Scientific machine learning: some methods and applications

J. Aghili[2], L. Bois[12], E. Franck[12], V. Michel Dansac[12], L. Navoret[12], V. Vigon[12]

SISMA, Cea DAM

[1]Inria Nancy Grand Est, France
[2]IRMA, Strasbourg university, France

Outline

Introduction

Maching learning: principle

- Set of methods to build models **from data**.
- In general, approaches use parametric functions f_θ where the parameters are chosen by optimization
- Three main types of ML problems:
  - Supervised learning: construct models like

    $$y = f(x) + \epsilon, \text{ or } \quad \mathbb{P}(y|x)$$

    with ϵ some noise, using inputs and outputs examples. We solve:

    $$\min_\theta \sum_i^n L(f(x_i), y_i),$$

    with L a loss function.
  - Unsupervised learning: construct models like

    $$\mathbb{P}(x), \text{ or } \quad \mathbb{P}(x|z),$$

    which explain data structure/probability data with some examples (z potential latent variables), where ϵ is some noise, and using inputs and outputs examples.
  - Reinforcement learning which considers time control problems like:

    $$s_{n+1} = f(s_n, a_n)$$

    with s_n a state and a_n an action, and constructs the model $\pi(a_n|s_n)$ which decides the best action to maximize some criterion.

Deep learning: neural networks

- **Which parametetric functions**? Polynomial? (not in high dimension problems).
- Current choice: kernel approximation or neural network.

Layer

A layer is a function $L_l(\mathbf{x}_l) : \mathbb{R}^{d_l} \rightarrow \mathbb{R}^{d_{l+1}}$ given by

$$L_l(\mathbf{x}_l) = \sigma(A_l \mathbf{x}_l + \mathbf{b}_l),$$

$A_l \in \mathbb{R}^{d_{l+1}, d_l}$, $\mathbf{b} \in \mathbb{R}^{d_{l+1}}$ and $\sigma()$ a nonlinear function applied component by component.

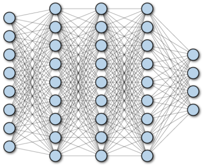
Neural network

A neural network is parametric function obtained by composition of layers:

$$f_\theta(\mathbf{x}) = L_n \circ .... \circ L_1(\mathbf{x})$$

with θ the trainable parameters composed of all the matrices $A_{l,l+1}$ and biases $\mathbf{b}_l$.

- Fully connected neural network (FCNN): the matrices $A_{l,l+1}$ are dense.



Deep learning: neural networks II

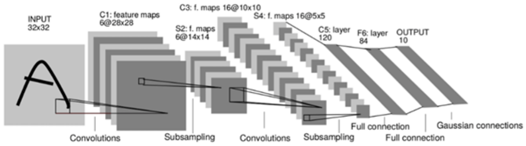
Neural network

For structured data like pictures, time signals or functions on structured grids, there exist more powerful NN: convolutional neural networks.

- 1D convolutional neural network: the matrices L_i are sparse Toeplitz matrices.

$$A = \begin{pmatrix} a & c & 0 & 0 \\ b & a & c & 0 \\ 0 & b & a & c \\ 0 & 0 & b & a \end{pmatrix}$$

- This is equivalent to applying a small convolution kernel to the 1D signal.
- We often apply some convolutions to the signal on each layer, to create several new signals.
- 2D convolutional networks:



- **Ingredients**: translation equivariance/invariance, scale separation, deformation stability (Mallat 2016).

Our objectives

Limit of ML for PDE

- We can directly solve or approximate PDEs with ML (next section) but without **convergence/stability** or other guarantees.

Objective I

- Structure- and property-preserving reduced models with ML (ANR with IPP Garching).

Objective II

- Construct new hybrid numerical methods with ML, conserving the classical properties of PDE approximations.

Objective III

- Construct a ML framework to tend towards self-adapting simulation codes.

Physics-based learning

PINN's

- How to directly solve PDEs with NNs? Physics-Informed neural network (M. Raissi, G. E. Karniadakis et al, 2017).
- Problem:

$$\begin{cases} \partial_t \mathbf{U} = \mathcal{N}(\mathbf{U}, \partial_x \mathbf{U}, \partial_{xx}\mathbf{U}, \boldsymbol{\beta}) \\ \mathbf{U}_h(t,x) = g(x), \quad \forall x \in \partial\Omega \\ \mathbf{U}(0,x) = \mathbf{U}_0(x, \boldsymbol{\alpha}) \end{cases}$$

  with $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ some parameters.
- **Idea I**: represent/approximate solutions of PDEs by a NN. So we define a FCNN: $\mathbf{U}_\theta(t,x)$.
- **Idea II**: Neural networks are $C^p(\mathbb{R}^d)$ functions so we can exactly compute the derivative appearing in the PDE.

Optimization problem of PINN's

$$\min_\theta J_r(\theta) + J_b(\theta) + J_i(\theta)$$

with

$$J_r(\theta) = \int_0^T \int_\Omega \| \partial_t \mathbf{U}_\theta(t,x) - \mathcal{L}(\mathbf{U}_\theta, \partial_x \mathbf{U}_\theta, \partial_{xx}\mathbf{U}_\theta, \boldsymbol{\mu})(t,x) \|_2^2 \, dxdt$$

and

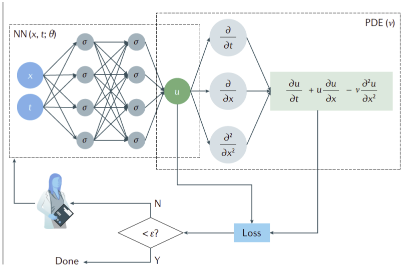
$$J_b(\theta) = \int_0^T \int_{\partial\Omega} \| \mathbf{U}_\theta(t,x) - g(x) \|_2^2 \, dxdt, \quad J_i(\theta) = \int_\Omega \| \mathbf{U}_\theta(0,x) - \mathbf{U}_0(x) \|_2^2 \, dx$$

PINN's II

- **How to approximate the previous problem?** Using the Monte-Carlo method.
- We randomly choose collocation points: $(t_1, x_1, ...t_N, x_N)$. Thanks to these points, we approximate the loss function as follows:

$$J_r(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} \parallel \partial_t \mathbf{U}_\theta(t_i, x_i) - \mathcal{L}(\mathbf{U}_\theta, \partial_x \mathbf{U}_\theta, \partial_{xx}\mathbf{U}_\theta, \boldsymbol{\beta})(t_i, x_i) \parallel_2^2$$

- General PINNs behavior:



- PINNs variants: add data, BC/initial condition strongly imposed, causal training (time sub-interval by time sub-interval), variational PINNs, entropic PINNs, preferential sampling using residuals, . . .

Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2}\right) = \nu \partial_{xx} \rho$

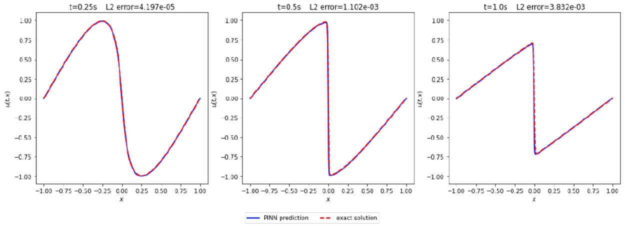


Figure 3. Burgers' equation: Comparison of the predicted and exact solutions corresponding to three temporal snapshots. $\nu = 10^{-3}$

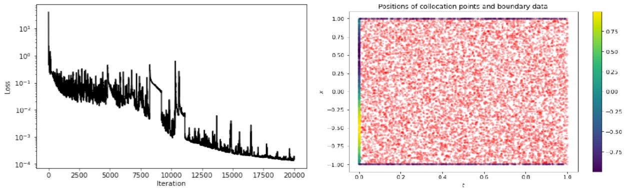


Figure 4. Loss history during training (left) distribution of training points. The colormap correspond to the values of the initial condition function and the boundary condition (right)

Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2}\right) = \nu \partial_{xx} \rho$

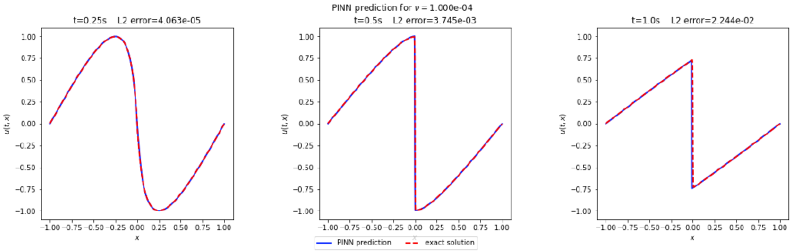


Figure 5. Burgers' equation: Comparison of the predicted and exact solutions corresponding to three temporal snapshots. $\nu = 10^{-4}$

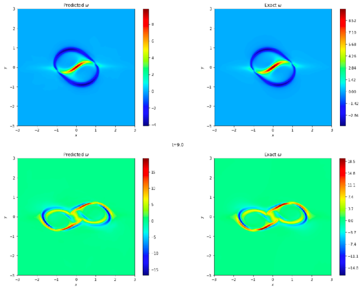
- Training for viscosity 10^{-4}: 2h.

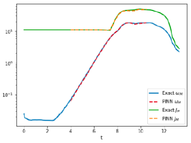
Example: Reduced MHD

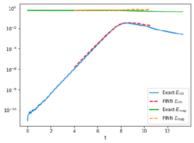
- Application: reduced MHD equations

$$
\begin{cases}
\frac{d\omega}{dt} + [\phi, \omega] = [\psi, j] + \nu\Delta\omega, \\
\frac{d\psi}{dt} + [\phi, \psi] = \eta\Delta\eta, \\
\omega = -\Delta\phi, \\
j = -\Delta\psi
\end{cases}
$$

- Test case: Tilt instability.
- ω prediction (left), energy evolution (right)







- Since it is a multi-scale problem, the training is complicated, and requires 10 data points in time to get an accurate description of the instability.

PINN's and parametric PDEs

- **Advantages of PINNs**: mesh-less approach, not too sensitive to the dimension.
- **Drawbacks of PINNs**: they are not competitive with classical methods.
- Interesting possibility: use the strengths of PINNs to solve parametric PDEs.
- The neural network becomes $\mathbf{U}_\theta(t, x, \boldsymbol{\alpha}, \boldsymbol{\beta})$.

New Optimization problem of PINN's

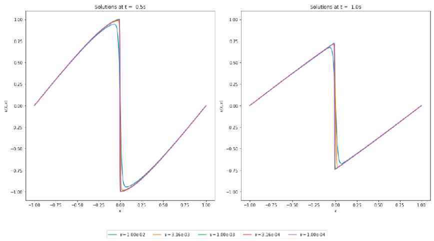
$$\min_\theta J_r(\theta) + ...$$

with

$$J_r(\theta) = \int_V \int_0^T \int_\Omega \parallel \partial_t \mathbf{U}_\theta(t,x) - \mathcal{L}(\mathbf{U}_\theta, \partial_x \mathbf{U}_\theta, \partial_{xx}\mathbf{U}_\theta, \boldsymbol{\mu})(t,x) \parallel_2^2 \, dxdt$$

with V a subspace of the parameters $(\boldsymbol{\alpha}, \boldsymbol{\beta})$.

- Application for the Burgers equations with many viscosities $[10^{-2}, 10^{-4}]$:



Operator learning

- We consider the following problem:

$$\begin{cases} G_{\boldsymbol{\mu}}(u(x)) = \partial_t u(x) + \mathcal{L}_{\boldsymbol{\mu}}(u(x)) = 0 \quad \text{on } \Omega, \\ u(x) = g(x) \quad \text{on } \partial\Omega, \\ u(t=0,x) = u_0(x) \end{cases}$$

- Formally, there exists a pseudo inverse operator G^+, such that $G^+(u_0, g, \boldsymbol{\mu}) = u(x)$.

Operator learning

Approximate G^+ by a neural network on a subspace of the data, to quickly compute an approximation of the solution.

First approach: discrete approach

We discretize the data on a mesh $u_{0,h}, g_h, \boldsymbol{\mu}_h$, and we construct a neural network $G^+_\theta(u_{0,h}, g_h, \boldsymbol{\mu}_h)$ (in general, a CNN) which minimizes $\mathcal{J}(\theta) = \mathcal{J}_1(\theta) + \mathcal{J}_2(\theta)$, with

$$\mathcal{J}_1(\theta) = \int_G \int_{U_0} \int_{\boldsymbol{\mu}} \sum_{n=1}^{n_T} \parallel G^+_\theta(\boldsymbol{g}_h, \boldsymbol{u}^n_h, \boldsymbol{\mu}_h) - \boldsymbol{u}^{n+1}_h \parallel_2^2 dg dU_0 \boldsymbol{\mu}$$

and

$$\mathcal{J}_2(\theta) = \int_G \int_{U_0} \int_{\boldsymbol{\mu}} \sum_{n=1}^{n_T} \parallel G_{\boldsymbol{\mu},\Delta t}(G^+_\theta(\boldsymbol{g}_h, \boldsymbol{u}^n_h, \boldsymbol{\mu}_h), \boldsymbol{u}^n_h) \parallel_2^2 dg dU_0 \boldsymbol{\mu} dx dt,$$

with $G_{\boldsymbol{\mu},\Delta t}(\boldsymbol{u}^{n+1}_h, \boldsymbol{u}^n_h)$ a scheme, and where integrals are approximated by MC.

Neural operator

Second approach: continuous approach

We construct a neural network $G_\theta^+(u_{0,h}, g_h, \boldsymbol{\mu}_h)$, which minimizes $\mathcal{J}(\theta) = \mathcal{J}_1(\theta) + \mathcal{J}_2(\theta)$, with

$$\mathcal{J}_1(\theta) = \int_G \int_{U_0} \int_{\boldsymbol{\mu}} \int_\Omega \int_0^T \parallel G_\theta^+(t, x, g(x), \boldsymbol{u}_0(x), \boldsymbol{\mu}_h(x,t)) - \boldsymbol{u}(x,t) \parallel_2^2 \, dg dU_0 d\boldsymbol{\mu} dx dt$$

and

$$\mathcal{J}_2(\theta) = \int_G \int_{U_0} \int_{\boldsymbol{\mu}} \int_\Omega \int_0^T \parallel G_{\boldsymbol{\mu}}(G_\theta^+(t, x, g(x), \boldsymbol{u}_0(x), \boldsymbol{\mu}_h(x,t))) \parallel_2^2 \, dg dU_0 d\boldsymbol{\mu} dx dt,$$

where the integrals are approximated by MC.

- This approach leads to so-called neural operators (N. Kovachki, Z. Li et al 2021).
- Which neural network to use?
- Example:

$$\begin{cases} -\nabla \cdot (a(\boldsymbol{x})\nabla u) = f(\boldsymbol{x}), & \forall x \in \Omega \\ u = 0, & \forall x \in \partial\Omega \end{cases}$$

- The solution is given by

$$u(x) = \int_\Omega G_a(x,y) f(y) dy$$

with G_a a Green kernel. **Important**: the operator in non-local.

Neural operator: Fourier NN I

Integral kernel

We call integral kernel applied to a function $v(y) \in C^0(D_t; \mathbb{R}^{n_t})$ the quantity

$$\mathcal{K}(v)(x) = \int_{D_t} k(x,y)v(y)d\nu(y),$$

with $k(x,y) \in C^p(D_{t+1} \times D_t; \mathbb{R}^{n_{t+1}} \times \mathbb{R}^{n_t})$ and ν a measure.

Neural operator layer

We call a integral kernel layer an operator which transforms $v_l(x)$ into a function $v_{l+1}(x)$, and which has the form:

$$\forall x \in D_{l+1},\ v_{l+1}(x) = \sigma_{l+1}\left(W_l v_l(\pi_l(x)) + b(x) + \mathcal{K}^t(v)(x)\right)$$

where $W_t \in \mathbb{R}d_{l+1}, d_l$ is a weight matrix and where Π_l is a mapping between D_{l+1} and D_l.

- It requires computing the integral kernel many times.

Neural operator: Fourier NN II

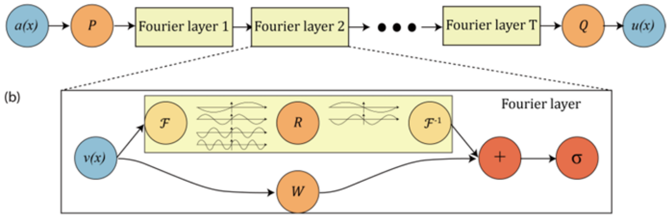
Fourier Neural Network

The FNOs use neural operator layers with an integral kernel:

$$\mathcal{K}(v)(x) \approx \mathcal{F}^{-1}(R_\theta \mathcal{F}(v(x)),$$

with R_θ the learnable filters in the Fourier space. In practice it is computed with an FFT.

- Principle:



- Contrary to the discrete CNN case, we can change the mesh resolution (it is also possible with CNNs, provided interpolation is used), and we could adapt the approach to unstructured grids.

Physics-based Learning

Summary of the main goal

Here, the neural network takes the parameters and predicts the solution. The NN approximates the map from data or parameters to solutions.

Application of Physics-based learning

Nonlinear elliptic problems and Newton's method

- We want to solve the following elliptic problem:

$$u - \alpha_0 \nabla \cdot (A(x,y)k(u)\nabla u) = f(x,y).$$

- It also corresponds to the implicit part of a diffusion equation.

Solver

- Finite difference or Finite element (here on structured meshes)
- Newton-Krylov method (Jacobian-free approximation + GMRES for linear part)

- After discretisation, we solve the problem:

$$G_{A_h, f_h, \alpha_0}(\mathbf{u}_h) = 0$$

with $\mathbf{u}_h$ a discretization of $u(x)$.

Difficulties

- The more the equation is non-linear, the more the Newton convergence is difficult.
- The more $A(x,y)$ is anisotropic and $\alpha_0 \gg 1$, the more the condition number is large and the convergence (linear/nonlinear) is hard.

Initial guess and FNO

Convergence of Newton's method

The convergence depends on the initial guess. If the initial guess is too far from the solution, Newton's method converges slowly, or even does not converge.

Idea

Train a Fourier Neural Operator (FNO) to approximate the solution of the elliptic equation and use it as an initial guess.

- We keep the convergence properties of the scheme, and we hope to accelerate Newton's method.
- Algorithm:
  1. Fix a mesh, fix α_0 and $k(\cdot)$,
  2. Randomly generate many data u_h^i, A_h^i (random Fourier coefficients, sum of random Gaussian functions),
  3. Compute the right-hand side associated with f_h^i,
  4. Train the neural network $G_\theta^+(A_h^i, f_h^i)$, by minimizing

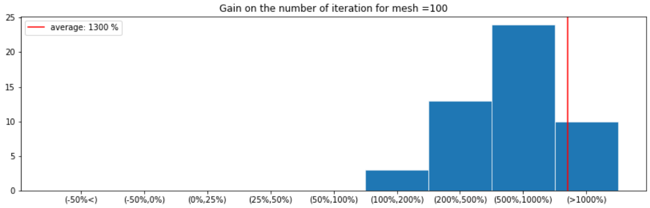
$$J(\theta) = \omega \sum_{i=1}^{n} \| G_\theta^+(A_h^i, f_h^i) - u_h^i \| + (1-\omega) \sum_{i=1}^{n} \| G_{A_h^i, f_h^i, \alpha_0}(G_\theta^+(A_h^i, f_h^i)) \|.$$

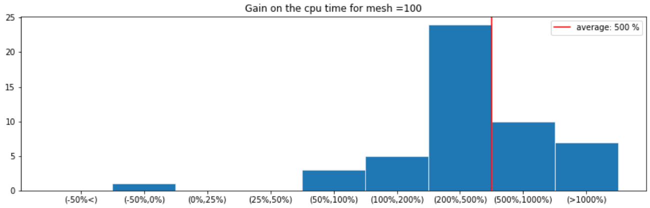
Results I

- We consider the equation

$$u(x) - \partial_x(2\alpha(x)u^4\partial u) = f$$

- We learn only with the data loss and the residue loss.
- Ratio number of iterations (top) and CPU time (bottom) for classical/NN on 100 cells



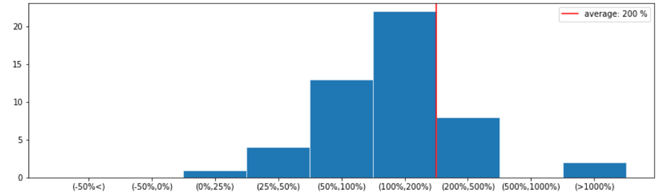


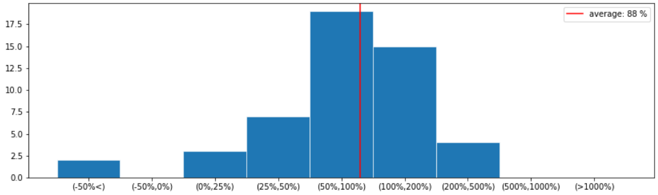
Results I

- We consider the equation

$$u(x) - \partial_x(2\alpha(x)u^4\partial u) = f$$

- We learn only with the data loss and the residue loss.
- Ratio number of iterations (top) and CPU time (bottom) for classical/NN on 200 cells



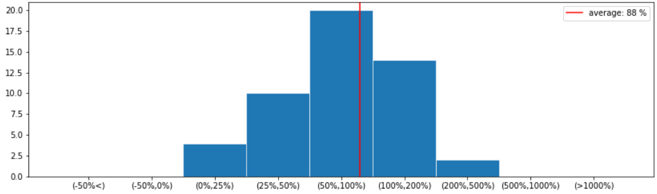


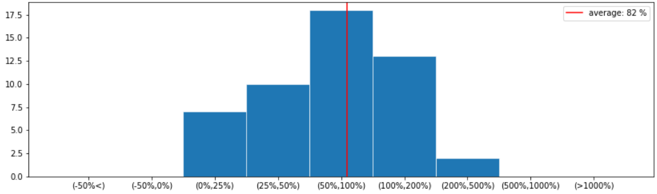
Results I

- We consider the equation

$$u(x) - \partial_x(2\alpha(x)u^4\partial u) = f$$

- We learn only with the data loss and the residue loss.
- Ratio number of iterations (top) and CPU time (bottom) for classical/NN on 400 cells



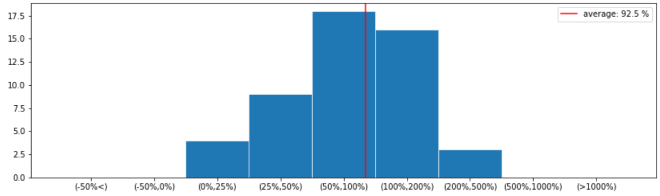


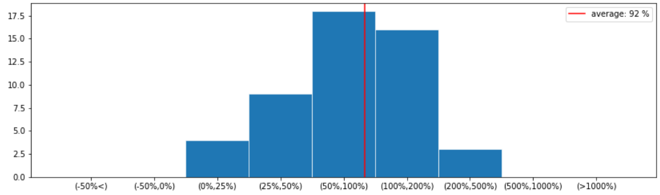
Results I

- We consider the equation

$$u(x) - \partial_x(2\alpha(x)u^4\partial u) = f$$

- We learn only with the data loss and the residue loss.
- Ratio (nb iter Newton/nb iter Newton +NN) on 600 cells





Results II

- What happens when we increase α_0 to get a stronger non-linearity?
- We only compare the average results:

mesh	$\alpha_0 = 2$ (40 sim)	$\alpha_0 = 5$ (25 sim)	$\alpha_0 = 8$ (25 sim)
100 cells	+500%	+1800%	+5000%
200 cells	+88%	+230%	+620%
400 cells	+82%	+150%	+220%
600 cells	+92%	+220%	+250%

Table: Comparison of the mean "gain" for different values of α_0.

- **Fails**: on all the tests, we have 0% of fail (our method being less efficient than the classical one) for the iteration criterion, and around 2% of fail on the CPU time criterion.
- On more refined meshes, the gain is smaller (the network acts only at the beginning of the convergence).
- More the system is nonlinear more the method is efficient.

Differentiable physics

Principle I

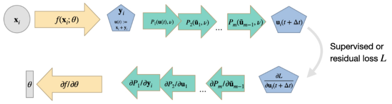
- At the beginning, PINNs/neural operators are used to solve PDEs.
- **Other approach**: supervised learning, Hesthaven et al 2017, 2018, 2020, 2022, B. Desprès and H. Joudren 2020, R. Loubère et al 2020, etc.
- Differentiable physics, Michael P. Brenner et al 2018, 2020, "Physics-based deep learning", Nils Thuerey et al, 2021.

Coming back to neural networks

- The neural networks are trained with stochastic gradient descent.
- How is the gradient computed? Back-propagation.

Back-propagation and automatic differentiation

- Function: $f_\theta(x) = f_1 \circ .... \circ f_n$
- Automatic differentiation methods are able to deal with deep function composition.



$\mathbf{x}_i$
$f(\mathbf{x}_i;\theta)$
$\mathbf{y}_i$
$\mathbf{u}(t) := \mathbf{x}_i + \mathbf{y}_i$
$P_1(\mathbf{u}(t),\nu)$
$P_2(\tilde{\mathbf{u}}_1,\nu)$
...
$P_m(\tilde{\mathbf{u}}_{m-1},\nu)$
$\mathbf{u}_i(t+\Delta t)$
Supervised or residual loss L
$\frac{\partial L}{\partial \mathbf{u}_i(t+\Delta t)}$
$\partial P_m/\partial \tilde{\mathbf{u}}_{m-1}$
...
$\partial P_2/\partial \mathbf{u}_1$
$\partial P_1/\partial \mathbf{y}_i$
$\partial f/\partial \theta$
θ

Principle II

Differentiable physics

Write the scheme such that we can apply automatic differentiation and back-propagation to compute the gradient of each function of the scheme in the code, and each composition of these functions.

- Using that, we can compute the gradient with respect to all inputs of the solver, or of sub-parts of the solver.
- **Consequences**: We can put a NN anywhere in the solver and optimize it with respect to a criterion on the simulation result.

Link with optimal control

In optimal control we compute the gradient of the loss with respect to the input with an adjoint method. It is another use of automatic differentiation methods.

Drawback

With back-propagation, stability problems (vanishing or exploding gradient) can arise when composing too many functions.

Differentiable physics learning

Summary of the main goal

Here, the neural network approximates one subpart of the global map between data/parameters and solution. This subpart can be: the full operator, a sub-function, just one or two parameters, . . .

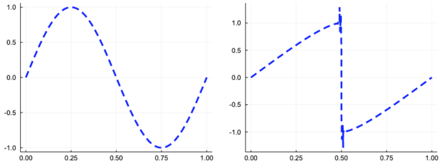
Application of differentiable physics

General problem

- We want to solve general hyperbolic PDEs:

$$\partial_t \mathbf{U} + \partial_x \mathbf{F}(\mathbf{U}) = 0$$

- High order method (MUSCL, HO finite volumes or DG) generate oscillations around areas with strong gradients or shock waves: Gibbs phenomenon.
- Example on the Burgers equation:



- **Solutions**: slope limiting, artificial viscosity, filtering, etc.

Goal

Design slope limiting for MUSCL or artificial viscosity for DG using neural networks.

Artificial viscosity problem for DG

- We have a DG scheme, written under the form

$$\partial_t^{rk}\mathbf{U}_h + \partial_x^{DG}\mathbf{F}(\mathbf{U}_h) = 0.$$

- The idea of the artificial viscosity method is to add a diffusion operator, which acts on the oscillations.
- Modified scheme:

$$\partial_t^{rk}\mathbf{U}_h + \partial_x^{DG}\mathbf{F}(\mathbf{U}_h) = \partial_x^{DG}(\mathbf{D}(\mathbf{U}_h)\partial_x^{DG}\mathbb{U}_h).$$

- **How to construct D?**
- Derivative-based approach:

$$D(\mathbf{U}_h) = \lambda_{max} h|\partial_x^{DG}\mathbf{U}_h)|$$

- MDH approach: we reconstruct the modes within the cells, and apply viscosity to decrease the highest modes.
- Other approaches: MDA, entropy-based, etc.
- How to use neural networks? Approach from J. Hesthaven: compute the best viscosity on many test cases, and learn this viscosity with a NN.
- The NN interpolates between known viscosities.
  - There is no new viscosity model,
  - and we cannot use this method to tune a scheme where we do not have a prior viscosity model.

Differentiable physics approach I

Tool

We propose to use differentiable physics (control optimal approach) to design new types of viscosity model.

- Formalism of optimal control and RL.
- We define a NN $D_\theta(\mathbf{U}_h(t))$ with $\mathbf{U}_h(t)$ the discrete solution.
- We define a value function:

$$V_\theta^T(\mathbf{U}_0) = \int_0^T C(\mathbf{U}_h(t))dt,$$

with C a cost function and $\mathbf{U}_0 = \mathbf{U}_h(0)$ an initial condition.

Goal

Our objective to find a solution of the minimization problem:

$$\min_\theta \int_{U_0} V_\theta(\mathbf{U}_0)dp(\mathbf{U}_0)d\mathbf{U}_0 \qquad (1)$$

with $p(\mathbf{U}_0)$ a probability law of initial data on $\mathbf{U}_0$.

Differentiable physics approach II

- After Monte-Carlo discretization, we obtain the minimization problem:

$$\min_\theta J(\theta) = \min_\theta \sum_{i=1}^{n_{\text{data}}} V_\theta^T(\mathbf{U}_{i,0}).$$

- We provide an approximation in time of the value function:

$$V_\theta^T(\mathbf{U}_0) = \Delta t \sum_{t=1}^{T} C(\mathbf{U}_h^t)$$

- The transition between two time steps is given by $\mathbf{U}_h^{n+1} = S_h(\mathbf{U}_h^n, D_\theta(\mathbf{U}_h^n))$ with our scheme. As a consequence, we have:

$$V_\theta^T(\mathbf{U}_0) = C(\mathbf{U}_0) + C(S_h(\mathbf{U}_0, D_\theta(\mathbf{U}_0))) + C(S_h(S_h(\mathbf{U}_0, D_\theta(\mathbf{U}_0)), D_\theta(S_h(\mathbf{U}_0, D_\theta(\mathbf{U}_0))))) + \dots,$$

- As previously mentioned in the paradigm of differential physics, we can compute by automatic differentiation:

$$\nabla_\theta V_\theta^T(\mathbf{U}_0)$$

- We solve the minimization problem on $J(\theta)$ using a gradient method, with

$$\nabla_\theta J(\theta) = \sum_{i=1}^{m} \nabla_\theta V_\theta^T(\mathbf{U}_{i,0})$$

Differentiable physics approach III

- To complete the algorithm, the NN and loss function still have to be defined.

Neural network

A ResNet convolution neural network (without coarsening operator) with q channels (polynomial order q); once trained, it can be used on arbitrary uniform grids, by sliding the convolution window.

Loss function

The cost function $C()$ is composed of three parts:

- L^2 error compared to a MUSCL solution on a fine grid:

$$C_{\text{error}}(\mathbf{U}_h^n) = h_{FV}\sum_{i=1}^{n} \|\Pi_{FV}(\mathbf{U}_h^n)_i - \mathbf{U}_{i,ref}\|_2^2\,,$$

- L^1 error on the Laplacian compared to the Laplacian of the reference solution

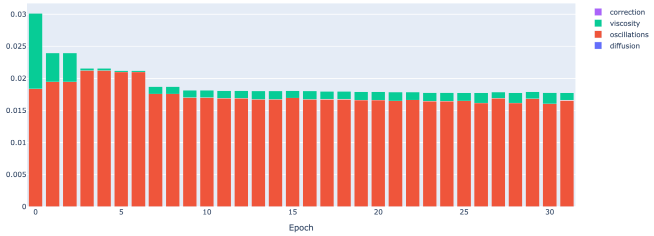
$$C_{\text{osc}}(\mathbf{U}_h^n) = h_{fv}\sum_{i=1}^{n} \left\| D_{xx}^{fv}(\Pi_{fv}(\mathbf{U}_h^n))_i - D_{xx}^{fv}\mathbf{U}_{j,ref}\right\|_1 .$$

- L^2 norm of D_θ:

$$C_{\text{vis}}(\mathbf{U}_h^n) = \|D_\theta(\mathbf{U}_h^n)\|_2^2 .$$

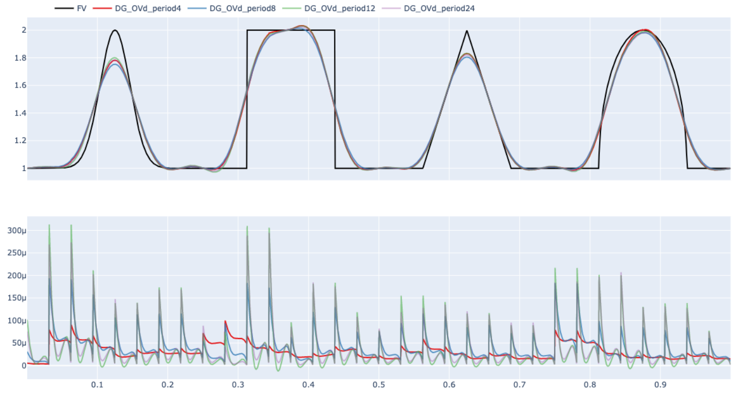
Results I

- We consider two losses: C_{vis} and C_{osc}.
- Loss evolution:



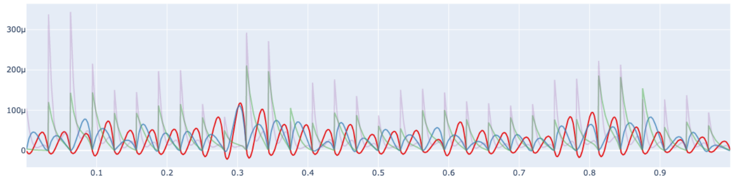
Results I

- We consider two losses: C_{vis} and C_{osc}.
- Test case after some training epochs:



Results II

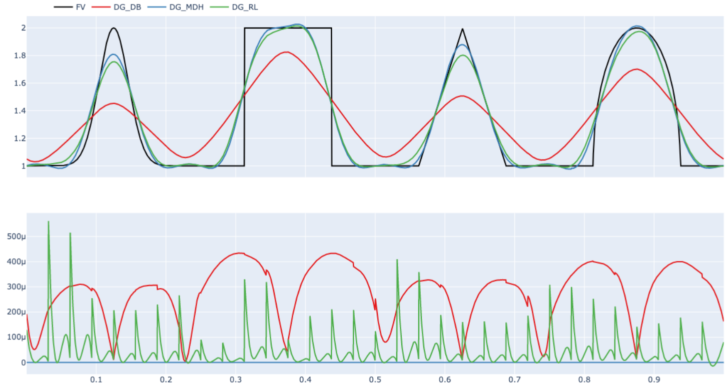
- We observe the effect of the parameters m (number of time iteration in the gradient) on the viscosity:



- If m is too small the effect of viscosity is not visible and the model does not learn to modify the viscosity.

Results II

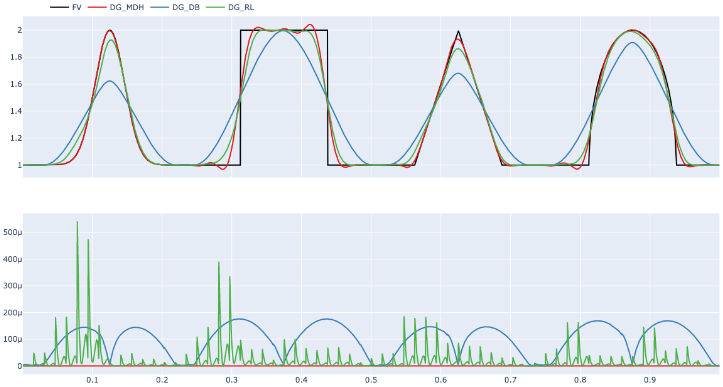
- We compare the solution on 32 cells (training on 32)



- Results depend of the balance between the two losses.

Results II

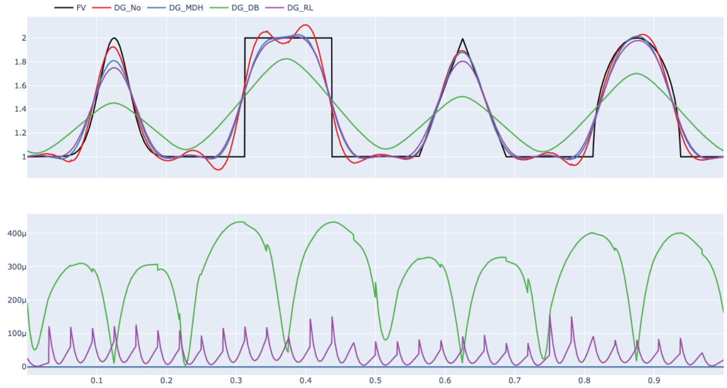
- We compare the solution on 64 cells (training on 32)



- Results depend of the balance between the two losses.

Results III

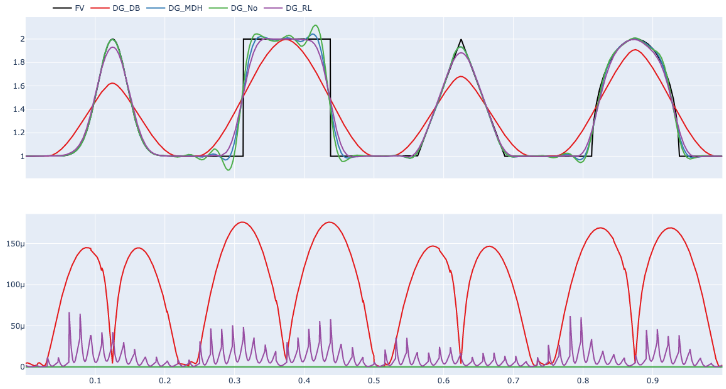
- We consider two loss: C_{error} and C_{osc}. We consider the grids: 32 cells, 64 cells and 128 cells.



- 32 cells

Results III

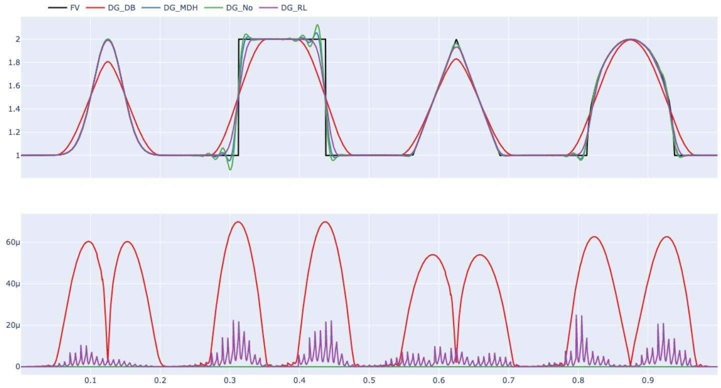
- We consider two loss: C_{error} and C_{osc}. We consider the grids: 32 cells, 64 cells and 128 cells.



- 64 cells

Results III

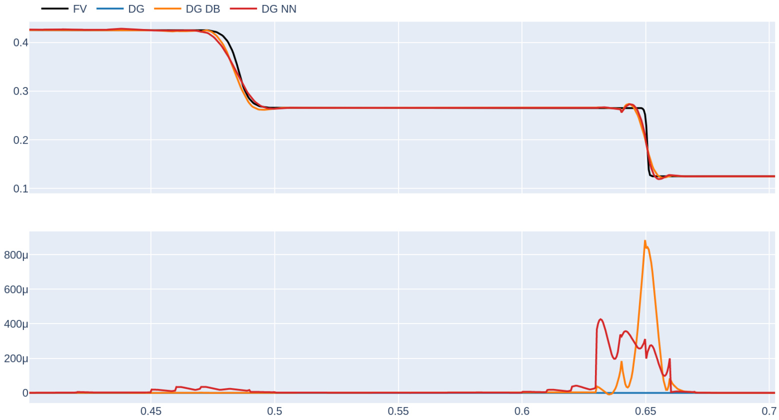
- We consider two loss: C_{error} and C_{osc}. We consider the grids: 32 cells, 64 cells and 128 cells.



- 128 cells

Results IV

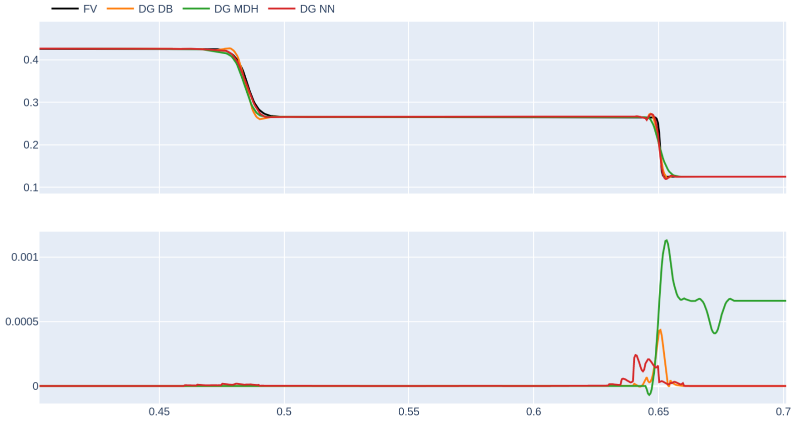
- Euler equations, viscosity model: $D_\theta(\rho, U, E)$ on the three equations.
- Sod problem (zoom on contact and shock wave):



- 100 cells

Results IV

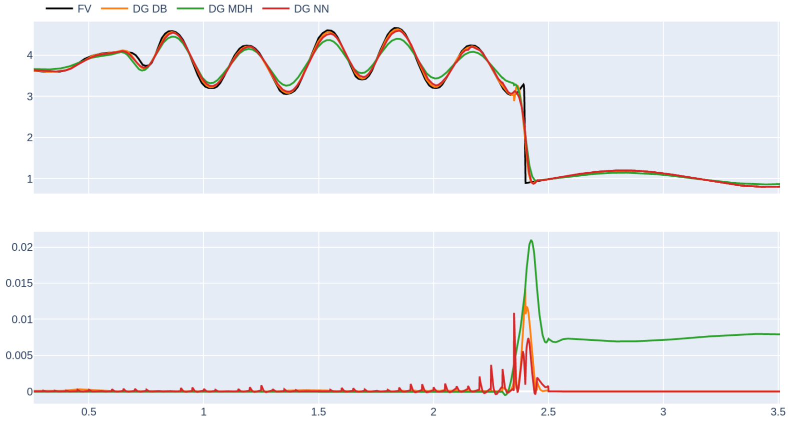
- Euler equations, viscosity model: $D_\theta(\rho, U, E)$ on the three equations.
- Sod problem (zoom on contact and shock wave):



- 200 cells

Results IV

- Euler equations, viscosity model: $D_\theta(\rho, U, E)$ on the three equations.
- Shu-Osher problem:



- 200 cells

Conclusion

Conclusion

Physics-Informed neural networks

The PINNs/neural operator approach allows to approximatively solve a family of problems. This framework is interesting to improve numerical methods.

Next steps

- Finish implementing Newton in 2D and apply this to more complex, time-dependent problems like the reduced MHD. Go towards self-optimizing code?
- Use PINNs/Neural operator to improve spatial numerical methods like FE or DG.

Differential physics

The DF approach allows optimizing some part of a numerical solver using NNs and training with complex losses, depending on the output of the solver. A supervised part can be added.

Next steps

- Finish implementing the viscosity for the Euler equations,
- Find better loss to detect the oscillations, find a loss which does not require a fine solution,
- Use DF to improve spatial numerical methods like FE or DG.

Bonus: Unstructured meshes

Graph neural networks

CNN and signal processus

Convolutional neural networks are very useful to analyze pictures and detect patterns (segmentation, . . .). For PDEs, they can be useful for discontinuity or front tracking.

- How to use them on unstructured meshes?
- CNN have been made for pictures, and for regular grids.

GNN

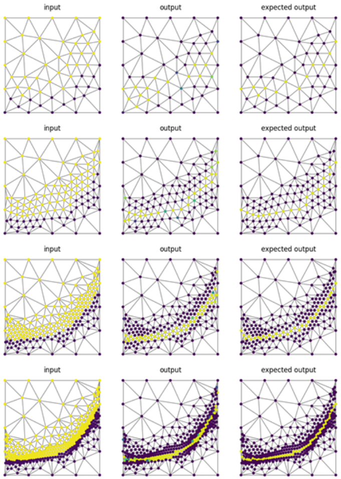
In the last five years, many Graph convolutional neural network have been proposed and can be used on general meshes.

Important

Choose the network type such that the performance is not impacted when changing the mesh but not the topology.

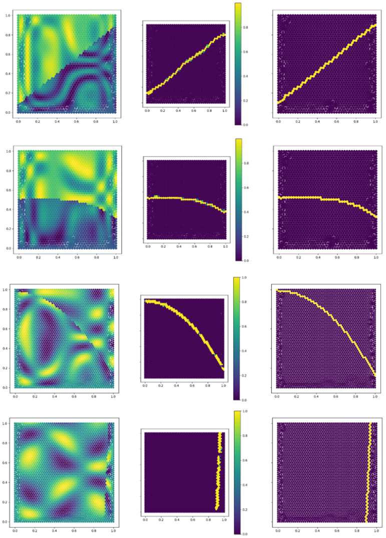
Discontinuity Tracking I

- **A first application**: discontinuity/shock detection and mesh refinement.
- **Case 1**: constant by part function. Localization of discontinuity by GCN and iterative refinement.



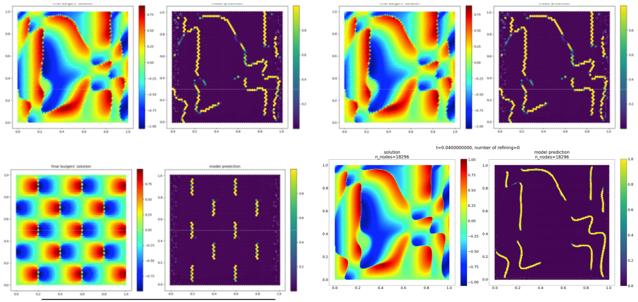
Discontinuity Tracking II

- **Case 2**: Discontinuity detection for non piecewise constant function. Unet architecture with Chebnet and geometric pooling layers.
- Training on a single mesh (for computational reasons)

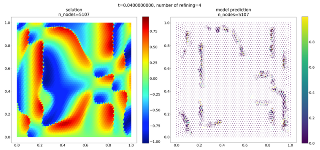


Discontinuity Tracking III

- **Case 3**: Discontinuity detection in Burgers simulation using previous training



- A first refinement approach



- After refining the mesh, the discontinuity remains detected. This effect dampens after four refinements, possibly due to training on a single mesh.