm for the finite horizon finite state problem generates an improving sequence of policies.

It remains to show that the algorithm terminates with an optimal policy. This follows easily from the fact that, since we only have finite states, there are only finitely many stationary policies. Since the sequence improves, we will have reached equality after finitely many steps. That means there exists some k such that

$$V^{\mu_k}(i) = V^{\mu_{k+1}}(i).$$

Then by definition of the policy improvement (2.11) we must already have

$$V^{\mu_k}(i) = \max_{a \in \mathcal{A}} \left[C(i, a) + \sum_{j=1}^{n} p_{ij}(a) V^{\mu_k}(j) \right].$$

By Proposition 2.9 (b) and (d) V^{μ_k} is then the value function V and μ_k is optimal. □

In each iteration, we have to solve a linear system in the policy evaluation step, formed by the Bellman equations for each state i. In practice this is often a bottleneck and computationally expensive. Instead of solving this via direct methods (which would require $O(n^3)$ arithmetic operations), we typically approximate the solution. We will address this in the later chapters.

Algorithm 1: Policy iteration

Input: tolerance θ, reward function C, transition probabilities p
Output: an approximation v of the value function V, an approximation π of the optimal policy π^*

Initialize $v(s)$ and $\pi(s)$ for all $s \in \mathcal{S}$;
while *not policy-stable* **do**
  while $\Delta \geqslant \theta$ **do**
    $\Delta \leftarrow 0$;
    foreach $s \in \mathcal{S}$ **do**
      $w \leftarrow v(s)$;
      $v(s) \leftarrow C(s, \pi(s)) + \sum_{s'} p_{ss'}(\pi(s))v(s')$;
      $\Delta \leftarrow \max(\Delta, |w - v(s)|)$;
    end
  end
  policy-stable $\leftarrow$ true ;
  foreach $s \in \mathcal{S}$ **do**
    old-action $\leftarrow \pi(s)$;
    $\pi(s) \leftarrow \arg\max_a C(s, a) + \sum_{s'} p_{ss'}(a))v(s')$;
    if *old-action* $\neq \pi(s)$ **then**
      policy-stable $\leftarrow$ false
    end
  end
  if *policy-stable* **then**
    return *v and* π;
  end
end

See Algorithm 1 for a specific implementation of the policy iteration algorithm (adapted from [SB17, p.65]). Here the policy evaluation is solved approximately in an in-place iteration.

As we have seen above, policy iteration converges globally. However, we would like to have some results on the convergence rate, too. In order to do that, we can alternatively define the problem we want to solve as

$$\text{Find } v \in \mathbb{R}^N \text{s. t. } \min_{a \in \mathcal{A}^N} (B(a)v - c(a)) = 0, \tag{2.16}$$

where $\mathcal{A}$ is our finite action set as before, $B(a)$ is a monotone $N \times N$- matrix for every $a \in \mathcal{A}^N$ and $c(a) \in \mathbb{R}^N$. [7] The policy iteration algorithm solves 2.16 (analogous to before) by iterating policy improvement and policy evaluation. These two steps now take on the following form:

$$\text{Solve } a^{k+1} = \arg\min_{a \in \mathcal{A}^N} \left(B(a)v^k - c(a)\right) \quad \text{(policy improvement)} \tag{2.17}$$

[7] A matrix is *monotone* if and only if its inverse exists and is positive.

$$\text{Find } v^{k+1} \text{ s.t. } B(a^{k+1})v^{k+1} - c(a^{k+1}) = 0 \quad \text{(policy evaluation)} \tag{2.18}$$

We arrive at (2.16) by using the notation from before; a is the policy vector, c is the vector of rewards, and B is the matrix necessary for policy evaluation: As an illustration we can rewrite 2.10 as

$$\underbrace{\left(v^{\mu_k}(i)\right)_i}_{:=v^k(\mu_k)} = \underbrace{\left(C(i,\pi(i))\right)_i}_{:=C(\mu_k)} + \left(\sum_{j=1}^{n} \underbrace{p_{ij}(\mu_k(i))}_{:=p_{ij}(\mu_k)} v^{\mu_k}(j) \right)_i .$$

Then we have, for $P(\mu_k) := \left(p_{ij}(\mu_k)\right)_{ij}$ the matrix of transition probabilities,

$$v^k(\mu_k) = C(\mu_k) + P(\mu_k)v^k(\mu_k) \Rightarrow \underbrace{\left(\mathbb{1} - P(\mu_k)\right)}_{:=B(\mu_k)} v^k(\mu_k) - C(\mu_k) = 0.$$

Renaming the policy, we obtain the formula for the policy evaluation (2.18):

$$B(a)v^k - C(a) = 0.$$

As for the policy improvement, we want to minimize the term in (2.17) because

$$\begin{aligned} \min_{a\in\mathcal{A}^N}(B(a)v^k - c(a)) &= \min_{a\in\mathcal{A}^N}(v^k - (Pv^k + C(a))) = \max_{a\in\mathcal{A}^N}(-v^k + (Pv^k + C(a))) \\ &= \max_{a\in\mathcal{A}^N}(Pv^k + C(a))), \end{aligned} \tag{2.19}$$

as v^k is constant (independent of a). The last expression in (2.19) is then equivalent to (2.11). The idea behind expressing the problem as (2.16) is that it is then in the right form to apply Newton's method to it.

Puterman and Brumelle [PB79] were among the first to show that under certain conditions policy iteration is equivalent to Newton's method. However, they used hard-to-verify Lipschitz order conditions and also required the exact computation of the value functions. Bokanowski et al. [BMZ09] followed this up with a global superlinear convergence result for easier-to-verify assumptions, in particular a monotonicity assumption on the matrices B. They compare policy iteration to a semismooth Newton method based on the concept of slant differentiability. We note that in the discounted case, B always satisfies this assumption. However, this does not necessarily hold for the undiscounted case.

Unfortunately, there is evidence that not all MDPS fulfill either of these assumptions mentioned above. There are examples where a finite state MDP problem with N states requires about N policy iteration steps to converge to the optimal policy - and there might even be less-behaved examples [SR04, pp.4-5]. So there is no general result that policy iteration can converge in a small number of steps, that is, policy iteration is only efficient in some cases.

Comparison of value and policy iteration. We can actually view value iteration as a form of policy iteration: It equals policy iteration with a truncated policy evaluation, where we just have one update per step. Value iteration combines both steps (policy evaluation and policy

improvement) in *one* update rule by directly maximizing in the policy evaluation step. Instead of computing the policy explicitly for each i, we just compute it within the right-hand side of (2.10) whenever needed, that is, whenever we come upon state i.

This leads to different convergence behaviour: Policy iteration tends to converge fast in number of iterations, but each iteration takes much longer and is much more expensive. In the discounted infinite horizon case, policy iteration is guaranteed to find an optimal stationary policy in a finite number of steps, while value iteration generally needs infinitely many.

Remark 2.12. A question that naturally arises: Why do we need value or policy iteration at all? After all, for the finite horizon finite state MDP problem from above, we can just use the DP Algorithm to solve it exactly. Furthermore, as we have seen in Proposition 2.9 and the considerations preceding Remark 2.10, value iteration is basically the same as the DP Algorithm (for finite horizon finite state MDPs). We have also seen that policy iteration converges after finitely many steps. One might wonder, therefore, why we introduced them, considering that they are very similar to the DP Algorithm and are also not in any way less computationally expensive.

Indeed, for finite horizon finite state MDPs, policy iteration and value iteration are somewhat trivial. Nevertheless, we chose the framework of finite horizon finite state MDPs as it is well suited for demonstrating how these algorithms work. Moreover, value and policy iteration are useful because they can be applied in settings *where the DP Algorithm is not applicable anymore*. An example of this is the infinite horizon case, which we briefly discuss below.

Convergence for the infinite horizon case. As we will later on consider the finite horizon case, we only considered the finite horizon cases for the DP algorithms. Of course, for a finite horizon we technically do not need either value or policy iteration, we can simply use exact recursion. The DP algorithms were originally developed for the infinite horizon case where exact recursion (and going through all stages) is impossible. One might wonder why it is interesting to consider the infinite horizon case at all, because in practice horizons are always finite. However, as it turns out, modelling with an infinite horizon can be a better representation for very long planning periods or when the termination is contingent on the achievement of some goal where the achievement of the goal is not associated to a certain time stage (as with stochastic shortest path algorithms).

Classical DP results for policy iteration and value iteration are centered upon the infinite horizon case. We outline key results. We first make some general remarks and classifications and then present the convergence results for certain classes in both the cases with finitely many and infinitely many states.

In the following we assume a stationary system (no time-dependence). The infinite horizon reward (in this paragraph we will just use "reward" instead of "total expected reward") of a given policy is defined as the limit of its corresponding N-stage rewards where N goes to infinity: $V^\pi(s) = \lim_{N\to\infty} V_N^\pi(s)$.

We would like the following results to hold:

(A) *Convergence of the DP Algorithm:* The optimal infinite horizon reward is the limit of the corresponding N-stage optimal rewards,

$$V(s) = \lim_{N\to\infty} V_N(s).$$

(B) *Bellman equation:* The optimal reward satisfies the Bellman equation,

$$V(s) = \sup_{a\in\mathcal{A}} \mathbb{E}_{\xi} \left[C(s,a) + V(g(s,a,\xi))\right].$$

(C) *Characterization of optimal stationary policies:* If $\mu : \mathcal{S} \to \mathcal{A}$ attains the optimum on the right-hand side of the Bellman equation for all states s, the stationary policy $\pi = \{\mu, \mu, \ldots\}$ should also be optimal.

As we will see, these results only hold for some cases.

Bertsekas [Ber17, Vol.I, p.235] classifies infinite horizon total reward problems as follows (using the terminology of costs):

- *contractive problems* (including discounted problems with bounded cost per stage) - they are "well-behaved"),
- *noncontractive problems*, and
- *semicontractive problems* (including the stochastic shortest path problems from above[8])- here the contraction property is only valid for a subclass of policies satisfying additional constraints, in the best cases we can restrict the search for an optimal policy to this subclass.

First, we consider the convergence results for the case with a finite number of states. For stochastic shortest path problems, the results (A) - (C) hold. Value iteration only terminates finitely for some special cases and needs in general infinitely many iterations. As we have seen (the proof of Proposition 2.11 is the same for the infinite horizon case), we have finite termination for policy iteration. Discounted problems can just be converted into stochastic shortest path problems (by suitably adding the discount factor to the transition probabilities), therefore the same results hold.

Now we consider the case with an infinite number of states. The discounted case with bounded reward per stage is the "well-behaved" case, the results (A) - (C) hold. The undiscounted case with unbounded reward per stage (as we have in our energy storage model, though with a finite horizon) is much more complicated as the value function can now also be infinity for some states. These problems do not have a contraction property, instead we need additional assumptions which enforce monotonicity properties. The Bellman equation (B) still holds, and value iteration is valid (under additional assumptions). However, policy iteration is not valid anymore (the main problem is that (C) does not hold anymore).

For an in-depth analysis of the infinite horizon case with many more results and examples we refer to [Ber17, Vol. II].

[8] While not fully contractive such as discounted problems, here ρ performs a similar function to a discount factor.

Operator notation. Another way to express DP results is to write them down in operator form. This shorthand notation is often found in books, and can simplify expressions especially for the infinite horizon case. The central component is the mapping T, which is motivated by the DP Algorithm. It is defined by:

$$\forall s \in \mathcal{S} : (T(V))(s) := \sup_{a \in \mathcal{A}} \left[C(s,a) + \mathop{\mathbb{E}}_{\xi} \left[V(g(s,a,\xi)) \right] \right],$$

where g is the transition function from (2.1) and $V : \mathcal{S} \to \mathbb{R}$ is a scalar function (the value function)[9]. For readability, we will just write TV instead of $T(V)$ in the following.

The operator T is a transformation of V. It is an application of the DP Algorithm: TV is the optimal expected reward for the two-stage problem with reward C in the first stage and reward V in the second stage. Similarly, we can define an operator T^μ for a given stationary policy μ:

$$\forall s \in \mathcal{S} : (T^\mu(V))(s) := C(s,\mu(s)) + \mathop{\mathbb{E}}_{\xi} \left[V(g(s,\mu(s),\xi)) \right],$$

We define for T (and similarly for T_μ)

$$T^k V := T(T^{k-1}V), \quad T^0 V := V.$$

Using induction, one can easily show the following monotonicity property, which plays a crucial role in convergence proofs in Dynamic Programming:

Theorem 2.13 (Monotonicity Lemma)**.** *[Ber17, Vol.II, Lemma 1.1.1]*
For any functions $J, J' : \mathcal{S} \to \mathbb{R}$, such that $J \geqslant J'$ pointwise for all $s \in \mathcal{S}$ and any stationary policy $\mu : \mathcal{S} \to \mathcal{A}$, we have for $k \in \mathbb{N}$

$$T^k J \geqslant T^k J', \quad T_\mu^k J \geqslant T_\mu^k J' \quad \textit{(pointwise).}$$

In the case with finitely many states, $\mathcal{S} = \{1, \ldots, n\}$, we can write down the expectation explicitly:

$$\forall i \in \mathcal{S} : (TV)(i) := \max_{a \in A} \left[C(i,a) + \sum_{j \in S} p_{ij}(a) V(j) \right]$$

We can now use these operators to simplify the notation for value and policy iteration:

- We can express value iteration as starting with an initial value function guess V_0 and then just successively computing $TV_0, T^2V_0, T^3V_0, \ldots$.
- In policy iteration we start with a stationary policy μ_0. Then we iterate the following two steps:
  1) determine V^{μ_k} as the solution of $T^{\mu_k} J = J$ (policy evaluation)
  2) determine the new policy μ_{k+1} which is defined by $T^{\mu_{k+1}} V^{\mu_k} = TV^{\mu_k}$ (policy improvement)

[9] We implicitly add sufficient assumptions such that the term is well-defined.

The DP results (A) - (C) from before can also be simplified:

(A') *Convergence of the DP Algorithm:* The optimal infinite horizon reward satisfies $V(s) = \lim_{k\to\infty} T^k V(s)$.

(B') *Bellman equation:* The optimal reward satisfies the Bellman equation $V = TV$.

(C') *Characterization of optimal stationary policies:* A stationary policy $\mu : \mathcal{S} \to \mathcal{A}$ is optimal if and only if $T^\mu V = TV$.

2.3 Post-Decision States

In this section we explain the method of post-decision states. They are an alternative way to formulate the Bellman equation and are used in (Approximate) Dynamic Programming. In particular, we will use them later in the algorithm for our energy storage problem.

Recall that the central relation between the value functions is the Bellman equation

$$V_t(s_t) = \max_{x_t \in X_t} \left[C(s_t, x_t) + \mathbb{E}\left[V_{t+1}(s_{t+1}) | s_t \right] \right]. \tag{2.20}$$

Note that the expectation is inside the maximization operator. Unfortunately, the expectation can be difficult or even impossible to compute. Consequently, this makes the optimization more difficult.

A trick to avoid this is to use *post-decision states* (sometimes also called *afterstates*, see for example [SB17]). The idea is to use the fact that the transition function consists of a deterministic and a stochastic component. Considering the discrete system dynamics 2.1, we define the post-decision state as

$$s_t^{a_t} := s_t + f(s_t, a_t), \tag{2.21}$$

where a_t is the decision taken at time t based on knowledge of the state s_t. Powell [Pow07, Section 4.6] accordingly calls the "normal" state (that is, s_t, or what we called state before), the *pre-decision state*. With post-decision states we separate the effect of the decisions and the incoming stochastic information: We see in (2.21) that the post-decision state captures only the deterministic effect of the decision a_t. Then, by definition, the difference between being in the post-decision-state $s_t^{a_t}$ and the state s_{t+1} is the exogenous information that has arrived between time t and $t+1$:

$$s_{t+1} = s_t^{a_t} + \xi_{t+1} \tag{2.22}$$

We can now formulate the optimality equation using post-decision states. Equivalently to V_t, we now define V_t^p, the *post-decision value function*: By 2.22 we have

$$V_t^p(s_t^{a_t}) = \mathop{\mathbb{E}}_{\xi_{t+1}} \left[V_{t+1}(\underbrace{s_t^{a_t} + \xi_{t+1}}_{=s_{t+1}}) | s_t^{a_t} \right]. \tag{2.23}$$

Plugging 2.23 into the Bellman equation we obtain

$$\begin{aligned} V_t(s_t) &= \max_{a_t \in \mathcal{A}_t} \left[C(s_t, a_t) + \mathbb{E}\left[V_{t+1}(s_{t+1}) | s_t \right] \right] \\ &= \max_{a_t \in \mathcal{A}_t} \left[C(s_t, a_t) + \mathbb{E}\left[V_{t+1}(s_{t+1}) | s_t^{a_t} \right] \right] \\ &= \max_{a_t \in \mathcal{A}_t} \left[C(s_t, a_t) + V_t^p(s_t^{a_t}) \right] \end{aligned} \tag{2.24}$$

We can use $s_t^{a_t}$ instead of s_t in the second equation because in the first equation we already implicitly condition on a_t in the computation of the expectation $\mathbb{E}\left[V_{t+1}(s_{t+1}) | s_t \right]$ (due to being in the maximization operator).

Instead of plugging 2.23 into the Bellman equation we can also plug the last equation in 2.24 into 2.23 to obtain the post-decision version of the Bellman equation. Reducing the time index by one we obtain

$$V_{t-1}^p(s_{t-1}^{a_{t-1}}) = \mathbb{E}\left[\max_{a_t \in \mathcal{A}_t} \left[C(s_t, a_t) + V_t^p(s_t^{a_t}) \right] | s_t^{a_t} \right]. \tag{2.25}$$

The equations 2.23 and 2.24 show that the Bellman equation for the pre-decision state and the Bellman equation for the post-decision state are equivalent.

The main advantage of using post-decision states is the computational benefit: We avoid the repeated computation of the expectation within the optimization. Instead, we only need to compute the expectation once in 2.25, that is, once after we have solved the optimization. This is beneficial if the computation of the expectation is difficult or impossible (for example the case when the outcome space is very large, or when the transition probabilities are unknown). (2.25) also means that we now have a *deterministic* optimization.

Even in the deterministic optimal control case it can be beneficial to use post-decision states. We illustrate this with the board game tic-tac-toe: We define the pre-decision state as the state of the board we see before we make a move, the post-decision state is then the state of the board our opponent sees when he or she makes a move. For a fixed state, there are more actions than post-decision states, because multiple actions can lead to the same post-decision state:

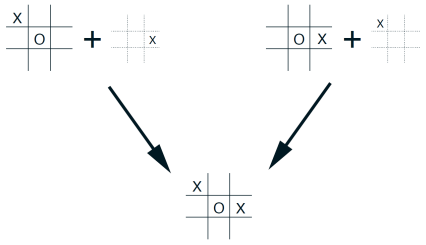


Figure 2.1: Figure from [SB17, p.112]

We thus see that an advantage of post-decision states is that, compared to the optimization in (2.20), we only have to optimize over a potentially much smaller set in (2.25). This is another computational benefit and holds in the stochastic case as well. Related to this, the post-decision state variable often has a smaller dimension than the pre-decision state variable.

Despite these benefits, post-decision states are not very common in the approximate dynamic programming literature. Powell [Pow07] attributes this to the fact that historically in the development of Markov Decision Processes the ability to compute the expectation exactly was taken for granted. We refer to [Pow07] or [SB17] for more details on post-decision states as well as example applications.

3 Model of the energy storage problem

In this chapter we present the Approximate Dynamic Programming (ADP) model of the energy storage problem and motivate the modelling decisions.

3.1 The ADP model

The following model of energy storage is taken from Jiang et al. [JPP+14], with minor modifications.

We now apply the concepts from Chapter 2 to our specific application: We model the energy storage problem as a discrete time, finite horizon, and undiscounted stochastic optimal control problem. We consider the optimization problem of a firm which operates an electric utility based on wind energy. Given generated electricity, the firm faces two options: it can either use the electricity directly to satisfy demand or the firm can store it for later use at a certain cost. The firm's objective is to maximize its profit, therefore "optimally controlled" equals maximal profit here.

The state variable consists of the four factors that influence which decisions the firm can make at time t:

- r_t, the *storage level* at time t
- w_t, the amount of *energy* available from wind at time t
- p_t, the *price* of energy at time t
- d_t, the size of *demand* that must be satisfied at time t

As units we choose megawatt hour (MWh) for r_t, w_t and d_t as well as \$ / MWh for p_t, but the exact units are not of importance. All of the four variables are modelled as non-negative and bounded at every time ($x_t \in [x_{min}, x_{max}]$ for $x \in \{r, w, p, d\}$). The bounds are modelled both for computational purposes as well as real-world constraints (such as that storage facilities have limited capacity).

Our *state* at time t is then

$$s_t = (r_t, w_t, p_t, d_t).$$

The state s_t consists of two parts: r_t, and (w_t, p_t, d_t). The storage r_t constitutes the endogenous part, it is the only one we can control. Wind energy, price and demand constitute the *exogenous information*; they develop stochastically. An example of actual whole-sale electricity prices for an interval in July can be seen in Figure 3.1 (figure taken from [Sma]).

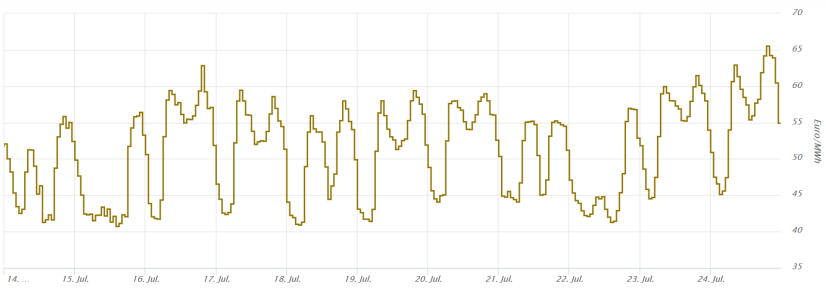


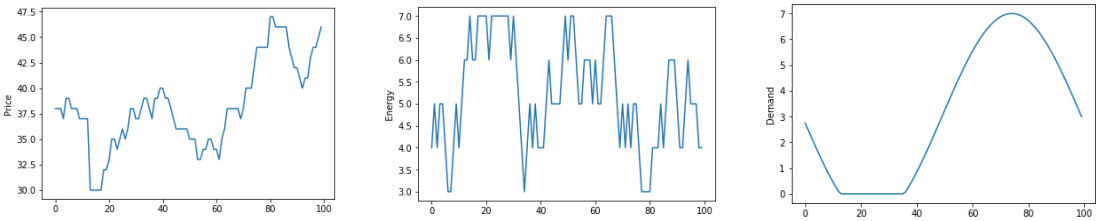
Figure 3.1: Example for actual whole-sale electricity prices on the day-ahead market for July 2018 (taken from [Sma])

Formally, we define the random variable $z_t := (w_t, p_t, d_t)$ (the exogenous information), and the random variable $\hat{z}_{t+1}$ as the incremental change in wind energy, price and demand between times t and $t+1$:

$$z_{t+1} = z_t + \hat{z}_{t+1}.$$

We define $\mathcal{F}_t := \sigma(\hat{z}_1, \ldots, \hat{z}_t)$. Then $\mathcal{F}_0 \subset \mathcal{F}_1 \ldots \subset \mathcal{F}_T$ is a filtration. We adopt the convention that all variables with index t are $\mathcal{F}_t$-measurable. Our problem is then defined on a probability space $(\Omega, \mathcal{F}, P)$, where $\mathcal{F} := \mathcal{F}_T$. Furthermore we assume that w_t, p_t, and d_t develop independently of each other and independently of r_t.

In the numerical implementation, different processes and functions were used to model available wind energy, price and demand. For an example of how sample paths of these processes looked like in the implementation, see Figure 3.2.



(a) Price **(b)** Energy **(c)** Demand

Figure 3.2: Example samplepaths (over the full time period, $T = 100$)

Apart from the capacity constraints the storage device is also characterized by its *(dis)charge efficiency*, its *maximal (dis)charge* and the *holding cost*. The (dis)charge efficiency is denoted by β_d and β_c, where $\beta_d, \beta_c \in (0, 1]$, and the maximal amount that can be (dis)charged is denoted by γ_d and γ_c. The *holding cost* is denoted by c_h per stored unit of energy. The existence of the (dis)charge efficiency and the holding cost together represent that storage is costly. Storing in itself incurs costs (for instance rent or maintaining the storage facility), and we additionally face conversion losses when we store energy (represented by the efficiencies).

Considering this, the firm has six decisions to make at time t, denoted as

$$a_t = (a_t^{wd}, a_t^{rd}, a_t^{gd}, a_t^{wr}, a_t^{gr}, a_t^{rg}). \tag{3.1}$$

The double indices denote the origin i and the destination j of the energy amount x^{ij}. Here r stands for storage (repository), d for demand, w for wind energy and g for grid. Figure 3.3 depicts how storage, wind, demand and grid are related through the possible decisions: Demand can be satisfied by either of the other three, wind energy can be used for demand or for storage, storage and grid can interact in both ways.

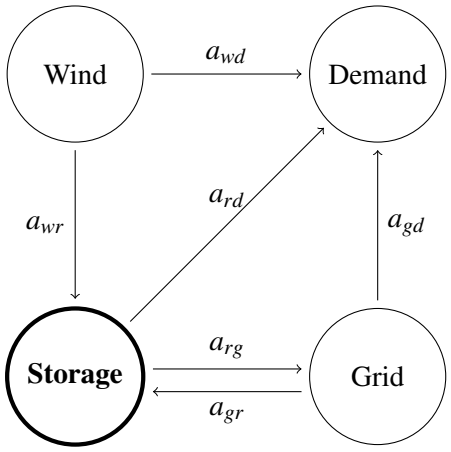


Figure 3.3: The relation between the state components

A decision a_t at time t is *admissible* if it fulfills the following constraints:

$$\begin{aligned} a_t^{wd} + \beta_d a_t^{rd} + a_t^{gd} &= d_t, \\ a_t^{wr} + a_t^{wd} &\leqslant w_t, \\ a_t^{rd} + a_t^{rg} &\leqslant \min\{r_t, \gamma_d\}, \\ a_t^{wr} + a_t^{gr} &\leqslant \min\{r_{max} - r_t, \gamma_c\}. \end{aligned} \tag{3.2}$$

In words: The first constraint states that demand must always be satisfied, either from the energy source, the storage or the grid (and for the storage we have to factor in the conversion loss). The second constraint implements the definition of the variables a_t^{wr} and a_t^{wd} by limiting their sum to the available wind energy. The last two constraints express the constraints of the storage device; we can only discharge what is stored and we can only charge till the maximal capacity is reached, while simultaneously being bound by the (dis)charge limitations.

We define $\mathcal{A}_t(s_t)$ as the space of all admissible decisions at time t given state s_t,

$$\mathcal{A}_t(s_t) := \{a_t : a_t \text{ satisfies (3.2)}\}.$$

Each decision potentially affects the current storage. We describe this deterministic part of the system dynamics in the *transition function*:

$$T(s_t, a_t) := r_t + \beta_c a_t^{wr} + \beta_c a_t^{gr} - a_t^{rd} - a_t^{rg}.$$

We obtain it by simply adding energy coming from the energy source or grid, and subtracting energy sold to the grid or to satisfy demand. Note that a_t^{wr} and a_t^{rg} are not multiplied by β_d, since

here only the *extracted* amount matters, not the delivered amount. Then the storage in the next time period can be written as $r_{t+1} = T(s_t, a_t)$. Alternatively, in order to see the structure clearer, we can also write

$$T(s_t, a_t) := r_t + \phi \cdot a_t, \tag{3.3}$$

where $\phi = (0, -1, 0, \beta_c, \beta_c, -1)$ is the fixed transition vector.

Then the *system dynamics* for all four state components are

$$s_{t+1} = g(s_t, a_t, \xi_{t+1}) := s_t + f(s_t, a_t) + \xi_{t+1},$$

where $\xi_{t+1} = (0, \hat{w}_{t+1}, \hat{p}_{t+1}, \hat{d}_{t+1}) = (0, \hat{z}_{t+1})$ are the independently distributed shocks for wind, price and demand, and

$$f(s_t, a_t) = \begin{pmatrix} T(s_t, a_t) \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Here, the *post-decision state* is simply

$$s_t^a = s_t + f(s_t, a_t) = (r_t + T(s_t, a_t), w_t, p_t, d_t).$$

That is, the only difference between the pre-decision state s_t and the post-decision state s_t^a is the difference in storage.

In this optimal control problem, the *reward function* is the profit:

$$C(s_t, a_t) := p_t(d_t + \beta_d a_t^{rg} - a_t^{gr} - a_t^{gd}) - c_h \underbrace{(r_t + \beta_c a_t^{wr} + \beta_c a_t^{gr} - a_t^{rd} - a_t^{rg})}_{=T(s_t,a_t)=r_{t+1}}.$$

The profit is obtained in the following way:

$$\begin{aligned} C &= \text{profit} \\ &= \text{revenue} \quad - \quad \text{costs} \\ &= \text{sold electricity} - (\text{bought electricity} + \text{storage costs}). \end{aligned}$$

The goal is to find an admissible policy which maximizes expected profits, the *objective function* to be optimized is therefore the expected cumulated profit

$$\mathbb{E}\left[\sum_{t=0}^{T} C(s_t, a_t)\right].$$

Note that we do not discount future rewards since we optimize over a finite time horizon. Only the total sum of rewards matters for the firm. Having a finite time horizon also allows the policies to be time-dependent.

Properties of the ADP energy storage problem. From the application perspective, our problem belongs to dynamic resource allocation and inventory problems. This includes problems such as scheduling people and machines or supply chain management. Related problems are optimization of financial portfolios, demand management or engineering control problems such as optimizing the flight of an aircraft (see [Pow11] for a concise general overview).

Furthermore, we can see that the (post-decision) value function has two important features, namely concavity and monotonicity:

Proposition 3.1. *For each time $t \leqslant T-1$, the post-decision value function $V_t^p(r^p, z_t)$ is concave in r_t^p, where r_t^p is the post-decision storage variable and $z_t = (w_t, p_t, d_t)$.*

Proof: Using the post-decision value function and (2.24) we can write our problem as

$$V_t(s_t) = \max_{a_t \in \mathcal{A}_t} C(s_t, a_t) + V_t^p(s_t^{a_t}) \quad \forall t \in T.$$

This in turn can be written as a *deterministic* maximization of a linear program where the constraints 3.2 are written in the form $Ax \leqslant b$. Then the resource variable is part of the right-hand side constraint b. For maximizing linear programs the objective value function (in this case V_t) is concave in the right-hand side constraints b [Van01]. □

Proposition 3.2. *[JPP+14] For each time $t \leqslant T-1$, the post-decision value function $V_t^p(r^p, z_t)$ is nondecreasing in r_t^p, where r_t^p is the post-decision resource variable and $z_t = (w_t, p_t, d_t)$.*

Note however, that the proof of Proposition 3.2 in [JPP+14] only holds if there are no holding costs[10].

3.2 Modelling the energy trade and storage

In this subsection, we motivate the model and explain how it relates to energy storage in practice.

Energy market and trade. We first need to gain an understanding of how the general wind energy market works. The exact design of energy markets varies slightly across the world. We therefore focus on the main characteristics which most energy markets share, to allow for a wide applicability.

In a electrical power system, also called (power) *grid*, there are three main processes: generation, transmission and distribution. All these processes can be carried out by separate firms. After generation, the generated energy can be traded on the whole-sale electricity market. There, electricity generators can sell their output to retailers who in turn sell the electricity to end-users. The electricity is then delivered to the end-users via transmission and distribution. Trade takes place in large part over-the-counter transactions, the rest is traded on the power exchange. Trade

[10] In the respective proof (proof of Proposition 1 in [JPP+14]) the argument for both inclusions relies on the fact that the contributions $C(S_t, x_t) = C(\bar{S}_t, x_t)$ and $C(S_t, x_t) \leqslant C(\bar{S}_t, \bar{x}_t)$ which both break down if there is a holding cost: Then $C(S_t, x_t) > C(\bar{S}_t, x_t)$ and in the relation the inequality can go in either direction, depending on the size of the storage and the holding cost.

at the power exchange is organized into two distinct groups: long-term delivery (*futures market*) and short-term delivery (*spot market*).

On the futures market long-term contracts are traded which specify the constant amount of electricity the seller provides each day for a certain amount of time, usually months or years.

The spot market is further divided into *intra-day trade* and *day-ahead trade*. The intra-day ahead takes place continuously. Its main function is compensatory; it can compensate deviations from the original schedule. For example, firms can buy electricity to satisfy demand despite sudden and unexpected shortages. In the day-ahead market electricity is typically sold in hour-blocks for the following day. Demand and supply are matched once a day via a blind auction [11], the electricity price is then determined as the the market clearing price for each hour of the following day (that is, the market clearing price is the price for which supply equals demand). The same categories exist for over-the-counter trades. The only difference is that the products traded there tend to be less-standardized and that trade takes place directly between the trade partners, that is, without the power exchange as an intermediary [PIB, WID].

For renewable energies the futures market does not play a significant role as renewable energy sources like wind are inherently volatile and unpredictable. Trading of energy from renewable sources thus takes place on the spot market. We therefore focus on the spot market, which is also a common choice in the literature [KP11, HD11]. As trade takes place only once a day in the day-ahead market and only in hour blocks, we model time discretely.

Note that here, that the demand d_t from above equals the commitment that has been made in the auction of the day-ahead trade the day before. This commitment has to be fulfilled (thus also the binding equality constraint in (3.2)). If the current energy production is too low, we have to buy compensatory energy on the intra-day market which is denoted as the grid g in the model above.

Energy storage devices. In our model we characterize energy storage devices by five parameters: its charge and discharge efficiency, the maximal amount of charge and discharge (injection and withdrawal capacity), as well as its maximum capacity. The model is therefore not limited to a specific application and can be used for all kinds of storage devices. Common energy storage devices are hydroelectricity in the form of pumped storage or dams, batteries or hydrogen in compressed or liquid form. Their *energy efficiency* (i.e., the ratio of energy out per energy in, also known as round-trip efficiency) varies hugely, hydroelectricity and batteries have an efficiency of 70%-80%, while hydrogen only has an efficiency of 20%-45%. They also vary strongly in capacity and costs.

Modelling decisions. We make the following modelling decisions and assumptions:

- In optimal control the state space is typically modelled as continuous. Here our state represents the physical state - storage level, available wind energy, demand, price - which in practice develops continuously (in space) and which we therefore also model continuously (but still discrete in time).

[11] In a blind auction all bids are given simultaneously, so that no bidder knows the bid of any other bidder.

- We assume that the day-ahead and the intra-day ahead price are the same.
- We assume that there is some distribution modelling how much energy we will commit to for each time period. Additionally, uncertainty of the commitment has to be factored in, as we only make a commitment if the price in our bid is lower than the market clearing price. Therefore, the demand or commitment d_t can be modelled stochastically.
- We assume that the firm is only a small player in a large market. This means that its bid has only a negligible influence on the overall price. We therefore consider it as a price taker and model the price as an exogenous factor. (In other words, we consider the market to be competitive and to consist of many market participants.)
- We make the simplifying assumption that all the exogenous information components are independently distributed. In particular, price and energy are modelled as independent, as well as price and demand (commitment). In reality however, prices on the spot market are also influenced by physical circumstances such as available wind energy. Kim and Powell [KP11] argue that price and wind energy can be modelled as independent because the electricity price is mainly influenced by controllable energy sources such as coal plants.
- In general, costs are composed of variable and fixed costs. We neglect fixed costs as they do not affect the behaviour of an optimal policy ("sunk costs"). The only variable costs we consider are the holding costs for storing electricity. While wind turbines are capital-intensive, they require only a negligible amount of maintenance once installed (unlike for example coal which needs to be extracted at a cost).

4 Approximate Dynamic Programming

The Dynamic Programming Principle (2.3) is the central building block of Dynamic Programming algorithms, as it breaks the original optimizing problem into smaller subproblems which are much easier to solve. Specifically, as introduced in Chapter 2, we can simply solve the problem by backward induction in the discrete time case with finite horizon. That means, since $V_T \equiv 0$, we can solve the equation for V_{T-1} for every state. Knowing V_{T-1}, we can in turn solve the equation for V_{T-2} for every state. We do this iteratively till we reach the initial time $t = 1$. In the infinite horizon case we can often use policy iteration or value iteration to obtain the optimal value functions.

However, in practice this is often not computationally tractable - if possible at all! Specifically, solving the problem via backward induction requires computing the value function for every single state and computing the expectation. Solving the problem with the classical DP algorithms at least requires a finite state space and we still need to repeatedly update the values for every single state.

For our energy storage problem, this poses a problem:

- We do not know the underlying distribution of the exogenous information (wind, energy, demand), therefore we can not compute the expectation.
- We have a *continuous* state space. It is consequently impossible to compute the value function exactly for every state via backwards induction.

The natural approach is then to *approximate* both the expectation and the update of the value function. The development of approximate versions of the classical DP algorithms constitutes the field of *Approximate Dynamic Programming*[12] (ADP). [13]

In this chapter, we mainly use [Pow07] as a rough guideline, in particular Chapters 6 and 8.

4.1 Approximating the components of DP

There is of course a multitude of ways in which the DP methods can be approximated and generalized; we can approximate the value function, the policy, the computation of the expectation, and much more. Arguably the central question in ADP is in general how to approximate the value function.

In the first section, we will go through some common ways to approximate the DP components and some crucial features to consider when designing ADP algorithms: model-based versus model-free, problem approximation, policy types, on-policy and off-policy learning, approximating the value function, and obtaining value estimates on which these value function approximations are based on. In the second section we will discuss the properties of such an ADP algorithm and how to evaluate its quality.

[12] Depending on the literature, it is also called *neuro-dynamic programming*, *reinforcement learning* or *adaptive dynamic programming*.

[13] As an alternative approach to ADP, a lot of DP problems can also be equivalently reformulated as *nonlinear (stochastic) programming* problems. Nonlinear programming is often used in practice, especially in the deterministic case with continuous states and controls.

What we want or need to approximate also depends heavily on the specific problem and how much we know. Reasons for using ADP can be like above that the state space is too large or that we do not know the distribution of the exogenous information. It can also be that some of the other components are unknown, for example the transition function. Each of these cases requires a specific kind of approximation. If the state space is too large, reducing the dimensionality of the problem or the state space is an option, while this does not help in the case where the transition function is unknown. If the transition function is unknown, we might either try to model it or simply work with samples.

Model-based and model-free. Depending on we what we know, we talk about *model-based* or *model-free* algorithms. As the name suggest, model-free methods are not based on a specific model and work on a trial-and-error basis. Instead of working with the transition function and the reward function, they only work with samples (however, some model-free methods try to learn (approximate) the model based on the samples). Prominent model-free examples are Monte Carlo methods and temporal difference methods. Model-based algorithms, such as dynamic programming, "plan"; that is they use and improve a certain policy. Model-free methods can be useful for complex problems where finding a sufficiently accurate model is difficult.

The two approaches share many similarities, in particular the approximation of the value functions is a central component for both. Sutton and Barto discuss the similarities between model-based and model-free algorithms and unify both methods in a certain framework [SB17, Chapter 8].

Problem Approximation. In problem approximation we simplify the structure of the original problem. We discuss two ways: discretization and aggregation.

If we discretize the state space, then we can solve the discretized problem exactly. A problem however is that this approach is quickly affected by the curse of dimensionality: Adding more dimensions or expanding the state space leads to an exponential increase in its states. The approach therefore does not scale well and quickly becomes computationally intractable.

For example, let us consider a finite horizon optimization problem with T stages and continuous state, control and exogenous information space $\mathbb{R}^n, \mathbb{R}^m, \mathbb{R}^r$. We can then for example discretize these spaces as grids, with d points in every dimension. Let us consider the costs of exactly computing an optimal policy for the discretized problem with the Dynamic Programming Principle. For every stage and for every state we have to solve a minimization problem over all possible actions, that is, we have to compare d^m values. Each of these values is an expectation that has to be computed; we have to sum up over d^r numbers. As this optimization has to be computed for every stage and every state, we can end up with up to Td^{n+m+r} operations, with the lower bound of Nd^n operations. There are of course also other discretization methods like sparse grids which require considerably less points. Still, the curse of dimensionality affects all discretization methods.

In *aggregation* we simply combine states into aggregate states. Again, the goal is to obtain a system of smaller dimension which we can solve exactly. While discretization transforms continuous models into discrete ones, we can also apply aggregation to already discrete spaces in order to reduce their size. In aggregation methods, we choose certain probability distributions

(called dis-/aggregation probabilities) to relate the original state space with the aggregate state space. For more details we refer to [Ber17, Section 6.2.3].

Value Function Approximation (VFA). Arguably the most popular approach for solving complex ADP problems is to approximate the value function by some function $\tilde{V}$. There are many different types, they can be broadly sorted into three categories: lookup table methods, parametric models, and non-parametric models.

Lookup table methods are the simplest way of representing value functions. They assume no model or structure, and instead explicitly store a value $\tilde{V}(s)$ for each state s individually. In particular, they are only applicable to discrete state spaces. As another consequence, the update of a state does not contain any information on the value of any other state. We therefore need to visit each state often enough, in order to obtain a good approximation of the true value function. As a drawback, the method becomes computationally expensive as we estimate as much parameters as we have states (as with discretization in general, lookup tables are vulnerable to the curse of dimensionality). An advantage is however that we can approximate any value function arbitrarily well if we only update the states enough.

In *parametric models* the value function is approximated by a linear combination of basis functions. We first identify a set of *features* $\mathcal{F}$. That is, we reduce our potentially high-dimensional state variable to its values with regard to certain properties (features) we consider important. We use the *basis functions* $\{\phi_f\}_{f\in\mathcal{F}}$ to draw the feature information from the states. Then the value function approximation is of the general form

$$\tilde{V}(s) = \sum_{f\in\mathcal{F}} \theta_f \phi_f(S), \tag{4.1}$$

where the θ_f are the weights of the basis functions. The goal is then to estimate the parameter vector θ. The idea is that the number of features is much lower than the number of states ($|\mathcal{F}| \ll |\mathcal{S}|$) and we thus need only few observations to estimate the parameters. Thus, especially in contrast to lookup tables, parametric models are computationally comparatively cheap. A drawback is however that we can only achieve an accurate approximation if we have chosen the right basis functions. Additionally, there is no canonical way to choose the right features. Powell [Pow07, p.236] calls choosing the features an "artform" and "finding θ [...] the science within the art".

An approximation such as in (4.1) is called *linear*. Note however that the the approximation is not linear in the states themselves - rather, it is only linear in the basis functions (that is, the basis function may be nonlinear). Examples for parametric models are for example linear regression or support vector regression. Parametric models are widely used because they are a simple and compact form of representation.

Nonparametric models are basically applications of nonparametric statistics. Examples here are k-nearest neighbour, kernel regression, (local) polynomial regression or neural networks. In a sense, we can generally consider VFAs as statistical models that we update using statistical techniques. For the statistical methods mentioned here and in the paragraph above we mention [HTF09] for further reference, which covers a breadth of statistical methods.

Nonparametric models are fundamentally different from parametric models: Instead of fixing one model for all states, we have in some sense a different function for every state. In nonparametric models, "the data forms the model". The approximation of the value function at a state s is computed as a weighted estimate of local observations. That means, given our observations up till this point, where we denote the observations as $\{\hat{v}_{i=1}^N\}$ corresponding to some states $\{s\}_{i=1}^N$, we approximate the value function at s using

$$\tilde{V}(s) = \frac{\sum_{i=1}^N w_h(s, s_i)\hat{v}_i}{\sum_{i=1}^N w_h(s, s_i)}.$$

Here w_h is the weighting function, which specifies how much the observation from a specific state enters into the approximated value $\tilde{V}(s)$. It is parameterized by a *bandwidth* h which determines the range of states that influence $\tilde{V}(s)$. A typical choice is the Gaussian density

$$w_h(s, s_i) = \exp - \left(\frac{s - s_i}{h}\right)^2.$$

Compared to parametric models, nonparametric models are much more flexible since they do not fixate any kind of structure. However, as with lookup tables the other side of the coin is that they are computationally expensive (for every new point we have to compute the weights anew).

Remark 4.1. With regard to the labelling: As lookup tables are not parametric, they are of course only a subcategory of non-parametric methods. But we consider them separately as they are fundamentally different from the non-parametric statistical methods mentioned above: They are not based on a model of any kind.

As a specific example for a nonparametric model we discuss Gaussian Process Regression in the next chapter. This is done in great detail, as it will be our specific choice for the value function approximation in the algorithm.

Policies. A few general remarks on policies: So far we have only considered policies implicitly defined by the maximization of the value function (approximation). But there are also other ways to define policies, we can divide them roughly into three groups: myopic policies, lookahead policies and policy function approximations.

Myopic policies are the most elementary: They are greedy functions and do not consider the future in any way, the prototype of a myopic policy is simply

$$\pi(s) = \arg\max_{a \in \mathcal{A}(s)} C(s, a).$$

Remark 4.2. Depending on the problem, we might have time indices C_t or π_t. In the following we will generally omit time indices as they only clutter the notation and distract from the essence of the concepts.

Lookahead policies are more advanced in that they actually factor in the future. The main idea is to simplify the problem by optimizing over some horizon shorter than the full time horizon. This simplified problem is then solved exactly. Despite this simplification, however, lookahead policies can be quite computationally expensive, because even exact optimization over a small number of time steps can lead to an exploding problem size. Examples for lookahead policies are tree search or rolling horizon. The simplest form of lookahead is a one-step lookahead policy: At each time step t and at each state $s_t \in \mathcal{S}_t$ we solve

$$\pi_t(s_t) := \underset{a_t \in \mathcal{A}_t}{\arg\min}\, \mathbb{E}\left[C_t(s_t, a_t) + \tilde{V}_t(g(s_t, a_t, \xi_{t+1}))\right], \tag{4.2}$$

where $\tilde{V}_t$ is some approximation of the true value function V_t.

An m-step lookahead policy is analogously computed: At each time step t we solve an m-step DP problem exactly, where s_t is the starting state. Then, for $r = t+m-1, t+m-2, \ldots, t$ we compute the policy π_r as a one-step lookahead policy based on an approximation $\tilde{V}_{r+1}$ of the true value function V_{r+1}. For each state s_r, the policy $\pi_r(s_r)$ is the *argmin* of the equation corresponding to (4.2), while the approximation $\tilde{V}_r$ is the *min* of that equation. In other words, we start with some approximation $\tilde{V}_{m+t}$ and then use the DP algorithm to solve the m-step DP problem exactly. After solving the m-step DP problem, we only use the last policy π_t, however, and discard the other policies.

In rolling horizon (also called *model predictive control*), we choose a horizon H. Then, at time t, we optimally solve the simplified problem which only goes from time t to $t+H$. We then use this policy to compute the decision leading to the next state s_{t+1}. At time $t+1$ we then repeat this for the problem from $t+1$ to $(t+1)+H$. In this sense, we "roll" the horizon forward.

Policy function approximations are fundamentally different from all the methods mentioned above. In myopic policies, lookahead policies as well as in policies defined by value function maximization, the crucial step is *optimization* of some function. With policy function approximation we instead just define the policy as some function. This is useful if we already have a good idea of how an optimal policy should look like. As with value function approximations, we can choose lookup tables, parametric or non-parametric methods to represent the policy.

On-policy and off-policy learning. Another central categorization of ADP algorithms is whether they employ *off-policy learning* or *on-policy learning*.

When approximating the optimal policy, we face an essential problem: On the one hand we want the policy obtained from the algorithm to get as close to the optimal policy as possible. On the other hand, in order to find the optimal policy, we also need to try out actions that might be considered suboptimal under the current policy. In order to clarify these roles, we consider two policies: We call the policy which approximates the optimal policy and which will be returned by the algorithm *target* or *learning policy*. We call the policy which actually generates the states the *sampling policy*. If target and sampling policy agree, we speak of *on-policy learning*, otherwise of *off-policy learning*. On-policy learning is conceptually simpler, off-policy learning is more general but also converges slower.

This is essentially the "exploration versus exploitation" dilemma, which is a problem inherent to learning in ADP. The question is the following: Do we try out new actions just in order to

know their value (explore) or do we choose the action we currently consider optimal (exploit)? Exploring too much leads to slow convergence and is expensive, exploiting too much can result in not finding the optimal policy. Visiting every state is often a prerequisite in convergence proofs. It is therefore crucial to factor in how certain we actually are of a value of a state.

Estimating the value of a state. So far we have only discussed how to estimate the value function *given* some observations of the values of some state. We now address how we actually obtain these observations.

In the finite horizon case, this is easy: In order to evaluate the value function for a given policy π we simply follow a sample path: Let s_t be the state we are in at time t. We choose the action a_t according to our policy π, that is $a_t = \pi(s_t)$. The exogenous information ξ_{t+1} is realized and we obtain the reward $\hat{C}_t = C_t(s_t, a_t, \xi_{t+1})$. Then we transition to the next state according to the system dynamics and reach time $t+1$. We do this until we reach the time horizon T. Our estimate of the value of the state s_0 is then $\hat{v} = \sum_{\tau=0}^{T} \hat{C}_\tau$. This process of generating a sampling path from time $t = 0$ to T is called an *episode*. We typically generate many episodes in one iteration, and then use the observations (the $\hat{v}$'s) to estimate the value function. Another way to see this is that (in the finite horizon case) evaluating a fixed policy is the same as making unbiased observations of a noisy function.

In the infinite horizon case it is impossible to sample directly like above. One way to deal with this is to simply cut off the sampling at some point. These *partial simulations* assume that if we choose a "long enough" time horizon the sample is a "good enough" estimate. This is especially applicable if we use discounting in the model: The impact of the rewards become smaller as time progresses.

Another approach is to use *temporal differences*. They are the difference between our current estimate of the value of a state and the update estimate of that value. Temporal differences are used to update the value function iteratively. More on temporal difference learning can be found for example in [SB17, Chapter 6] or [Ber17, Vol. II, Chapter 6].

Remark 4.3. There are multiple ways to generate a realization of exogenous information, or rather, obtain random samples: Sometimes the random realizations are given by actual physical and real-world processes (for example sequences of actual demands and prices or wind energy data). An alternative way is to use as a black box computer simulation modelling a complex process. A third way is to simply draw from a random variable, if it is known.

4.2 Putting it together: Approximate policy iteration

There is no fixed definition for approximate policy iteration [14] - approximate policy iteration is simply policy iteration for which at least one of its components (for instance, the value function, the sampling, the maximization) is approximated. The specific approximation choices are endless. Typically only the policy evaluation is approximated. (Finding the global optimum in the policy improvement step is crucial for fast convergence.)

[14] The same holds for approximate value iteration.

Convergence results. Though classical policy iteration converges, the convergence of methods using approximate value functions is only poorly understood. Results only exist for special cases and many cases are known where algorithms using VFAs diverge or oscillate between poor policies.

For example, for linearly parametrized VFAs (such as e.g. LSTD) convergence of the approximate policy evaluation can be guaranteed more easily. If the errors of the approximate policy evaluation and approximate policy improvement are bounded, then approximate policy iteration algorithms can be proved to produce policies with bounded suboptimality in the discounted case [BSB10, Section 5.3]. However, convergence to a fixed policy is not guaranteed, oscillating policies are possible. These results can also be applied to general approximation methods. Issues such as policy oscillation, susceptibility to noise or convergence are discussed for instance in [Ber11]. We note that this lack of theoretical convergence proofs holds for ADP in general. Nevertheless, ADP algorithms have been successfully applied to a variety of problems, also to very complex problems [Pow07].

When discretizing, we would like the property of *consistency* to hold. By this we mean that the finer the discretization is chosen, the better should the result of the algorithm become, and the optimal value function of the original problem should be obtained in the limit of the discretization refinement. There are no general results for consistency. It is typically guaranteed by some continuity conditions, such as the (Lipschitz) continuity of the optimal value function or the optimal policy. This in turn only holds for specific subclasses of the problem, for example only MDPs with a (Lipschitz) continuous reward function and system dynamics ([Ber17, Vol.I, p.406], or see [LGM16] for a concrete example, here for a classification problem).

Evaluation of an algorithm. In practice, we would like to have a measure of how "good" an algorithm is or not. Most of the time, we do not have an analytical solution or a way to compute the solution exactly. If we do have an exact solution, we can measure the optimality of the policy obtained by the algorithm by the values of the objective function. If we do not, we have to resort to general characteristics such as the convergence rate. In practice, algorithms are not rated absolutely, but judged in comparison to other algorithms, typically based on their performance on some test problems.

Another viewpoint: Powell finds that it can often be more beneficial in industrial applications to have a highly accurate model of the transition function than to obtain optimal policies[Pow07, p.148]. He argues that the quality of policies is often hard to measure in practice, while a simplified model of the system dynamics can lead to an incorrectly evolving system and thus can have a stronger detrimental effect than a suboptimal policy.

Another thing to keep in mind is that real-time constraints of the problem have to be respected. That is, we need to factor in the computing capacities available which influence whether we need the solution of the problem within an hour, a day, or a week.

5 Application to the energy storage problem

In this chapter, we apply the theory of ADP from the previous chapters to our specific application. We will use the approximate policy iteration approach from [JPP+14], in which the authors test multiple methods as value function approximations. From these methods we choose Gaussian Process Regression. Before we discuss the algorithm, we give a general introduction on Gaussian processes and how they can be applied to regression and prediction problems.

5.1 Gaussian Process Regression

In recent decades, Gaussian processes have become very popular, particularly in the machine learning community in the field of supervised learning. They are part of a trend for kernel-based methods (another example are Support Vector Machines). It is also used in other fields: In fact, it originates from geostatistics where it has been developed under the name of "kriging". Gaussian processes can be applied to many different tasks, such as regression, classification or optimization. Here we are only going to focus on *Gaussian Process Regression* (GPR). Gaussian Process Regression is a nonparametric regression technique and a kernel-based learning method. GPR has strong connections to many other popular methods: It is closely related to Support Vector Machines and even mathematically equivalent to other popular models, such as Bayesian linear models or spline models [RW05, Chapter 6]. Neal [Nea96] showed that, under suitable conditions, neural networks with one hidden layer converge to Gaussian processes as the number of hidden neurons tends to infinity.

For a general introduction of Gaussian processes with a particular focus on their use in machine learning we refer to [RW05]. MacKay [Mac98] mainly discusses the relation between Gaussian processes and neural networks, but also gives a general introduction on Gaussian processes.

A few general words on regressions. The goal in regression is to model and make inferences about the relation between inputs and their outputs. We typically consider real-valued scalar outputs. Given data points, in regression the aim is therefore to fit a curve through potentially noisy data. That is, given a vector of inputs $\mathbf{x}$ and a vector of datapoints $\mathbf{y}$, we want to find a function f dependent on some parameters β such that

$$\mathbf{y} \sim f(\mathbf{x}, \beta).$$

The form of the function, f, is specified beforehand, and the goal is to estimate the parameters β. Note however, that - unlike for interpolation - we do not require the curve to pass exactly through those data points, because we assume noise.

Doing so, we often face the problem of *overfitting*. The given dataset is only a sample of the underlying distribution, the latter is what we want to model. However, we can only base our modelling on the given sample. Overfitting then means that the model resulting from regression memorizes the specific properties of our sample too much and thus does not reflect the true distribution anymore. The more degrees of freedom there are in the model, the higher the chance of overfitting is. For example, if we have a linear model and more parameters than

observations, then the fitted function can simply reproduce the observations exactly. There is a general trade-off between data-fit and predictive power. In GPR we try to limit overfitting by using a prior.

Stochastic processes. Stochastic processes are, roughly speaking, a generalization of probability distribution to functions (which we can also view as an "infinitely-dimensional vector") instead of finite-dimensional random variables. Thus, Gaussian processes are generalizations of multivariate Gaussian distributions. They have found widespread use in *Bayesian inference* (a method of statistical inference where the updating of the probability of a hypothesis is based on Bayes' theorem, see below for more details) where, compared to other stochastic processes, Gaussian processes require only relatively simple computations and still yield good results. In the following paragraphs, we will roughly follow Chapter 2 in [RW05].

Definition 5.1. A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution.

Remark 5.2. Alternatively, MacKay [Mac98] defines: "The probability distribution of a function $y(x)$ is a Gaussian process if for any finite selection of points $x_1, x_2, \ldots, x_n$ the marginal density $P(y(x_1), y(x_2), \ldots, y(x_n))$ is a Gaussian."

Just like Gaussian distributions, Gaussian processes are fully specified by their mean and their covariance. Note that for processes, both mean and covariance are functions. Note also that the covariance function plays the much more important role; the properties of Gaussian processes are mainly determined by their covariances.

Analogous to the finite-dimensional case, we define the *mean function* $m(x)$ and the *covariance function* $k(x, x')$ of a real stochastic process $f(x)$ as:

$$\begin{aligned} m(x) &= \mathbb{E}[f(x)], \\ cov(f(x), f(x')) = k(x, x') &= \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))]. \end{aligned}$$

The mean function is here assumed to be zero for notational simplicity. Also, note that the covariance of the output is written as a function of the input variables. In the machine learning context, the covariance function is typically called *kernel*. It measures the similarity between points.

Then we can write the Gaussian process as

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')). \tag{5.1}$$

Gaussian Process Regression. The setting is as follows: Our given data consists of *n training points*, $(x_i, y_i)_{i=1}^n \in \mathbb{R}^d \times \mathbb{R}$ (sometimes also called observations). Our training points $(x_i, y_i)_{i=1}^n$ are composed of the d-dimensional input variable x_i and the scalar output variable, or *target*

variable, y_i [15]. In the following calculations we will represent the input data in the $d \times n$-matrix X, where column i represents the input vector x_i.

GPR is an application of Bayesian inference. As the name already indicates, Bayesian inference is based on Bayes' Theorem:

$$P(H|D) = \frac{P(D|H) \cdot P(H)}{P(D)}. \tag{5.2}$$

Here H stands for hypothesis and D for data. We call $P(H|D)$ *posterior* (probability), $P(H)$ *prior* (probability), $P(D|H)$ *likelihood*, and $P(D)$ *marginal likelihood*.

In this case, our prior is over the function space. In other words, our prior is our belief of the distribution over the function space before any data is observed. We assign a prior probability to every possible function, with regards to how likely it is that this function underlies the given data.

Specifically, our prior is that a Gaussian process f underlies the data, where

$$f(x) \sim \mathcal{GP}(0, k(x, x'))$$

for all inputs and for a specified covariance function k. When doing GPR, we can *choose* which covariance function we use (and of course we can also use another mean than zero).

In this case we choose the *Gaussian kernel* (also called squared exponential or Radial Basis Function) as our covariance function, so that

$$k(x, x') = \sigma^2 \exp\left(-\frac{|x - x'|^2}{2l^2}\right), l > 0. \tag{5.3}$$

The Gaussian kernel is stationary, that is, its value is only determined by the relative position of its arguments, not their absolute location. We see that the exponential part attains values in the interval $(0, 1]$. It tends to one the closer the inputs get, and tends to zero as the distance between the inputs increases.

It has two parameters, the *lengthscale* l and the *signal variance* σ^2. The lengthscale determines how close two points have to be to influence each other. In general we are not able to extrapolate more than l units away from the data. The signal variance is a scale factor, it determines how much the function varies from its mean on average.

The Gaussian kernel has many useful properties: It is infinitely differentiable, it is universal (see [MXZ06][16], and it can easily be integrated against. As a result the Gaussian kernel has become the default kernel in many kernel-based methods, such as Gaussian processes or Support Vector Machines. Other standard kernels are periodic kernels or linear kernels. More information on which kernel to choose and how to combine them can be found for example in [Duv14].

Recall that in regression we are interested in making inferences about the relation between inputs and outputs. That means that we are *not* interested in modelling the distribution of the the inputs itself, but in modelling the *conditional* distribution of the outputs *given the inputs*.

[15] In statistics the input variables are called *independent variables* and the output variables *dependent variables*.

[16] "Universal" means that any continuous target function can be uniformly approximated on any compact subset of the input space.

In general, GPR is not only used to describe the given data, but to moreover predict the target for some given *testpoints*. In our case it is sufficient to consider only one testpoint (see the algorithm later). Let this testpoint be denoted by x_*. We are interested in calculating, or rather, approximating the unknown test output, that is $y_* := f(x_*)$. Furthermore, we denote the vector of all training targets with $\mathbf{y} = (y_1, \ldots, y_n)^T$.

In reality, we often only have access to noisy observations,

$$y = f(x) + \varepsilon,$$

where we typically assume that we have additive i.i.d. Gaussian noise ε with variance σ_N^2 (the subscript stands for "noise"). The following derivations also holds for the noise-free case, by simply omitting ε. We call the Gaussian noise variance σ_N, the signal variance σ and the lengthscale l the *hyperparameters* of the GPR. This term reflects that they are parameters of a non-parametric model. We can vary the hyperparameters. With this noise distribution the covariance becomes

$$cov(y_i, y_j) = k(x_i, x_j) + \sigma_N^2 \delta_{ij},$$

or written as a matrix

$$cov(\mathbf{y}) = K(X, X) + \sigma_N^2 I_n, \tag{5.4}$$

where $(K(X, X')_{kl} = k(x_k, x'_l)$ for matrices X and X'.

The joint prior distribution of the training outputs $\mathbf{y}$ and the test output y_* is, according to our prior (5.1),

$$\begin{pmatrix} \mathbf{y} \\ y_* \end{pmatrix} \sim \mathcal{N}\left(0, \begin{pmatrix} K(X,X) + \sigma_N^2 I_n & K(X, x_*) \\ K(x_*, X) & k(x_*, x_*) \end{pmatrix}\right).$$

We now obtain the posterior distribution (recall (5.2)) by conditioning the joint prior distribution (corresponding to the numerator in (5.2)) on the observed data (the conditioning corresponds to dividing by the marginal likelihood in (5.2)), cf. [RW05] for the detailed derivations:

$$\begin{aligned} y_* | X, \mathbf{y}, x_* &\sim \mathcal{N}\left(\bar{y}_*, cov(y_*)\right), \text{where} \\ \bar{y}_* &= \mathbb{E}[y_* | X, \mathbf{y}, x_*] = K(X, x_*)\left[K(X,X) + \sigma_N^2 I_n\right]^{-1} \mathbf{y} \text{ (the predictive mean)} \\ cov(y_*) &= k(x_*, x_*) - K(x_*, X)\left[K(X,X) + \sigma_N^2 I_n\right]^{-1} K(X, x_*). \end{aligned} \tag{5.5}$$

In $\bar{y}_*$ we have thus obtained our prediction for the test output. Note that the prediction is actually a distribution in itself. We *choose* to use the mean as the returned prediction value. Furthermore, note that the vector $\mathbf{v} := [K(X,X) + \sigma_N^2 I_n]^{-1}\mathbf{y}$ does not actually depend on the chosen test point. In other words, we use the vector $\mathbf{v}$ for all test outputs we predict. Thus, we can rewrite

$$\bar{y}_* = K(X, x_*)^T \mathbf{v} = \sum_{i=1}^{n} k(x_i, x_*) v_i.$$

In other words, the (mean) prediction is a linear combination of kernel functions, each one centered on a training input.

GPR scales cubically in the number of training points; its runtime is $O(n^3)$, where n is the number of training points. This is due to calculating the determinant and matrix inversions. Furthermore, it has $O(n^2)$ memory complexity. GPR therefore quickly becomes intractable for large datasets (ten thousand or more). As a result, sparse Gaussian processes are an active research field.

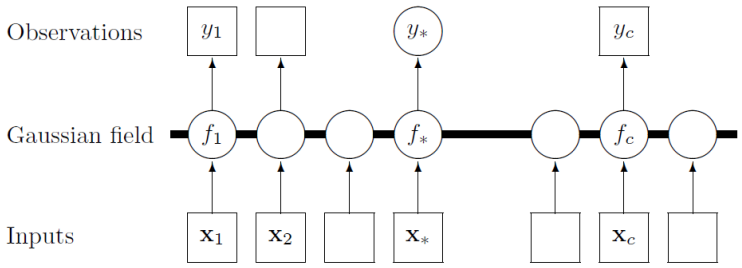


Figure 2.3: Graphical model (chain graph) for a GP for regression. Squares represent observed variables and circles represent unknowns. The thick horizontal bar represents a set of fully connected nodes. Note that an observation y_i is conditionally independent of all other nodes given the corresponding latent variable, f_i. Because of the marginalization property of GPs addition of further inputs, $\mathbf{x}$, latent variables, f, and *unobserved* targets, y_*, does not change the distribution of any other variables.

Figure 5.1: Figure 2.3 from [RW05]

Optimized choice of hyperparameters using the marginal likelihood. We would like to choose our hyperparameters l, σ and σ_N optimally. The most common way to do this is to maximize the marginal likelihood. The marginal likelihood, also called *(model) evidence*, is defined as $P(D)$ in (5.2). We can write it as the integral over the product of likelihood and prior, $P(D) = \int_H P(D|H)P(H)$. In this case the hypothesis H consists of the exact (noise-free) function values $\mathbf{f}$ of our training points, and the data D consists of the noisy target values $\mathbf{y}$ of our training inputs. Our marginal likelihood is $p(\mathbf{y}|X)$ (for all three probabilities (marginal likelihood, likelihood, prior) we additionally condition on the training inputs X), we thus have

$$p(\mathbf{y}|X) = \int p(\mathbf{y}|\mathbf{f}, X)p(\mathbf{f}|X)d\mathbf{f}.$$

The marginal likelihood is then the likelihood that we obtain the noisy training outputs, based on the training inputs. Using the fact that for our Gaussian Process framework the likelihood is a factorized Gaussian and the prior is Gaussian (for more details see [RW05, p.37]), we can write the *log marginal likelihood* as

$$\log p(\mathbf{f}|X) = -\frac{1}{2}\mathbf{y}^T(K + \sigma_N^2 I)^{-1}\mathbf{y} - \frac{1}{2}\log|K + \sigma_N^2 I| - \frac{n}{2}\log 2\pi. \tag{5.6}$$

The three summands have the following meaning: $-\frac{1}{2}\mathbf{y}^T(K + \sigma_N^2 I)^{-1}\mathbf{y}$ represents the data-fit, $-\frac{1}{2}\log|K + \sigma_N^2 I|$ is a complexity penalty and $\frac{n}{2}\log 2$ is a normalization constant. With a growing lengthscale l the (negative) data-fit decreases (the model becomes less flexible) and the

(negative) complexity penalty increases (with a higher lengthscale the Gaussian kernel becomes more peaked).

Defining optimal as "maximal marginal likelihood", we can now optimize the hyperparameters by maximizing the marginal likelihood (the marginal likelihood is a suitable candidate for an optimality criterion because it incorporates an automatic trade-off for data-fit and complexity). We need the partial derivatives for the first order condition, they can be written as

$$\frac{\delta}{\delta\theta_j}\log p(\mathbf{y}|X,\theta) = \frac{1}{2}tr(((K^{-1}\mathbf{y})(K^{-1}\mathbf{y})^T - K^{-1})\frac{\delta K}{\delta\theta_j}), \tag{5.7}$$

where θ are our hyperparameters.

5.2 Approximate policy iteration algorithm

We now finally turn to the specific implementation of API for our energy storage problem. The following algorithm is based on the API algorithm in [JPP+14]. Boiled down to its essence, we approximate exact policy iteration only in the policy evaluation step: We use GPR as an approximation of the value function, based on Monte Carlo samples.

Algorithm 2: Approximate policy iteration

Input: n_{max}, m, ApproximationArchitecture
Output: Approximation of optimal policy

for $n = 1, ..., n_{max}$ **do**
  Set $v_{i,t} := 0$ for all $i = 1, ..., m$ and for all $t = 1, ..., T$
  for $i = 1, ..., m$ **do**
    Initialize $r_{i,1}$
    for $t = 1, ..., T$ **do**
      Sample $w_{i,t}, p_{i,t}, d_{i,t}$
      Set $s_{i,t} := (r_{i,t}, w_{i,t}, p_{i,t}, d_{i,t})$
      $a_{i,t} := \begin{cases} \text{initial policy} & n = 1 \\ \arg\max_{a\in\mathcal{A}_t(s_t)} C(s_{i,t},a) + V^p_{n-1,t}(s^a_{i,t}) & n > 1 \end{cases}$
      $c_{i,t} := C(s_{i,t}, a_{i,t})$
      for $\tau = 1, ..., t$ **do**
        $v_{i,\tau}$ += $c_{i,t}$
      end
      $r_{i,t}$ += $\Phi \cdot a_{i,t}$
    end
  end
  for $t = 1, ..., T$ **do**
    $V^p_{n,t}(\cdot) := \text{ApproximationArchitecture}\left(\left\{s^{a_{i,t}}_{i,t}\right\}_{i=1,...,m}, \{v_{i,t}\}_{i=1,...,m}\right)$
  end
end

As before, $\Phi = (0, 0, -1, \beta^c, \beta^c, -1)$ denotes the transition factors for each component of the decision vector as defined in (3.3), and $\mathcal{A}_t(s_t)$ denotes the space of admissible decisions for the state s_t as defined in (3.2). n is the iteration number, and i is the index for the samples or episodes. Note that the return value of the algorithm (the final approximation of an optimal policy) is only returned implicitly: In order to evaluate it at a given state, we have to first perform the optimization in line 8.

We now describe the algorithm based on the notions introduced in the previous chapter.

Since the energy storage problem is continuous in space, we have to use ADP methods. Here, we use approximate policy iteration. In the algorithm, the (exact) policy improvement takes place in the optimization in line 8, while the (approximate) policy evaluation ranges from lines 2-18 , excluding line 8 (that is to say, it takes up the rest of the algorithm).

In the policy improvement step we use a one-step lookahead step in the optimization. Note also, that the policy is not explicitly represented: It is implicitly defined by the current value function approximation and only computed for the states that occur during the sampling.

As a value function approximation we choose GPR which is a nonparametric model. The value function observations $v_{i,t}$ for the value function approximation are computed by simple Monte Carlo sampling where the sampling of the exogenous information in line 6 is uniform. At the beginning of every episode, the initial storage value is also sampled uniformly (line 4).

Note that these observations $v_{i,t}$ are noisy: They are not exact samples from the value function, but only *approximate* samples; that is, samples from value functions corresponding to suboptimal policies. This is because our approximations are based on our value function approximation $V^p_{n,t}$, but these approximations may be quite bad, especially in the first few iterations (depending on the quality of the intial policy). We therefore have a two-fold approximation: First, we approximate the value function using GPR. Second, we approximate the approximate value samples the GPR is based on.

During the sampling for the policy evaluation, we use an on-policy approach. There is no exploration. Arguably, we already explore to some degree by the fact that we have approximate samples of the true value functions which can also be seen as exact samples of suboptimal policies, as mentioned above. However, these suboptimal policies are not chosen randomly as one would do in exploration.

The algorithm is not model-free since we already know the transition function and the contribution function, but it can be considered as *distribution-free* (by this we mean that we do not know the distribution of the exogenous variables w, p, d [Pow07, p.604]).

Two additional notes on the algorithm: There are no "memory effects"; there is no learning effect within one iteration. Learning only takes place in one sweep between iterations. Furthermore, we assume perfect state information, that is, we know all relevant information, in particular the price.

Initial policy. There is no canonical choice for an initial policy. In our implementation, we do not define the initial policy globally (for example implicitly via a value function), but only compute the respective action for the specific state we are in. In particular, we do not determine the respective action deterministically, but *sample* it from $\mathcal{A}_t$.

Rejection sampling ("sample till you hit a feasible action"), while simple, is computationally expensive and very slow, in our experiments it took up to 16 million samples till a feasible decision was sampled. That is because, depending on the state, the set of feasible decisions is fairly small compared to the total set of decisions (see Section 6.4). Thus the rate of acceptance is very low and the algorithms takes a long time. Instead, we sample our admissible decision from a *subset* of admissible decisions.

Concretely, given a state (r_t, w_t, p_t, d_t), we sample the components of the decision

$$(a_t^{wd}, a_t^{rd}, a_t^{gd}, a_t^{wr}, a_t^{gr}, a_t^{rg})$$

consecutively and *dependent* of each other (the dependency is required in order to produce an admissible decision that lies within the set $\mathcal{A}_t$). We sample in the following order:

$$\begin{aligned}
a_t^{wd} &\in [0, \min\{d_t, w_t\}], \\
a_t^{rd} &\in [0, \min\{r_t, \gamma_d, \beta_d(d_t - a_t^{wd})\}], \\
a_t^{gd} &= \max\{0, d_t - a_t^{wd} - a_t^{rd} * \beta_d\}, \\
a_t^{wr} &\in [0, \min\{w_t - a_t^{wd}, \gamma_c, r_{max} - r_t\}], \\
a_t^{gr} &= 0, \\
a_t^{rg} &= 0.
\end{aligned} \tag{5.8}$$

Lemma 5.3. *A decision generated by (5.8) yields an admissible decision.*

Proof: This can be checked by a quick case distinction: Since the constraints (1)-(4) in (3.2) are only (in-)equalities, it is sufficient to show the admissibility of the generated sample by taking the min and max in each of the decision. That is, we do not sample from the interval, but set each variable as its maximal possible value. Thus we have

$$a^{wd} = \min\{d, w\}, \;\; a^{rd} = \min\{r, \gamma_d, \beta_d(d - a^{wd})\}, \text{ and } a^{wr} = \min\{w - a^{wd}, \gamma_c, r_{max} - r\}.$$

For readability we also omit the time index.

Case 1: $d \leqslant w$
Then $a^{wd} = d$. Thus we have $a^{rd} = 0$ and $a^{gd} = 0$. $\Rightarrow$ (1) is fulfilled. $\Rightarrow a^{wr} + a^{wd} = \min\{w, \gamma_c + d, r_{max} - r + d\} \leqslant w$. $\Rightarrow$ (2) is fulfilled by definition of the minimum, the same holds for (4). Inequality (3) is trivially fulfilled, as the left side is zero.

Case 2: $d > w$
Then $a^{wd} = w$. Thus we have $a^{wr} = 0$. Then (2) is fulfilled. (4) trivially holds because the left side is zero. Inequality (3) holds by definition of the minimum (the upper bound is already incorporated in the definition of a^{rd}). It remains to show equality (1):
Case 2.1: $d - w - \beta_d a^{rd} > 0 \Rightarrow a^{wd} + \beta_d a^{rd} + a^{gd} = w + \beta_d a^{rd} + d - a^{wd} - \beta_d a^{rd} = d$.
Case 2.2: $d - w - \beta_d a^{rd} = 0 \Rightarrow a^{wd} + \beta_d a^{rd} + a^{gd} = w + \beta_d a^{rd} = d$. □

The motivation for this choice of initial policy is the following intuitive idea: Use all available wind energy. Use as much of it for demand as needed and as possible. If there is not enough, stock up with energy from the storage, and if that is not enough, with energy from the grid. If there is excess wind energy left, store it. Note that a_t^{gr} and a_t^{rg} are zero for all states. (This only concerns the initial policy, during the course of the algorithm the policies can potentially be very different.)

We set a_t^{gr} equal to zero in the initial policy because we want to minimize costs and therefore only buy from the grid sparingly. Intuitively, we prefer stocking the storage with excess wind energy rather than with bought energy from the grid. The idea is that even in the worst case (that is, when demand exceeds the combined energy from wind and storage), we can simply compensate with energy from the grid. Put differently: We can always just buy energy from the grid when we need it, this might be cheaper. a_t^{rg} is the most difficult decision and heavily depends on the state of storage and the state. For example, it is wise to sell when the price is high, and when the storage is high. However, if we expect that d_t will be low in the next few timesteps, than we might as well accept selling from an already low storage, if the price is very high. This is only one of many considerations that influence the size of a_t^{rg}. Thus, we simply set it as a constant in the initial policy. In order to ensure an admissible decision at all times, we set it to zero.

5.3 Approaches to the wind energy storage problem in the literature

The specific wind energy storage problem described in Chapter 3 was first formulated in [SP17] by Powell and Salas, where a lookup table approximate value iteration approach exploiting concavity is used. In a follow-up paper [JPP$^+$14] the same authors (as well as Jiang, Pham and Scott) then compared the performance of various common ADP approaches. In another follow-up paper [JP15], Jiang and Powell applied a lookup table approximate value iteration approach exploiting monotonicity. Using the same problem formulation, Han and E [HE16] use a neural network approach to the problem. In the following we summarize the various approaches and their results.

Lookup table approximate value iteration exploiting concavity (Salas, Powell 2017). In this paper[SP17] Salas and Powell develop the energy storage problem used in [JPP$^+$14]. They use a structured lookup table with approximate value iteration. Their algorithm also uses post-decision states and is based on piecewise linear and separable VFAs $\bar{V}$ of the form

$$\bar{V}_t(r_t^p, w_t, p_t) = \sum_{i=1}^{K_t} \bar{v}_{t,i}(w_t, p_t) b_{t,i}. \tag{5.9}$$

Here r_t, w_t and p_t have the same meaning as in Chapter 3 (r_t^p is the post-decision storage variable), K_t is some integer number, $\sum_i b_{t,i} = r_t^p$ and $\bar{v}_{ti}(w_t, p_t)$ is the slope for segment i. The VFA is completely determined by the slopes $\{\bar{v}_{t,i}(w_t, p_t)\}_{i=1}^{K_t}$ and the breakpoints $\{b_{t,i}\}_{i=1}^{K_t}$.

The crucial feature of the algorithm is that it exploits the concavity of the value function in the resource dimension 3.1. As the authors write: "[Exploiting concavity] allows us to use a pure exploitation policy, avoiding the need for exploration policies that are characteristic of

all reinforcement learning policies for problems with discrete actions." At every update of the VFA its concavity is enforced using the Concave Adaptation Value Estimation algorithm from [GP01].

As shown in the paper, the algorithm performs very well: On the same dataset as used here, its returned near-optimal policy is within 0.1% of optimal for the deterministic datasets and within 0.9% of optimal for the stochastic ones. For almost all datasets, it performs better than model predictive control. Additionally, the authors demonstrate that the algorithm is well-suited for multiple devices and that it scales linearly with the number of devices. Its drawback is however that it is limited to a low-dimensional state-of-the-world-variable, because (in addition to the curse of dimensionality) a problem with complex state variables typically has much less structure to be exploited.

Monotone-ADP (Jiang, Powell 2015). In [JP15] Jiang and Powell develop an approximate value iteration lookup table method which they call *Monotone-ADP*. It is intended for problems which have monotone structure in some input dimensions of the value function and enforces this monotonicity for the VFA after every update. The algorithms requires a full exploration policy. The authors demonstrate that the algorithm outperforms several popular ADP algorithms (among them approximate policy iteration and kernel-based reinforcement learning) at multiple applications. Almost sure convergence to the optimal value function is proved under suitable conditions.

Comparison of ADP methods (Jiang et al. 2014). In their paper [JPP+14], Jiang et al. develop the model described in Chapter 3.

The objective of the paper is to compare and evaluate the performance of several popular ADP methods on the same energy storage problem. The methods are:

(1) approximate policy iteration where the value function is approximated by both linear and nonparametric methods (Support Vector Machines, Gaussian Process Regression (with Gaussian kernels), local polynomial regression, Dirichlet cloud with radial basis functions)

(2) the lookup table approximate value iteration exploiting monotonicity developed in [JP15]

(3) the lookup table technique exploiting concavity developed in [SP17]

(4) direct policy search

The performance is compared to the exactly computed benchmark solutions for the discretized version of the problem. The authors come to the conclusion that none of these techniques works really well. Their results can be summarized as follows: Firstly, the approximate policy iteration and approximate value iteration algorithms ((1) and (2)) perform poorly. Among them the support vector machines work best, with an average optimality of 90%. Secondly, while the two latter methods ((3) and (4)) work well, (especially the structured lookup table approximate value iteration which has optimality rates of $98 - 100\%$), they do not scale well to large-dimensional problems, however.

As a consequence of the bad performance of the general-purpose ADP methods, the authors conclude that an efficient method should exploit the structure of the problem.

Neural network approach (Han, E 2016). In [HE16] Han and E use feedforward neural networks with two hidden layers to approximate the optimal controls. Their model is principally the same as in [JPP+14] with minor modifications (no holding costs, no dis-/charge efficiencies, no option to stock the storage with electricity from the grid). They use standard stochastic gradient descent with backpropagation and batch normalization in the subnetworks. Though their algorithm is suitable for continuous variables, they confine it to a discretized set in order to compare it to the benchmark solutions in [JPP+14].

Han and E present the results only for time horizons $T = 10$ and $T = 15$. For these two time horizons, their algorithm performs near-optimal, for $T = 10$ it performs even better than the benchmark lookup table policy. They concede however that both the variance of the optimality results and the number of iterations required for convergence grow with increasing time horizon. This seems to imply that the algorithm would not scale well to the full-time-horizon problem in Chapter 3 ($T = 100$).

6 Numerical Results

In this chapter, the approximate policy iteration from Chapter 5 is analysed numerically. In Section 6.1 we introduce the benchmark datasets that we used for the computations. We present our implementation approach and the main results in Section 6.2. In our implementation the algorithm achieves a much higher optimality rate than in [JPP+14]. We discuss how the choice of initial policy and the choice of hyperparameters contribute to this. In Section 6.3 we analyse other measures of optimality, we find that they partly give discouraging results. We discuss the computation time in Section 6.4. We conclude in Section 6.5 with a summary of our results and suggest ideas for further investigation.

6.1 Benchmark datasets

We use the same benchmarking datasets as Jiang et al. in [JPP+14]. The datasets originate from [SP17] and consist of both deterministic and stochastic cases. We focus on the stochastic datasets for comparative reasons, as results of the approximate policy iteration with GPR are only provided in [JPP+14] for the stochastic datasets.

In the stochastic case the exact solution (the benchmark) was computed by discretizing the state space and then solving the Markov Decision Process exactly using backwards induction. In this respect, the solutions for the stochastic datasets can only be considered as optimal *contingent on the chosen discretization*. They are only approximations of the true optimal solution of the energy storage problem in Chapter 3. The discretization is rather coarse: The resulting grid contains $7 \times 13 \times 41 = 8897$ states: we have 7 uniformly distributed levels of energy, 41 uniformly distributed levels of price and 31 uniformly distributed levels of storage. Demand is assumed to be deterministic, thereby further reducing the number of states. We note that the coarseness is necessary in order to have a computationally tractable problem.

The energy storage problem we consider is a comparatively simple problem. However, we note that it only constitutes a *benchmark problem*. It is not meant to represent the real-world application as accurately as possible, but instead to provide a simple problem for the evaluation of algorithms. Moreover, even though this is a simple problem, we see that the existing methods do not work that well [JPP+14].

The exogenous factors are modelled as follows:

- Demand is assumed to be deterministic and sinusoidal (representing a seasonal structure):

$$d_t = \max\left\{0, 3 - 4\sin\left(\frac{2\pi t}{T}\right)\right\}$$

- Two types of price processes are tested: A sinusoidal price process (similar structure to demand) and a Markov chain (for some datasets also with jumps), both perturbed by Gaussian shocks. The Markov chain is time homogenous, first-order and bounded in state space.

- Energy is also modelled as a time homogenous, first-order, bounded Markov Chain with Gaussian or uniform shocks.

Example sample paths were shown in Figure 3.2.

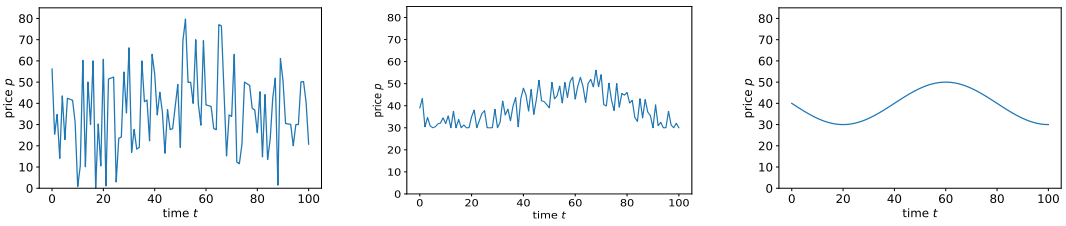
Discharge efficiency and charge efficiency are modelled as perfect; $\beta_d = \beta_c = 1$. This is necessary for the trajectories to stay on the discretization grid. Furthermore, the (dis)charging capacity is restricted to $\gamma_d = \gamma_c = 5$. For more details we refer to [JPP+14] or [SP17].

Optimality is measured against the exact solution which is determined by backward dynamic programming. The performance metric M of an algorithm is the ratio of the total profit obtained by the algorithm, TP_A, and the total profit obtained by the exact solution, TP_E, on the same set of sample scenarios (that is, for those sample scenarios the stochastic factors are identical, only the storage trajectories are different):

$$M = \frac{TP_A}{TP_E} \tag{6.1}$$

Inconsistencies in the datasets. Each dataset contains 256 sample paths [17]. For each sample path the state (price, energy, demand, storage), the optimal decision, and the resulting profit are provided for every point of time.

We found that the files of some datasets were partially inconsistent with the description given in the corresponding papers ([JPP+14, SP17]): This mainly concerned the price samples for the datasets S1-S4. The price process for the datasets with the sinusoidal price process (S1-S4) does not seem to be sampled randomly: For each time step t, the decimal places across all 256 samples are identical. The plots according to the given price samples are significantly different from the plots generated according to the definition of the price process: In the latter ones the underlying sinusoidal structure is clearly identifiable while it is not in the former (see Figure 6.1). Additionally, while $p_{min} = 30$ and $p_{max} = 70$ in [JPP+14], the provided price-data for S1-S4 does not comply with these bounds, instead it has a range of $[0, 80]$. For the dataset S1 37% of the provided price samples lay outside the bounds in the paper (same for the datasets S2-S4). As a consequence of these inconsistencies, we did not use these datasets for the comparison to the results of Jiang et al. [JPP+14].



(a) Dataset **(b)** Simulation **(c)** Unperturbed price

Figure 6.1: Comparison of provided, simulated and unperturbed sinusoidal price process

Another inconsistency is the use of holding costs in the benchmarks in the datasets, but not

[17] In the computations below, we typically consider only a subset of 30 sample paths. We found that this yielded the same results as for the full 256 sample paths. Accordingly, whenever we compare the mean of results from our own implementation with the mean of results from the benchmark datasets, we compare both means on the same set of 30 sample paths.

in the paper. In [JPP+14] the Monotone-ADP algorithm [JP15], which requires monotonicity of the value function in the storage dimension, is also applied on the benchmark datasets from above. However, in the paper [JPP+14] the authors only prove the required monotonicity in the case without storage costs. It is not obvious why even with storage costs the monotonicity in the storage dimension should still hold (see footnote regarding Proposition 3.2). Therefore the application of the Monotone-ADP to the problem in the datasets seems inadmissible. However, this does not directly affect the use of the benchmarks here, since we do not use the monotonicity property in the approximate policy iteration algorithm.

6.2 Main results and algorithm tuning

Approach and implementation details. In the following subsections we discuss the findings based on our implementation of the approximate policy iteration algorithm in [JPP+14]. In our implementation (see enclosed CD) we use the *GPy* software [GPy12] for the Gaussian Process Regression (a recent comparison of different Gaussian Process modelling software can be found in [EAS17]). In their implementation of the Gaussian kernel, three hyperparameters can be set: the lengthscale and the variance (l and σ in (5.3)), and the Gaussian noise (σ_N in (5.4)). The maximization in the policy improvement step is implemented by brute-force gridsearch: we simply compute the values for all admissible actions and choose a maximal value (if there are several maxima, we choose the one discovered first). We note that the algorithm is able to handle continuous variables and the discretization is only performed to enable the comparison with the benchmarks.

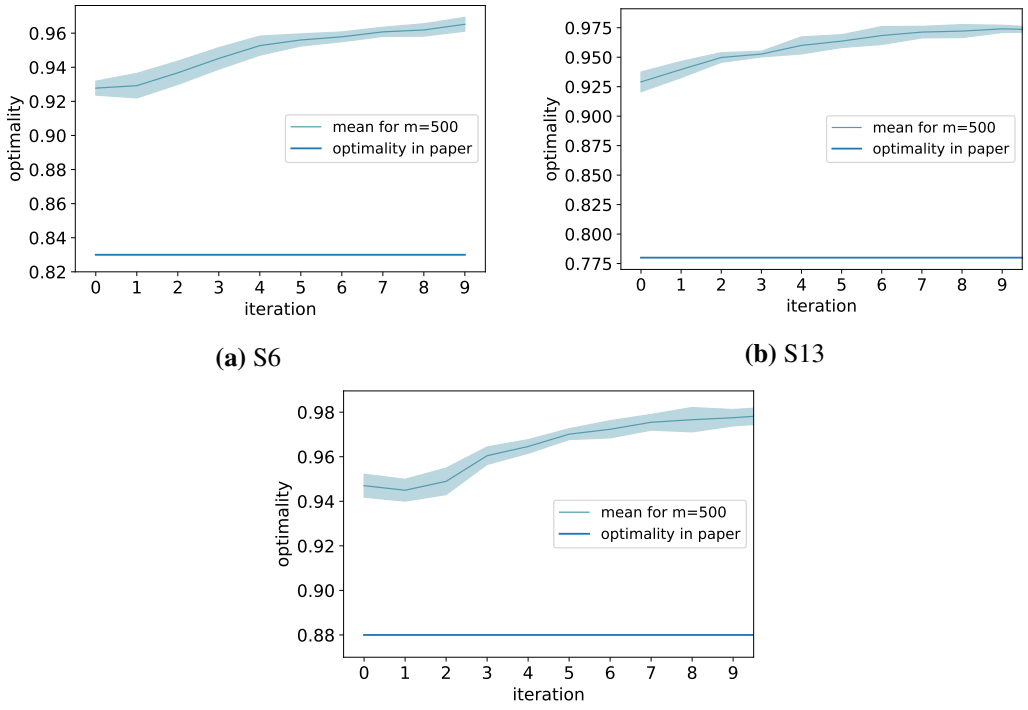
We note that some parameters and essential choices for the computation are not specified in the [JPP+14]; in particular the choice of initial policy, the number of samples in the policy evaluation step, and the hyperparameters of the GPR. As default values we choose the discretized version of the initial policy described in (5.8) (to comply with the discretization in the benchmarks), 500 samples in the policy evaluation step ($m = 500$ in Algorithm 2), and fixed hyperparameters ($l = 1, \sigma = 1, \sigma_N = 1$).

In the following, we will compare the performance of our implementation to the one in [JPP+14] on three different test sets, the benchmark datasets S6, S13, S17. In all three cases, the price shocks are modelled according to a discretized form of the normal distribution $\mathcal{N}(0,1)$ (called discrete pseudonormal distribution in [JPP+14]). The price is modelled as a first-order Markov chain, in S6 and S13 we additionally have jumps (see [JPP+14]). The wind energy shocks are modelled uniformly ($\mathcal{U}(-1,1)$) in S6 and discrete normally in S13 ($\mathcal{N}(0,0.5^2)$) and S17 ($\mathcal{N}(0,1)$).

Despite comparatively small differences, the difference in performance, as given in [JPP+14] is comparatively big: After 10 iterations of the approximate policy improvement algorithm as described in the previous chapter, 83% optimality (measured as in (6.1)) are obtained for S6, 78% for S13, and 88% for S17. We chose these three datasets as they represent the range of performance in Jiang et al.[JPP+14]: S13 is the worst-performing dataset, S6 is average and S17 is one of the best-performing datasets. If not specified differently, we always use the dataset S6 in the computations below.

Improvement of performance. In our numerical computations we obtain significantly better results than the results of Jiang et al. in [JPP+14], this comparison is shown in Figure 6.2. With our default values as described above we obtain after 10 iterations an average of 96.5% over 5 different random seeds. For S13 and the same setting, we obtain 97.6%, and for S17 we obtain 98.2%. This is an improvement of about ten percentage points compared to the results of Jiang et al. Also note that the variation in performance across the datasets is much smaller than in [JPP+14]. As there are only minor differences in the model parameters for each dataset, this seems plausible. As before, the best results are obtained for S17. This might be attributable to the presence of jumps in S6 and S13 and the consequently stronger stochasticity: With jumps, there can be a higher approximation error in the policy evaluation step, as a sample path with a price jump may be less representative of the actual value of the policy.

Note that these optimality values do not represent the best performances of the algorithm: In the majority of the computations the optimality rate had not converged yet and continued to increase (in computations over 20 iterations the optimality rate increased by up to $1-2\%$ and then converged). However, in [JPP+14] only the values for 10 iterations are computed and we therefore limit the number of iterations to 10 in order to be able to compare them to the results in [JPP+14].



(a) S6

(b) S13

(c) S17

Figure 6.2: Comparison on multiple datasets

Two factors may explain why we obtain better results: the choice of hyperparameters and the choice of initial policy. Both choices were not specified in [JPP+14]. In the following we

illustrate the impact of each choice on the performance of the algorithm.

In order to put the optimality results of the algorithms presented in this chapter into context, it is worth noting that for the VFA $\tilde{V} \equiv 0$ the optimality is roughly 57% (for S6 and $m = 500$). That is, we actually perform policy improvement without considering any value function at all (the policy evaluation becomes redundant and we only need one iteration), and we only optimize the reward for the current time period. The algorithm thus reduces to a simple greedy algorithm. The comparatively high optimality comes from the fact that the optimality is measured by cumulated profit. However, merely satisfying demand (which is a binding constraint in every time step) already generates profit in most cases.

Choice of initial policy. We find that the choice of the initial policy drastically affects the outcome of the algorithm, see Figure 6.3.

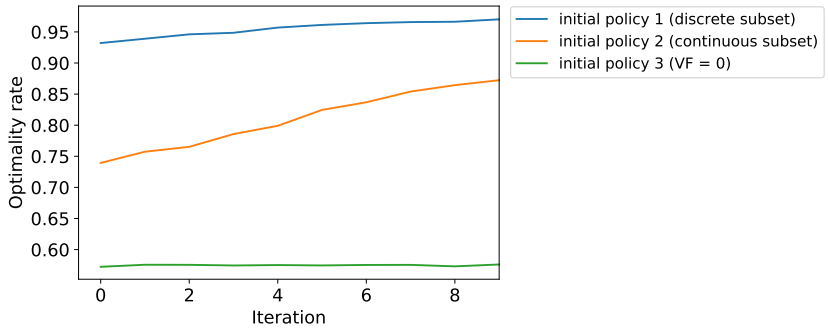


Figure 6.3: Comparison of initial policies

Here, "initial policy 2" denotes the policy described in (5.8). "Initial policy 1" represents the discretized version of this policy, that is, we only sample from the points which additionally lie on the discretization grid. "Initial policy 3" represents an initial policy where the policy is again implicitly defined by the maximization of an objective function including a VFA, which is here set to zero for all states. In other words, we consider a greedy initial policy which only maximizes over the rewards.

We see that the greedy initial policy performs worst and does not display any improvement tendency. It performs at roughly the same level as the simple greedy algorithm without VFA. This seems to suggest that the VFA can not successfully improve the values obtained by the greedy policy. Initial policy 1 and 2 display some improvement over the number of 10 iterations. Initial policy 1 improves at a much slower rate than initial policy 2 but is on a much higher absolute level. Note that initial policy 1 starts of with a much higher optimality rate than initial policy 2 (20 percentage points higher), though they are basically the same, apart from the discretization in initial policy 1 [18]. A reason for this could be that since we only consider the discretized values in the algorithm, we obtain much better samples of the real value functions if

[18]Initial policy 2 works despite not using the discretized values: Note that the initial policy only influences the samples for the fitting of the first GPR, and the GPR can handle continuous values.

we sample directly from the discretized state space (we get information directly from the grid points instead of the points around them).

As a conclusion we see that the choice of the initial policy has a significant and extremely big impact on the performance of the algorithm. Already within these three initial policies we have a difference in performance of almost over 40%. These results reflect the fact that the initial policy is of crucial importance for the fitting of the VFA: In order for the algorithm to work well, we need to ensure that the approximate value samples $v_{1,t}$, and thus the VFA $V^p_{1,t}$, in the first iteration of the algorithm are not too inaccurate. If the VFA is too inaccurate, the algorithm does not recover and stagnates (see "initial policy 3"). (The first VFA $V^p_{1,t}$ is based on the cumulative rewards $v_{1,t}$ of the sample paths in the first iteration. But if the $v_{1,t}$ are very far from reflecting the actual value of a state, our VFA is a poor approximation of the actual value function. This poor VFA then in turn creates inaccurate value samples in the policy evaluation step of the next iteration, which then lead to another poor VFA and so on.) The only way for us to influence the quality of the value samples in the first iteration is the choice of initial policy, it is therefore of crucial importance.

Hyperparameter tuning. The second algorithmic choice that hugely influences the performance of the algorithm is the choice of hyperparameters for the Gaussian Process Regression. Recall that we have three hyperparameters: the lengthscale l, the signal variance σ and the Gaussian noise variance σ_N. An unsuitable choice of hyperparameters leads to a bad fitting of the Gaussian Process Regression to the approximate value samples which in turn leads to a bad approximation of the GPR to the true value function. In this case we test two values for each of the two hyperparameters and consider all hyperparameter configurations from the product space $\{0.5, 2\} \times \{0.2, 1\} \times \{0.2, 1\}$. For each configuration we compute the mean over two random seeds, the results are displayed in Figure 6.4, as well as the mean for $l = 1, \sigma = 1, \sigma_N = 1$ from Figure 6.2(a).

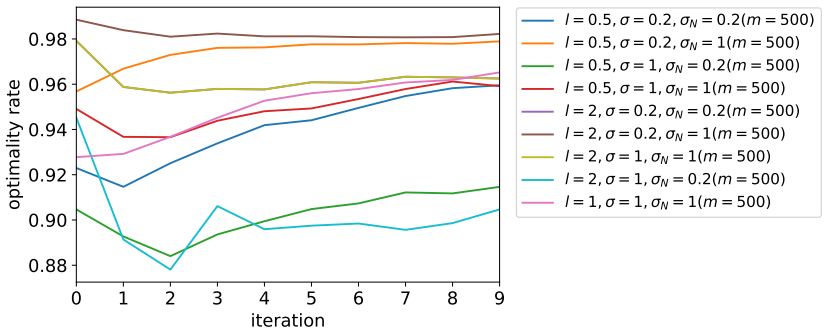


Figure 6.4: Comparison of hyperparameters ($m = 500$)

The best performance is achieved by a small signal variance ($\sigma = 0.2$) and a bigger Gaussian noise variance ($\sigma_N = 1$); for either lengthscale we achieve 98% after 10 iterations. In particular we achieve a better result than for our default values $l = 1, \sigma = 1, \sigma_N = 1$. The good performance for $\sigma = 0.2$ and $\sigma_N = 1$ may reflect the fact that the true value function does not have a

high variance, but that our approximate value samples are in fact quite noisy: Since our sample paths are based on our inaccurate value functions, our samples are not not very good samples of the true value function. Interestingly, although we find similar values for both lengthscales, the behaviour of the optimality graphs is different: For $l = 2$ we obtain an approximately monotonically decreasing graph, while the graph for $l = 0.5$ is monotonically increasing. However, this might just be a stochastic difference, as we only consider two random seeds each.

For the opposite configuration (a bigger signal variance ($\sigma = 1$) and a smaller noise variance ($\sigma_N = 0.2$)), we obtain only around 91% - 92% optimality and thus the worst performance among the combinations tested. This can be attributed to overfitting: With a higher signal variance and a lower noise variance, we assume that our training outputs are very close to the exact functional values of the training inputs. Thus, if there is high variance between the training outputs, we assume this is due to the underlying true function having high variance there. We thus end up modelling the noise of these specific training points (the real noise is unknown, it might be quite high) instead of the real function. However, we note that even the worst hyperparameters tested in Figure 6.4 performs better than the results by Jiang et al [JPP+14].

We observe that the graph for $H_1 := (l = 2, \sigma = 0.2, \sigma_N = 0.2)$ is not visible, as the values in our computations happen to be identical to those for $H_2 := (l = 2, \sigma = 1, \sigma_N = 1)$. Though the respective Gaussian Processes themselves are not identical, they are still similar enough that the maximization in Line 8 of the algorithm yields the same results. (To explain this, consider (5.1): Since the lengthscale is the same for H_1 and H_2, and since we have $\sigma = \sigma_N$ in both cases, the respective Gaussian Processes for H_1 and H_2 only differ by a constant and thus display similar behaviour.)

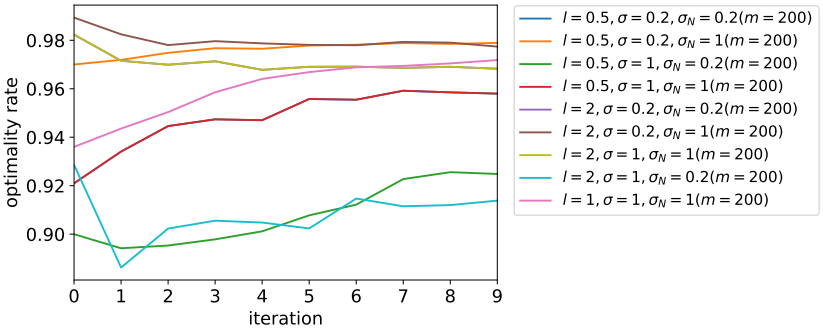


Figure 6.5: Comparison of hyperparameters ($m = 200$)

For a smaller sample size ($m = 200$) we obtain similar results, see Figure 6.5. Though the behaviour of the graphs is slightly different from the ones in Figure 6.4, the ranking of the hyperparameter combination among each other is still the same (with the only exception of the two best-performing combinations (brown and orange graph), which basically yield the same optimality from iteration 5 onwards).

Again we have identical values for $(l = 2, \sigma = 0.2, \sigma_N = 0.2)$ and $(l = 2, \sigma = 1, \sigma_N = 1)$. Additionally, we obtained identical values for $(l = 0.5, \sigma = 0.2, \sigma_N = 0.2)$ and $(l = 0.5, \sigma = 1, \sigma_N = 1)$.

We show the results for both $m = 200$ and $m = 500$ in the same figure in Figure 6.6. Comparing the absolute optimality rates, we see that for $(l = 2, \sigma = 1, \sigma_N = 1)$, $(l = 2, \sigma = 0.2, \sigma_N = 0.2)$, $(l = 0.5, \sigma = 1, \sigma_N = 0.2)$, $(l = 2, \sigma = 1, \sigma_N = 0.2)$ the algorithm performs better for the smaller sample size. That is, $m = 200$ obtains better results in the case that the signal variance is bigger and the noise variance smaller, as well as in the case that the lengthscale is bigger and the variances are set to the same value. For the other hyperparameter combinations, there is no significant or only a small difference in performance between the two sample sizes. Among the hyperparameter and sample size combinations tested, the best performance was achieved by $l = 2, \sigma = 0.2, \sigma_N = 1$ and $m = 500$.

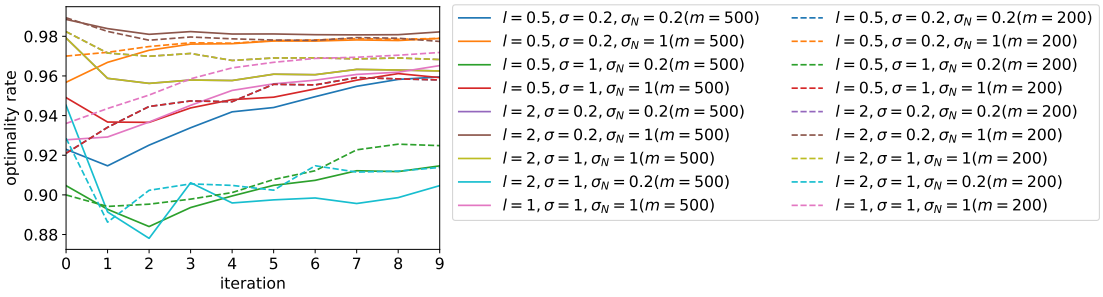


Figure 6.6: Comparison of hyperparameters ($m = 200, 500$)

We conclude that different sample sizes require different sets of hyperparameters, and that our default choice can be improved. However, this does not yield a general rule how to choose the hyperparameters optimally. We consider an automated approach to optimize the hyperparameters in the following paragraph.

Automated hyperparameter tuning. As described in Section 5.1, the optimization of the hyperparameter choice can be automated using the marginal likelihood. In our implementation we used the built-in optimization in [GPy12].

However, the results are not encouraging: We found that after the optimization (maximizing the marginal likelihood) all three hyperparameters are set to unrealistically high values (of order 10^6 or higher). Due to the extremely large lengthscale the Gaussian kernel is zero almost everywhere. Moreover, with such a large noise variance, all of the data is de facto interpreted as noise. The obtained Gaussian Process subsequently has only very little explanatory power and is almost constant everywhere. As a result of this poor VFA, the performance of the algorithm decreases over the course of iterations (in our experiments we obtained 77% optimality after 10 iterations).

Reasonable "optimized" hyperparameter values (that is, values not greater than order 10^1) were only achieved if in the fitting of the GPR the target values were additionally scaled. However, scaling the target values in itself already seems to worsen the performance significantly, see Figure 6.7. Additionally optimizing the hyperparameters via the marginal likelihood only seemed to deteriorate the performance further.

The high noise after optimization is the result of the fact that the matrix K in (5.6) is very

ill-conditioned. As a result, the matrix $K + \sigma_N^2 I$ is only numerically invertible for an extremely large σ_N^2. However, while this stabilizes the inversion, adding the identity matrix multiplied with a constant of order 10^6 or greater highly distorts the covariance matrix, and the end result is very far away from the true solution. This is a possible explanation why the hyperparameter optimization does not work in this case.

It is still unclear why the matrix K is so ill-conditioned. One idea that we tested is that the generated samples are "too close" to each other. We tried removing points which were "too close" to one of the other points, where "too close" is defined by some threshold distance (we tried various threshold distances). We did not obtain any significant improvement in the condition number of K however. Another idea is that the bad condition arises from the dependency between the samples. Though the starting states are sampled independently, the sample paths in the policy evaluation (lines 2-18 in Algorithm 2) all develop according to the same VFA. The sample paths therefore do not develop independently. In particular, the sample storage values $r_{i,t}$ tend to concentrate around the same values; in the beginning (for small t) storage is quite high and then goes down to zero towards the end (see also under Section 6.3).

A further possible reason for the poor performance of the hyperparameter optimization is that at the end of every iteration of the algorithm, a new Gaussian Process is fitted for every time step. Theoretically, the hyperparameters obtained after the optimization can change drastically from iteration to iteration, as they are highly dependent on the samples obtained in the respective iteration. This instability is alleviated if we fix the hyperparameters, as for the results in Figures 6.2 - 6.6.

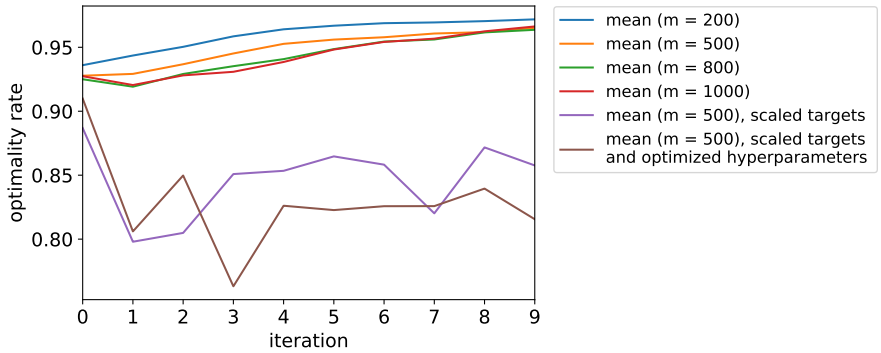


Figure 6.7: Comparison with automated hyperparameter optimization

Effect of sample size. Increasing the sample size does not necessarily improve the optimality rate: For all other parameters fixed, we compare the performance of the algorithm for various sample sizes m. The mean over five different randomseeds is shown in Figure 6.8 for four different sample sizes. We see that there is no clear correlation between sample size and performance. Indeed, the smallest sample size ($m = 200$) surprisingly performs best, while the other sample sizes end up with very similar optimality rates after 10 iterations. This might seem counterintuitive: A higher sample size corresponds to more data for the fitting of the Gaussian Process and we expect that more data can only improve the result, not worsen it. To understand this, we

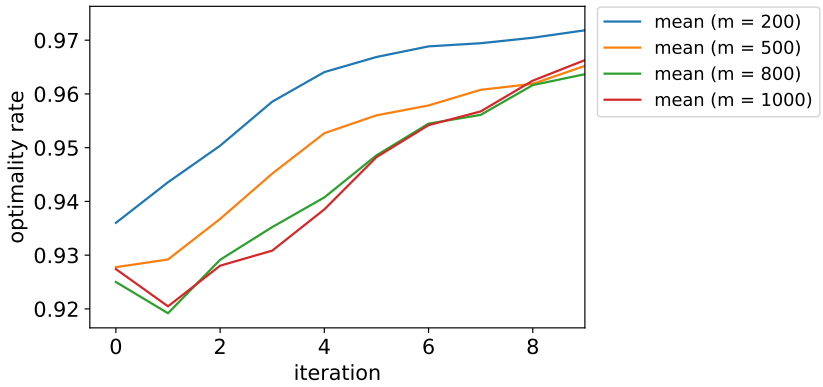


Figure 6.8: Comparison of different sample sizes

have to take into account that all results in Figure 6.8 were computed with the hyperparameters fixed to one (our default hyperparameters). However, the optimal choice of hyperparameters also depends on the sample size. As we saw in Figure 6.6, depending on the hyperparameter combination, either of two sample sizes can be better, no sample size dominates.

Effect of scaling. We discuss two kinds of scaling to improve the performance of the algorithm: feature scaling and target scaling. Feature scaling is a common step in machine learning to preprocess raw data, since many algorithms do not work without it or only work much slower. There is no canonical scaling method, popular choices are standardization, normalization and scaling to unit length. In this case, we specifically aim to improve the accuracy of the GPR in line 17. It is fitted on the inputs $s_{i,t}$ (states) and the outputs $v_{i,t}$ (approximate value function samples).

Firstly, we scale the features $s_{i,t}$ by standardizing them (that is, we remove the mean and scale to unit variance). We use the "StandardScaler" from [PVG+11]. In Figure 6.9 we see that without standardization, a lower optimality rate is achieved than with standardization (compare with Figure 6.8). Additionally, the graph is characterized by a much higher variance than its counterpart with scaling. We attribute this to the scaling making the inverting of the matrix in the calculation of the prediction (5.5) numerically more stable, as the matrix is very ill-conditioned (see above).

We note that while scaling the inputs improves the performance, scaling the targets results in the opposite (see Figure 6.7). This might be because the targets (value samples) are more affected by stochasticity than the inputs: The value samples are computed as sums of rewards, which in turn are computed as products of price and demand as well as price and storage. This "squares" the stochasticity in some sense. Indeed, we find that the target values can vary from each other by several orders of magnitude, while the state components all range in the same order of magnitude. However, considering that the linear scaling of the target $\mathbf{y}$ should only affect the result linearly in (5.5), it is surprising that it affects the performance at all. Why this is, remains an open question.

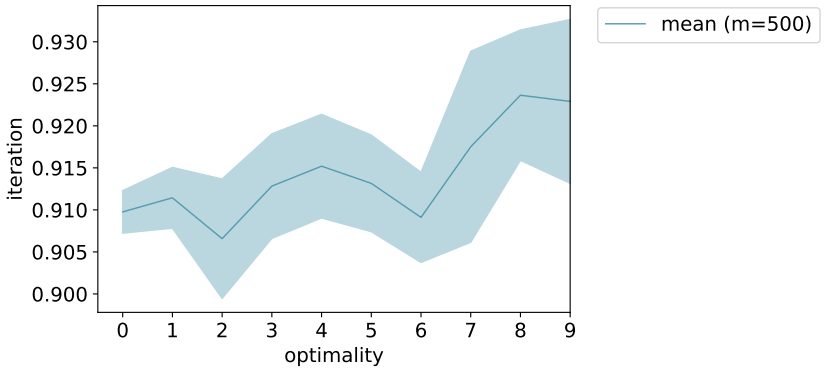


Figure 6.9: Algorithm 2 without feature scaling (mean over 3 random seeds, with standard deviation)

6.3 Error measures

We will now discuss other ways to evaluate the performance of the algorithm, in particular we try to capture the approximation error in several ways.

Mean squared error to optimal decision. So far we have only measured the performance of the algorithm based on the optimality ratio compared to the benchmark solution. That is, we have only compared the *obtained total profits* on a set of sample scenarios.

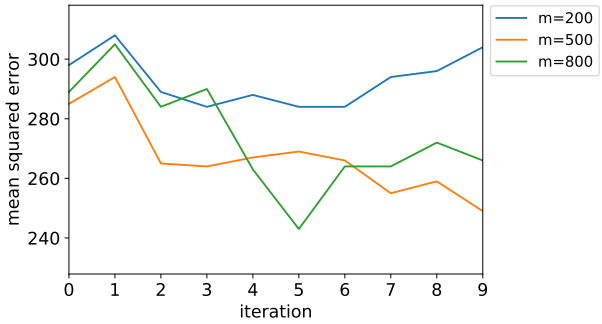


Figure 6.10: Mean squared error

Since the optimal decisions are provided in the datasets, we can measure the distance between *the optimal decisions* and the decisions computed by the algorithm 2 (for the sample scenarios which we also used to compute the optimality rate in (6.1)). We compute the distance to the optimal decisions in the following way: For every time step we compute the mean squared error of the computed decision to the optimal decision (the decisions are six-dimensional, and the ranges are similar for each dimension). Then for every sample path in the policy evaluation we compute the mean squared error over all time steps (we compute the error to the reference value zero), and then for every policy iteration step we compute the mean squared error of the errors

for each sample path in the policy evaluation. The results for three different sample sizes are shown in Figure 6.10.

We see that the mean squared error seems to go down for $m = 500$ and $m = 800$, however it does not decrease for $m = 200$ for which the algorithm actually achieves the best total performance (see Figure 6.8). Note however that a near-optimal policy does not necessarily need to be close to the provided optimal policy. The set of near-optimal policies can be quite large. The aim is only to design a policy that is near-optimal; the VFA, and thus the approximate optimal policy, can differ from their exact counterparts. Therefore, while a small mean squared error would be an indicator of near-optimality, a high and non-decreasing mean squared error is not necessarily an indicator for bad performance.

Optimal storage paths. We can also compare the optimal storage paths (again for the sample scenarios used for the computation of the optimality rate) to the ones computed by the algorithm. That is, we now compare the *outcome* of the optimal actions versus the outcome of the actions computed by the algorithm 2.

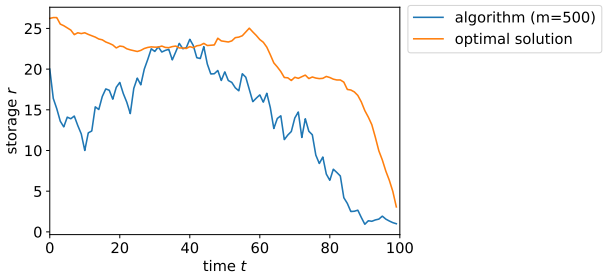
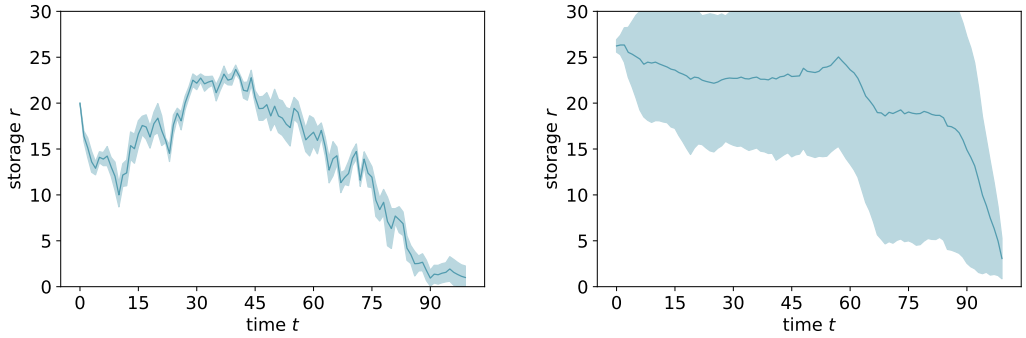


Figure 6.11: Comparison of storage paths (mean over 30 samples)

In Figure 6.11 we compare the mean of the optimal storage paths from the benchmark datasets to the mean of the ones obtained by the algorithm after 10 iterations (where the mean is computed over the same 30 sample scenarios). Though the rough behaviour is the same ("keep the storage well filled for the first half of the time period, then empty it over the second half"), we see that there is quite a big difference between the two means. In particular, the optimal solution maintains a higher storage on average. We also see for each of the 30 samples (not shown here) that the storage paths chosen by the algorithm and the benchmark differ substantially. This goes hand in hand with the observation from above that the computed decisions differ strongly between algorithm and benchmark solution.



(a) Algorithm 2 **(b)** Benchmark

Figure 6.12: Comparison of mean sample paths with standard deviation

Figure 6.12 depicts the means from Figure 6.11 with their respective standard deviation. The standard deviation for the benchmark is much higher than for the algorithm.

This might be due to the specific structure of the computed VFA in the algorithm. We first note that in the policy improvement step (line 8 in Algorithm 2), we are given a price p and an amount of available energy w, only the storage value can be changed (within a small interval). The VFA however does not vary a lot in the price direction. The graphs

$$\{((r,w,p), V_{n,t}(r,w,p)) \mid r \in [r-5, r+5]\}^{19}, \quad w \text{ and } p \text{ fixed},$$

do not vary too much across either wind or price. Even when the absolute values vary, the essential behaviour of the function (especially with regard to optima) remains the same. Consider for instance the VFA after 10 iterations in Figure 6.13. In time $t = 0$ we start at $r_0 = r_{start} = 25$. For every price p, the behaviour of the value function $V_{n,t}(\cdot, 4, p)$ in the interval $[20, 30]$ is basically the same (except near the maximal and minimal price), and favours a maintaining of the storage value or even a slight reducing. Thus, considering that for all sample paths the start value for the storage is the same and considering that the value function tends to dominate the contribution function in the policy improvement objective function (the value function as an expectation of the cumulated profit to be obtained in the future tends to be much bigger than the profit of a single time step), we end up with a similar storage development. This carries through the other time steps $0 < t \leqslant T$ as well.

Not only do the storage sample paths stay close to the mean over all 30 evaluation sample paths of a single dataset, the mean also does not vary much across different datasets. In the figure in Table 1 we consider three different datasets, with different energy and price processes (where the difference is expressed mainly in the variance of the energy and price shocks). However, the means over the evaluating sample paths are very similar.

Training and test error. As another way of evaluating the performance of the algorithm we look at the development of the training errors and test errors. In Figure 6.14 we plot the training and the cross-validation test error. These are computed on a set S_{or} of state and target samples,

[19] It suffices to consider an in-/decrement of 5, as this is the maximal dis-/charge in this case.

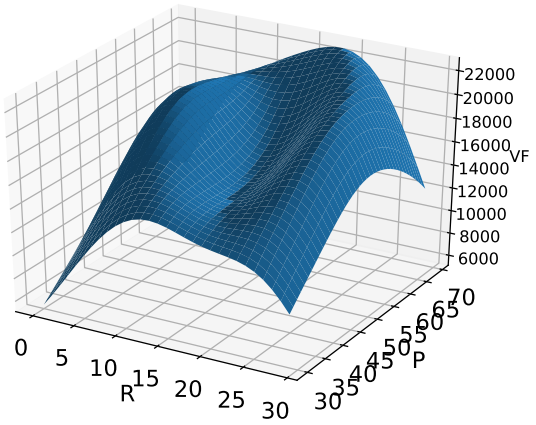


Figure 6.13: VFA after 10 iterations ($t = 0, w = 4, m = 500$)

Dataset	$\hat{e}_t$	$\hat{p}_t$
S6	$\mathcal{U}(-1,1)$	$\mathcal{N}(0,1^2)$
S12	$\mathcal{N}(0,2^2)$	$\mathcal{N}(0,5^2)$
S13	$\mathcal{N}(0,0.5^2)$	$\mathcal{N}(0,1^2)$

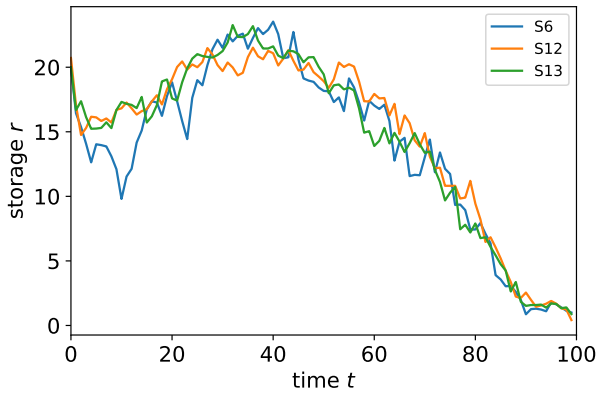


Table 1: Mean sample paths over different datasets (with differently distributed energy and price shocks, $\hat{e}_t$ and $\hat{p}_t$)

which we obtain in the 10th iteration in line 17 of the algorithm ($m = 1000$). Then, for every $N_u \leqslant 1000$ we repeat the following a 100 times: We divide a random subset $S_{N_u} \subset S_{or}$ of the original sample set, where $|S_{N_u}| = N_u$, randomly into a training set (90% of S_{N_u}) and a test set (10% of S_{N_u}). We train the GPR on the training set and then cross-validate it on the test set. For both we compute the mean squared error to the respective target samples of S_{or} (we assume that the target samples are a close approximation to the true value functions). We then average over all 100 errors for N_u, respectively for the training error and the test error. We see that both the training and test error are quite high and plateau relatively fast. (Note though, that this is not the true error: Since we do not have the true value function values, we can only compute the error to our targets, which are approximate value samples. Depending on how good these approximations are, the training and test error might look different.) This error behaviour seems to suggest a high modelling bias and an underfitting problem [Ng17]. The

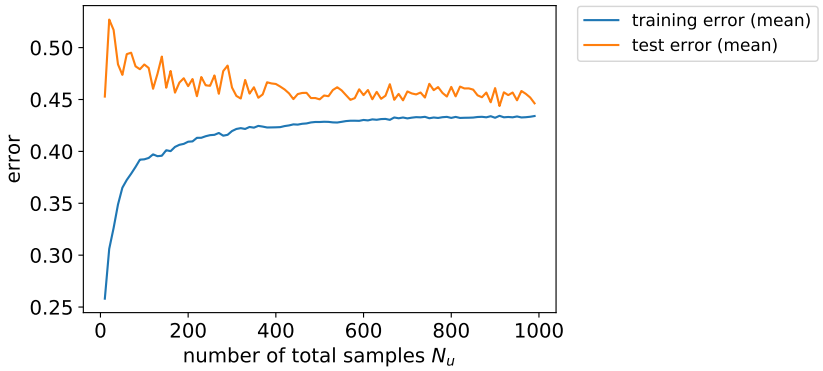


Figure 6.14: Training and cross-validation test error

reason for this could either be that GPR is in general not a very accurate modelling choice for the value function, or that the hyperparameters are simply not chosen well enough. Note also that the errors are surprisingly high considering that the algorithm achieves a tolerably good performance (for $m = 1000$, see Figure 6.8).

6.4 Computational aspects

The bottlenecks of the algorithm are the fitting of the Gaussian Process Regression in the policy evaluation step and the gridsearch optimization in the policy improvement step.

The fitting of the GPR varies hugely over time: For 500 training points, the fitting for one single regression ranges between 2 and 20 seconds. The mean lies at around 6 seconds, the median slightly below that. For 800 training points, the fitting for one single regression ranges between 2 and 40 seconds. The mean lies at around 9.5 seconds, the median at 8.5 seconds.

A single gridsearch optimization takes less time: For 500 episodes (the number of episodes influences the complexity of the GPR, therefore also the objective function and the length of the gridsearch) one single optimization takes between 0.2 and 1.6 seconds, with a mean of 0.6 seconds, for 800 episodes it takes between 0.2 and 2 seconds, with a mean of 0.85 seconds.

However, since the gridsearch optimization is performed many times more often than the fitting of the Gaussian Process Regression, it affects the total computational time more than the GPR fitting. In total we have $|\text{iterations}| \times |\text{episodes}| \times |\text{timesteps}|$ optimizations, compared to $|\text{iterations}| \times |\text{episodes}|$ fittings; in this case we thus perform $|\text{timesteps}|$=100 optimizations for every fitting. Making the gridsearch more efficient and faster would therefore yield a significant improvement in computational performance. In total, the algorithm needs 3 days for 500 samples and a week for 800 samples. Summing up the total amount of time needed for optimization and fitting, we see that these two steps make up the total computation time for the whole algorithm.

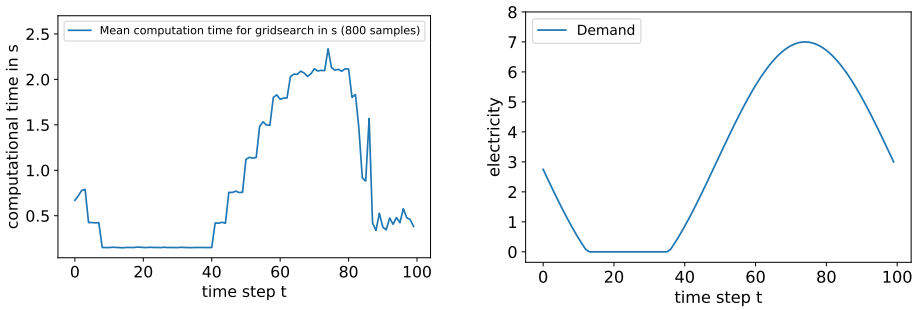
We already consider a more efficient version of gridsearch optimization as we do not go over the complete action space. This is because in the gridsearch optimization we need to evaluate the objective function (reward plus VFA) only for the *admissible* actions. The advantage then lies in the fact that the set of admissible actions is on average much smaller than the whole set of actions: In our computations it constituted on average under 1% of the entire (discretized)

action space [20], often even significantly lower. However, even though we do not necessarily need to evaluate the objective function for each action, we still need to check for each action *if* the resulting state is feasible. We can reduce the computational complexity of this check by exploiting the constraints (3.2): These constraints yield upper bounds for the variables. Thus, the search can be limited to

$$a_t = (a_t^{wd}, a_t^{rd}, a_t^{gd}, a_t^{wr}, a_t^{gr}, a_t^{rg}) \in A := [0, c_3] \times [0, c_1] \times [0, d_t] \times [0, c_2] \times [0, c_2] \times [0, c_1], \quad (6.2)$$

where $c_1 = \min\{r_t, \gamma_d\}$, $c_2 = \min\{r_{max} - r_t, \gamma_c\}$ and $c_3 = \min\{w_t, d_t\}$. This reduces the computational burden: In our case, the feasible points now make up between $2.5\% - 10\%$ of the constrained action space A.

At closer inspection, we see that the length of the optimization step is closely linked to the size of demand, as shown in Figure 6.15:



(a) Average time for the optimization

(b) Demand

Figure 6.15: Relation between optimization length and demand function

This can be attributed to the structure of (6.2): Of the relevant variables r_t, w_t, d_t, demand most directly influences the size of A. In most stages, the storage exceeds the discharge capacity but is also well under the maximum storage amount. We then have $c_1 = \gamma_d$ and $c_2 = \gamma_c$, therefore the storage does not influence the size of the action space A in practice. It is only in the later stages (around $t = 80$) that we typically have storage close to zero while demand is not zero. This is also where the plot of the policy improvement length departs more strongly from the shape of the demand curve. The discrete nature of the state space is reflected in the stepfunction-like ascent in the function.

[20] Here we consider the continuous action space as

$$\mathcal{A} = [0, \max d] \times [0, \max d] \times [0, \gamma_d] \times [0, \gamma_c] \times [0, \gamma_c] \times [0, \gamma_d],$$

where d is again demand. Considering that for these benchmark problems $\max d = 7$, $\gamma_c = \gamma_d = 5$, and the discretization step is of size one, the discretized action space then has $8 \times 8 \times 6 \times 6 \times 6 \times 6 = 82944$ actions in total.

6.5 Summary of the evaluation and outlook

In this chapter, we tested and analysed the approximate policy iteration algorithm using GPR as a VFA which originates from [JPP+14]. Our main result is that with the right choice of parameters, the algorithm performs much better than previously found: In particular, with a suitable choice of the initial policy and of the hyperparameters we achieved an increase in optimality rate of 10 percentage points. Both choices determine the range of optimality as well as overall behaviour (for example whether the performance improves over the course of iterations or not). The big impact of the initial policy choice (see Figure 6.3) reflects the fact that the initial policy is the driving factor of the quality of the approximate value samples on which the first VFA is trained. We also note that while we use general purpose methods (API, GPR), the algorithm's good performance relies on a problem-specific choice of initial policy.

The choice of hyperparameters is equally important. Among the values tested we found that a fixed small signal variance and a bigger noise variance work best. Hyperparameter optimization via maximization of the marginal likelihood did not yield optimal parameters and instead lead to a poor performance. Our results suggest that this failure is not due to numerical instabilities caused by "too close" points, but it remains an open question why exactly the hyperparameter optimization does not work. We also saw that the choice of hyperparameters depends on the sample size used in the policy evaluation. Consequently it is not clear whether (with a right choice of hyperparameters) increasing the sample size can always improve the performance of the algorithm.

Scaling the state variables improves the performance of the algorithm, while scaling the target variables fundamentally worsens it. The former can be attributed to increased numerical stability in calculating the inverse in the evaluation of the GPR. The latter may be caused by the high variance of the target values (in particular the targets have a higher variance than the state variables).

However, although the results are promising, the approach might be limited: We observe that training and test error are high and do not improve, even when overall performance is good. This seems to suggest a high bias in the modelling and that the GPR may not be a very accurate model for the true value function. We also note that the approximate optimal policy computed by the algorithm varies significantly from the exact optimal policy. Similarly, the storage paths produced by the approximate optimal policy do not coincide with the exact optimal storage paths, though the overall behaviour is the same. In particular, the algorithm stores less on average and has a lower variance in storage outcomes. However, deviation from the optimal policy and the optimal storage paths is not necessarily an indicator for bad performance.

We suggest the following ideas for further investigation and for further improvement of the algorithm: It still remains to find a rule for the optimal choice of hyperparameters. Automated hyperparameter optimization using marginal likelihood does not seem to work. Gridsearch is a possibility, but is in general too expensive, since we need to run the whole algorithm to evaluate the performance (the optimality rate). Additionally, the performance can potentially be increased by improving the VFA: Proposition 3.1 shows that the true value function is concave in the storage dimension. As of now, this is regularly violated in the implementation (consider for example Figure 6.13). Enforcing concavity through shape-constrained GPR as described in [WB16] might thus reduce the approximation error (it might also address the high bias from

above) and consequently lead to better results. Likewise, testing other kernels in the GPR might address the high bias problem. Another starting point for improving the algorithm can be to choose a different method of sampling of initial states in the policy evaluation step (see Line 6 in the algorithm). Currently this is done by uniform sampling, but of course the states do not all have the same likelihood under the true stochastic exogenous information processes.

7 Conclusion

In this thesis we investigated and improved the performance of an existing approximate policy iteration approach by Jiang et al. [JPP+14] using Gaussian Process Regression and applied it to a wind energy storage problem.

For that purpose, we first gave an extensive overview over the theoretical framework of Dynamic Programming (DP) in Chapter 2 and derived the policy iteration method from the Dynamic Programming Principle. We discussed convergence and other theoretical properties. In Chapter 3, we provided a detailed description of the wind energy storage application. The problem in this storage application is to find a profit maximizing storage policy which smoothens fluctuations in wind energy, price and demand. We motivated the modelling decisions and assumptions. In Chapter 4 we generalized the DP methods from Chapter 2. We presented Approximate Dynamic Programming (ADP) methods such as value function approximation and discussed other options for algorithm design.

As the final building block of the necessary theoretical groundwork for the algorithm we gave a detailed introduction to Gaussian Process Regression (GPR) in Chapter 5. We explained the main ideas and addressed the topic of hyperparameter selection using marginal likelihood as a measure of optimality. We then presented the approximate policy iteration approach using GPR by Jiang et al. [JPP+14] and explained how the ADP methods were applied in this specific algorithm. We proposed a heuristically motivated initial policy choice. We also addressed how this specific wind energy storage problem was solved in the literature.

In Chapter 6, we analysed the approximate policy iteration algorithm in detail and discussed our own implementation choices. We found that with our choice of initial policy and a suitable choice of hyperparameters of the GPR we could achieve an improvement of about 10 percentage points (yielding an overall optimality rate of about 97%). These results show that general purpose methods such as approximate policy iteration using popular machine learning methods as value function approximations perform better than previously found in the literature [JPP+14]. We note however that the good performance of the algorithm relies on incorporating problem-specific information in form of an initial policy tailored to the application. We illustrated the impact of the initial policy on overall performance and demonstrated the effectiveness of our heuristically motivated initial policy choice. We analysed the impact of the hyperparameter choice on overall performance. We found that for this application, GPR with a small signal variance relative to the noise seems to work best. Additionally, we showed that feature scaling improves the algorithm performance significantly.

Though our results are promising and imply that the approximate policy iteration can be an effective choice, a high training and testing error suggest that this particular choice of value function approximation may not be a very accurate choice. Possible steps to improve the model choice would be to test other kernels or to add shape constraints in the GPR to exploit properties of the underlying value function such as concavity. Another important question that requires further investigation is how to choose the hyperparameters of the GPR optimally. We found that model selection by maximizing the marginal likelihood did not achieve desirable results. Similarly, the tuning of the initial policy could increase the optimality further.

References

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Ber11] D. P. Bertsekas. Approximate policy iteration: a survey and some new methods. *Journal of Control Theory and Applications*, 9(3):310–335, Aug 2011.

[Ber17] D. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1-2. Athena Scientific, 4th edition, 2017.

[BMZ09] O. Bokanowski, S. Maroso, and H. Zidani. Some convergence results for Howard's algorithm. *SIAM Journal on Numerical Analysis*, 47(4):3001–3026, 2009.

[BR11] N. Bäuerle and U. Rieder. *Markov Decision Processes with Applications to Finance*. Springer-Verlag, 2011.

[BS07] D. P. Bertsekas and S. E. Shreve. *Stochastic Optimal Control: The Discrete-Time Case*. Athena Scientific, 2007.

[BSB10] L. Buşoniu, B. D. Schutter, and R. Babuška. Approximate dynamic programming and reinforcement learning. In *Interactive collaborative information systems*, pages 3–44. Springer-Verlag, Berlin, Heidelberg, 2010.

[CBJK08] L. M. Costa, F. Bourry, J. Juban, and G. Kariniotakis. Management of energy storage coordinated with wind power under electricity market conditions. In *Proceedings of the 10th International Conference on Probabilistic Methods Applied to Power Systems*, pages 259–266. 2008.

[Duv14] D. Duvenaud. *Automatic model construction with Gaussian processes*. PhD thesis, University of Cambridge, 11 2014.

[EAS17] C. Erickson, B. E. Ankenman, and S. M. Sanchez. Comparison of Gaussian process modeling software. *European Journal of Operational Research*, 2017.

[ES98] L. Evans and A. M. Society. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 1998.

[FF13] M. Falcone and R. Ferretti. *Semi-Lagrangian Approximation Schemes for Linear and Hamilton-Jacobi Equations*. Society for Industrial and Applied Mathematics, 2013.

[FS02] E. A. Feinberg and A. Shwartz. *Handbook of Markov Decision Processes - Methods and Applications*. Springer US, 2002.

[GP01] G. A. Godfrey and W. B. Powell. An adaptive, distribution-free algorithm for the newsvendor problem with censored demands, with applications to inventory and distribution. *Management Science*, 47(8):1101–1112, 2001.

[GPy12] GPy. GPy: A gaussian process framework in python. `http://github.com/SheffieldML/GPy`, since 2012.

[GWE] Global Wind Energy Council. *Wind in numbers*. http://gwec.net/global-figures/wind-in-numbers/. Accessed: 2018-08-05.

[HD11] L. Hannah and D. Dunson. Approximate dynamic programming for storage problems. *Proceedings of the 28th International Conference on Machine Learning*, pages 337–344, 2011.

[HE16] J. Han and W. E. Deep learning approximation for stochastic control problems. *Deep Reinforcement Learning Workshop, NIPS*, 2016.

[How60] R. Howard. *Dynamic Programming and Markov Processes*. Published jointly by the Technology Press of the Massachusetts Institute of Technology and Wiley, New York, 1960.

[HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning - Data Mining, Inference, and Prediction*. Springer-Verlag New York, 2nd edition, 2009.

[IEAa] International Energy Agency. *Share of sources in renewable electricity generation*. https://www.iea.org/statistics/?country=WORLD&year=2015&category=Key%20indicators&indicator=ShareRenewGen&mode=chart&categoryBrowse=false. Accessed: 2018-08-05.

[IEAb] International Energy Agency. *Wind electricity generation*. https://www.iea.org/statistics/?country=WORLD&year=2015&category=Key%20indicators&indicator=WindGen&mode=chart&categoryBrowse=false. Accessed: 2018-08-05.

[JP15] D. R. Jiang and W. B. Powell. An approximate dynamic programming algorithm for monotone value functions. *Operations Research*, 63(6):1489–1511, 2015.

[JPP+14] D. R. Jiang, T. V. Pham, W. B. Powell, D. F. Salas, and W. R. Scott. A comparison of approximate dynamic programming techniques on benchmark energy storage problems: Does anything work? In *2014 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 1–8. 2014.

[KHH03] M. Korpaas, A. T. Holen, and R. Hildrum. Operation and sizing of energy storage for wind power plants in a market system. *International Journal of Electrical Power and Energy Systems*, 25(8):599 – 606, 2003. 14th Power Systems Computation Conference.

[KP11] J. H. Kim and W. B. Powell. Optimal energy commitments with storage and intermittent supply. *Operations Research*, 59(6):1347–1360, 2011.

[LGM16] A. Lazaric, M. Ghavamzadeh, and R. Munos. Analysis of classification-based policy iteration algorithms. *Journal of Machine Learning Research*, 17(19):1–30, 2016.

[Mac98] D. J. MacKay. Introduction to gaussian processes. *NATO ASI Series F Computer and Systems Sciences*, 168:133–166, 1998.

[MXZ06] C. A. Micchelli, Y. Xu, and H. Zhang. Universal kernels. *Journal of Machine Learning Research*, 7:2651–2667, December 2006.

[Nea96] R. M. Neal. *Bayesian Learning for Neural Networks*. Springer-Verlag, 1996.

[Ng17] A. Ng. Lecture notes in machine learning (CS229), 2017.

[Øk00] B. Øksendahl. *Stochastic Differential Equations: An Introduction with Applications*. Springer-Verlag, 2000.

[PB79] M. L. Puterman and S. L. Brumelle. On the convergence of policy iteration in stationary dynamic programming. *Mathematics of Operations Research*, 4(1):60–69, 1979.

[PIB] Presse- und Informationsamt der Bundesregierung. *Wie funktioniert der Strommarkt*. https://www.bundesregierung.de/Content/DE/Artikel/2014/08/2014-08-04-so-funktioniert-der%20strommarkt.html. Accessed: 2018-07-24.

[Pow07] W. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley Series in Probability and Statistics. Wiley, 2007.

[Pow11] W. Powell. Approximate Dynamic Programming I: Modeling. In *Encyclopedia of Operations Research and Management Science*. John Wiley and Sons, 2011.

[Put94] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Rus96] J. Rust. Chapter 14: Numerical dynamic programming in economics. volume 1 of *Handbook of Computational Economics*, pages 619 – 729. Elsevier, 1996.

[RW05] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[SB17] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd (complete draft, forthcoming) edition, 2017.

[Sha53] L. Shapley. Stochastic games. *Proceedings of the national academy of sciences*, 39:1095–1100, 1953.

[Sma] Bundesnetzagentur. *Großhandelspreise*. https://www.smard.de/blueprint/servlet/page/home/wiki-article/446/562. Accessed: 2018-07-24.

[SP17] D. F. Salas and W. B. Powell. Benchmarking a scalable approximate dynamic programming algorithm for stochastic control of grid-level energy storage. *INFORMS Journal on Computing*, 30(1):106–123, 2017.

[SR04] M. Santos and J. Rust. Convergence properties of policy iteration. *SIAM Journal on Control and Optimization*, 42(6):2094–2115, 2004.

[SS16] A. A. Shardin and M. Szölgyenyi. Optimal Control of an Energy Storage Facility Under a Changing Economic Environment and Partial Information. *International Journal of Theoretical and Applied Finance*, 19(4), 2016.

[Van01] R. Vanderbei. *Linear Programming: Foundations and Extensions*. International Series in Operations Research & Management Science. Springer US, 2001.

[WB16] X. Wang and J. Berger. Estimating shape constrained functions using gaussian processes. *SIAM/ASA Journal on Uncertainty Quantification*, 4(1):1–25, 2016.

[WID] Windindustrie in Deutschland. *The power market*. `https://www.windindustrie-in-deutschland.de/fachartikel/the-power-market-how-it-works-its-players-and-pricing/`. Accessed: 2018-07-24.