



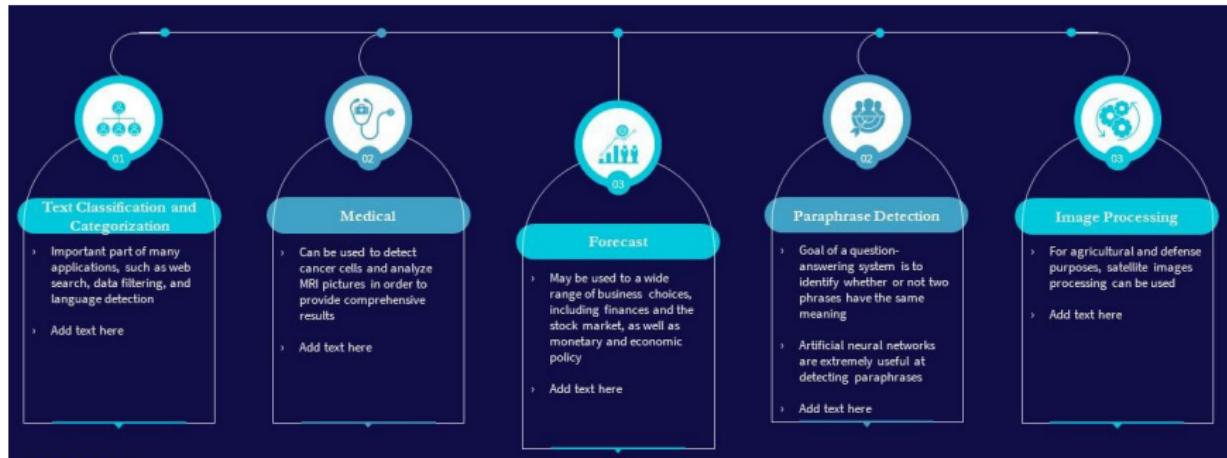
Solving differential equations using physically informed neural networks (PINNs)

Klaudio Peqini

Department of Physics, Faculty of Natural Sciences, University of Tirana

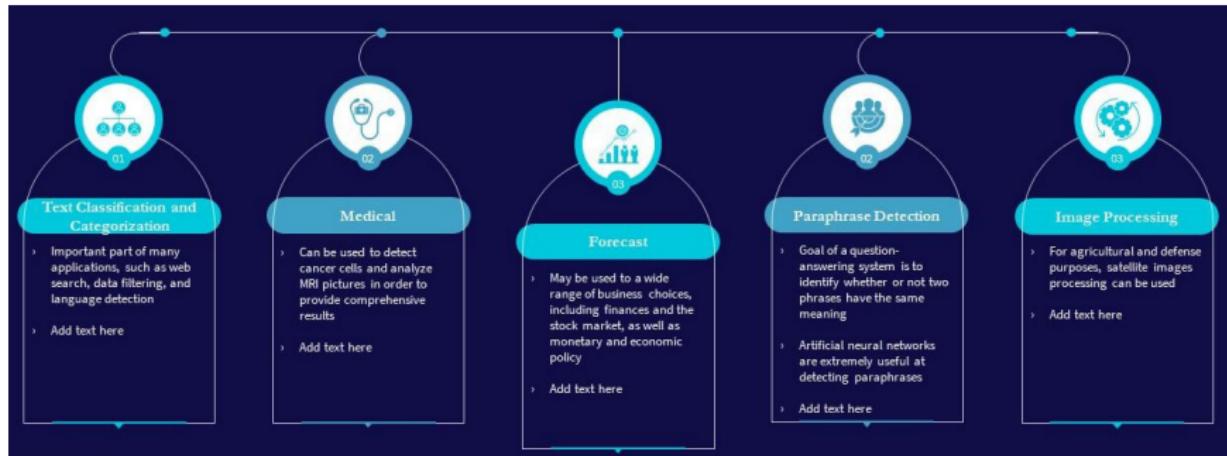
The Data Science Balkan School 2024
January 22–26, 2024

Neural Networks



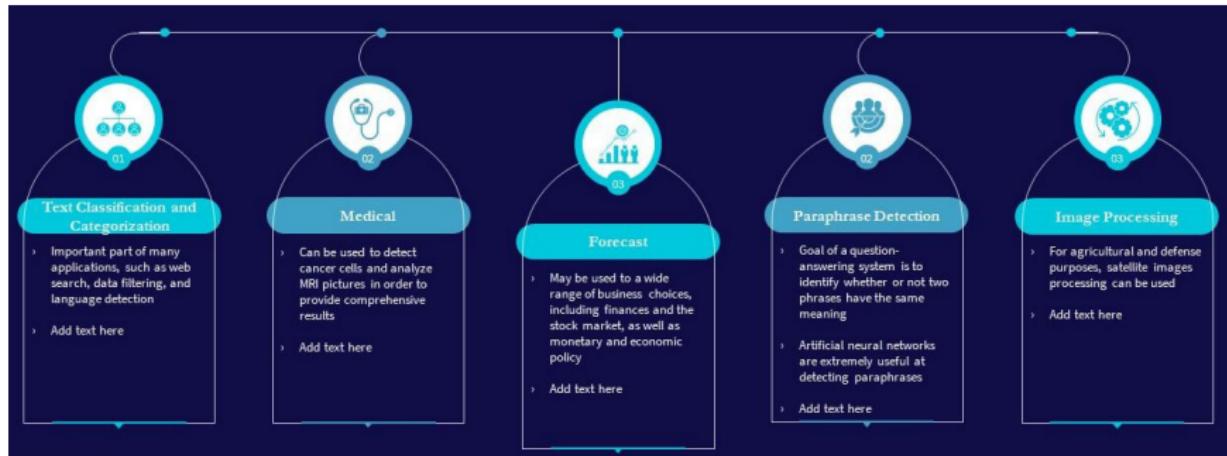
- The most famous successful applications of Deep Learning Algorithms and Neural Networks (NN) in various tasks seem less physics-related

Neural Networks



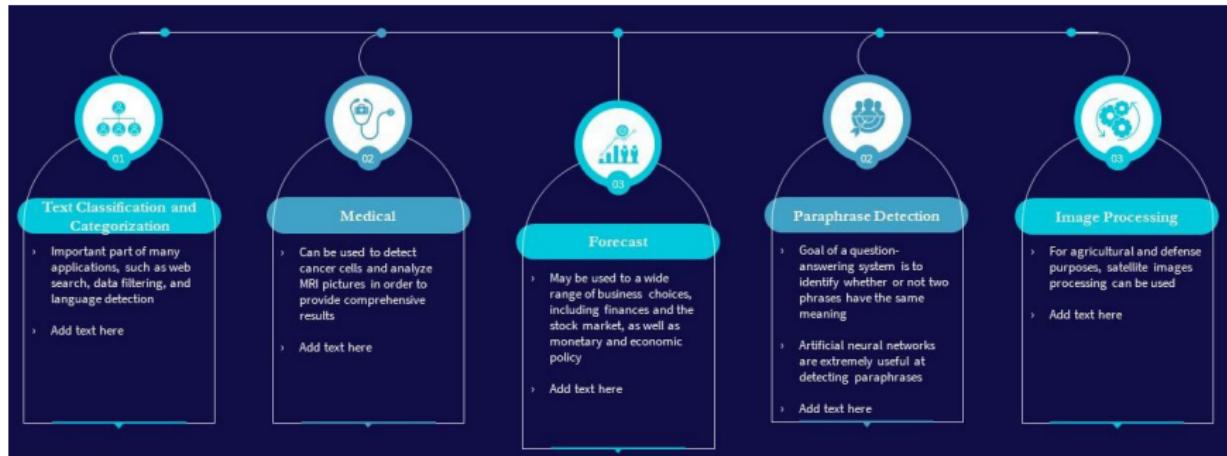
- The most famous successful applications of Deep Learning Algorithms and Neural Networks (NN) in various tasks seem less physics-related
- Personally, I hope this school sheds light on some cool science-related applications

Neural Networks



- The most famous successful applications of Deep Learning Algorithms and Neural Networks (NN) in various tasks seem less physics-related
- Personally, I hope this school sheds light on some cool science-related applications
- Since 1990 there have been several tentatives to employ NNs to solve Ordinary Differential Equations (ODEs)

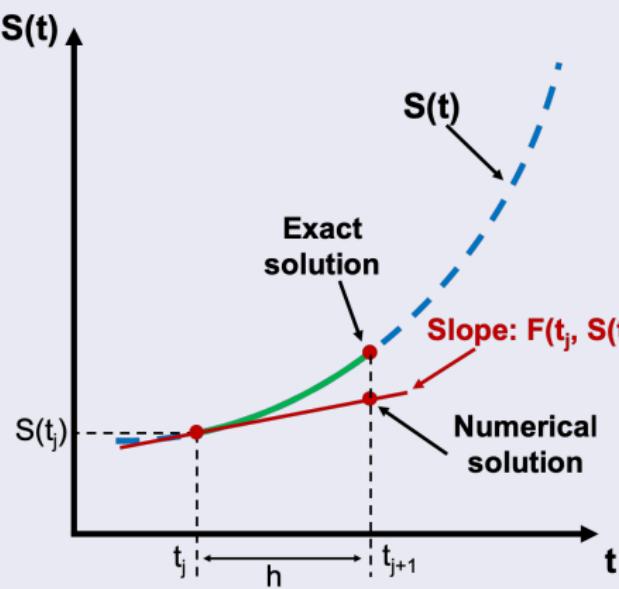
Neural Networks



- The most famous successful applications of Deep Learning Algorithms and Neural Networks (NN) in various tasks seem less physics-related
- Personally, I hope this school sheds light on some cool science-related applications
- Since 1990 there have been several tentatives to employ NNs to solve Ordinary Differential Equations (ODEs)
- Why would we have such approach and where may it lead?

Forward Euler Scheme

Consider a 1st order ODE with respect to time: $\frac{dS}{dt} = f(t, S(t))$



Notice how following the tangent line for a large step increases the error

Discretizing the derivative

The derivative is instantaneous change, that is physically non-measurable. In reality we can measure the average change in a given interval Δt , between two specific time instants t_j and t_{j+1} . The derivative need to be **discretized**:

$$\frac{dS}{dt} \rightarrow \frac{\Delta S}{\Delta t} = \frac{S(t_{j+1}) - S(t_j)}{h},$$

h being the step. The ODE is converted into an equation that takes the previous value of $S(t_j)$ as known and calculate the following value $S(t_{j+1})$ as follows:

$$S(t_{j+1}) = S(t_j) + h f(t_j, S(t_j))$$

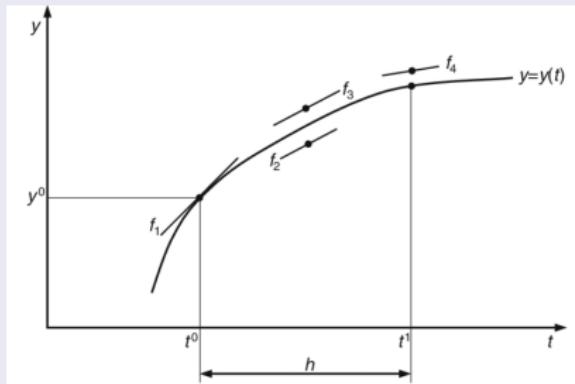
Runge-Kutta Method (4th order)

Can we improve the numerical scheme? **YES!**

Calculate more points inside the step following the scheme to the right.

There are more calculations, but the overall errors are *considerably smaller*.

Geometrical interpretation of RK4 Scheme of RK4



$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$k_1 = f(x_i, y_i)$$

$$k_2 = f(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_1)$$

$$k_3 = f(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_2)$$

$$k_4 = f(x_i + h, y_i + hk_3)$$

where $h = x_{i+1} - x_i$ is the step size.

Local error is of $O(h^4)$. Global error is of $O(h^3)$.

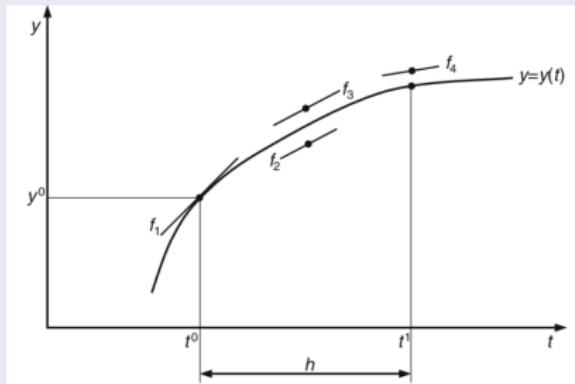
Runge-Kutta Method (4th order)

Can we improve the numerical scheme? **YES!**

Calculate more points inside the step following the scheme to the right.

There are more calculations, but the overall errors are *considerably smaller*.

Geometrical interpretation of RK4 Scheme of RK4



$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$k_1 = f(x_i, y_i)$$

$$k_2 = f(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_1)$$

$$k_3 = f(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_2)$$

$$k_4 = f(x_i + h, y_i + hk_3)$$

where $h = x_{i+1} - x_i$ is the step size.

Local error is of $O(h^4)$. Global error is of $O(h^3)$.

These methods (yes there are more) can be adapted for higher order ODEs. They are deterministic and work on the derivative which stems out naturally.

What about PDEs?

Numerically solving ODEs is manageable, but applying such methods on PDEs can become extremely time- and resource-consuming.

What about PDEs?

Numerically solving ODEs is manageable, but applying such methods on PDEs can become extremely time- and resource-consuming.

- ① This is observed when the step becomes very small

What about PDEs?

Numerically solving ODEs is manageable, but applying such methods on PDEs can become extremely time- and resource-consuming.

- ① This is observed when the step becomes very small (to get smoothness and more realistic results)

What about PDEs?

Numerically solving ODEs is manageable, but applying such methods on PDEs can become extremely time- and resource-consuming.

- ① This is observed when the step becomes very small (to get smoothness and more realistic results)
- ② The domain of integration is very large

What about PDEs?

Numerically solving ODEs is manageable, but applying such methods on PDEs can become extremely time- and resource-consuming.

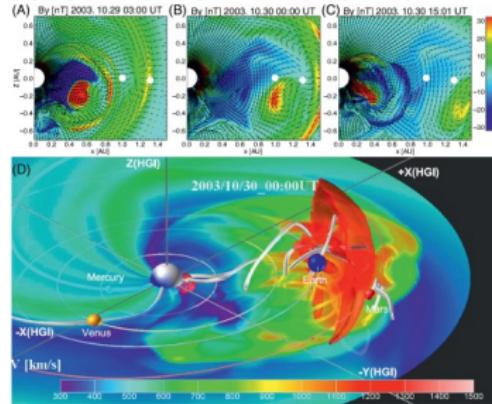
- ① This is observed when the step becomes very small (to get smoothness and more realistic results)
- ② The domain of integration is very large to study multiple scales, either in time or space

What about PDEs?

Numerically solving ODEs is manageable, but applying such methods on PDEs can become extremely time- and resource-consuming.

- ① This is observed when the step becomes very small (to get smoothness and more realistic results)
- ② The domain of integration is very large to study multiple scales, either in time or space

Navier-Stokes come to mind or full MHD models!

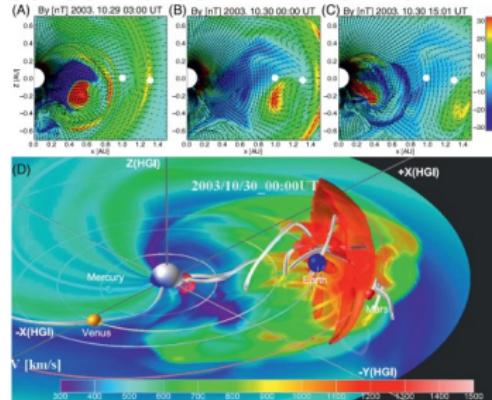


What about PDEs?

Numerically solving ODEs is manageable, but applying such methods on PDEs can become extremely time- and resource-consuming.

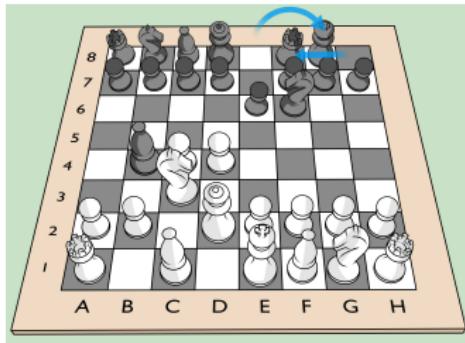
- ① This is observed when the step becomes very small (to get smoothness and more realistic results)
- ② The domain of integration is very large to study multiple scales, either in time or space

Navier-Stokes come to mind or full MHD models!



Can we be smarter!

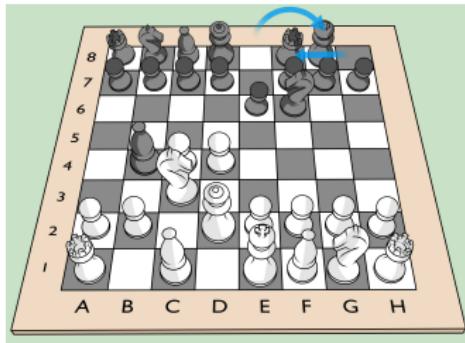
How to do better?



- ① A chessplayer sees each setup for 10 seconds and is asked to set pieces in the same way on an empty board. In which case will he/she perform better?

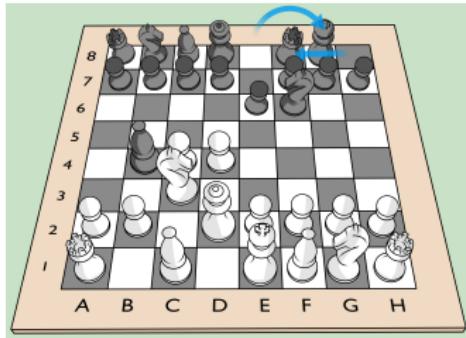


How to do better?



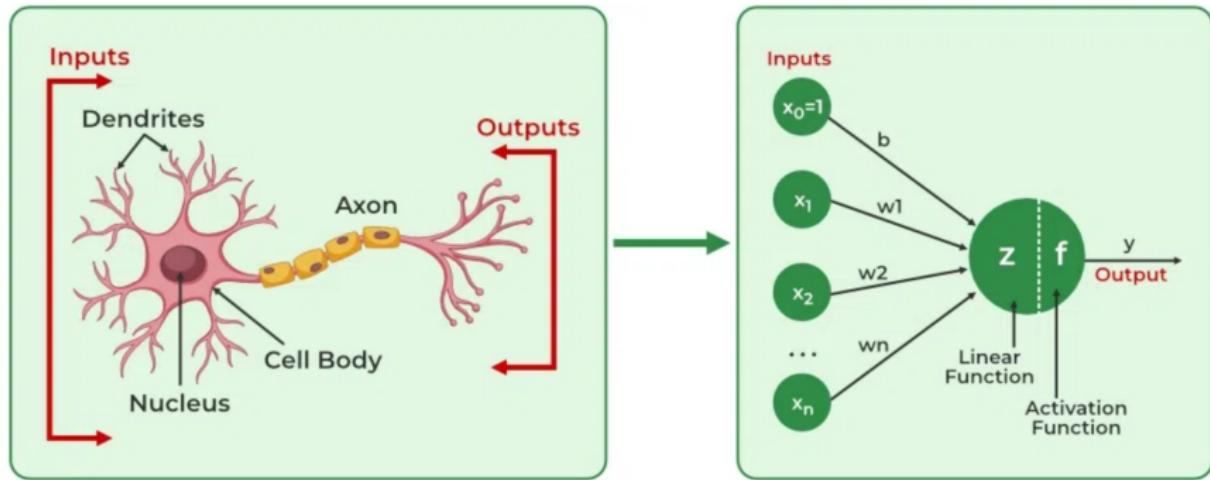
- ➊ A chessplayer sees each setup for 10 seconds and is asked to set pieces in the same way on an empty board. In which case will he/she perform better?
- ➋ Instead of remembering the position of each individual piece, it is much more efficient to identify typical structures or groups and memorise only the position of a central piece (usually the most important in the group).

How to do better?



- ➊ A chessplayer sees each setup for 10 seconds and is asked to set pieces in the same way on an empty board. In which case will he/she perform better?
- ➋ Instead of remembering the position of each individual piece, it is much more efficient to identify typical structures or groups and memorise only the position of a central piece (usually the most important in the group).
- ➌ Instead of solving an ODE for a lot of points in a specific domain or boundary, one can identify some patterns and reduce the computational costs.
- ➍ These patterns are not like patterns sought in images, but are based on the physics of the system described by the differential equation

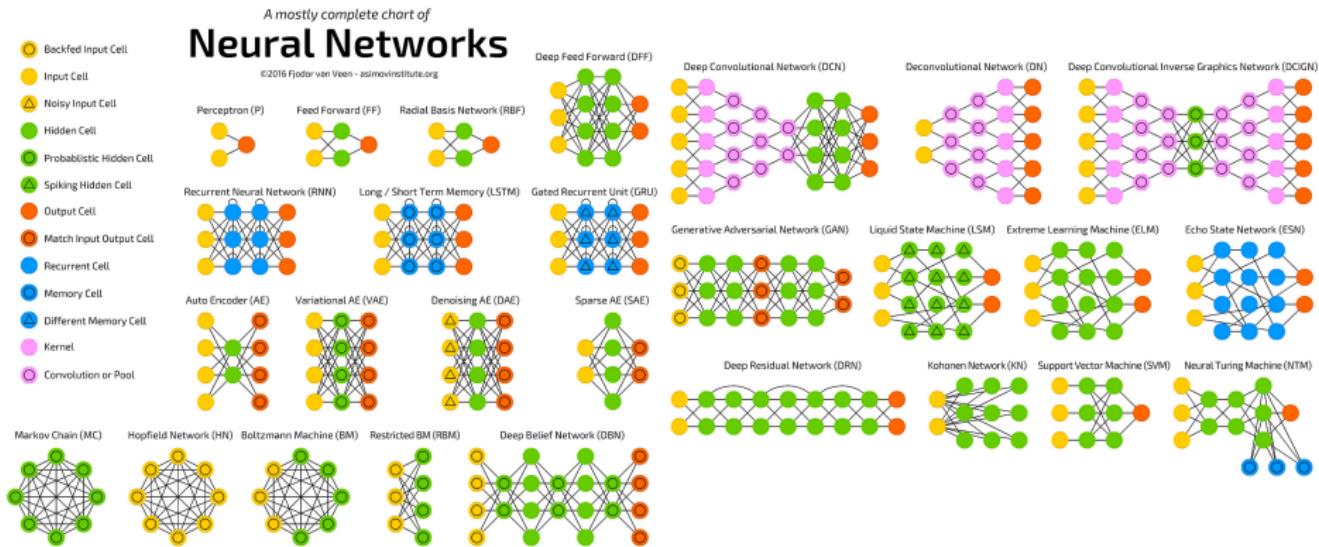
Neural Networks (NN)



Neurones get electrical impulses through the dendrites (**the inputs**). After a specific threshold potential is crossed, the neuron fires and sends an impulse through the axon (**the output**). Such artificial neurons are not supposed to act alone!

NN Architectures

The artificial neurons are connected in a myriad of architectures. In our case, several hidden layers are suitable (RNN).

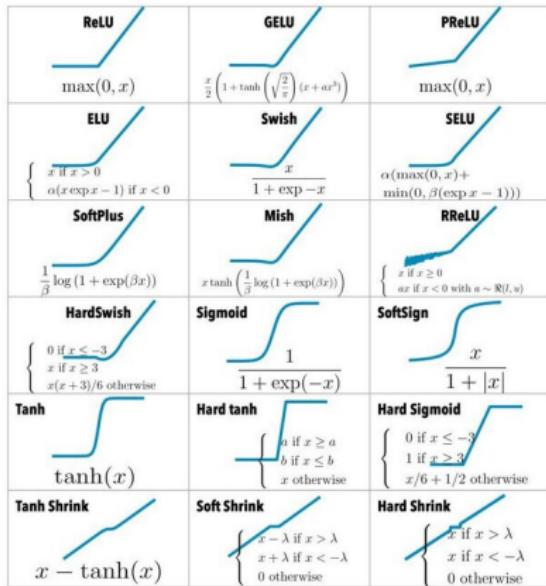


Firing neurons

Firing a neuron means that the neuron transmits signal (a number) after a specific threshold is reached. We say the neuron is activated. There are many so-called activation functions. In the following, a **small subset**.

Firing neurons

Firing a neuron means that the neuron transmits signal (a number) after a specific threshold is reached. We say the neuron is activated. There are many so-called activation functions. In the following, a **small subset**.



Firing neurons

Firing a neuron means that the neuron transmits signal (a number) after a specific threshold is reached. We say the neuron is activated. There are many so-called activation functions. In the following, a **small subset**.

| | | |
|--|---|--|
| ReLU $\max(0, x)$ | GELU $\frac{x}{2} \left(1 + \tanh \left(\sqrt{\frac{2}{\pi}} (x + \alpha x^2) \right) \right)$ | PRelu $\max(0, x)$ |
| ELU $\begin{cases} x & \text{if } x > 0 \\ \alpha(x \exp x - 1) & \text{if } x \leq 0 \end{cases}$ | Swish $\frac{x}{1 + \exp -x}$ | SELU $\alpha(\max(0, x) + \min(0, \beta(\exp x - 1)))$ |
| SoftPlus $\frac{1}{\beta} \log(1 + \exp(\beta x))$ | Mish $x \tanh\left(\frac{1}{\beta} \log(1 + \exp(\beta x))\right)$ | RReLU $\begin{cases} x & \text{if } x \geq 0 \\ ax & \text{if } x < 0 \text{ with } a \sim \mathcal{R}(l, u) \end{cases}$ |
| HardSwish $\begin{cases} 0 & \text{if } x \leq -3 \\ x & \text{if } x \geq 3 \\ x(x+3)/6 & \text{otherwise} \end{cases}$ | Sigmoid $\frac{1}{1 + \exp(-x)}$ | SoftSign $\frac{x}{1 + x }$ |
| Tanh $\tanh(x)$ | Hard tanh $\begin{cases} a & \text{if } x \geq a \\ b & \text{if } x \leq b \\ x & \text{otherwise} \end{cases}$ | Hard Sigmoid $\begin{cases} 0 & \text{if } x \leq -3 \\ 1 & \text{if } x \geq 3 \\ x/6 + 1/2 & \text{otherwise} \end{cases}$ |
| Tanh Shrink $x - \tanh(x)$ | Soft Shrink $\begin{cases} x - \lambda & \text{if } x > \lambda \\ x + \lambda & \text{if } x < -\lambda \\ 0 & \text{otherwise} \end{cases}$ | Hard Shrink $\begin{cases} x & \text{if } x > \lambda \\ x & \text{if } x < -\lambda \\ 0 & \text{otherwise} \end{cases}$ |

You will try different activation functions on your own as a homework!

Understanding NN (some definitions)

Mathematically speaking, an artificial neuron is defined as a function that takes as an input $\vec{x} = \{x_1, x_2, \dots, x_N\}$ and returns the single value (in this case, but it can be an array, etc.) $a(z)$, where $z = \sum_{i=1}^n \omega_i x_i + b$. Here a is the activation function, $W = \{\omega_i\}$ is the set of weights, and b is the bias.

$$\begin{array}{ccc} \mathbb{R}^n & \longrightarrow & \mathbb{R} \\ \{x_n\} & \longrightarrow & z = \sum_{i=1}^n \omega_i x_i - b \end{array} \longrightarrow a(z)$$

Understanding NN (some definitions)

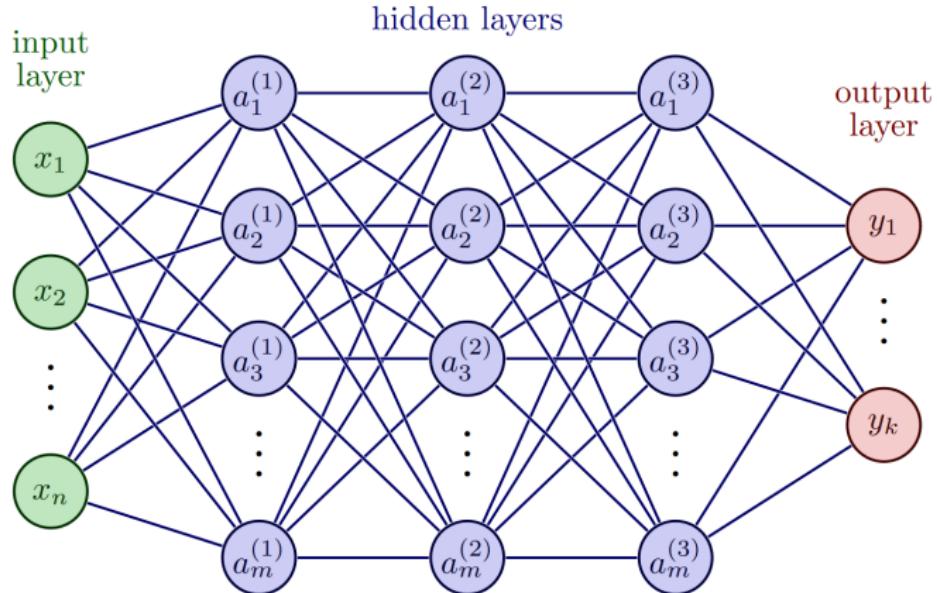
Mathematically speaking, an artificial neuron is defined as a function that takes as an input $\vec{x} = \{x_1, x_2, \dots, x_N\}$ and returns the single value (in this case, but it can be an array, etc.) $a(z)$, where $z = \sum_{i=1}^n \omega_i x_i + b$. Here a is the activation function, $W = \{\omega_i\}$ is the set of weights, and b is the bias.

$$\begin{array}{ccc} \mathbb{R}^n & \longrightarrow & \mathbb{R} \\ \{x_n\} & \longrightarrow & z = \sum_{i=1}^n \omega_i x_i - b \end{array} \longrightarrow a(z)$$

What if the output of a neuron serves as the input of another neuron? What if you organize neurons in layers that are somehow connected to each other? What if you use many layers?

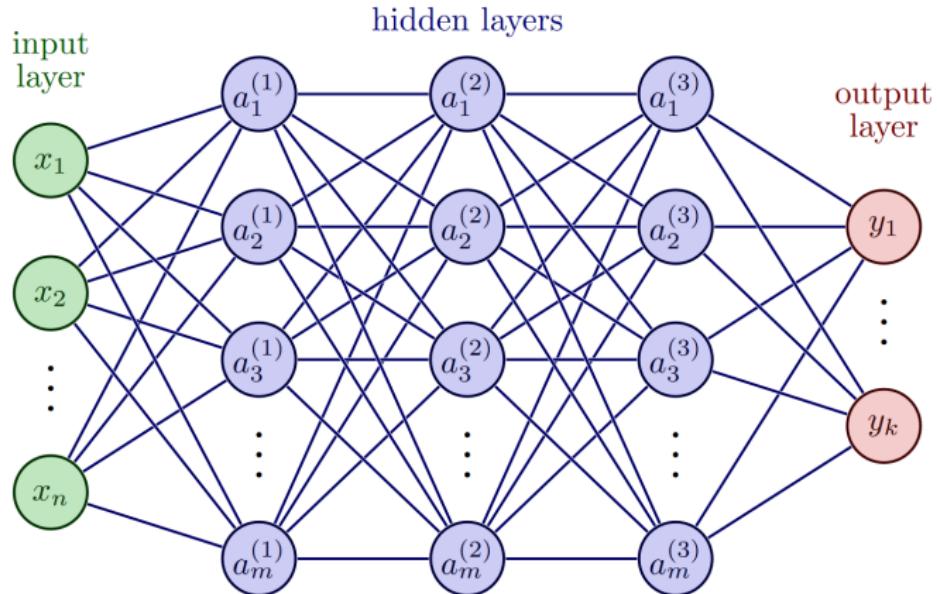
NN structure (not a PINN yet!)

A typical **Deep Learning Neural Network** (DLNN)



NN structure (not a PINN yet!)

A typical **Deep Learning Neural Network** (DLNN)



The output of the NN is $y_{NN}(\vec{x}, \theta)$, where $\theta = W_i, b_i, 0 \leq i \leq l$, l being the *number of all hidden layers*. θ is the set of all weights and biases.

DLNN Learning

The output of the NN has to be compared with the true solution and the learning process consists in minimizing a given measure, known as **the loss function** L . The simplest one is the *Mean Squared Error* (MSE)

$$MSE = \frac{1}{n} \sum_{i=1}^n |y_{NN} - y_i|^2$$

The minimization of MSE cannot be achieved immediately because the weights and biases are *initially arbitrary*. One cycle of learning is known as an **epoch**. After epoch $j+1$ the weights and biases of epoch j are updated

$$\begin{aligned}\omega_i^{j+1} &= \omega_i^j - \eta \frac{\partial L}{\partial \omega_i} \\ b_i^{j+1} &= b_i^j - \eta \frac{\partial L}{\partial b_i}\end{aligned}$$

η is known as the *learning rate*.

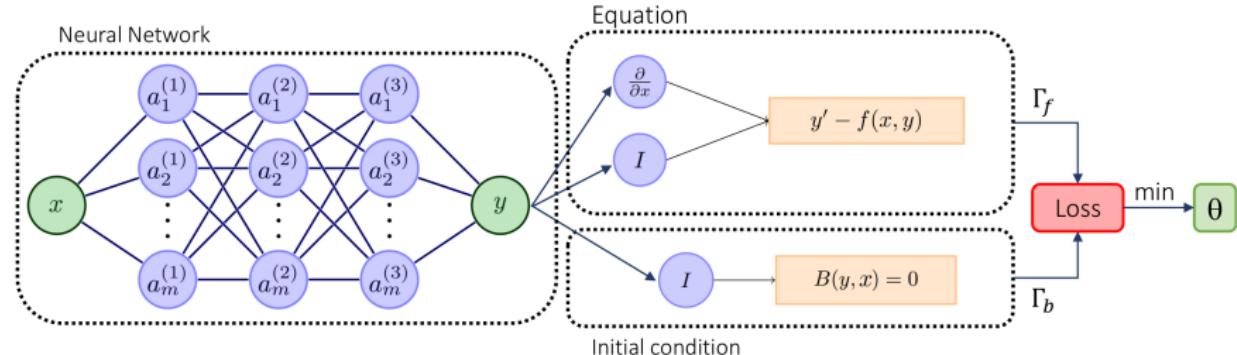
The aim of the whole learning process is to search for the **global minimum**. Anyways for an inappropriate learning rate the process may get stuck in a *local minimum*. The learning process is enabled by calculating the gradients above employing **Automatic Differentiation**, a method that permits calculating gradients using **Gradient Descent**.

Hyperparameters

- ① There are several parameters that can be changed manually in a NN, known as **hyperparameters** such as:
 - ① number of layers
 - ② number of neurons per layer
 - ③ number of *epochs*
 - ④ learning rate η , etc.
- ② There are no general rules in assessing these hyperparameters for a given NN. As a consequence it may require the adoption of *trial – and – error* approach. Possible heuristic rules may be considered.
- ③ Such lack of rigour may be concerning for some physicists may alienate them with the whole Deep Learning. Caution is always a good idea to succeed.

Building the PINN

The **Physically Informed Neural Network (PINN)** is built upon a typical NN by making a crucial addition.



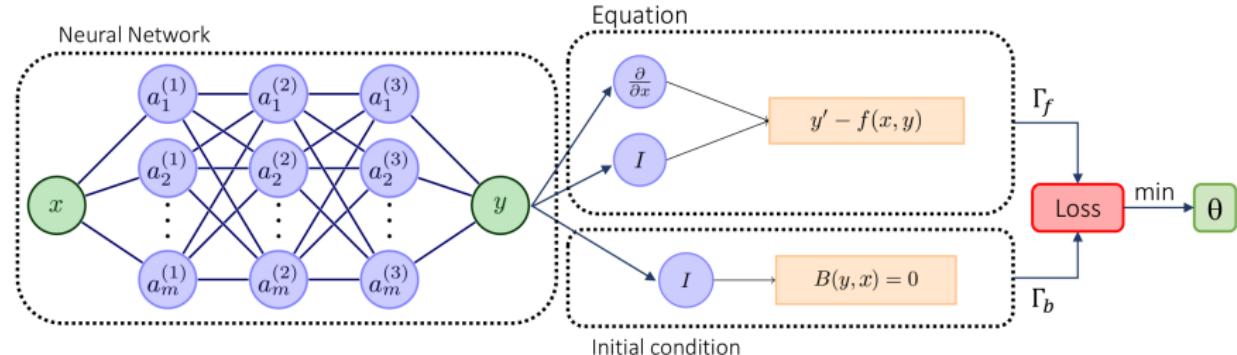
The Loss Function is modified by adding two more terms:

- ① Loss Function of the Domain L_D , calculated in points located in the domain of a function (similar to MSE)
- ② Loss Function of the Boundary L_B , calculated in points located in the boundary of the Domain

The total Loss Function: $L = \omega_D L_D + \omega_B L_B$. ω_D and ω_B are *hyperparameters* that must be tuned.

Building the PINN

The **Physically Informed Neural Network (PINN)** is built upon a typical NN by making a crucial addition.



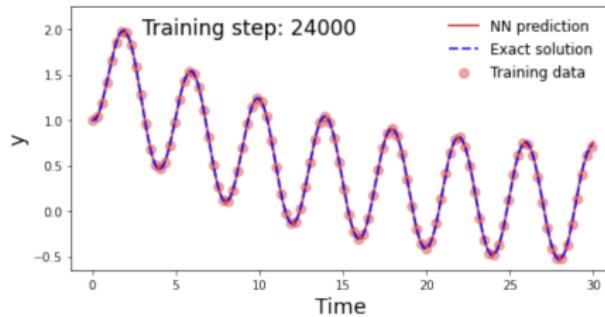
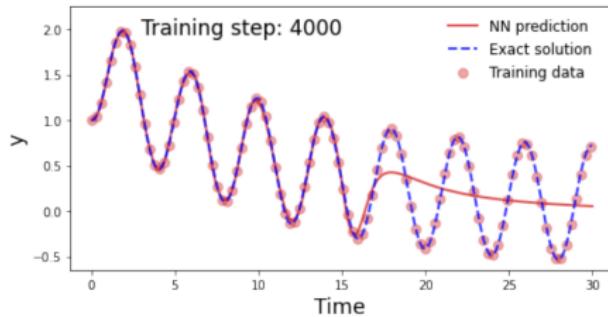
The Loss Function is modified by adding two more terms:

- ① Loss Function of the Domain L_D , calculated in points located in the domain of a function (similar to MSE)
- ② Loss Function of the Boundary L_B , calculated in points located in the boundary of the Domain

The total Loss Function: $L = \omega_D L_D + \omega_B L_B$. ω_D and ω_B are *hyperparameters* that must be tuned. **PINNs are mathematical tools with physical constraints.**

Why modifying the Loss Function

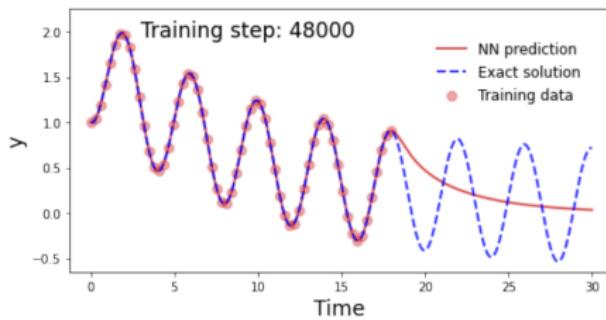
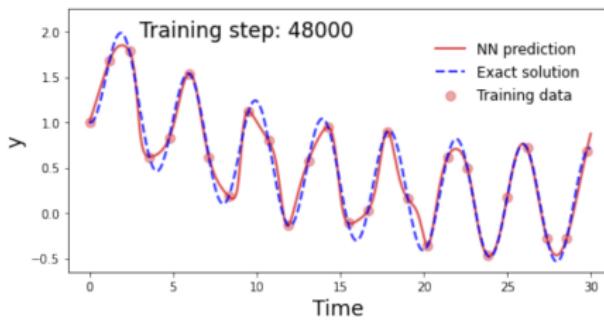
A simple NN requires training over many points of the domain and many, many epochs to be effective.



It is not better than the classical methods!

Why modifying the Loss Function

A sparse coverage of the domain, even after many epochs of training, does not produce an NN with extrapolating power.



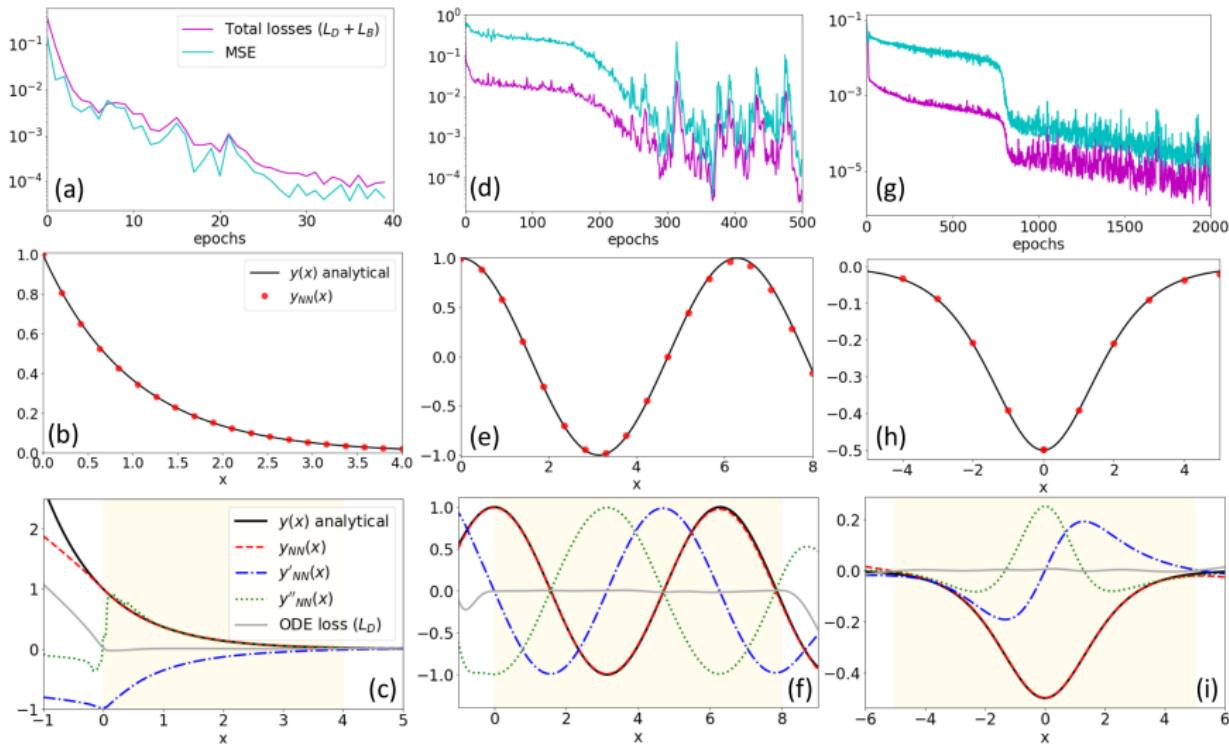
Simple NNs are good interpolators but bad extrapolators

Some examples with PINNs

The types of ODEs analyzed are:

- ① 1st order ODE of a typical exponential decay, like radioactive decay →
 $y'(x) + y(x) = 0, y(0) = 1$
- ② 2nd order ODE that describes the motion of a Harmonic Oscillator with unit angular frequency → $y''(x) + y(x) = 0, y(0) = 1, y'(0) = 0$
- ③ 2nd order ODE known as Korteweg-de Vries (KdV) equation, which can model solitons in ocean waves, fiber optics modes, and Bose-Einstein condensates in quantum mechanics →
 $y''(x) - y(x) + 3y^2(x) = 0, y(0) = -\frac{1}{2}, y'(0) = 0$

Some examples with PINNs



At last (brief)

Conclusions:

- It is possible to use (Physically Informed) NN to solve differential equations by decreasing computational costs
- With the current open-source libraries, Tensorflow or Pytorch, the implementations is straightforward

Future works:

- Apply such tools to solve numerically Navier-Stokes or MHD models

References

- ① Baty H., Baty L, 2023. Solving differential equations using physics informed deep learning: a hand-on tutorial with benchmark tests. hal-04002928v2
- ② Nair A. S., Sirignano J., Panesi M., MacArt J. F. 2023. Deep Learning Closure of the Navier–Stokes Equations for Transition-Continuum Flows 2303.12114v2
- ③ Navarro L. M., Moreno L: M., Rodrigo S. G. 2023. Solving differential equations with Deep Learning: a beginner's guide. 2307.11237v1using
- ④ Navarro L. M., Moreno L: M., Rodrigo S. G. 2023. PINNs-for-education. <https://github.com/IrisFDTD/PINNs-for-education>