Calcolo Scientifico - Progetto 3

Fusar Bassini Leonardo, Galimberti Chiara

July 2, 2025

1 Heat equation

Abstract

One of the most important equations in the history of mathematics is the Heat equation, not only for its physical interest, but also because its study gave birth to entire fields of study. Although there is an extensive theory for analytically finding its solution, as often happens, sometimes it is preferable to find a numerical approximation, and it also showcases some common challenges of numerical approximation of PDE in general. Our goal is to find $u = u(x,t)$ solution of the following homogeneous heat problem with Dirichlet's border conditions and an initial condition. We will operate using the discretization of the problem with the CNCS and FTCS methods; we will then compare the two solutions together, underlining the importance of the choice of the time step, and with the exact solution, to see which makes a better approximation.

1.1 The CNCS method

The heat problem is the system:

$$\begin{cases} \dfrac{\partial u}{\partial t} - \dfrac{\partial^2 u}{\partial x^2} = 0 & (x,t) \in (0,1) \times (0,T] \\ u(0,t) = u(1,t) = 0 & t \in (0,T) \\ u(x,0) = \eta(x) & x \in (0,1) \end{cases}$$

where $u(x,0) = \eta(x) = x(1-x)$ is the initial condition, $u(0,t) = u(1,t) = 0$ are Dirichlet's boundary conditions and $f(x,t) = 0$ is the known term.

The CNCS method, that is the Crank Nicholson centered space method, is a Crank Nicholson implicit method in time with a finite difference spatial discretization. It allows us to rewrite the derivatives in the main equation of the problem in an explicit way. We discretize our space domain $(0,1)$ using Nx intervals, so $Nx + 1$ spatial nodes.
Considering a general heat equation of the form

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + f(x,t)$$

with the CNCS method it becomes

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{1}{2h^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n + u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}) + \frac{1}{2}(f_i^n + f_i^{n+1})$$

where we use the following notation: $u_i^n = u(x_i, t_n)$, $f_i^n = f(x_i, t_n)$, Δt is the time step and h is the space step.
In our case f is identically equal to 0, so the rightmost term of the previous equation is inexistent.
Now we define $r = \frac{\Delta t}{2h^2}$, thus our equation becomes

$$-ru_{i+1}^{n+1} + (1+2r)u_i^{n+1} - ru_{i-1}^{n+1} = ru_{i+1}^n + (1-2r)u_i^n + ru_{i-1}^n$$

This means that, generally, we have an implicit problem which requires, for every time step, to solve the linear system:

$$\begin{bmatrix} 1+2r & -r & 0 & \cdots & 0 \\ -r & 1+2r & -r & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & -r & 1+2r & -r \\ 0 & \cdots & 0 & -r & 1+2r \end{bmatrix} \begin{bmatrix} u_0^{n+1} \\ \vdots \\ \vdots \\ \vdots \\ u_m^{n+1} \end{bmatrix} = \begin{bmatrix} 1-2r & r & 0 & \cdots & 0 \\ r & 1-2r & r & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & r & 1-2r & r \\ 0 & \cdots & 0 & r & 1-2r \end{bmatrix} \begin{bmatrix} u_0^{n} \\ \vdots \\ \vdots \\ \vdots \\ u_m^{n} \end{bmatrix}$$

to which, we will now add the initial and boundary conditions.
First of all, the initial condition is fixed in time $t = 0$ but not in space, so, since our solution $u(x, t)$ is in matrix form, this means that the first column $u(:, 1)$ is composed of the function $\eta(x)$ evaluated in the spatial nodes. Secondly, the boundary conditions are fixed in space $x = 0$ and $x = 1$, but not in time, so the first and last line of the matrix $u(x, t)$ are made of zeros.
Thus, the first iteration vector is known and of the form $\underline{u}^0 = [0, \eta(x_1), \dots, \eta(x_{Nx}), 0]^T$.
The boundary conditions tell us that for every iteration vector $\underline{u}^n$ its first and last term is constant and equal to zero; consequently, we can implement the BCs directly in the matrices of the linear system if we put $u_0^{n+1} = u_0^n \ \forall n$, since the first equivalence computed is $u_0^1 = u_0^0 := 0$. The same goes for the last line.
So, the linear system implemented is:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -r & 1+2r & -r & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & -r & 1+2r & -r \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0^{n+1} \\ \vdots \\ \vdots \\ \vdots \\ u_m^{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ r & 1-2r & r & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & r & 1-2r & r \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0^{n} \\ \vdots \\ \vdots \\ \vdots \\ u_m^{n} \end{bmatrix}$$

defined as $A\underline{u}^{n+1} = B\underline{u}^n$, which is solvable with the Matlab command \.

1.2 Comparison between CNCS and FTCS

The FTCS method, that is the forward time centered space method, is a first order explicit method in time with a finite difference spatial discretization. With the FTCS, the heat equation becomes:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{1}{h^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

Using the parameter r it can be rewritten as $u_i^{n+1} = 2ru_{i+1}^n + (1-4r)u_i^n + 2ru_{i-1}^n$; this means that for every time step we have to solve an explicit problem. Thus, we can clearly see that FTCS has a smaller computational cost than CNCS.
Both methods are consistent of order 2, but the interesting part is about the stability of the methods, necessary for their convergence; in this context, CNCS is better than FTCS.
Indeed, FTCS behaves like the Explicit Euler method, so it has an absolute stability condition with a limited region of absolute stability. Using the method of lines, semi-discrete in space, FTCS is of the form $\frac{du_i(t)}{dt} = \frac{1}{h^2}(u_{i-1}(t) - 2u_i(t) + u_{i+1}(t)) \ \forall i$; gathering together the equations we obtain $\underline{u}'(t) = A\underline{u}(t)$, where A is the tridiagonal matrix of the coefficients $\frac{1}{h^2}$, $\frac{-2}{h^2}$, $\frac{1}{h^2}$. Computing the eigenvalues $\lambda_i = \frac{2}{h^2}(cos(ih\pi) - 1)$ of A, we ask that, $\forall i$, λ_i belongs to the absolute stability region; moreover the farthest negative eigenvalues, which is $\lambda_p = -\frac{4}{h^2}$ must belong to the region, and this is called the CFL condition that FTCS must satisfy to be stable. As we said, like Explicit Euler, the FTCS's stability function is $|1 + \Delta t\lambda_i| \leq 1$; we impose:

$$|1 - \Delta t\frac{4}{h^2}| \leq 1 \Longrightarrow -2 \leq -4\frac{\Delta t}{h^2} \leq 1 \Longrightarrow \Delta t \leq \frac{h^2}{2}$$

where the last inequation is called CFL condition, that is the Courant-Friedrichs-Lewy's condition, and it's needed to guarantee the stability.
More specifically, the CFL condition is a necessary condition for convergence that arises in the numerical analysis of explicit time integration schemes while solving certain partial differential equations numerically. Intuitively, a naive interpretation is that it states that the spatial step size Δx and the time step size Δt must be related by the inequality:

$$\Delta t \leq \frac{\Delta x}{c}$$

where c is the wave speed or the maximum eigenvalue of the system, ensuring that information does not travel further than one spatial step in one time step.

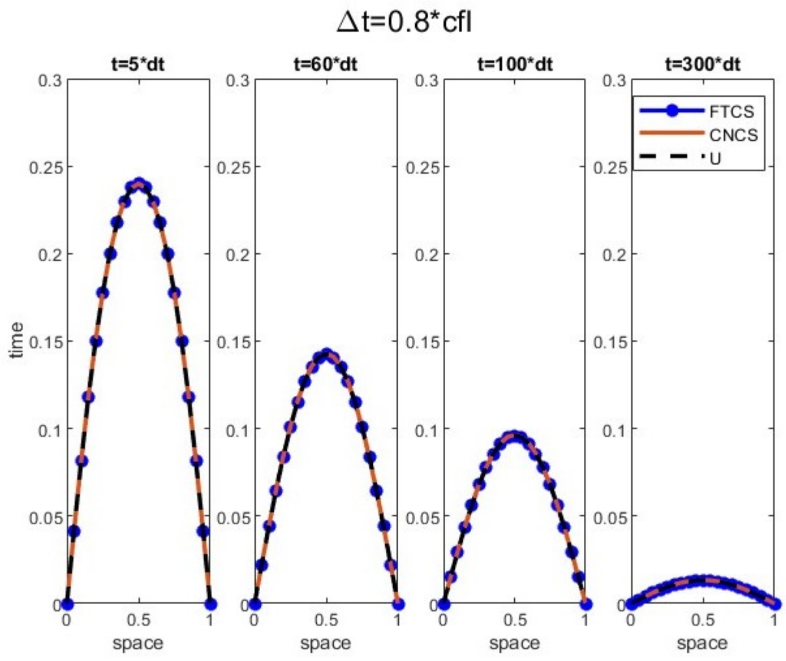
Instead, CNCS behaves like the Crank Nicholson method, so it doesn't have conditions to satisfy for its eigenvalues to fall in the stability region, since this region corresponds to $\mathbb{C}^-$; it's not only stable, but also A-stable. This is an advantage when one wishes to increase the time step Δt, for example if there is the requirement for longer time intervals, because the CNCS will not need the *cfl* requirements imposed upon Δt in FTCS.
This behavior recalls one of the possible definition of stiffness: stiffness occurs when stability requirements, instead of accuracy ones, are dictating the step size Δt. The heat equation is indeed a stiff problem in time, and consequently, implicit methods, like CNCS, usually work better.

The exact solution for the heat equation with initial condition $u(x, 0) = x(1 - x)$ and boundary conditions $u(0, t) = u(1, t) = 0$, calculated with Fourier series, is given by:

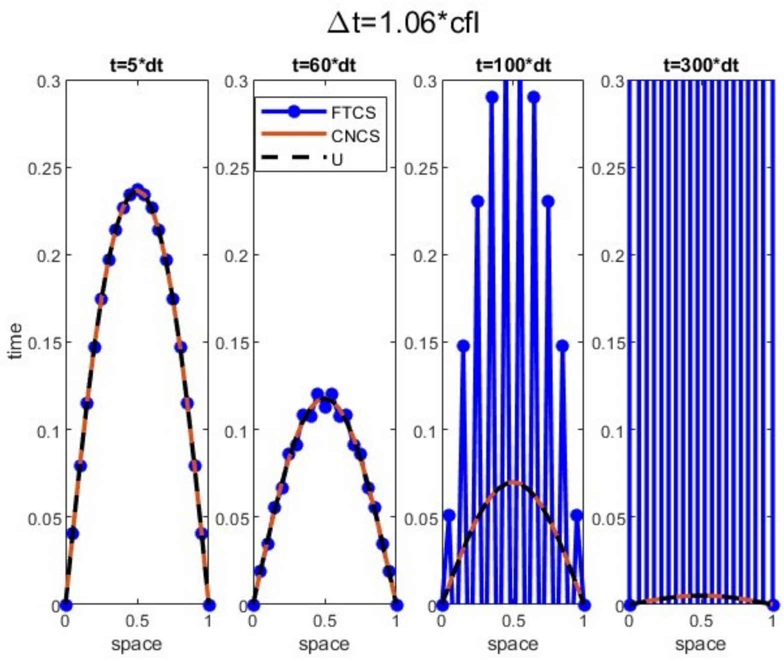
$$U(x, t) = \sum_{n=1}^{\infty} \frac{4 - 4(-1)^n}{(n\pi)^3} \sin(n\pi x) e^{-(n\pi)^2 t}$$

This formula describes the distribution of heat $u(x, t)$ at any point x in the interval $[0, 1]$ and at any time $t \geq 0$.

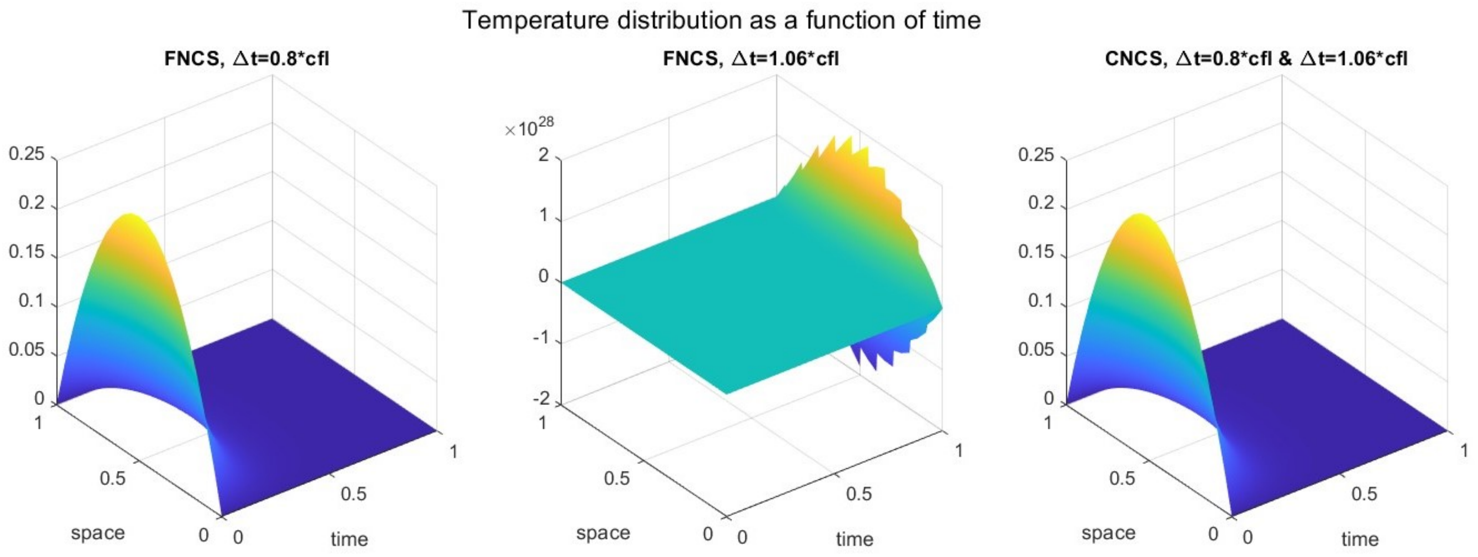
We will now take a fixed number of space intervals $Nx = 20$ to confront CNCS and FTCS with the exact solution U while changing the temporal step in the time domain $(0, T)$, with $T = 1$. Since Nx is fixed, so is h and the condition $\frac{h^2}{2} := cfl$.
Starting with a temporal step $\Delta t \leq cfl$, we take $\Delta t = 0.8 \cdot cfl$, we can see, that both methods approximate well the exact solution:



As soon as the time step exceeds cfl, FTCS starts to oscillate and diverges from U, while CNCS keeps on giving a well approximate solution. This is an example using $\Delta t = 1.06 \cdot cfl$.



This behavior can be seen also in the following tridimensional plots, where the heat is displayed as a function of space and time. When cfl is exceeded by Δt the FTCS's approximation curve changes completely shape.



In this last image we can see that even a "small" dissatisfaction of the *cfl* condition can make the approximate solution wildly diverge.

As a last note, one might ask why methods that must satisfy an additional STIFF condition are even studied, in a problem that as afored commented has an intrinsic STIFFNESS to it. The answer to this is similar to the answer to the general question: why do we use explicit methods if they tend to fail when applied to STIFF problems? In addition to be faster to implement, explicit methods are both less computationally expensive, not requiring the solution of implicit functions at each step, and can usually be taken of higher convergence orders. Also, not always the STIFF constraints are actually a problem; recalling the previously given STIFF statement: "stiffness occurs when stability requirements, instead of accuracy ones, are dictating the step size Δt", if one's desired error tolerance would already require a smaller time step than required by *cfl* condition, or alternatively if the time interval is small enough to make it computationally feasible, one might still choose the explicit versions.

2 Fredholm integral equation of the first kind

Abstract

Signal reconstruction plays a pivotal role in many numerical analysis applications. As a gentle but interesting approach to it, we are going to numerically study the Fredholm integral equation of the first kind. Starting from three simple but crucially important functions, we are going to numerically approximate their convolution with a Gaussian-like kernel function and then try to inversely reconstruct the original function from the "disturbed" output, as might for example happen in applications in geology, astronomy, or signal transmission. We are going to utilize and discuss the Singular Value Decomposition technique, and briefly comment the effects of stochastic rumours with hints to their impact on the higher frequencies numerical results.

2.1 Introduction to the problem

We are going to study the Fredholm integral problem of the first kind in a one-dimensional space:

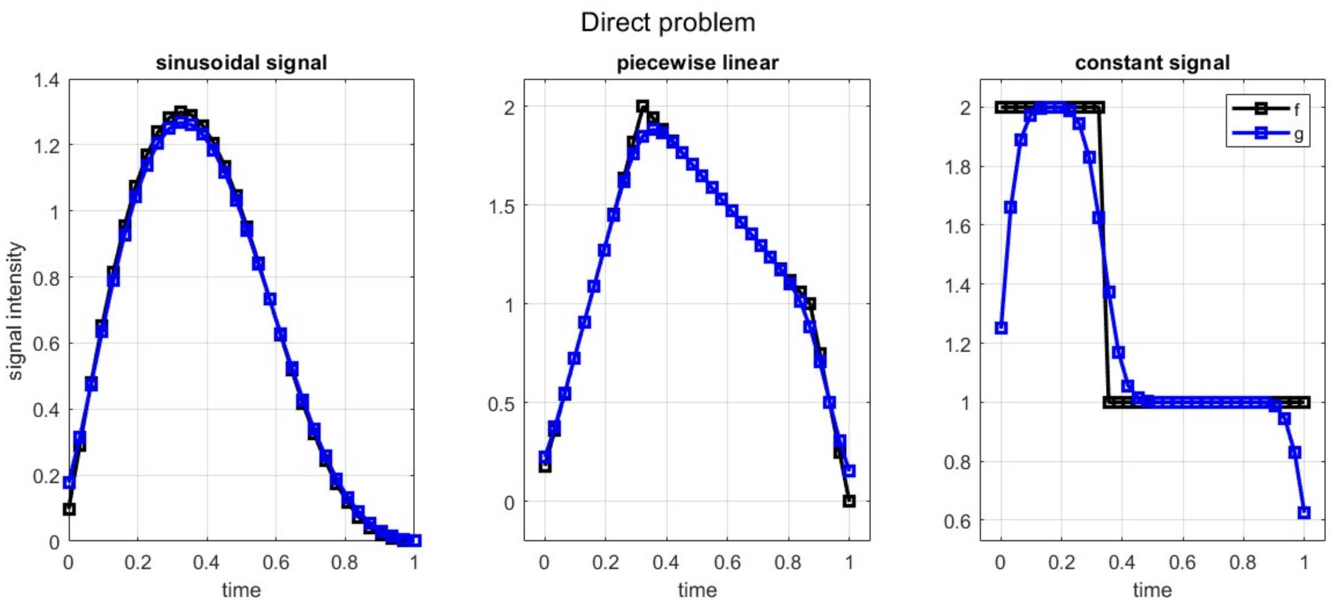
$$g(s) = \int_0^1 K(s-t)f(t)dt, \qquad 0 < s < 1$$

where f is the intensity of a light source in function of space and g the observed intensity, with the fixed parameters: $K(x) = C\exp(-x^2/(2\gamma^2))$, $C = 1/(\gamma\sqrt{2\pi})$ and $\gamma = 0.05$, and where $K(x)$ is also called the kernel.
Hence:

$$g(s) = \int_0^1 \frac{1}{\gamma\sqrt{2\pi}} e^{\frac{-(s-t)^2}{2\gamma^2}} f(t)dt, \qquad \gamma = 0.05$$

2.2 Direct problem

We are firstly going to study the direct discrete problem when applied to simple example functions f, and, since K is known, we can compute g. For f we will use three different types of functions, chosen for their importance in applications: first a sinusoidal one, a linear combination of two sin functions, secondly a piecewise linear function and thirdly a piecewise constant function.
To numerically approximate the results, we discretize both our space domain $(0, 1)$ and fixed time domain $(0, 1)$ with n intervals and use the midpoint quadrature rule to compute the integral problem; in this way, we generate a square matrix of coefficients which, when computed with the vector of the evaluations of f in the discrete space points, generate the time function g.
The following three plots show the different functions f with their respective g, all calculated using $n = 32$ intervals.



The forward problem is well conditioned, which means it has a low condition number, is stable to variations of f, and the error is not excessively amplified by numerical approximation. Although we are not going to thoroughly demonstrate it, we are going to note that it is reasonable to expect that because we are applying a convolution with a smooth function [QSS07].
This gives however birth to a smoothing effect, always present in integral equations, which as we will see might give conditioning problems to the inverse problem solution.
If the kernel K is absolutely integrable, as it is ($K \in L^1 \ \forall s$), then:

$$\lim_{p\to\infty} g_p(s) = \int_0^1 K(s,t) f_p(t) dt \longrightarrow 0,$$

where p are the frequencies; this implies that higher frequencies are smoothed in the convolution with K, hence they give a very small contribution to the observation and the integral "loses" information. This will be the main problem in signal reconstruction. As a last note, a similar effect also happens in the heat equation, where the inverse problem is really badly conditioned. It is enough to think that a finite insulated bar heat solution will always tend to a steady constant temperature, however the initial condition, as example of a very accentuated temporal smoothing effect [Eva10].

2.3 Inverse problem

The most important problem in applications however is to solve the inverse problem and reconstruct the original signal f from the disturbed one received. To do that, we use the SVD method, that is the Singular Value Decomposition method.

This method is a factorization of a, in our case, real matrix into, geometrically speaking, a rotation followed by a rescaling followed by another rotation. Specifically, the singular value decomposition of a real $n \times n$ matrix A is a factorization of the form $A = USV^T$, where U and V are orthonormal matrices and S is a diagonal matrix. The diagonal positive real entries $\mu_i = S_{ii}$ of S are uniquely determined by A and are called singular values, while the columns of U and the columns of V are called left-singular vectors and right-singular vectors respectively; they form two sets of orthonormal bases $u_1, \dots, u_n$ and $v_1, \dots, v_n$.
The SVD is not unique, however it is always possible to choose the decomposition such that the singular values μ_i are in decreasing order; since the number of non-zero singular values is equal to the rank r of A, the singular value decomposition can be written as $A = \sum_{i=1}^r \mu_i u_i v_i^T$.

In our problem the A matrix is the square matrix $K(s,t)$ of the kernel evaluated in a grid of discretized time and space with n intervals; so $K(s,t) = \sum_{i=1}^r \mu_i u_i(s) v_i^T(t)$, where r is the rank of K. Since v_i are orthonormal we have

$$\int_0^1 K(s,t) v_i(t) dt = \sum_{j=1}^r \mu_j u_j(s) \int_0^1 v_i(t) v_j^T(t) dt = \mu_i u_i(s)$$

Using $u_i(s)$ and $v_i(t)$ as base we can write $f(t) = \sum_{i=1}^r \langle v_i, f\rangle v_i(t)$ and $g(s) = \sum_{i=1}^r \langle u_i, g\rangle u_i(s)$. Thus g becomes

$$g(s) = \int_0^1 K(s,t) \sum_{i=1}^r \langle v_i, f\rangle v_i(t) dt = \sum_{i=1}^r \mu_i \langle v_i, f\rangle u_i(s)$$

from which we can also observe that, we have greater oscillation of u_i for increasing values of i, while μ_i decreases toward 0, so higher frequencies in f are smoothed in g, as expected.
Combining the previous formulae we obtain:

$$f(t) = \sum_{i=1}^r \frac{\langle u_i, g\rangle}{\mu_i} v_i(t)$$

In this way knowing K and g we can reconstruct f. That's the reconstruction without noise.
However, more realistically, the observations g contain some stochastic noise, like measurement errors. Indeed, a more realistic g is made like $g(s) = \overline{g}(s) + \epsilon$, where $\overline{g}(s)$ is the "clear part" and ϵ is the noise. In this case f is computed as:

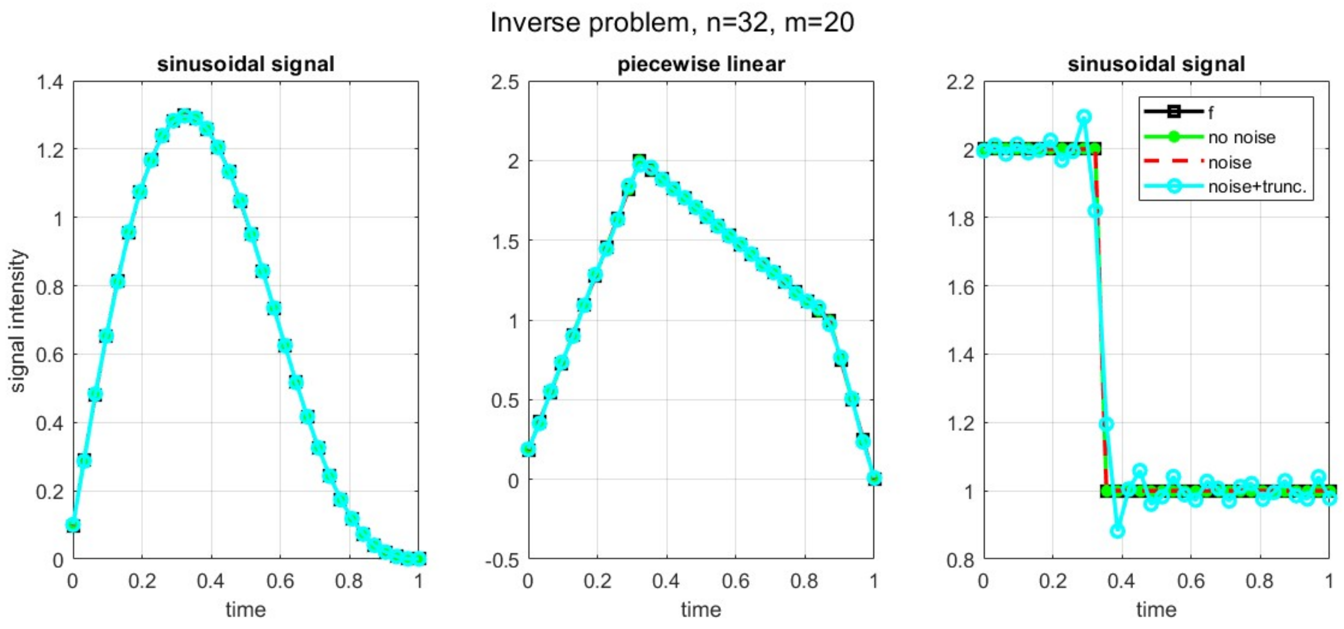
$$f(t) = \sum_{i=1}^r \frac{\langle u_i, \overline{g} + \epsilon\rangle}{\mu_i} v_i(t)$$

Thus, if μ_i decreases to 0 then the noise increases: this phenomenon is known as instability of the inverse problem.
To avoid this problem we use a "naive" but effective method, the truncated SVD. Since, as aforementioned, we can choose a decomposition such that the μ_i are decrescent, this means that the more problematic ones will be the last ones; thus, this truncated SVD method is a normal SVD method but with a finite smaller number of steps $m < r$, so that the smallest μ_i, that will increase the noise, are excluded:

$$f(t) = \sum_{i=1}^{m} \frac{\langle u_i, \overline{g} + \epsilon \rangle}{\mu_i} v_i(t)$$

The problem here becomes where to stop the sum, and that depends on the specific problem and might require a bit of art, or trial and error.

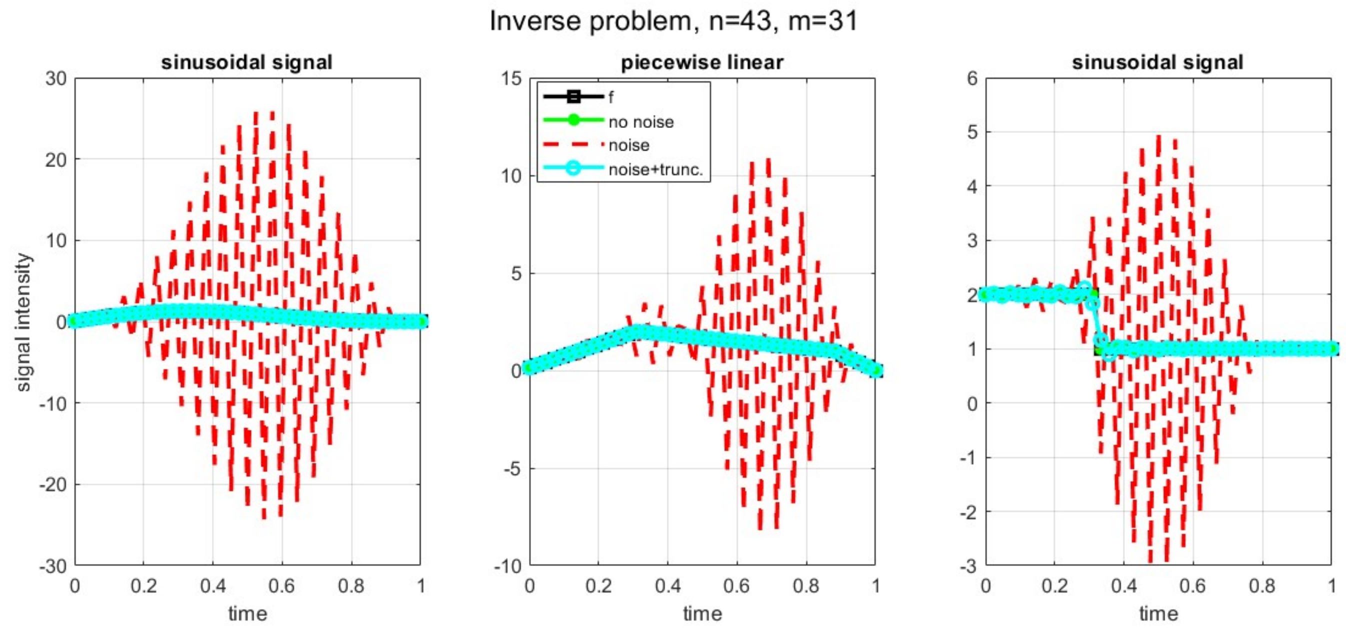
The following three plots display these different reconstructions of f with SVD for every example function f, using $n = 32$ discrete points and $m = 20$ as the truncation value.



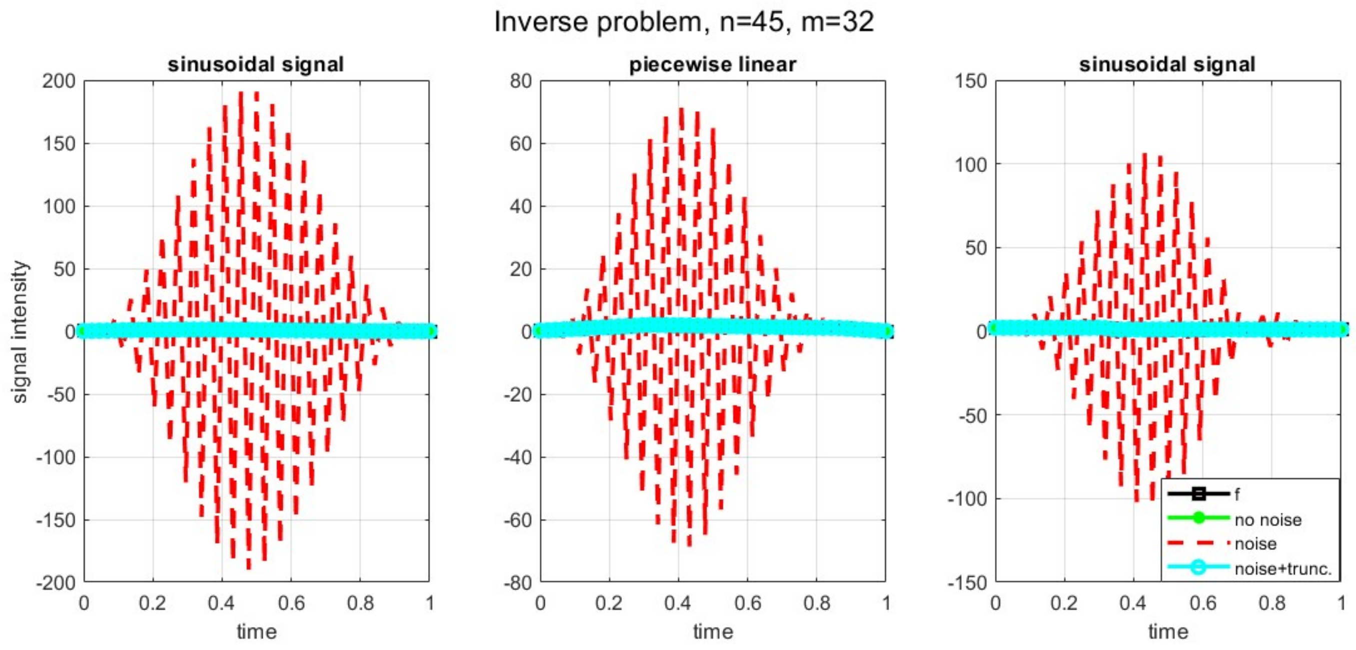
The aforementioned phenomenon of instability, where the decrease of μ_i makes the noise amplify, is well seen as soon as we start increasing the number n of intervals. Indeed, we can first observe the changes in the S matrix of the SVD method: when n increases the number of small μ_i also increases, for example:

n	30	35	40	45	50	55	60	65	70
# of $\mu_i < 1e-5$	0	3	8	13	18	23	27	32	37

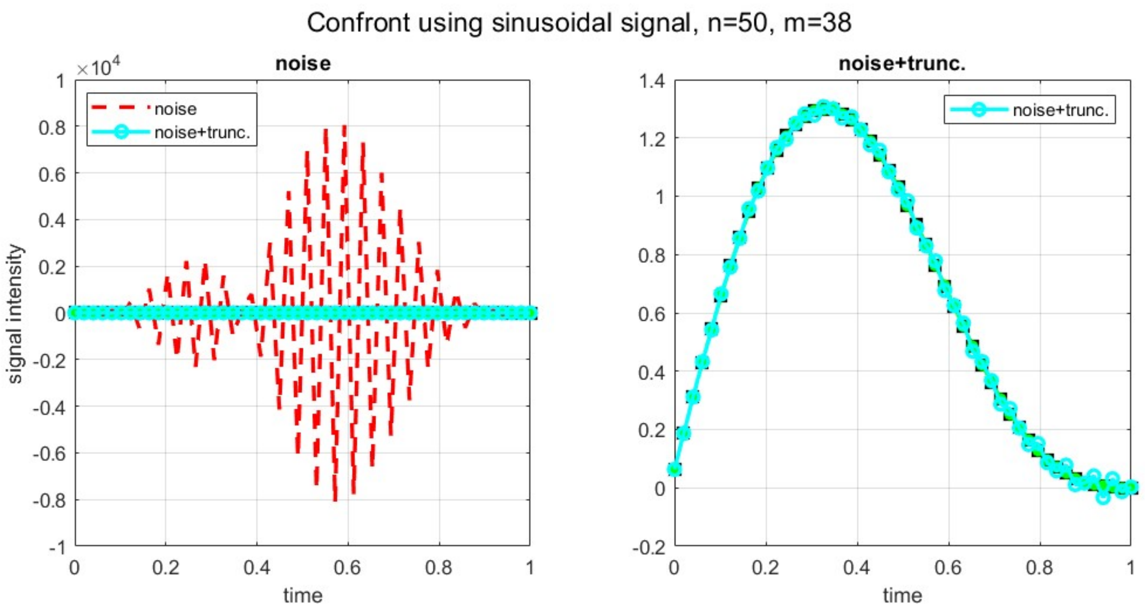
and that means that the noisy reconstruction of f will be less accurate. More precisely it diverges very quickly from f in a short increment of n. These are the plots with $n = 43$:



With only an increment of 2 intervals the noise diverges more than seven times with respect to what we see in the previous plots:



and going on, it will only diverge more.
Also the reconstruction with the truncated SVD will not make a perfect approximation of f, but its divergence is blocked by the truncation, as shown here:



so it's better than the noisy one if it's truncated in the right point, that is right before the last problematic μ_i.
Let us note that this is a hard limit on the precision of reconstructing the original signal with the SVD method, as a function of the intensity of the stochastic noise or of the interferences. One should then keep in mind that "strong" enough noises might compromise the possibility of reconstructing the original signal at all, as reason would suggest.

Another problem is that the inverse problem suffers of ill conditioning, so for a small change in the inputs there might be a large change in the answer.
This is seen studying the absolute value of the difference between f and f^δ and between their derivatives, where $f^\delta = f(x) + \epsilon^\delta(x)$ is a version of f with a noise with an upper bound δ, and $x \in (0,1)$, our domain. Starting with $|f^\delta(x) - f(x)| \leq \delta \ \forall x \in (0,1)$, we study the derivatives to see how fast f^δ diverge from f, using the central finite difference approximation with step h on the derivatives:

$$\left|f'(x) - \frac{f^\delta(x+h) - f^\delta(x-h)}{2h}\right| \leq \left|f'(x) - \frac{f(x+h) - f(x-h)}{2h}\right| + \left|\frac{(f-f^\delta)(x+h) - (f-f^\delta)(x-h)}{2h}\right|$$

where the second term is the sum of the approximation error and the error on data. Thus:

$$\left|f'(x) - \frac{f^\delta(x+h) - f^\delta(x-h)}{2h}\right| \lesssim \ ch^2 + \frac{\delta}{h}$$

where c is the value such as $\int_0^1 (\frac{d^2 f}{dx^2})^2 dx := c^2$.
The value $ch^2 + \frac{\delta}{h}$ increases, thanks to the first term, when h increases, and, thanks to the fraction, when the step h decreases to 0.

3 Neural network

Abstract

Among the rising numerical algorithms, both in industry and academy, are the Large Language Models (LLMs), which have also had a huge impact on society as a whole in the last years. This had also the effect of adding more attention to a sector that was already growing, Machine Learning, and especially Neural Networks. In this section, we are going to study a simple fully connected neural network, that, despite its apparent simplicity, is going to give us many hints in regard to neural networks' training, functioning, and limits.

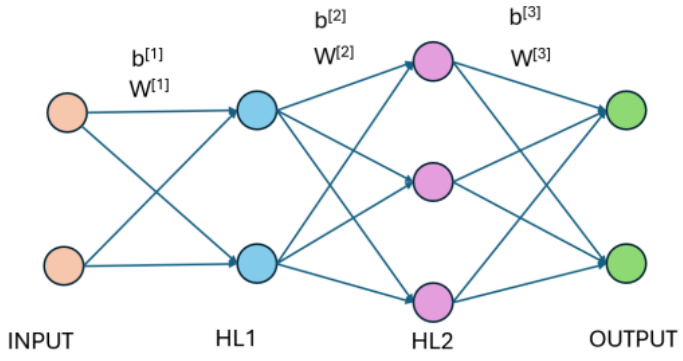
3.1 Neural Network's number of parameters

A simple idea that comes to mind when building Neural Networks is to connect each neuron with every one of the previous layer. This allows the Network to have a large space of parameters to vary in order to better solve the problems they are applied to. This type of networks, called fully

connected neural networks, has however the problem of quickly spiking the number of parameters, which can be a problem for optimization, computation, and memory requirements of the net. Hence the need to know in advance the number of parameters.
Let us now consider a general fully connected Neural Network with N hidden layers and let $E \in \mathbb{N}^{N+2}$ be a positive integers vector where $E(0)$ is the number of input neurons, $E(k)$ for $k \in \{1, ..., N\}$ the number of neurons in the k^{th} hidden layer, and $E(N+1)$ the number of output neurons.
For each neuron in the k^{th} layer, for $k > 0$, there are going to be $E(k-1)$ connections to the previous layer and a bias for each neuron, hence, the total number of parameters P will be given by:

$$P = \sum_{i=1}^{N+1} [E(i)E(i-1)] + \sum_{i=1}^{N+1} E(i) \tag{1}$$

Let us note that, supposing for example a constant number of neurons $E(1)$ in each layer, the number of parameters would asymptotically grow as $N \cdot E(1)^2$.
In our case, following (1):



The number of parameters will be $P = 23$, 6 from the first layer, 9 from the second and 8 from the third.

3.2 Explicit Operations of each Layer

Let us first set a notation: x_1 and x_2 will be the input neurons, $a_i^{(j)}$ the i^{th} neuron of the j^{th} hidden layer, and o_1 and o_2 will be the output neurons; $W_{ij}^{(k)}$ the weight of the connection from the ith neuron of the $(k-1)^{th}$ layer to the jth neuron of the $(k)^{th}$ layer, and $b_i^{(j)}$ the bias of the i^{th} neuron of the j^{th} layer. Furthermore, we suppose the sigmoid function as activation function:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Let's explicitly apply this to the linear operations done by the first layer of the neural network; the other two layers' operations will similarly follow.
For each neuron $j \in \{1, 2\}$ in the first hidden layer (HL1), with 2 input features $x = [x_1, x_2]$, weight matrix $W^{(1)}$ of size 2×2, and bias vector $b^{(1)}$ also of size 2, we compute:

1. **Linear Combination for Neuron j of first layer:**

$$z_j^{(1)} = \sum_{i=1}^{2} W_{ji}^{(1)} x_i + b_j^{(1)}$$

2. **Applying the Sigmoid Function:**

$$a_j^{(1)} = \sigma(z_j^{(1)}) = \frac{1}{1+e^{-z_j^{(1)}}}$$

Expliciting $z_j^{(1)}$:

$$a_j^{(1)} = \frac{1}{1+e^{-\left(\sum_{i=1}^{2} W_{ji}^{(1)} x_i + b_j^{(1)}\right)}}$$

3. **Vector Notation for first Layer:**

$$a^{(1)} = \left(\frac{1}{1+e^{-\left(\sum_{i=1}^{2} W_{1i}^{(1)} x_i + b_1^{(1)}\right)}}, \frac{1}{1+e^{-\left(\sum_{i=1}^{2} W_{2i}^{(1)} x_i + b_2^{(1)}\right)}} \right)$$

4. **Similarly, Vector Notation for second and third Layer:**

$$a^{(2)} = \left(\frac{1}{1+e^{-\left(\sum_{i=1}^{2} W_{1i}^{(2)} a_i^{(1)} + b_1^{(1)}\right)}}, \frac{1}{1+e^{-\left(\sum_{i=1}^{2} W_{2i}^{(2)} a_i^{(1)} + b_2^{(1)}\right)}}, \frac{1}{1+e^{-\left(\sum_{i=1}^{2} W_{3i}^{(1)} a_i^{(1)} + b_3^{(1)}\right)}} \right)$$

$$o = \left(\frac{1}{1+e^{-\left(\sum_{i=1}^{3} W_{1i}^{(3)} a_i^{(2)} + b_1^{(3)}\right)}}, \frac{1}{1+e^{-\left(\sum_{i=1}^{3} W_{2i}^{(3)} a_i^{(2)} + b_2^{(3)}\right)}} \right)$$

As a side note, let us observe that this computations show that the neuron structure is “just” a convenient way to include as many parameters as wanted, in a kind of smart way, in a function from the input space to the output space, where we want to train our neuron network. There are many reasons for why this is a “smart” way to do so, but this is not in the scope of this brief analysis [GBC16].

3.3 Training our sample Network

To train our sample Neural Network we are going to implement the Stochastic Gradient Descent Technique.
The base idea is that, given a Loss Function, we can measure how “off” is the current parameters' distribution from the desired one, and we can try to minimize such function with the Gradient Descent technique. However, this is often unfeasible in practice, because of the size of practical data sets. To circumvent that, at each step, we can use the Gradient Descent technique on a random subset of the sample size. This added randomness also has the added benefit of helping the algorithm escape the local minimum of the loss function.
In more detail, this backpropagation algorithm can be broken down as follows.
The input data is firstly propagated through the network to compute each neuron's activation, then the loss, or distance between output and desired value, is calculated for the current example or batch by comparing the network's output to the true label; this phase is often referred to as Forward Propagation and loss propagation. Then we study how the parameters' local changes influence the error starting from the output layer and proceeding backward, as suggested by the name backpropagation; in this way, the gradients of the loss with respect to each weight are computed as:

$$\delta_L = \frac{\partial L}{\partial z_L} = \frac{\partial L}{\partial a_L} \cdot \sigma'(z_L)$$

For each layer l moving backwards:

$$\delta_l = ((w_{l+1})^T \delta_{l+1}) \odot \sigma'(z_l)$$

where δ_l is the error for layer l, w_{l+1} are the weights connecting layer l to $l+1$, $\odot$ denotes element-wise multiplication. The gradient for each weight w_{ij}^l from neuron i in layer $l-1$ to neuron j in layer l is:

$$\frac{\partial L}{\partial w_{ij}^l} = a_i^{l-1} \delta_j^l$$

where a_i^{l-1} is the activation of neuron i in the previous layer, as discussed in the previous section.

After computing the gradients for the batch, which can be empirically viewed as studying the direction to follow in the parameters' space, we can finally update the weights:

$$w_{ij}^l = w_{ij}^l - \alpha \cdot \frac{\partial L}{\partial w_{ij}^l}$$

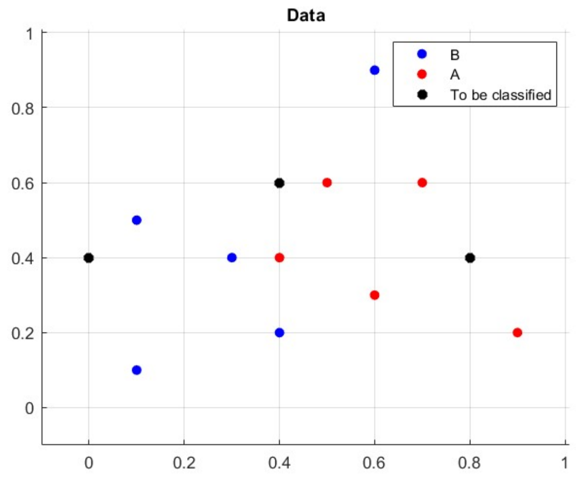
Here, α is the learning rate; it influences the step size as we will discuss later in depth.
To calculate the displayed accuracy we will instead use the following formula:

$$accuracy = 1 - avg(||true\_labels - predicted||_{Input,2})$$

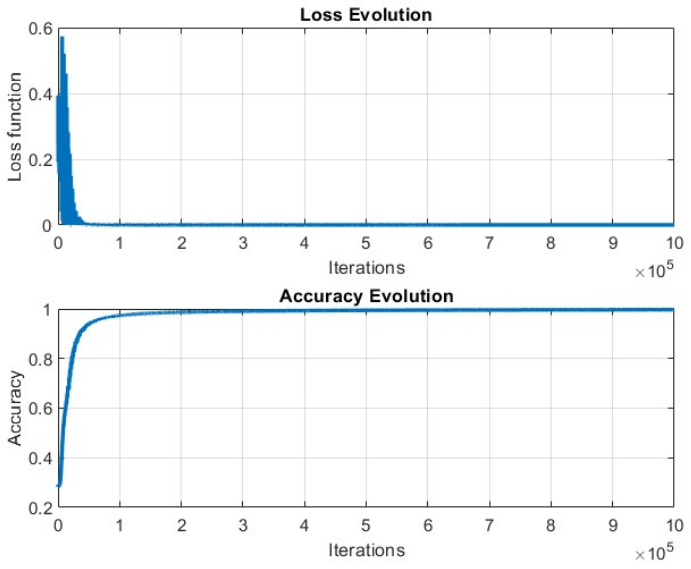
3.4 Test and results

We will now use the previous explained Stochastic Gradient Method to optimize the parameters of our net using batch size = 1, learning rate $\alpha = 0.05$ and number of epoch = 10^6; furthermore we initialize the parameters with a standard normal distribution and we use the mean square error as a loss function.
We will test the so composed net on the points: $x_1 = [0.1, 0.3, 0.1, 0.6, 0.4, 0.6, 0.5, 0.9, 0.4, 0.7]$ and $x_2 = [0.1, 0.4, 0.5, 0.9, 0.2, 0.3, 0.6, 0.2, 0.4, 0.6]$, with the corresponding labels $y_1 = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0]$ and $y_2 = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]$. If its label is 0 the point belongs to the A category, else to the B category.

In the first image we have a sample plot of both the data we have as labeled input and the data we have to give a prediction to. It is somewhat evident the reasonable result, except maybe for the one in the middle.

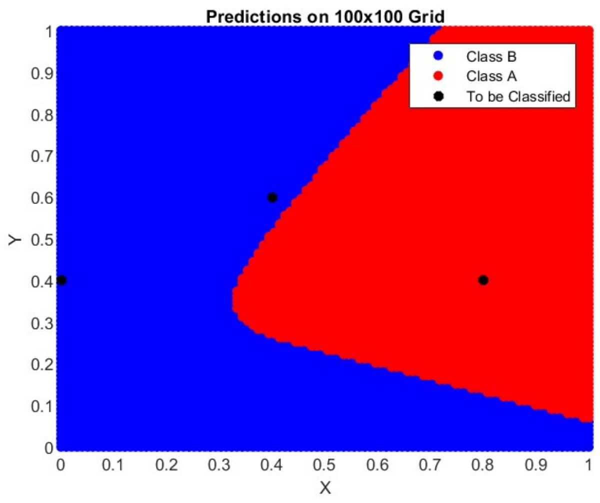


In the second image we can see both the evolution of the loss function, that tends to zero, and the evolution of the accuracy, that tends to one, as desired. We can also see that most of the change in the accuracy happens in the $[0, 10^5]$ training epochs; we will use this to choose the epochs interval for the comparison among different learning rates.

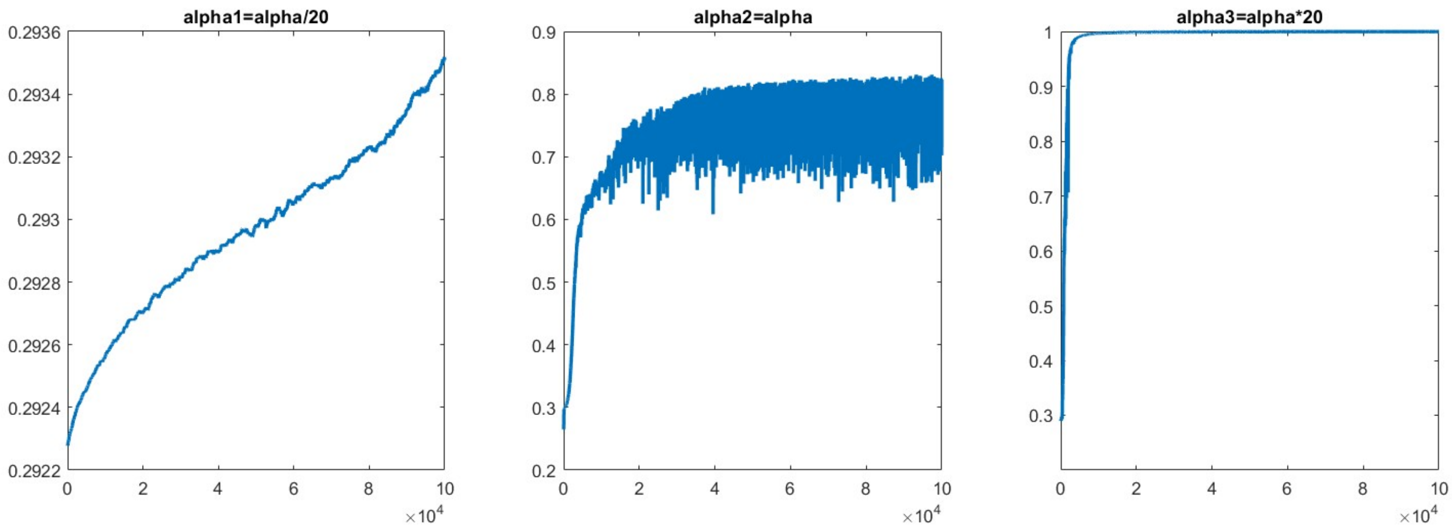


Finally, the following image shows the predicted label for a 100×100 grid in the plane.
It is interesting to note that, even if the assignment suggested an unsure categorization for the point to label in the middle, the found weights and biases make it so that the unknown point is not close enough to the edge between the two classes, and because of that the algorithm classifies it as class B ,$[1.000, 0.000]$ (where $[1.000, 0.000]$ output corresponds to category B and $[0.000, 1.000]$ output

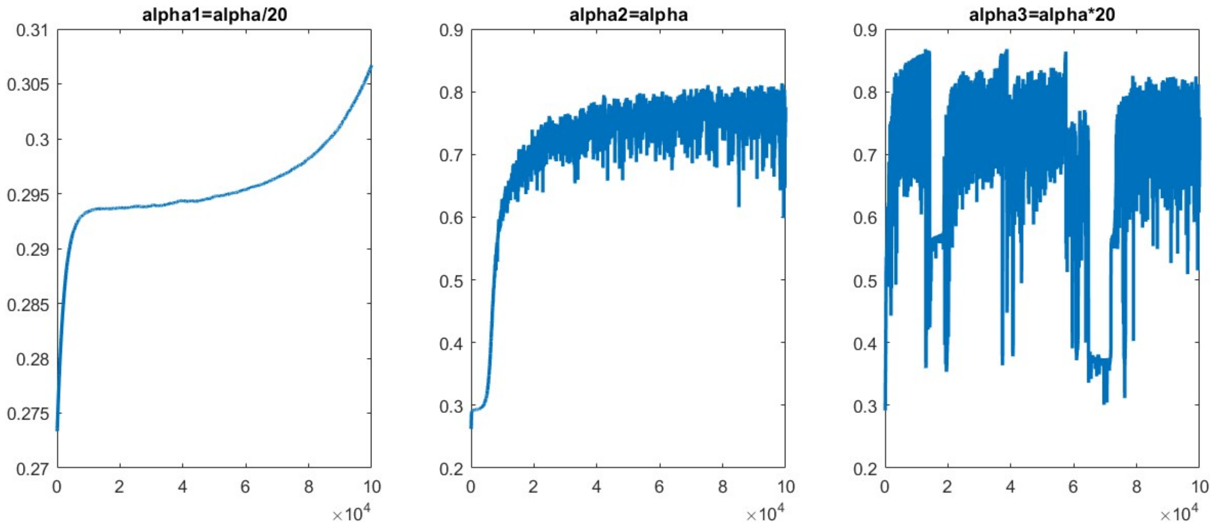
corresponds to category A). However, one must keep in mind that, given a low ratio of known labels and parameters, there might be many different configurations of these, hence different prediction areas.
The trained net correctly classifies the points not on the edge $[0, 0.4]$ and $[0.8, 0.4]$ as $[0.000, 1.000]$ and $[1.000, 0.000]$; for the central point $[0.4, 0.6]$ it gives $[1.000, 0.000]$, whilst for the point $[0.444, 0.6]$ it gives $[0.5187, 0.5287]$.



One of the most important parameters that must be adjusted for each problem is the learning rate, α. However, this requires a bit of art (or trial and error). In fact, the learning rate dictates the size of the steps taken in the direction of deepest descent on the loss function surface.
If the learning rate is too low, the training might required too much time, get stuck in local minima or risk overfitting. Smaller steps will in fact intuitively take more time to reach destination, might not be able to "jump" potential wells, or find patterns that do not exist but are occurring due to the finiteness of input data.
Conversely, if the learning rate is too high, there might be problem of divergence, oscillation, and numerical instability. Intuitively, if small steps would not let you "wander", too big ones would let you "wander around" too freely.



In this image, we see that a bigger learning rate might get the net trained really fast, however, the suggested learning rate is much more stable.
The difference in behavior is given by both the initial randomness (initial weights and biases are here taken as Normal distribution) and the randomness in choosing the batch to calculate the (S)GD on. Indeed:



As aforementioned, the smaller learning rate is by far the most stable one, however, it only gets to about $\frac{1}{2}$ of the accuracy reached by α in the same number of training epochs. It also appears to have been stuck in a local minimum in the $(2\ to\ 6) \cdot 10^4$ training epochs.
Conversely, the bigger learning rate potentially has an incredibly high reward, as shown in the first picture, but such prize is more than offset by the aforementioned risks of oscillation and instability.

We can have a similar but opposite discussion regarding the batch size.
Smaller ones are less likely to get stuck in local minimum but are far more susceptible to oscillations and instability, have slower convergence speed, and are more prone to overfitting. Bigger ones instead are usually smoother, and might be better at generalising, and are better at reducing random noise on the input data impact, however they are more likely to get stuck in local minimum and require more memory and computational strain per epoch.
To try to address these challenges trial and error is the first solution that comes to mind, however many modern algorithms have adaptive learning rate and batch sizes. Usually they have decreasing learning rates, for faster initial convergence and better stability later on, or might try to reduce the batch size if the solution isn't evolving much (to prevent the possibility of being stuck in a local minimum). Other interesting approaches might be to add a "momentum", that make faster changing parameters change faster and slower ones slower, or to introduce random noise in the net evaluation (to try to make it generalize more or make it more stable); however, expanding more on this is beyond the scope of our brief overview.

References

[Eva10] Lawrence C. Evans. *Partial Differential Equations.* American Mathematical Society, 2010. In Chapter 2.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. In Chapter 6, "Deep Neural Networks".

[QSS07] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical Mathematics.* Springer-Verlag, 2007. Chapter 11.