Machine Learning for Probability

Version vom: 7. Mai 2025

Prof. Dr. Thorsten Schmidt[1]

[1]Universität Freiburg. www.stochastik.uni-freiburg.de/schmidt

INHALTSVERZEICHNIS

I Introduction

Machine Learning is a powerful tool which can be applied in many areas, in particular in probability. In this lecture we will examine recent advances in the field and explore the exciting possibilities which arises from this new field.

In this lecture, we will cover several topics and start with some motivating examples in the introduction. Our first chapter will be on reinforcement learning and Markov decision processes.

1 Markov decision processes

A *Markov* process X is a process whose future evolution, viewed from time t on, does only depend on the current value X_t of the process and not its past. This provides a highly versatile framework for many applications in practice[1]

1.1 Driving a vehicle

Consider an autonomous vehicle deciding how to cross an intersection with traffic signals. The vehicles state is described by three parameters, say the velocity, the current state of the traffic signal and the location of the vehicle relative to the intersection.

The vehicle can be *controlled* by increasing or decreasing the speed (in rapid / soft manners for example). The control influence the transition of the system - which, in the setting here, could also be modelled in a deterministic way. It turns out that in many cases (for example if we want to incorporate external influences), the transition should be modelled in a random way. For this we would use the transition probability of the system, which then naturally depends on the control / action taken.

The overall performance is measured by a reward functions which incorporates how fast we crossed the intersection - which we of course want to optimise. An optimal balance between safety, efficiency and compliance with the traffic rules is certainly a challenge.

1.2 Ecosystem management under climate risk

Consider an ecological management scenario where the aim is to manage a complex ecosystem with the challenges the climate change introduces. The state of the ecosystem could be sizes of certain species, like several types of trees, together with their health state.

The control in this case could be to extract (possibly unhealthy) trees, introduce new trees, possibly also new species. The evolution of the system certainly depends on many external fac-

[1]Increasing the dimension, we can also incorporate additional values, as for example the past of X, into the state space - it turns out that this is also a highly general approach.

tors, weather, rain, temperature, etc. and needs to be modelled by a random system. Uncertainty could be covered through probabilities or[2].

The reward function could be the state of the ecosystem in the future, say 50 or 100 years measured for example by population size and their health. In Forest Economics, one could construct also monetary measures for the future value of the forest.

AlphaGo Zero

As a final example we consider a game - and explore AlphaGo Zero's reinforcement learning process for playing Go or Chess.

The playing table consists of 19 × 19 intersecting lines, and on each intersection a black or white stone can be placed. Therefore, the state of the game can be encoded via the board configuration

$$s \in \{0, 1, -1\}^{19\times 19}.$$

The action which can be taken depends of course on the current state and we could denote by $\mathcal{A}(s)$ the set of all possible actions for state s. Our action can be responded by an action taken by the opponent which we could represent by a transition probability

$$P(s'|s, a) = P(S = s'|S = s, A = a)$$

decoding the probability of going from state s to state s' when the action a is taken. Here it does not play a big role if A is actually a random variable or not (we could also look at $P_a(s'|s)$ for all a. Having the transition kernel will help solving some measurability issues later.

Finally, we need to decode a reward which is depending on the final state s of the game and would be 1 for winning and 0 otherwise.

AlphaGo Zero employs self-play combined with neural network policy/value approximation. The key feature is to encode the action via a random strategy (policy) for the player and the opponent and update the value on numerous random games.

[2]Which is called *Knightian uncertainty* through families of probability measures.

II Markov Decision Problems

Following Bäuerle and Rieder, consider a measurable state space $(E, \mathcal{E})$. A transition kernel between measurable spaces $(E, \mathcal{E})$ and $(F, \mathcal{F})$ is a mapping:

$$Q : E \times \mathcal{F} \to [0, 1],$$

such that:

- $Q(\cdot|H)$ is a probability measure on $(E, \mathcal{E})$,
- $Q(A|\cdot)$ is measurable $\forall A \in \mathcal{E}$.

We consider a stochastic process (X_n), where at each time n, an action $a_n \in D_n(X_n)$ can be taken, and the state evolves according to a kernel $Q(\cdot|X_n, a_n)$. The overall reward is given by:

$$E\left[\sum_n r_n(X_n, a_n)\right].$$

1 Decision Rules and Policies

A measurable mapping $f_n : E \to A$ is called a decision rule. The set of all decision rules at time n is denoted by $\mathcal{F}_n$. A policy is a sequence of decision rules (f_n).

2 Optimality and Bellman Equation

A policy π is called optimal if it maximizes the value function $V_n(x)$, defined as:

$$V_n(x) = \sup_\pi E^\pi\left[\sum_{k=n}^{N} r_k(X_k, f_k(X_k))\right].$$

The Bellman equation provides a versatile computational tool:

$$V_n(x) = \sup_{a \in D_n(x)} \left\{ r_n(x, a) + \int V_{n+1}(x')Q(dx'|x, a) \right\}.$$

3 Consumption Problem Example

Consider an investor with initial capital, making consumption and investment decisions to maximize expected utility:

$$E\left[\sum U(c_n)\right],$$

where U is the utility function, typically assumed to be logarithmic.

4 Dynamic Programming

Under suitable assumptions, locally optimal strategies are globally optimal. The dynamic programming algorithm is as follows:

1. Initialize with terminal reward $V_N(x)$.
2. For $n = N-1, \dots, 0$, compute:

$$V_n(x) = \sup_{a \in D_n(x)} \left\{ r_n(x,a) + \int V_{n+1}(x') Q(dx'|x,a) \right\}.$$

5 Verification and Optimality

To verify optimality:

- Solve Bellman's equation,
- Ensure measurability and existence of maximizers,
- Verify that the computed strategy maximizes the value function.

6 Linear-Quadratic Problems

In linear-quadratic problems, the state transitions linearly with quadratic costs. Solutions involve explicit iterations and quadratic forms.

7 Financial Market Example

Consider assets with self-financing strategies, maximizing a strictly concave utility function:

$$\sup E[U(X_T)],$$

subject to self-financing constraints. Power utility functions simplify analysis and computations.

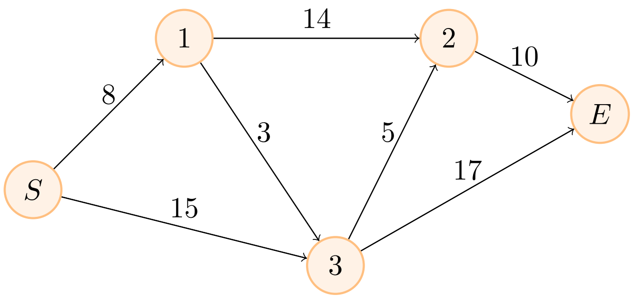


Abbildung II1: Our example of the search for a shortest path. S denotes start and E end, the numbers at the lines denote the length of the way. See Example 8.1 for details.

8 Dynamic Approximate Programming

- From now on, we study the field of dynamic approximate programming (ADP) following Powell (2011).
- As we already learned, there are many dialects in this field and we treat them here. This includes *reinforcment learning*, and a classic reference is Sutton and Barto (1998). For further references consider Powell (2011).
- The terminology reinforcementßtems from behavioural sciences. A positive reinforcer is something that increases the probability of a preceding response (in contrast to a negative reinforcer, like an electronic shock), see also Watkins (1989).
- Examples are: moving a robot, investing in stocks, playing chess or go.
- The system contains four main elements: a *policy*, a *reward function*, a *value function* and (optional) a *model* of the environment.

Let us start with a simple example on finding the shortest path, see Figure II1.

Example 8.1 (Shortest path)**.** It is our goal to find the shortest path from Start to End.

- By $\mathcal{I}$ we denote the set of intersections (S,1,. . . ,E),
- if we are at intersection i we can go to $j \in \mathcal{I}_i$ at cost c_{ij},
- we start at S and end in E. Denote

$$v_i := \text{cost from } i \text{ to } E$$

and we could iterate

$$v_i \leftarrow \min\Big\{v_i, \min_{j\in\mathcal{I}_i}(c_{ij}+v_j)\Big\}, \quad v_i \in \mathcal{I}$$

and stop if the iteration does not change.

Iteration	S	1	2	3	E
1	∞	∞	∞	∞	0
2	∞	∞	10	15	0
3	30	18	10	15	0
4	26	18	10	15	0

- What is an efficient algorithm for solving this problem ?

The above example is a *shortest-path problem* and therefore inherently difficult to solve. We can formulate this in the setting of MDPs (short recap). See (Bäuerle and Rieder, 2011) for details and further information.

8.1 Markov Decision Problems

Let us shortly describe the setting.

- We have a set $D_n(x)$ of possible actions at time n when the system is in state x.
- A *decision rule* at n is a measurable mapping f_n such that $f_n(x) \in D_n(x)$ for all $x \in E$.
- A *policy* is a collection of actions $\pi = (f_0, \dots, f_{N-1})$. We assume that the set of policies is non-empty.
- The dynamics of the model is specified via the transition kernel
$$Q_n(\,\cdot\,|x, s).$$
- Hence, the dynamics and with it the probability for evaluation depends on π. We denote
$$P^\pi_{n,x}(\cdot) := P^\pi(\,\cdot\,|X_n = x)$$
and by $E^\pi_{n,x}$ the associated expectation.

Our aim is to *maximise* the contribution given by the reward $r_n(x, a)$: our goal is to aim at the best possible result
$$\sup_\pi E^\pi\Big[\sum_{n=0}^N r_n(X_n, f_n(X_n))\Big],$$
where $r_N(x, a) = g_N(x)$ does not depend on a any more.

Remark 8.1. In general the supremum need not be measurable which causes a number of delicate problems, see Bertsekas and Shreve (2004) for a detailed treatment. The reason can be traced back to the fact that a projection of a Borel set need not be Borel (which leads to the fruitful notion of analytic sets, however).

- Define the value function
$$V_n(x) := \sup_\pi E^\pi\Big[\sum_{k=n}^N r_n(X_n, f_n(X_n))\,|\,X_n = x\Big]. \tag{1}$$

- Under the structural assumption, the *Bellman equation* holds, i.e.

  $$V_n = \mathcal{T}_n V_{n+1},$$

  with initial condition $V_N = r_N$ and

  $$\mathcal{T}_n v(x) = \sup_{a \in D_n(x)} r_n(x,a) + \int v(x')\, Q_n(dx'|x,a).$$

We therefore obtain the following algorithm to compute the value function iteratively:

Algorithm 8.1 Iterative computation of the Value function by the Bellman equation

```
Step 0  Initialize by the terminal condition $V_N(X_T)$ and set $n = N - 1$
Step 1  Compute
            $V_n(x) = T_n V_{n+1}(x)$
        for all $x \in E$ (and possibly find the optimiser $f^*$)
Step 2  Decrement $n$ and repeat Step 1 until $t = 0$
```

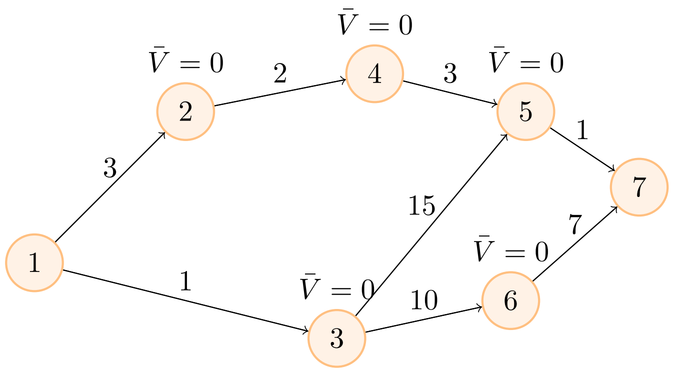
8.2 Approximate dynamic programming (ADP)

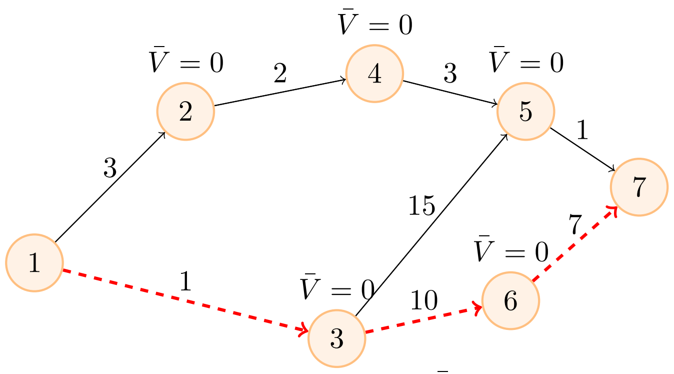
- While we introduced a nice theory beforehand, the core equation

  $$\sup_{\pi} E^{\pi}\Bigg[\sum_{n=1}^{N} r_n(X_n, f_n(X_n))\Bigg]$$

  may be intractable even for very small problems (higher than dimension 3!)
- ADP now offers a powerful set of strategies to solve these problems approximately.
- The idea stems from the 1950's while a lot of the core work was done in the 80's and 90's.
- We have the problem of curse of dimensionality in *state space*, *outcome space* and *action space.*

We illustrate this by an example: we approximate the value function by the function $\bar{V}$ which we update iteratively. The numbers at the arrows denote the associated *costs*. $\bar{V}$ hence is also the cost here. We go forward at the smallest cost.



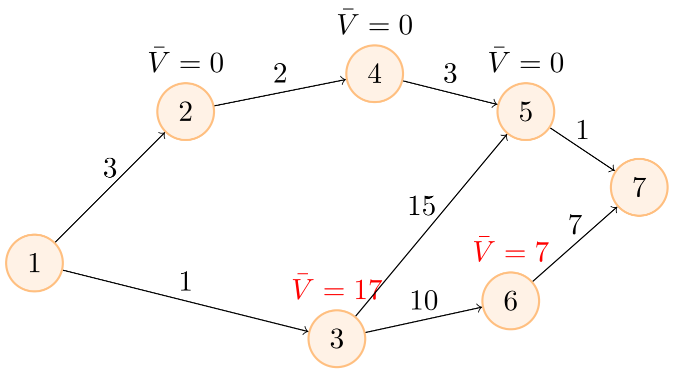


The approximation of the value function is *optimistic*: $\bar{V} = 0$ at all states. We start (forward !) in node $i = 1$ and choose the node j where

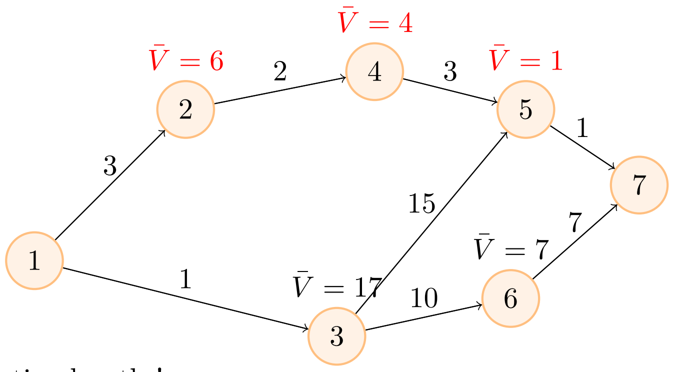
$$c_{ij} + \bar{V}(j)$$

is minimal. This means we choose $1 \to 3 \to 6 \to 7$ and update V accordingly:

$$\bar{V}(6) = 7, \quad \bar{V}(3) = 17.$$



Now we take another round and go (because we initiated optimistically) $1 \to 2 \to 4 \to 5 \to 7$, again updating the weights.



We have found the optimal path !
The above example (still a deterministic one) shows a number of interesting features:

- We proceed forward - which is suboptimal, but repeat until we have found an optimal (or close-to-optimal) solution.
- The value function is approximated.
- The choice of the initial $\bar{V}$ can make us explorative or less explorative - it will become important further on to have this in mind.
- Typical examples are the learning of a robot (for example to stop a ball) or balancing a standing stick on a platform (or robot hand).

8.3 Approximate dynamic programming - the basic idea

There are many variants of ADP - here we look at the basic idea: we proceed forward and approximate $\bar{V}$ iteratively. We start with an initial approximation

$$\bar{V}_t^0(s), \qquad \text{for all } t = 0, \dots, T-1, \ s \in \mathcal{S}.$$

Then we proceed iteratively, see Algorithm 8.2

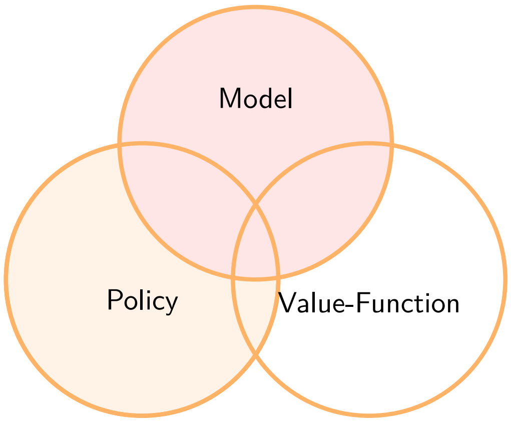
- Note that we still need to be able to compute the expectation (from the transition probabilities). This might be difficult (and for example for a robo running around in the world, infeasible and unwanted)
- We only update $\bar{V}$ for those states we visit. We therefore need to make sure that we are explorative enough to visit sufficiently many states
- We might get caught in a circle and a convergence proof is lacking.

Algorithm 8.2 Basic ADP algorithm

```
Starting from $\bar{V}^{k-1}$ we proceed as follows:
    (i) simulate a path $X(\omega) =: (x_0, x_1, \dots, x_N)$.
    (ii) at $n = 0$ we compute
        $\hat{v}_0^n = T_0 \bar{V}_1^{k-1}$
    (iii) Thereafter, we solve (forward !)
        $\hat{v}_n^k = T_n \bar{V}_{n+1}^{k-1}$
        and continue iteratively until $n = N - 1$.
    (iv) Finally, we update $\bar{V}$ by letting
        $\bar{V}_n^k(x) = \begin{cases} \hat{v}_n^k, & \text{if } x = x_t \\ \bar{V}_n^{k-1}(x) & \text{otherwise.} \end{cases}$
```

9 Solving the Bellman equation

We have three ingredients: model, policy, value-function. Consequently, we have associated groups: model-free / model-based, value-based and policy-based.



Due to the supremum, the Bellman equation is non-linear and a variety of methods for solving it exist:

- Value iteration
- Policy iteration
- Q-Learning
- SARSA

We start with Q-Learning, value and policy iteration typically apply to ∞-time horizon problems, but will be discussed shortly as well.

9.1 Q-Learning

A first model-free approach is the following:

- The Q-learning ADP was proposed in Watkins (1989) (an interesting read).
- The idea is again to approximate the value function. This time we look at the function $Q(x, a)$ (for quality) which gives the value of action a when being in state x, i.e. we are looking for
$$Q : E \times a \to \mathbb{R}$$
- This gives an immediate hand on the optimal policy, $a^*(x) = \arg\max_a Q(x, a)$.
- Again, we proceed iteratively. The assumption we make is that once we choose action a we observe the reward $r(X_n, a)$ and the next state X_{n+1}.
- We call an algorithm *greedy*, if it bases its decision on the value function.
- Assume we are only interested in $V_0(\cdot)$.

Algorithm 9.1 Q-Learning

- Start with an initial Q^0.
- Suppose we are in step k and at position x^k. We choose action a^k greedy, i.e.
$$a^k := \arg\max_{a \in D(x)} Q^{k-1}(x^k, a).$$
- We observe $r(x^k, a^k)$ and x^{k+1}.
- Compute
$$\hat{Q}^k = r(x^k, a^k) + \gamma Q^{k-1}(x^{k+1}, a^k)$$
and update with *stepsize* or *learning rate* α_k:
$$Q^k(x^k, a^k) = (1 - \alpha_k)Q^{k-1}(x^k, a^k) + \alpha_{k-1}\hat{Q}^k$$

Note that no expectation needs to be taken nor any model comes into play.

- A simple implementation just stores the values of Q in a table, which might be less efficient if the spaces get bigger.

- One possibility to solve this issue is to use an artificial network to learn this function (by the universal approximation theorem this is always possible), leading to ”deep reinforcement learning” schemes, as proposed by DeepMind for playing Atari Games.
- Other variants concern speeding up the rates of convergence, as in its current form Q-Learning can be quite slow.
- A variety of implementations are available:
- Car steering
  `http://blog.nycdatascience.com/student-works/capstone/reinforcement-learning-car/`
- a nice blog by Andrej Karpathy about the Atari game pong
  `http://karpathy.github.io/2016/05/31/rl/`
- The R package ReinforcementLearning from N Pröllochs (Freiburg!)[1]

9.2 Value iteration

Very similar to backward dynamic programming we can iterate the value function to find an optimal solution.

- We start with $v = 0$,
- for each $x \in E$ we set

$$V^k(x) = \max_{a \in \mathcal{A}} \Big(r(x,a) + \gamma\, E_{x,a}[V^{k-1}(X)] \Big)$$

- and stop if $\| V^k - V^{k-1} \|$ is sufficiently small.

See Powell, Section 3.10.3 for a proof of the convergence.

9.3 The magic of Reinforcement Learning

Its incredible performance in games[2]:

- RL play checkers *perfect*
- Backgammon, Scrabble, Poker *superhuman.*
- They play Chess and Go on the level of a Grandmaster

They produce interesting behaviour and results.

[1] `https://github.com/nproellochs/ReinforcementLearning`
[2] See David Silvers lectures

- https://www.youtube.com/watch?v=CIF2SBVY-J0
- Implementation in R:
  http://www.rblog.uni-freiburg.de/2017/04/08/reinforcementlearning-a-package-for-replic

Literaturverzeichnis

N. Bäuerle and U. Rieder. *Markov decision processes with applications to finance.* 2011.

D. P. Bertsekas and S. Shreve. *Stochastic optimal control: the discrete-time case.* 2004.

Warren B Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. John Wiley & Sons, 2011.

R. S. Sutton and A. G. Barto. *Reinforcement Learning : An Introduction.* MIT Press, 1998.

C. J. C. H. Watkins. *Learning from delayed rewards.* PhD thesis, King's College, Cambridge, 1989.