Numerik des Maschinellen Lernens

Jan Heiland

TU Ilmenau – Sommersemester 2024

Contents

Vorwort

Das ist ein Aufschrieb der parallel zur Vorlesung erweitert wird.

Korrekturen und Wünsche immer gerne als *issues* oder *pull requests* ans github-repo.

Chapter 1

Introduction

What is *Numerical Methods for Machine Learning*? (ML)

In short, for the training of an ML model, a computer steps through millions of instructions that are formulated in terms of mathematical expressions. Same holds for the evaluation of such a model. Then questions arise like *will there be a point when the training is comes to an end?* and *will the model be accurate?*.

In order to describe what is happening and for the analysis later, we introduce the general concepts of

- algorithm
- consistency/accuracy
- stability
- computational effort

some of which are classical *numerical analysis*.

1.1 What is an Algorithm

Curiously, the term *algorithm* is similarly intuitive and abstract. It took great efforts to come up with a general and concise definition that would meet requirements and limitations of all fields (ranging from, say, *cooking recipes* to the analysis of of *formal languages*).

Definition 1.1 (Algorithm)**.** A problem solution procedure is called an *algorithm* if, and only if, there exists a *Turing machine* that is equivalent to the procedure and that, for every input for which a solution exists, *stops*.

This definition is not too helpful in its generality – we haven't even defined what is a Turing machine.

A *Turing machine* can be described as a machine that reads a strip of instructions and that can write onto this strip. Depending on what it reads it may move forward, move backward, or stop (when the strip has reached a predefined state). The beauty is that this setup can be put into an entirely mathematical framework.

It is more helpful and more common, to look at the implications of this definition to check if a procedure meets at least the necessary conditions for being an algorithm

- The algorithm is described by finitely many instructions (finiteness).
- Every step is *feasible*.
- The algorithm requires a finite amount of memory.
- It will finish after finitely many steps.
- At every step, the next step is uniquely defined (*deterministic*).
- For the same initial state, it will stop at the same final state (*determined*).

Thus, an informal good-practice definition of an algorithm could be

Definition 1.2 (Algorithm – informally)**.** An procedure of finitely many instructions is called an *algorithm* if it computes a determined solution – if it exsists – to a problem in finitely many steps.

Note how some properties (like finitely many instructions) are assumed a-priori.

As an even more informal reference to algorithms we will use the term ***(numerical) method*** or ***scheme*** to address a procedure by listing its underlying ideas and sub procedures, whereas *algorithm* will refer to a specific realization of a *method*.

Furthermore, we will distinguish

- *direct* methods – that compute the solution exactly (like the solution of a linear system by *Gauss elimination*) and
- *iterative* methods – that iteratively compute a sequence of approximations to the solution (like the computation of roots using a *Newton scheme*).

1.2 Consistency, Stability, Accuracy

For the analysis of numerical methods the following terms are generally used:

Definition 1.3 (Consistency)**.** If, in exact arithmetics, an algorithm computes the solution to the problem with a given accuracy, it is called *consistent*.

Definition 1.4 (Stability (informal))**.** If the output of an algorithm depends continously on differences in the input and continously on differences in the instructions, then the algorithm is called *stable*.

The *differences in the instructions* are typically due to rounding errors as they occur in *inexact arithmetics*.

One could say that an algorithm is consistent if *it does the right thing* and that is stable *if it works despite all kinds of small inaccuracies.* If an algorithm is consistent and stable, it is often called *convergent* to express that it will eventually compute the solution even in inexact arithmetics.

Note that terms like

- *accuracy* – how close the computed output matches that actual solution or
- *convergence* – how fast (typically with respect to the computational effort) the algorithm approaches the actual solution

are not intrinsic properties of an algorithm because they depend on the problem that is to be solved. However, one can talk of *order consistency* of an algorithm to specify the expected accuracy for a class or problems and call an algorithm convergent or a certain order if it is stable too.

1.3 Computational Complexity

The *computational complexity* of an algorithm is important both theoretically (to estimate how the effort scales with, say, the size of the problem) and practically (to say how long the procedure will last and which costs in terms of CPU time or memory usage it will generate).

Typically, the complexity is measured by counting the elementary operations, often referred to as *FLOP*s, which is short for *floating point operations*. To classify the algorithms in terms of complexity versus problem size the following function classes are helpful

Definition 1.5 (Landau Symbols or big O notation)**.** Let $g\colon \mathbb{R} \to \mathbb{R}$ and $a \in \mathbb{R} \cup \{-\infty, +\infty\}$. Then we say for a function $f\colon \mathbb{R} \to \mathbb{R}$ that $f \in O(g)$ if

$$\limsup_{x\to a} \frac{|f(x)|}{|g(x)|} < \infty$$

and that $f \in o(g)$ if

$$\limsup_{x\to a} \frac{|f(x)|}{|g(x)|} = 0.$$

The sense and functionality of these concepts might become clear from looking at the typical applications:

- if $h > 0$ is a discretization parameter and, say, $e(h)$ is the discretization error, then we may say that $e(h) = O(h^2)$, if *asymptotically*, i.e. for ever smaller h – the error approaches 0 at least as fast as h^2
- if $C(n)$ is the complexity of an algorithm for a problem size n, than we could say that $C(n) = O(n)$ to express that the complexity grows *asymptotically*, i.e. for ever larger n, at the same speed as the problem size

Unfortunately, the common use of the Landau symbols is a bit sloppy.

1. the often used "="-sign is informal and by no means an equality
2. what is the limit a is hardly ever mentioned explicitly but fortunately generally clear from the context

As an example we look at two different ways to evaluate a polynomial p of the degree n at the abscissa x based on the two equivalent representations

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x) \cdots)) \end{aligned}$$

For a direct implementation of the first representation we obtain

```python
'''computation of p(x) in standard representation
'''
n = 10                                          # example value for n
ais = [(-1)**k*1/k for k in range(1, n+2)]  # list of example coefficients
x = 5                                           # an example value for x
cpx = ais[0]                                    # the k=0 case
for k in range(n):
    cpx = cpx + ais[k+1] * x**(k+1)             # the the k-th contribution
print(f'x={x}: p(x)={cpx:.4f}')                 # print the output
```

In the k-th step, the algorithm requires one addition (if we also count the initialization as an addition) and k multiplications. That makes an overall complexity of

$$C(n) = \sum_{k=0}^{n}(1+k) = n + 1 + \frac{n(n-1)}{2} = 1 + \frac{n}{2} + \frac{n^2}{2} = O(n^2)$$

For the second representation, we can implement the so-called *Horner scheme* that would read

```python
'''computation of p(x) using the Horner scheme
'''
n = 10                                          # example value for n
ais = [(-1)**k*1/k for k in range(1, n+2)]  # list of example coefficients
x = 5                                           # an example value for x
cpx = ais[n]                                    # the k=n case
for k in reversed(range(n)):
    cpx = ais[k] + x*cpx                        # the the k-th contribution
print(f'x={x}: p(x)={cpx:.4f}')                 # print the output
```

Overall, this scheme needs $n+1$ additions and n multiplications, i.e. $2n+1$ FLOPs, so that we can say that *this algorithm is* $O(n)$.

1.4 Exercises

1. Compare the two implementations for evaluating a polynomial by plotting the complexity as a function of n and by measuring and plotting the CPU time needed for an example evaluation versus n.

Further reading:

- wikipedia:Algorithmus

Chapter 2

Errors and Conditioning

Computations on a computer inevitably cause errors and the efficiency or performance of algorithms are always the ratio of costs versus accuracy. For example

- just looking at rounding errors, the accuracy can simply and significantly be improved by resorting to high-precision arithmetics which, however, comes at the cost of higher memory requirements and a higher computational load
- in iterative schemes, memory and computational effort can be saved easily by stopping the iteration at an early stage – at the expense of a less accurate solution approximation

Both these somehow trivial observations are fundamental components of training neural networks. Firstly, it has been observed that low-precision arithmetics can save computational costs with only minor effects on accuracy. Secondly, the training is an iterative process with often slow convergence so that the right time for a premature stop of the training is key.

Definition 2.1 (Absolute and relative errors)**.** Let $x \in \mathbb{R}$ be the quantity of interest and $\tilde{x} \in \mathbb{R}$ be an approximation to it. Then, the *absolute error* is defined as $|\delta x| := |\tilde{x} - x|$ and the *relative error* as $\frac{|\delta x|}{|x|} = \frac{|\tilde{x} - \tilde{x}|}{|x|}$.

Generally, the relative error is preferred as it puts the measured error into the right reference. For example, an absolute error of 10 km/h can be large or small depending on the context. On the other hand, the relative error requires knowledge of the actual value and the division by a value close to 0 can amplify the error estimate.

Next, we will define the *condition* of a problem A and, analogously, of an algorithm (that solves the problem). For that we let x be a parameter/input of the problem and $y = A(x)$ be the corresponding solution/output. The condition is a measure to what extend a change $x + \delta x$ in the input will affect the resulting relative change in the output. For that we consider

$$\delta y = \tilde{y} - y = A(\tilde{x}) - A(x) = A(x + \delta x) - A(x)$$

which after division by $y = A(x)$ and expansion by $x\,\delta x$ becomes

$$\frac{\delta y}{y} = \frac{A(x + \delta x) - A(x)}{\delta x} \frac{x}{A(x)} \frac{\delta x}{x}.$$

For infinitesimal small δx, the difference quotient $\frac{A(x+\delta x)-A(x)}{\delta x}$ becomes the derivative $\frac{\partial A}{\partial x}(x)$ so that we can estimate the condition of the problem/algorithm at x through

$$\frac{|\delta y|}{|y|} \leq |\frac{\partial A}{\partial x}(x)| \frac{|x|}{|A(x)|} \frac{|\delta x|}{|x|} =: \kappa_{A,x} \frac{|\delta x|}{|x|}. \tag{2.1}$$

and call $\kappa_{A,x}$ the condition number.

For vector valued problems/algorithm we can define the condition number through how a difference in the j-th input component x_j will affect the i-th component $y_i = A_i(x)$ of the output.

Definition 2.2 (Condition number)**.** For a problem/algorithm $A\colon \mathbb{R}^n \to \mathbb{R}^m$, we call

$$(\kappa_{A,x})_{ij} := \frac{\partial A_i}{\partial x_j}(x) \frac{x_j}{A_i(x)}$$

the partial *condition number* of the problem. A problem is called *well-conditioned* if $|(\kappa_{A,x})_{ij}| \approx 1$ and *badly-conditioned* if $|(\kappa_{A,x})_{ij} \gg 1$, for all $i = 1, \dots, m$ and $j = 1, \dots, m$.

Rather than using the scalar component functions of $A\colon \mathbb{R}^n \to \mathbb{R}^m$, one can repeat the calculations that led to (2.1) with vector valued-quantities in the corresponding norms.

2.1 Exercises

1. Derive the *condition* number as in (2.1) for a vector valued function $A\colon \mathbb{R}^n \to \mathbb{R}^m$. Where does a matrix norm play a role?
2. Derive condition number of an invertible matrix M, i.e. condition of the problem $x \to y = M^{-1}x$, by the same procedure. Where does the matrix norm play a role?

Referenzen