Solving Partial Differential Equations with Artificial Neural Networks

Louis Libat

Research Project
Master in Mechanics UGE – UPEC

Supervisors:
Vincent LE CHENADEC – Can SELÇUK

Reviewer:
Éric CHENIER

May 13, 2026

Université Gustave Eiffel

UPEC
Connaissance - Action
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE

Outline

Outline

Partial Differential Equations

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbb{F}(t, \mathbf{x}, \mathbf{u}, \nabla_x(\mathbf{u}), ...)$$

Objective: solve for a quantity as a function of space and time, given its initial and boundary conditions.
Modern numerical methods are impressive.

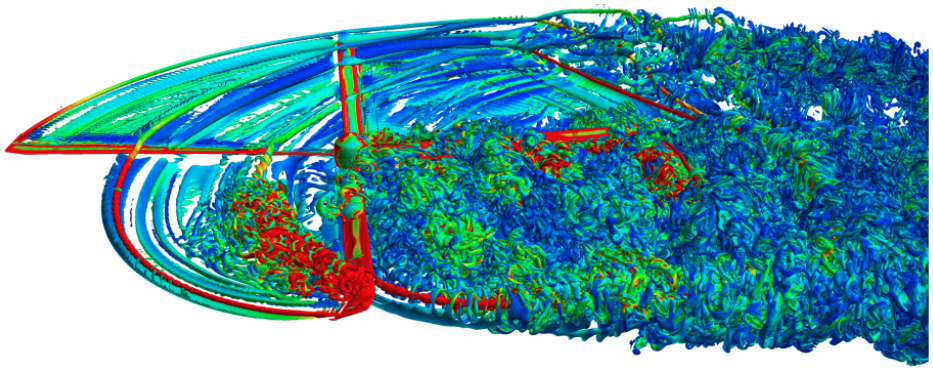


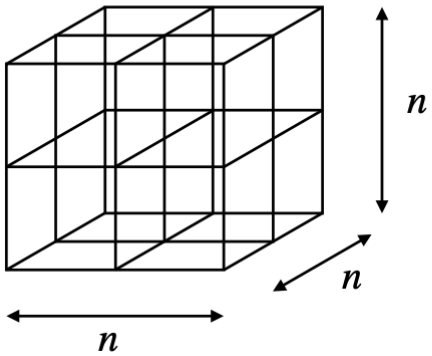
Figure: Simulation of dynamic stall on a helicopter rotor

BUT ...

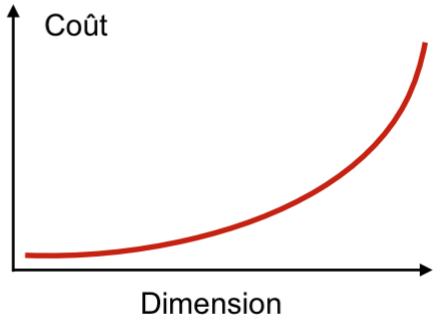
Many Challenges Remain

- The accuracy of the solution depends on mesh alignment and resolution.
- The mesh must be adapted to align with critical flow structures in order to maintain accuracy.
- The number of cells, and therefore the computational cost, increases exponentially with dimension.

Nombre de cellules = n^d



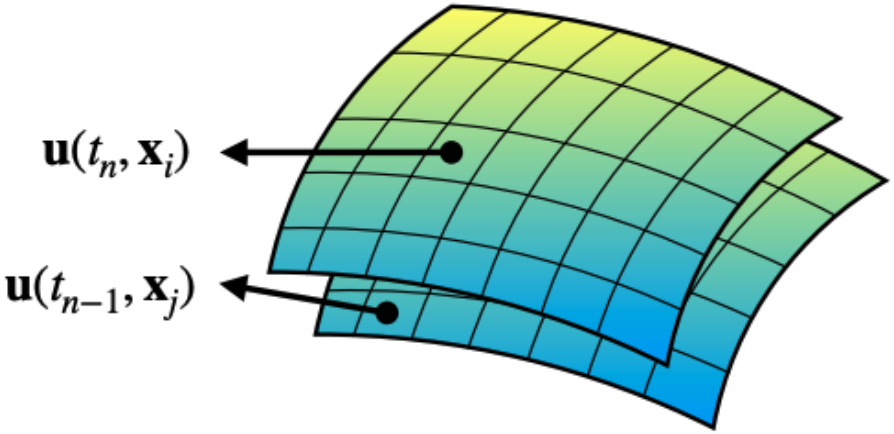
Coût CPU $\propto$ Nombre de cellules



Curse of dimensionality: multi-resolution, high-order methods or other advanced schemes are required to solve complex problems within a reasonable time.

Can We Remove the Mesh Entirely?

Classical discretization methods

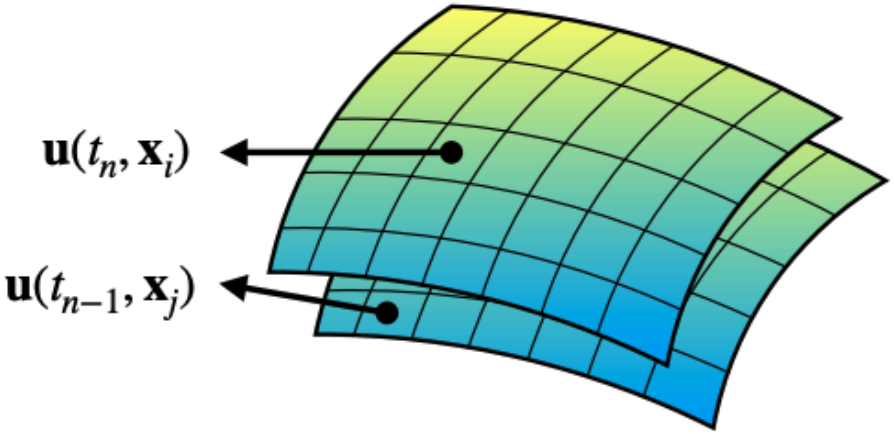


The problem is converted into a large system of ordinary differential equations:

$$\frac{\partial \mathbf{u_i}}{\partial t} = \mathbb{F}(\mathbf{u_1}, \mathbf{u_2}, ..., \mathbf{u_N})$$

Can We Remove the Mesh Entirely?

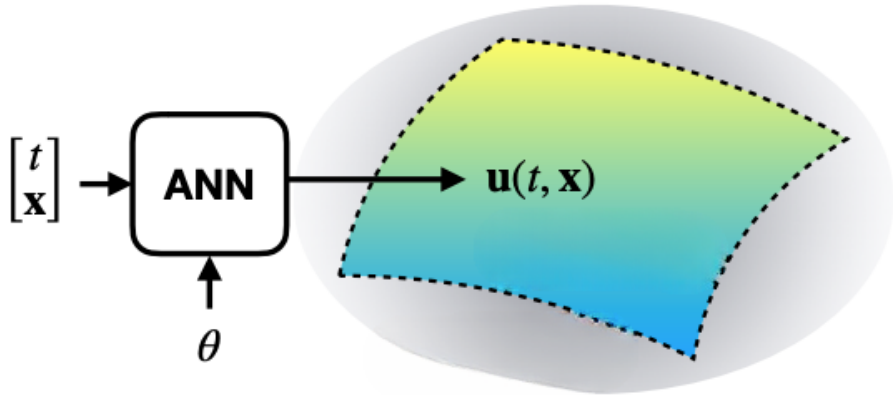
Classical discretization methods



The problem is converted into a large system of ordinary differential equations:

$$\frac{\partial \mathbf{u_i}}{\partial t} = \mathbb{F}(\mathbf{u_1}, \mathbf{u_2}, ..., \mathbf{u_N})$$

Deep learning approach

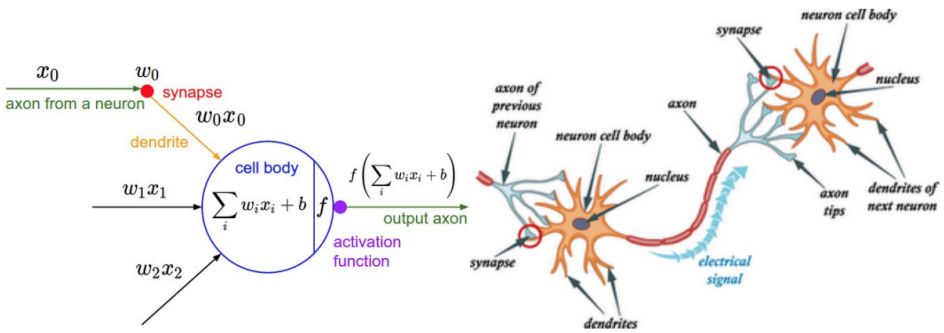


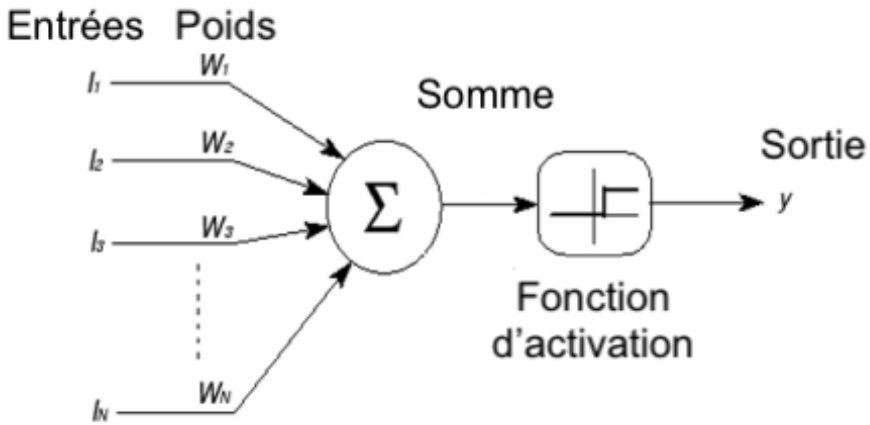
The problem is converted into an optimization problem over the parameters of a neural network:

$$\min_{\theta} \sum_{(t,x_i)} \left| \frac{\partial \mathbf{u}(\theta)}{\partial t} - \mathbb{F}(\mathbf{u}(\theta)) \right|$$

Neural Networks: Artificial?

1943: McCulloch and Pitts: formal computational neuron
1958: F. Rosenblatt: Perceptron





Multilayer Neural Networks

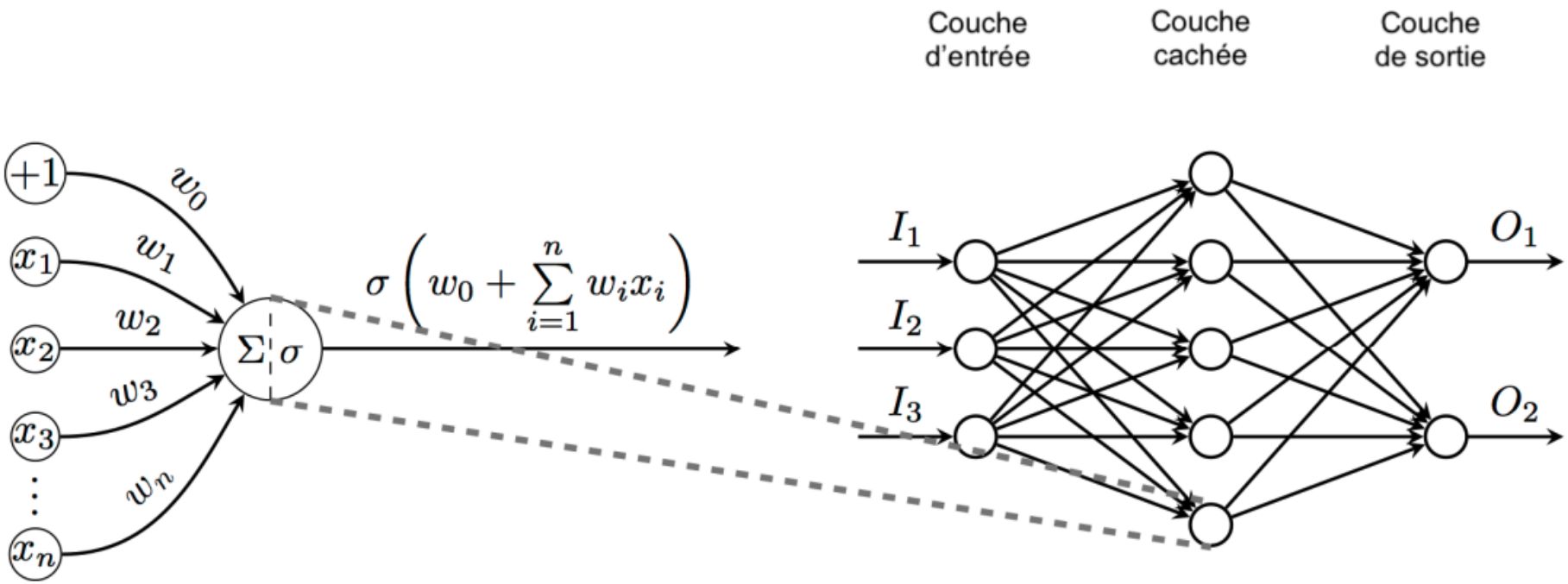


Figure: Credits: @github.com/PetarV-

Universal approximation theorem: A multilayer neural network with a locally bounded piecewise-continuous activation function can approximate any continuous function to a prescribed accuracy.

Learning

A network is said to **learn** when its **weights** are optimized with respect to an **objective function**. In practice, this usually means minimizing a **cost function**.

$$\min_{\theta} \mathbb{L}(\theta) = \min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \mathbb{L}_{loss}(y, F(x, \theta)) \tag{1}$$

Deep learning consists in training an artificial neural network with many hidden layers: the network is deep.

Stochastic Gradient Descent

- Training dataset: $\mathbb{D} = \{(X_1, Y_1), ..., (X_n, Y_n)\}$
- Build a neural network: $f(\mathbf{X}, \theta) \rightarrow \hat{Y}$
- Define a cost function: $\mathbb{L} = \frac{1}{n} \sum_{i=1}^{n} ||f(\mathbf{X_i}, \theta) - \mathbf{Y_i}||_2^2$
- Minimize the cost function: $\theta^* = \text{argmin}_\theta \mathbb{L}$

Stochastic Gradient Descent

- Training dataset: $\mathbb{D} = \{(X_1, Y_1), ..., (X_n, Y_n)\}$
- Build a neural network: $f(\mathbf{X}, \theta) \rightarrow \hat{Y}$
- Define a cost function: $\mathbb{L} = \frac{1}{n} \sum_{i=1}^{n} ||f(\mathbf{X_i}, \theta) - \mathbf{Y_i}||_2^2$
- Minimize the cost function: $\theta^* = \text{argmin}_\theta \mathbb{L}$

Algorithm:

1. Weight initialization: $\theta^0 = \mathbb{N}(0, \mu)$
2. Gradient-based update: $\theta^{k+1} = \theta^k - \lambda \nabla_\theta \mathbb{L}$
3. Repeat until convergence: $\theta \rightarrow \theta^*$

Outline

So, How Can We Learn Equations? – EDNN

Consider the following general nonlinear PDE:

$$\begin{cases} \frac{\partial \mathbf{u}}{\partial t} - \mathbf{N}_x(\mathbf{u}) = 0, & x \in \Omega,\ t > 0 \\ u(x, t = 0) = u_0(x), & x \in \Omega \\ + \text{ Boundary conditions} \end{cases}$$

So, How Can We Learn Equations? – EDNN

A neural network can represent the solution of the equation at each fixed time. Thus,

$$\mathbf{u}_h = \mathbf{u}_h(x, \theta(t))$$

where θ denotes the network parameters, assumed to depend directly on time. The initial condition is learned by training the network on the pairs $(\mathbf{x}, \mathbf{u}(\mathbf{x}, t = 0))$.

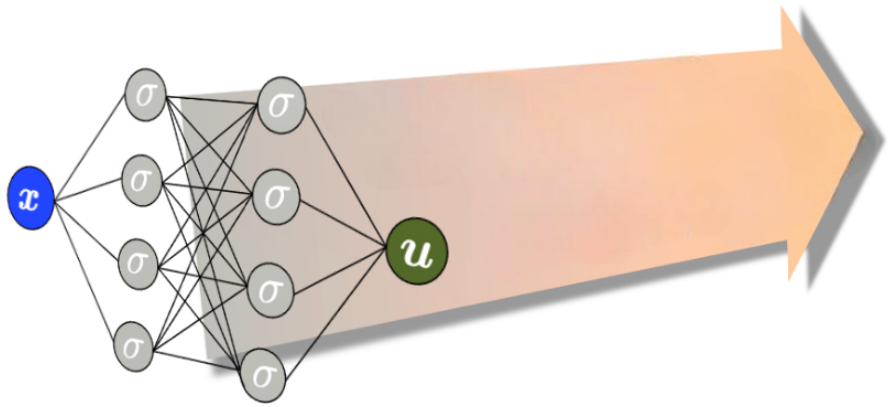


Figure: EDNN approach

So, How Can We Learn Equations? – EDNN

Using the chain rule:

$$\frac{\partial \mathbf{u}_h}{\partial t} = \frac{\partial \mathbf{u}_h}{\partial \theta}\frac{\partial \theta}{\partial t}$$

We can compute $\frac{\partial \theta}{\partial t}$ by minimizing:

$$\mathcal{J}(\gamma) = \frac{1}{2}\int_\Omega \left\| \frac{\partial \boldsymbol{u}_h}{\partial \theta}\gamma - \mathbf{N}_x(\boldsymbol{u}_h) \right\|_2^2 d\boldsymbol{x}$$

Therefore, at each time step:

$$\dot{\theta}(t) = \text{argmin}_\gamma \mathcal{J}(\gamma, \theta(t))$$

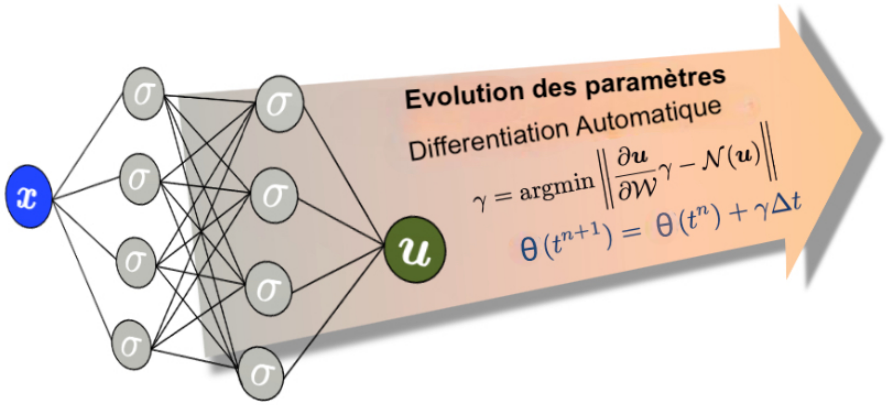


Figure: EDNN approach

So, How Can We Learn Equations? – EDNN

γ_{opt} is obtained from:

$$\nabla_\gamma \mathcal{J}(\gamma_{opt}) = 0$$

Hence:

$$\mathbf{J}^T \mathbf{J} \hat{\gamma}_{opt} = \mathbf{J}^T \mathbf{N}$$

Since direct inversion is too expensive, the system is solved in the least-squares sense.
Time marching can then be performed as:

$$\frac{\theta^{n+1} - \theta^n}{\Delta t} = \hat{\gamma}_{opt}$$

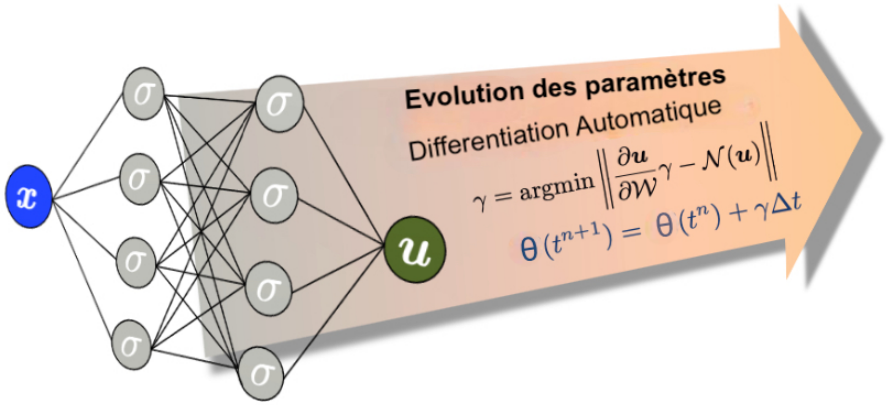


Figure: EDNN approach

Examples of PDE Solutions

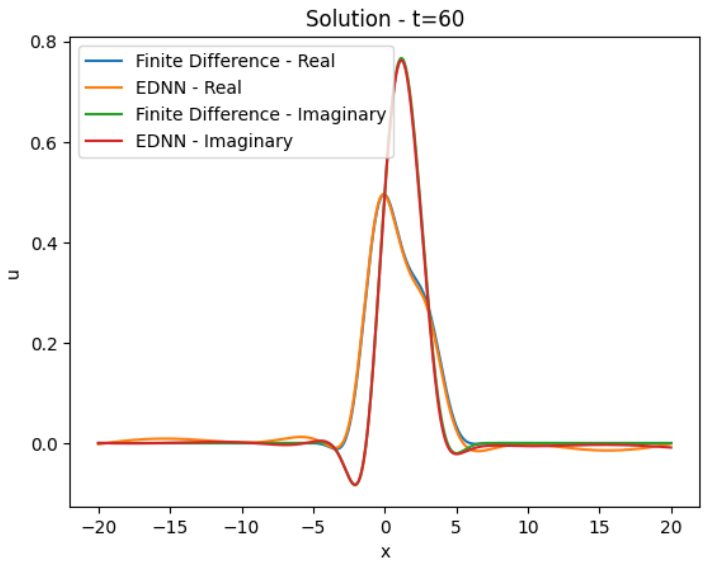
- Ginzburg–Landau equation: complex-valued equation leading to a coupled differential system

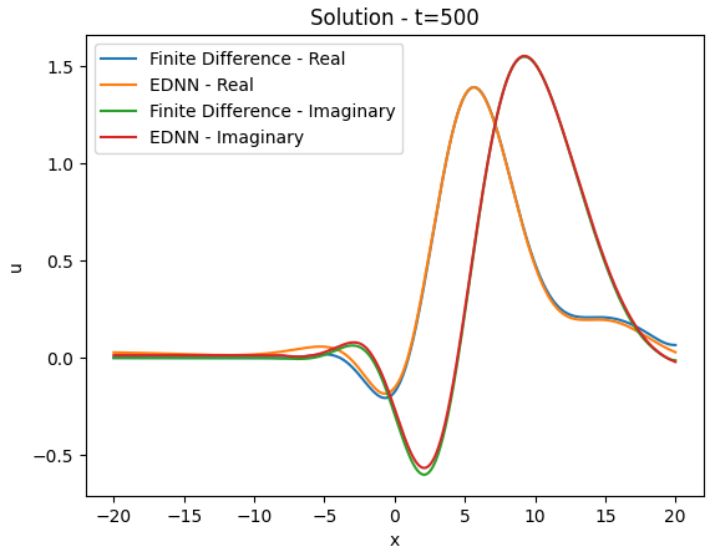
$$\frac{\partial q}{\partial t} = \left(-\nu \frac{\partial}{\partial x} + \gamma \frac{\partial^2}{\partial x^2} + \mu \right) q$$

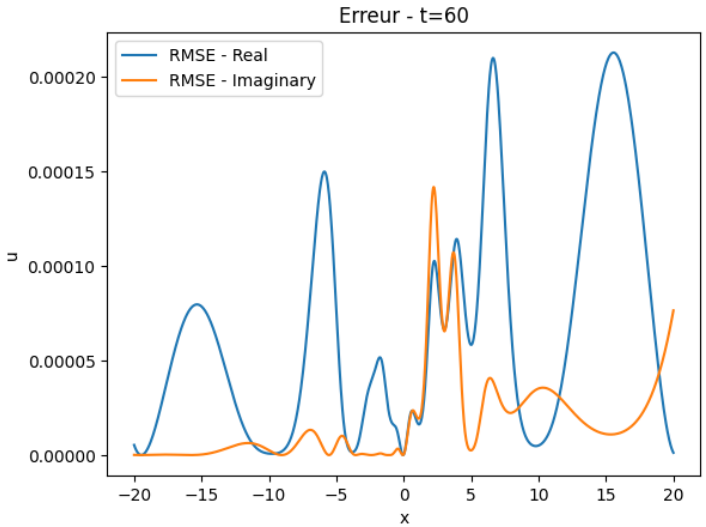
- Taylor–Green vortex: Navier–Stokes equations

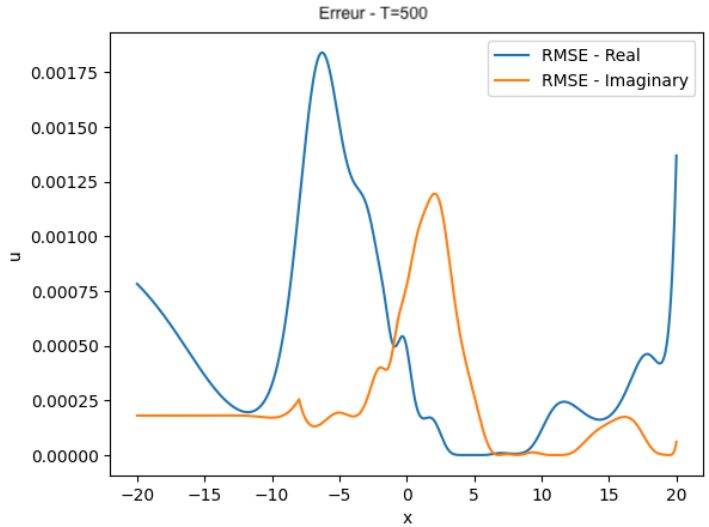
$$\begin{cases} \nabla \cdot \mathbf{u} = 0 \\ \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla P + \nu \Delta \mathbf{u} + \mathbf{f} \end{cases}$$

Ginzburg–Landau Equation









Taylor–Green Vortex

$\Omega = [0, 2\pi]^2,\ t \in \mathbb{R}^+$

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 \\ \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla P + \nu \Delta \mathbf{u} + \mathbf{f} \\ u(x, y, t = 0) = U_0 \cos(x) \sin(y) \\ v(x, y, t = 0) = -U_0 \sin(x) \cos(y) \\ + \text{ Periodic boundary conditions} \end{cases}$$

The exact solution of this problem is:

$$\begin{cases} u(x, y, t) = U_0 \cos(x) \sin(y) e^{-2\nu t} \\ v(x, y, t) = -U_0 \sin(x) \cos(y) e^{-2\nu t} \end{cases}$$

Taylor–Green Vortex

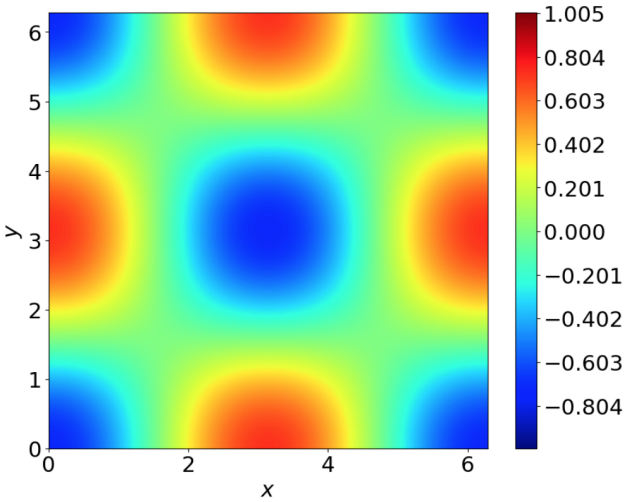


Figure: EDNN vorticity

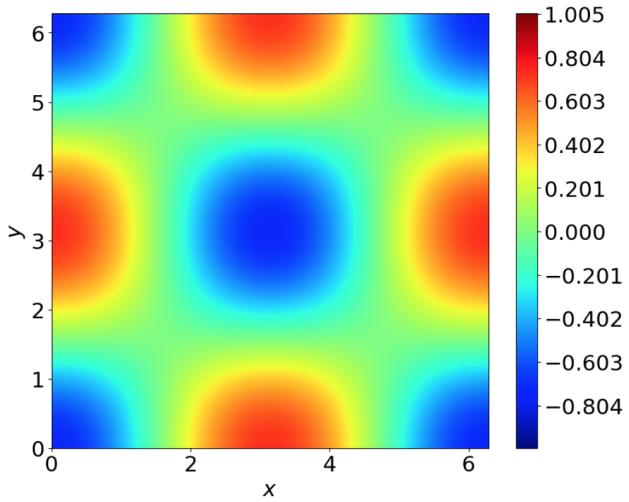


Figure: Exact vorticity

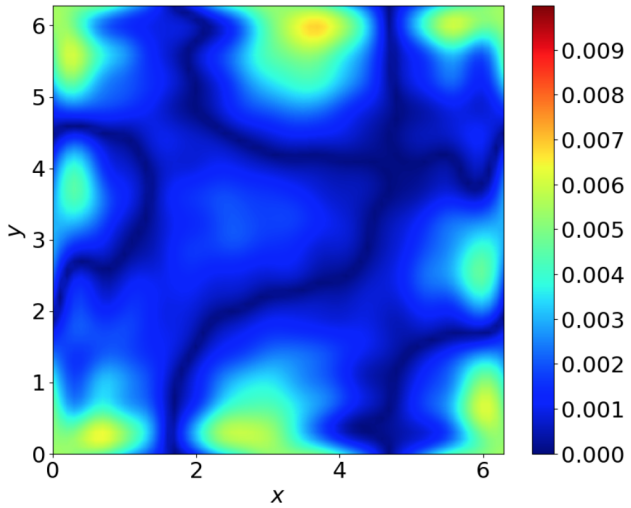
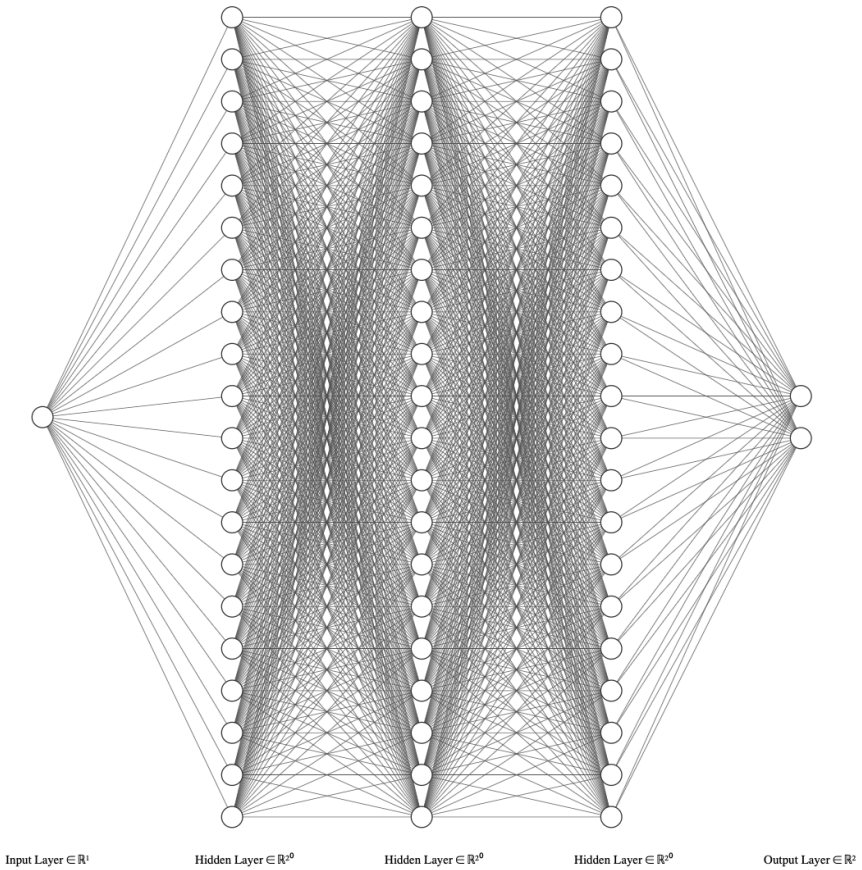


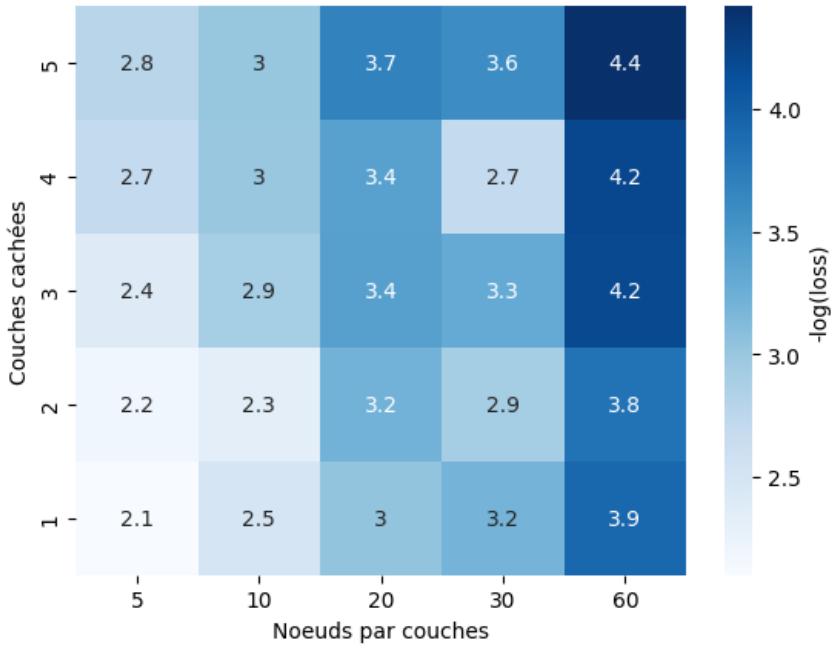
Figure: Vorticity error

- Training CPU time: 83 s
- Time-marching CPU time: 432 s

Outline

Impact of the Network Architecture





Impact of the Network Shape

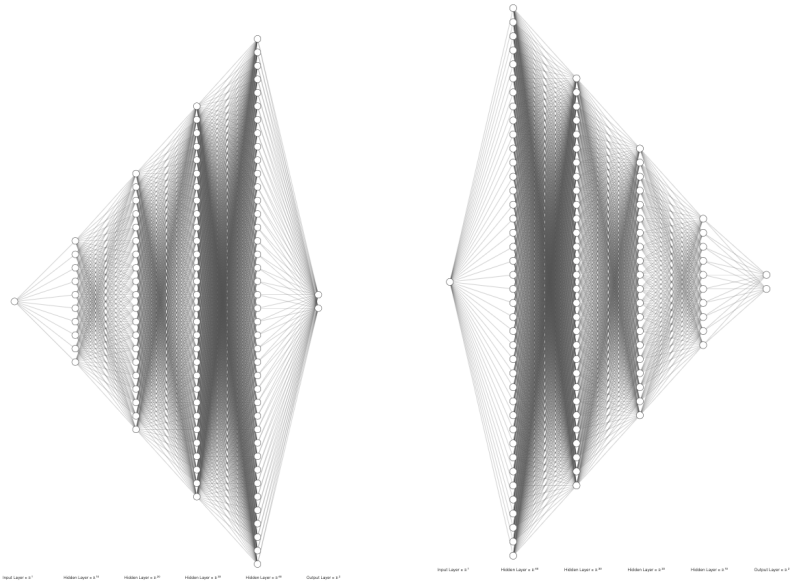


Figure: Network shape

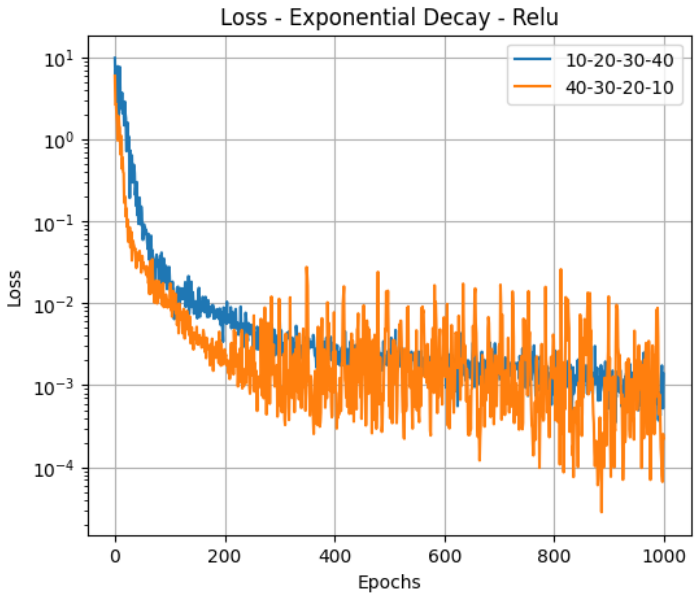


Figure: Loss – Network shape

- Early hidden layers $\leftrightarrow$ low-frequency components
- Later hidden layers $\leftrightarrow$ high-frequency components

Loss Landscape

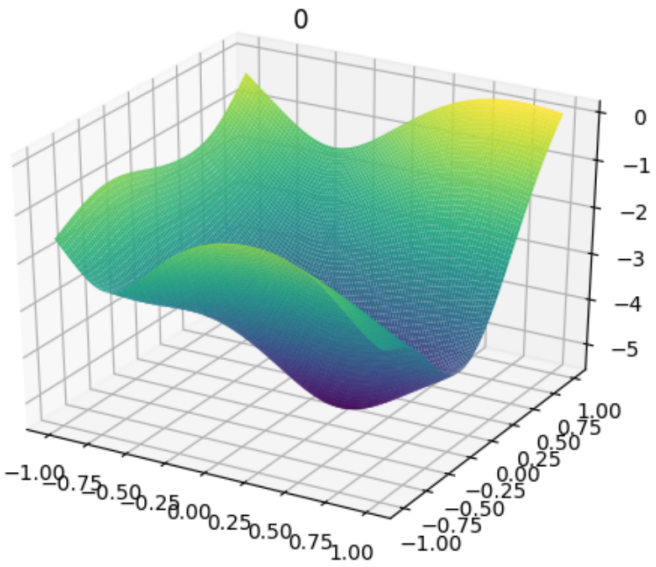


Figure: $\mu = 0$

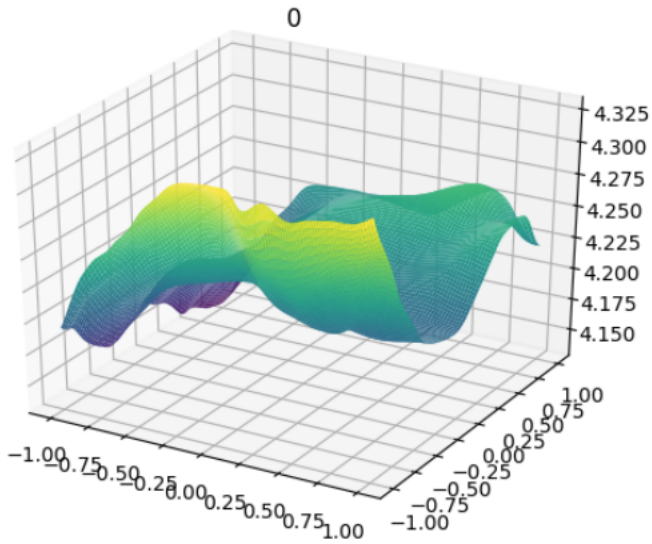


Figure: $\mu = 0.38$

Outline

Conclusion

Based on the results:

- Deep neural networks proved highly effective at representing complex functions.
- Using an ANN to solve a PDE together with its boundary and initial conditions leads to an optimization problem.
- The EDNN approach has the potential to solve highly nonlinear problems over long-time evolutions.
- Several aspects remain open: boundary-condition treatment, time integration, sampling strategies, and more.

Further developments are still needed:

- Treatment of additional boundary conditions.
- Parallelization and domain decomposition: one compute node $\leftrightarrow$ one neural network.

Conclusion

Thank you for your attention!


Yifan Du and Tamer Zaki.
Evolutional deep neural network.
Physical Review E, 104, 10 2021.


S Bagheri, Espen Åkervik, Luca Brandt, and D Henningson.
Input-output analysis and control design of spatially developing shear flows.
5th AIAA Theoretical Fluid Mechanics Conference, 06 2008.


Mariella Kast and Jan Hesthaven.
Positional embeddings for solving pdes with evolutional deep neural networks.
08 2023.


Joan Bruna, Benjamin Peherstorfer, and Eric Vanden-Eijnden.
Neural galerkin schemes with active learning for high-dimensional evolution equations.
Journal of Computational Physics, 496:112588, 10 2023.

Appendix – How Are Gradients Computed?

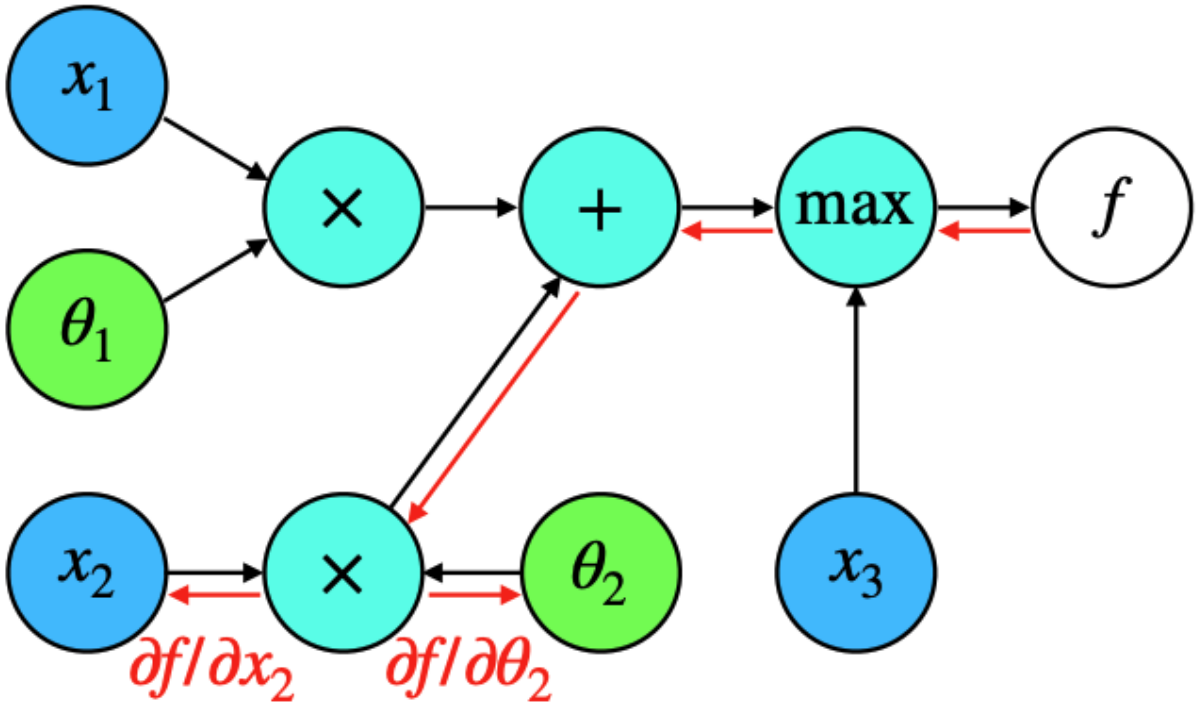
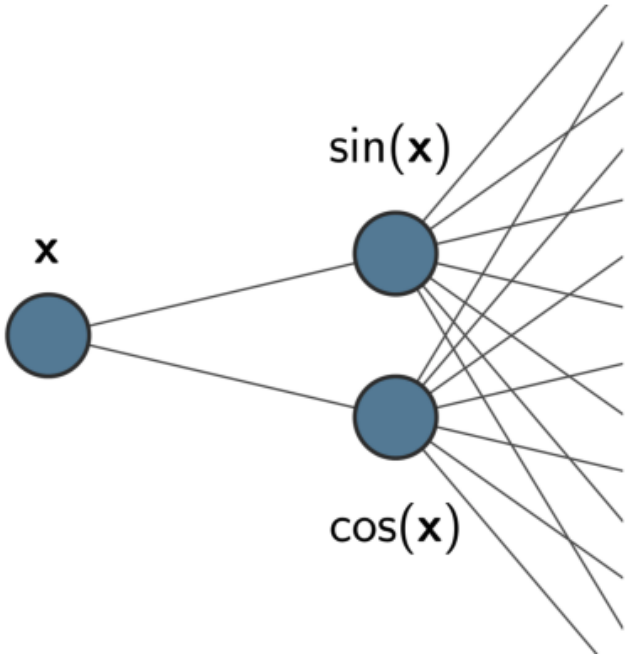


Figure: Computational graph

- Gradients are computed using the chain rule.
- Modern libraries implement gradient computation with respect to any node of the computational graph using backpropagation.

Appendix – What About Boundary Conditions?

Periodic boundary conditions



Dirichlet boundary conditions