Physics-informed neural networks: principles, limitations and extensions

E. Franck[1,2], V. Michel Dansac[1,2]

Journée IA pour la découverte scientifique

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

Outline

Introduction

Physics-informed Neural Networks

Operator learning

Application to numerical methods

Conclusion

Introduction to Numerical methods

Numerical methods

- We begin with a simple example:

$$\begin{cases} L_{t,x}u = \partial_t u - \Delta u = 0 \\ u(t=0,x) = u_0(x) \\ u(x) = g \text{ on } \partial\Omega \end{cases}$$

- Solving a PDE amounts to solving a infinite-dimensional problem.

Numerical methods

Reduce this problem to a finite-dimensional algebraic or optimization problem.

How?

We consider a known parametric function $f_\theta(x)$ with unknown parameters θ. Plugging this model into the equation (in some way), we obtain an equation like:

$$G(\theta) = 0, \quad \text{or} \quad \min_\theta F(\theta)$$

Approaches

- Mesh-based methods
- Spectral methods
- Mesh-free methods
- Deep model-based methods

Numerical methods on a mesh

- We begin with a simple example:

$$\begin{cases} L_{t,x}u = \partial_t u - \Delta u = 0 \\ u(t=0,x) = u_0(x) \\ u(x) = g \text{ on } \partial\Omega \end{cases}$$

- Solving a PDE amounts to solving a infinite-dimensional problem.

Polynomial Lagrange interpolation

We consider a domain $[a,b]$, $\exists P(x)$ a polynomial of degree k such that, for any $f \in C^0([a,b])$, we have

$$|f(x) - P(x)| \leq |b-a|^k \max_{x\in[a,b]} |f^{k+1}(x)|.$$

- On small domains ($|b-a| \ll 1$), this polynomial gives a very good approximation.

First step: choose a parametric function

We take a mesh splitting the geometry in small sub-intervals $[x_k, x_{k+1}]$, and propose the following candidate

$$u_{|[x_k,x_{k+1}]}(t,x) = \sum_{j=1}^{q} \alpha_j(t)\phi_j(x).$$

This is a piecewise polynomial representation.

Numerical methods on a mesh II

Second step: find the parameter equations

Plugging the representation in the equation, we have:

$$\sum_{j=1}^{q} \partial_t \alpha_j(t)\phi_j(x) - \sum_{j=1}^{q} \alpha_j(t)\partial_{xx}\phi_j(x) = 0$$

There are not enough equations, so we multiply by $\phi_i(x)$ and integrate, to obtain

$$\sum_{j=1}^{q} \partial_t \alpha_j(t) \int_{[x_k,x_{k+1}]} \phi_j(x)\phi_i(x) - \sum_{j=1}^{q} \alpha_j(t) \int_{[x_k,x_{k+1}]} \partial_{xx}\phi_j(x)\phi_i(x) = 0.$$

After integrating by parts, we obtain algebraic relation after some manipulations:

$$\frac{d}{dt}\boldsymbol{\alpha}(t) = K\boldsymbol{\alpha}(t).$$

with $\boldsymbol{\alpha}(t)$ the coefficients of all the cells.

- No continuous polynomial representation between cells: Discontinuous Galerkin method, finite volume methods.
- Continuous polynomial representation between cells: Finite element methods.
- Possibility to use non-polynomial bases (Trefftz methods for example).

Numerical methods on a mesh III

Space-time approach

We take a mesh splitting the space-time geometry in small subsets $\Omega_k^n = [x_k, x_{k+1}]x[t_n, t_{n+1}]$ and propose the following candidate

$$u_{|\Omega_k^n}(t,x) = \sum_{j=1}^{q} \alpha_j \phi_j(t,x)$$

We use a piecewise polynomial representation. Plugging this in the equation, we obtain:

$$A\alpha = b$$

with α the coefficients on all the time-space mesh.

Properties of mesh-based methods

- Use local polynomial/non-polynomial representations.
- Unknown parameters (or degrees of freedom) are associated with localized values on the mesh (average in the cell, point value at the nodes, etc).
- Convergence results in varying meshes.
- The parameters are determined thanks to an algebraic relation.
- Mesh dependency is complicated for complex geometry and high-dimensional problems.

Spectral Numerical methods

Hilbert space basis

Functional Hilbert space, such as L^2, admit countable bases. So, for $u \in L^2$, there exists

$$u(t,x) = \sum_{i=1}^{+\infty} \alpha_i(t)\phi_i(x), \quad \text{or} \quad u(t,x) = \sum_{i=1}^{+\infty} \alpha_i \phi_i(t,x)$$

with ϕ_i nonlocal functions (Fourier modes, Legendre polynomial, etc) and $\alpha_i = \int_\Omega u\phi_i$.

Spectral methods

We plug these (truncated) representations in the equation, and perform similar computations as for DG/FE methods to obtain algebraic relation.

Properties of spectral method

- Use a global polynomial/non-polynomial representation.
- Unknown parameters are associated with global values on the domain.
- Convergence results with the number of basis functions.
- The parameters are determined thanks to an algebraic relation.
- In practice, boundary conditions make mesh dependency comes back. It remains complicated for complex geometries and high-dimensional problems.

Mesh-free methods

- We introduce another type of global linear representation:

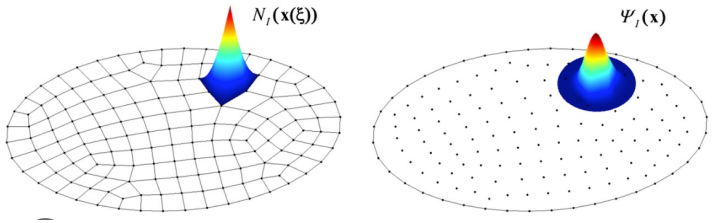
$$u(t,x) = \sum_{i=1}^{N} \alpha_i(t)\phi_i(|x - x_i|), \quad u(t,x) = \sum_{i=1}^{N} \alpha_i \phi_i(|(t,x) - (t_n, x_i)|)$$

  with $\phi_i(r)$ radial basis functions, such as Gaussian functions.

- Here, we again introduce this representation into the equation at some point x_j to obtain algebraic relations.

Example:

$$-\Delta u = f \rightarrow -\sum_{i=1}^{N} \alpha_i \partial_{xx}\phi(|x_j - x_i|) = f(x_j)$$



Properties of mesh-free based method

- Use a global non-polynomial representation.
- Slow convergence results with the number of points.
- The parameters are determined thanks to an algebraic relation (involving a dense matrix).
- No mesh dependency, but limited use for high-dimensional problems.

Physics-informed Neural Networks

Deep learning: neural networks

- Current choice: kernel approximation or neural network.

Layer

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

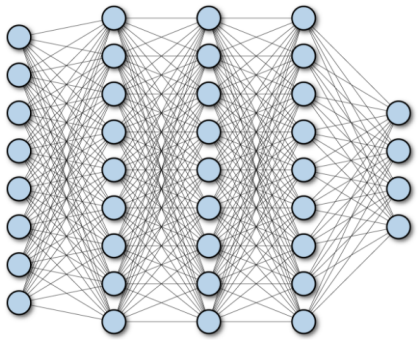
Neural network

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

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

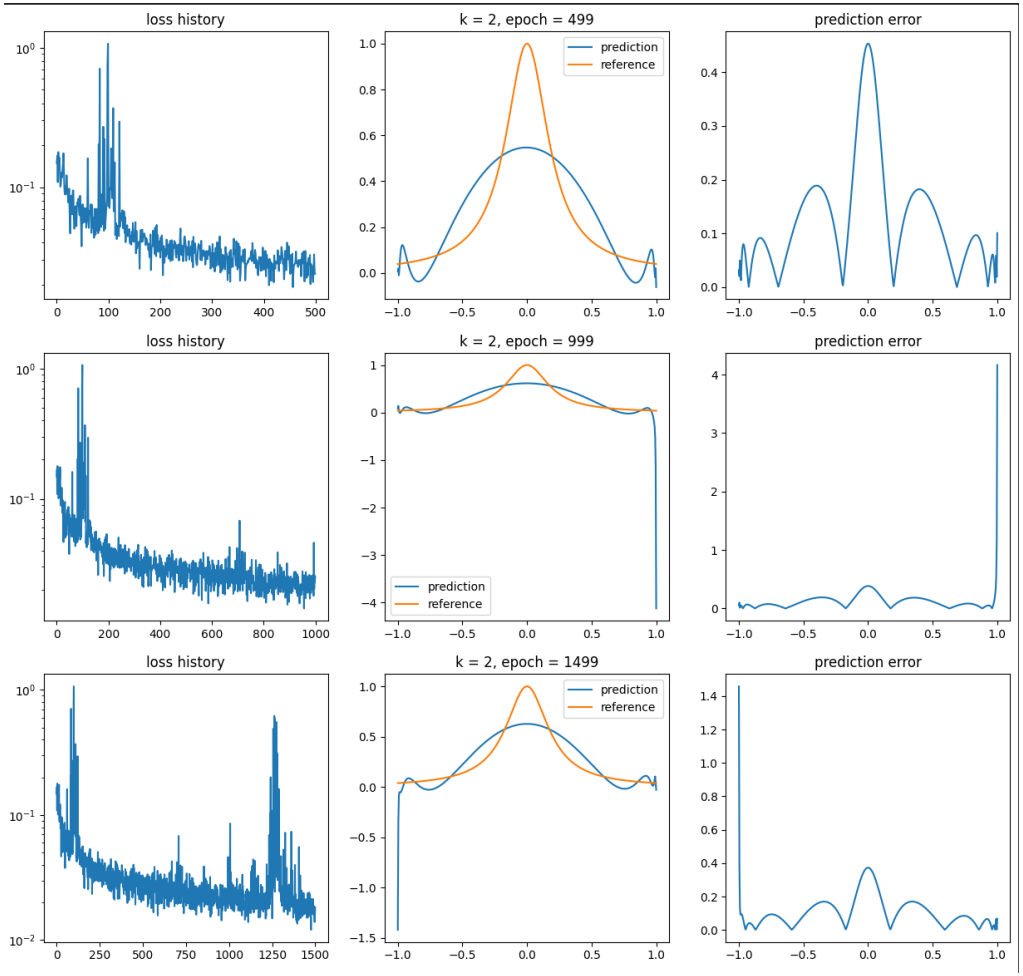
with θ the trainable parameters composed of all the matrices $A_{l,l+1}$ and biases $\mathbf{b}_l$.

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



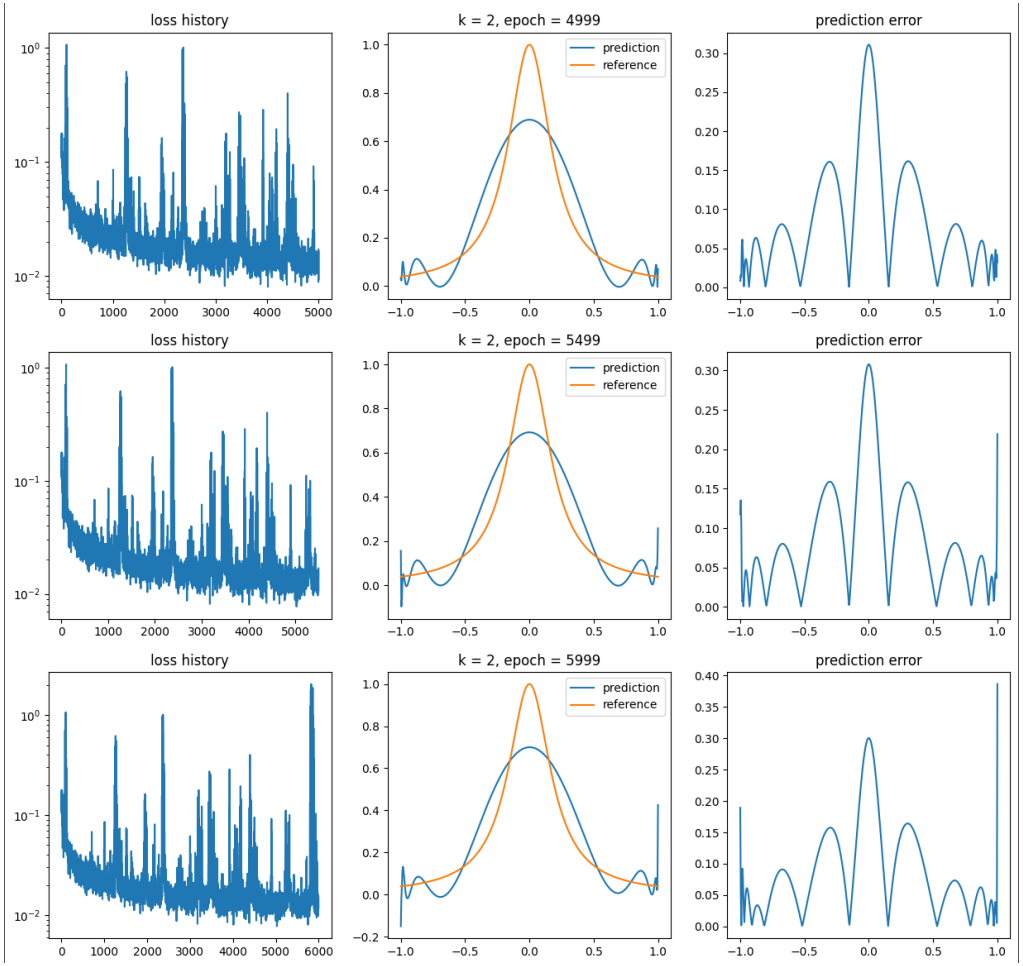
NN vs Polynomial

- We will compare over-parametrized NN and polynomial regression on the Runge function.
- 120 data and approximately 800 parameters in each model.



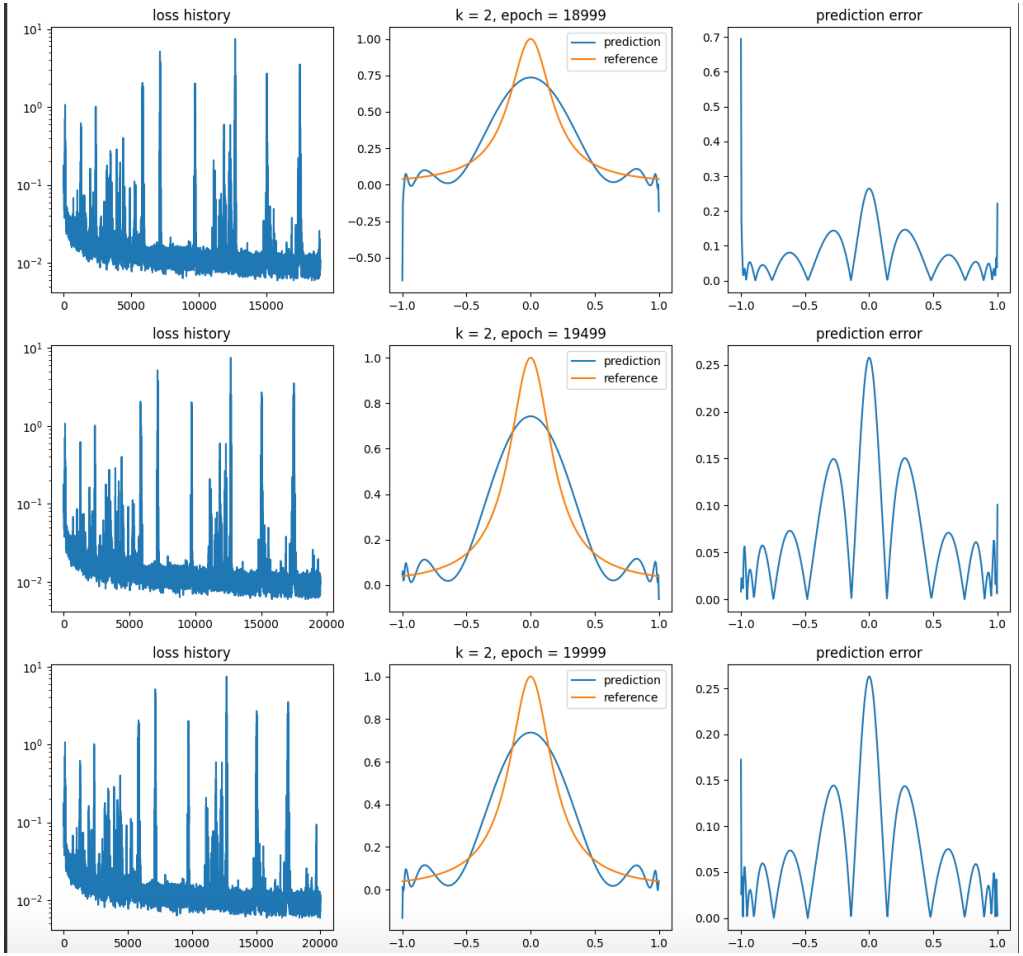
NN vs Polynomial

- We will compare over-parametrized NN and polynomial regression on the Runge function.
- 120 data and approximately 800 parameters in each model.



NN vs Polynomial

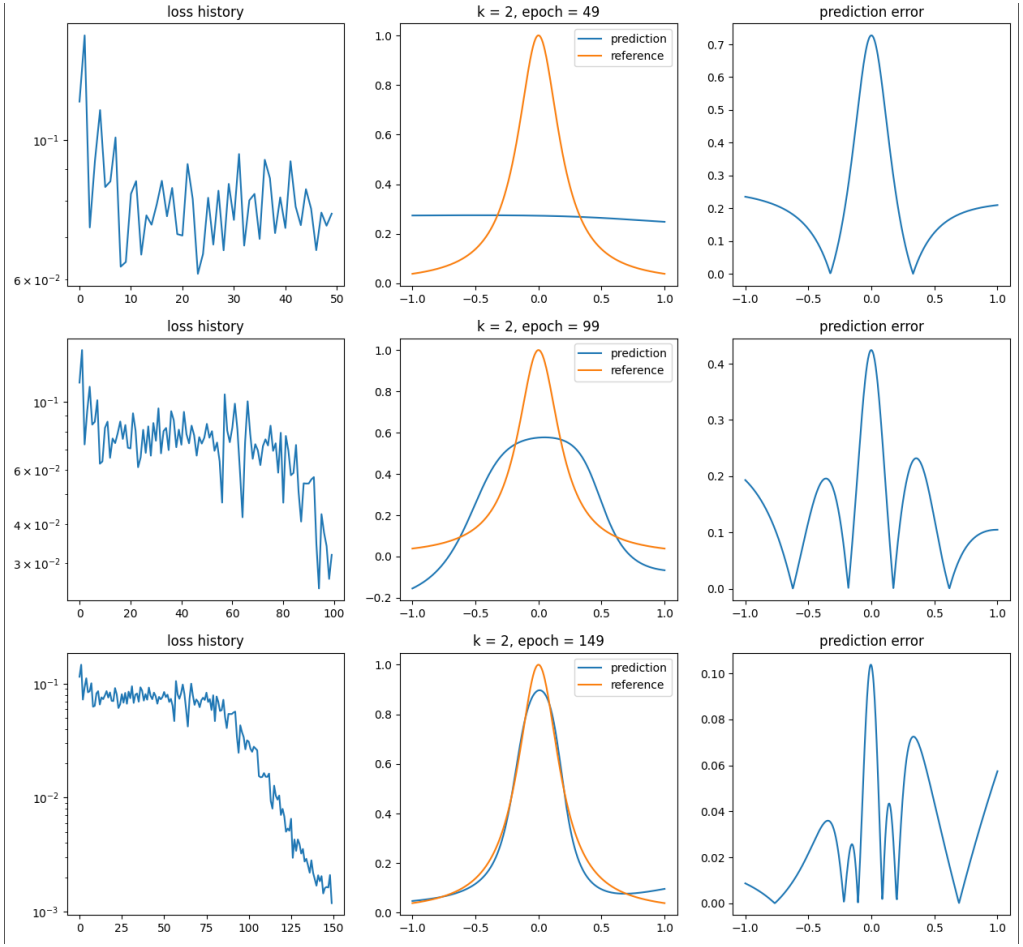
- We will compare over-parametrized NN and polynomial regression on the Runge function.
- 120 data and approximately 800 parameters in each model.



- The polynomial model tends to oscillate in the over parameterized regime. Problematic for overfitting.

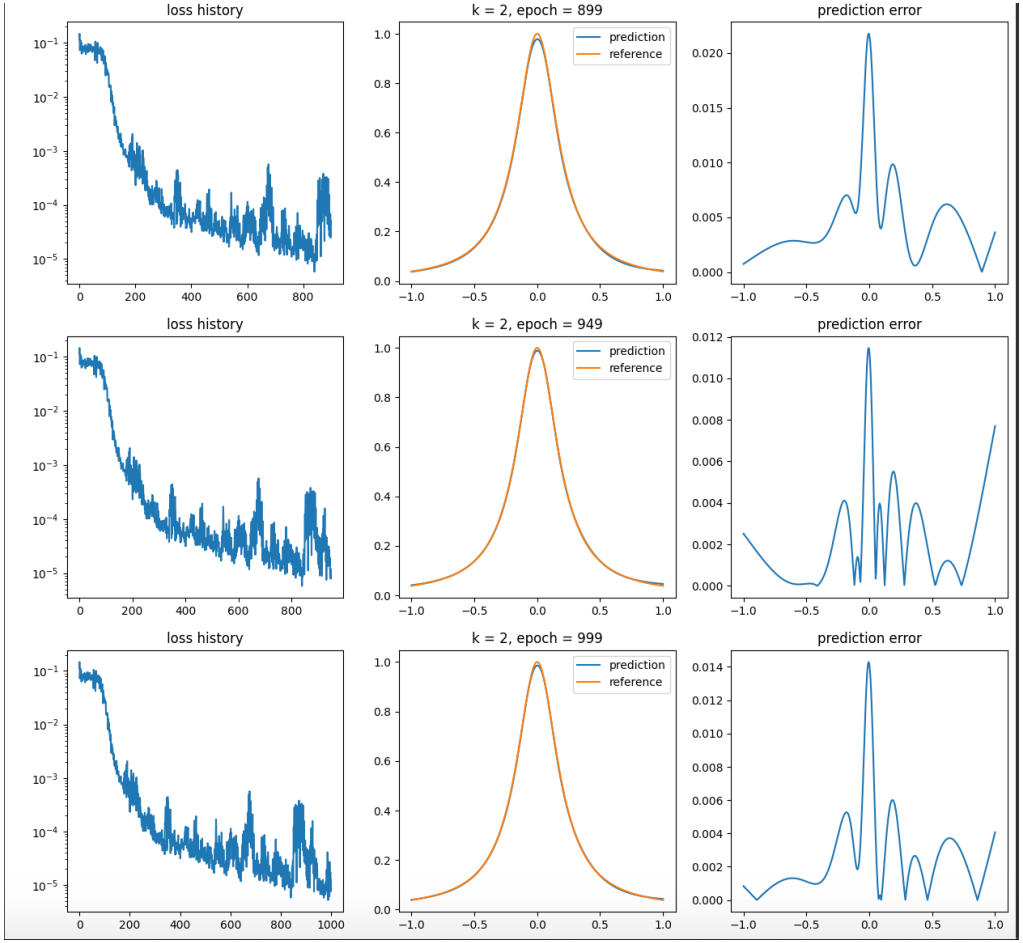
NN vs Polynomial

- We will compare over-parametrized NN and polynomial regression on the Runge function.
- 120 data and approximately 800 parameters in each model.



NN vs Polynomial

- We will compare over-parametrized NN and polynomial regression on the Runge function.
- 120 data and approximately 800 parameters in each model.



- The ANN generates very smooth/low frequency approximations.
- It is related to the spectral bias. The low frequencies are learned before the high frequencies. Seems very helpful for the generalization

Key idea of the PINNS/Neural Galerkin methods

Summary

Every previously mentioned space and space-time methods consists in:

1. choosing a linear representation (linear combination of basis functions), either local (on a mesh) or global;
2. pluging this representation into the equation to obtain algebraic relations (linear for linear problems, nonlinear for nonlinear problems)
3. solving this algebraic relation with a linear solver or Newton's method.

In general, the algebraic relation has a unique solution.

Idea

Choose a nonlinear representation given by a neural network. We replace a sum of simple functions with a composition of simple functions.

Important point

When we use linear representation, we restrict our PDE approximation to convex finite-dimensional vector subspaces, to ensure uniqueness of the solution in this discrete space. This is no longer the case with neural networks: we do not obtain an algebraic problem with a unique solution.

Spatial approach: Neural Galerkin I

- We solve the following PDE:

$$\partial_t u = \mathcal{F}(u, \nabla u, \Delta u) = F(u).$$

- Classical representation: $u(t,x) = \sum_{i=1}^{N} \theta_i(t)\phi_i(x)$
- Deep representation: $u(t,x) = u_{nn}(x;\theta(t))$ with u_{nn} a neural network, with parameters $\theta(t)$, taking x as input.
- We want that:

$$F(u_{nn}(x;\theta(t))) = \partial_t u_{nn}(x;\boldsymbol{\theta(t)}) = \left\langle \nabla_\theta u_{nn}(x;\boldsymbol{\theta}), \frac{d\theta(t)}{dt} \right\rangle$$

- How to find an equation for $\frac{d\theta(t)}{dt}$?
- We solve the minimization problem:

$$\min_{\boldsymbol{\eta}} J(\boldsymbol{\eta}) = \min_{\boldsymbol{\eta}} \int_\Omega |\langle \nabla_\theta u_{nn}(x;\theta), \boldsymbol{\eta}\rangle - F(u_{nn}(x;\theta(t)))|^2 dx.$$

- The solution is given by

$$\boxed{M(\theta(t))\frac{d\theta(t)}{dt} = F(x,\theta(t))}$$

with

$$M(\theta(t)) = \int_\Omega \nabla_\theta u_{nn}(x;\theta) \otimes \nabla_\theta u_{nn}(x;\theta) dx, \quad F(x,\theta(t)) = \int_\Omega \nabla_\theta u_{nn}(x;\theta) F(u_{nn}(x;\theta)) dx.$$

Spatial approach: Neural Galerkin II

- How to estimate $M(\theta(t))$ and $F(x, \theta(t))$?
- **Firstly**: we need to differentiate the network with respect to θ and to x (in the function F). This can easily be done with automatic differentiation.
- **Secondly**: How to compute the integrals? With a Monte Carlo approach.
  - The Monte-Carlo method stems from the **Law of large numbers**.
  - We consider a function $g : \mathbb{R}^d \to \mathbb{R}$. We define X a random variable with the law μ.
  - The method comes from:

$$\frac{Var(\mu)}{\sqrt{N}} \left(\frac{1}{N} \sum_{i=1}^{N} f(X_i) - \mathbb{E}_\mu[f(X)] \right) \to \mathcal{N}(0, 1)$$

    with X_i an random example sampled with the law μ
  - It makes it possible to compute integrals. Indeed:

$$\int_\Omega f(x)dx = \int_{\mathbb{R}^d} f(x)\mathcal{U}_\Omega dx = \mathbb{E}[f(X)]$$

    with $\mathcal{U}_\Omega$ the density of the uniform law Ω and X random variable following this law.
- So, we use: $M(\theta(t)) \approx \sum_{i=1}^{N} \nabla_\theta u_{nn}(x_i; \theta) \otimes \nabla_\theta u_{nn}(x_i; \theta)$, and the same for $F(x, \theta(t))$.
- **Summary**: we obtain an ODE in time (as usual) and mesh-less method in space.

Space-time approach: PINN's I

Idea of PINNs

- For u in some function space $\mathcal{H}$, we wish to solve the following PDE:

$$\partial_t u = \mathcal{F}(u, \nabla u, \Delta u) = F(u).$$

- Classical representation for space-time approach: $u(t,x) = \sum_{i=1}^{N} \theta_i \phi_i(x,t)$
- Deep representation: $u(t,x) = u_{nn}(x,t;\theta)$ with u_{nn} a neural network of parameters θ.

- Since ANN are C^p functions, we can compute $\partial_t u_{nn}(x,t;\theta)$, $\partial_{x^p} u_{nn}(x,t;\theta)$ and

$$r(x,t) = \partial_t u_{nn}(x,t;\theta) - \mathcal{F}(u_{nn}(x,t;\theta), \nabla u_{nn}(x,t;\theta), \Delta u_{nn}(x,t;\theta))$$

- First idea: we solve the nonlinear problem

$$r(x_i,t_n) = 0, \quad \forall 1 \leq j \leq N_x, \quad \forall 1 \leq n \leq N_t$$

  with $N_t * N_x$ equal to the number of parameters.
- **Problem**: There is the existence of the PDE solution in $\mathcal{H}$, but the subspace of NN functions is not a convex vector space. We cannot be ensured of the existence of the solution to the discrete problem.

Conclusion

We move away from algebraic equations on the parameters, and go towards non-convex optmization.

Space-time approach: PINN's II

- We define the residual of the PDE:

$$R(t,x) = \partial_t u_{nn}(t,x;\theta) - \mathcal{F}(u_{nn}(t,x;\theta), \partial_x u_{nn}(t,x;\theta), \partial_{xx} u_{nn}(t,x;\theta))$$

- To learn $u_{nn}(t,x;\theta)$, we minimize:

$$\min_\theta \left(J_r(\theta) + J_b(\theta) + J_i(\theta)\right),$$

with

$$J_r(\theta) = \int_0^T \int_\Omega |R(t,x)|^2 dx dt$$

and

$$J_b(\theta) = \int_0^T \int_{\partial\Omega} \|u_{nn}(t,x;\theta) - g(x)\|_2^2 dx dt, \quad J_i(\theta) = \int_\Omega \|u_{nn}(0,x;\theta) - u_0(x)\|_2^2 dx.$$

- To avoid an extra loss for the BC and initial conditions, we use:

$$\bar{u}_\theta(t,x) = u_0(x) + t(\phi(x) * u_\theta(x)),$$

with $\phi(x) = g(x)$ on the boundary and something inside.

Space-time approach: PINN's II

- We define the residual of the PDE:

$$R(t,x) = \partial_t u_{nn}(t,x;\theta) - \mathcal{F}(u_{nn}(t,x;\theta), \partial_x u_{nn}(t,x;\theta), \partial_{xx} u_{nn}(t,x;\theta))$$

To learn $u_{nn}(t,x;\theta)$, we minimize:

$$\min_\theta \left(J_r(\theta) + J_b(\theta) + J_i(\theta)\right),$$

with

$$J_r(\theta) = \sum_{n=1}^{N}\sum_{i=1}^{N} |R(t_n,x_i)|^2$$

with (t_n, x_i) sampled uniformly and

$$J_b(\theta) = \sum_{n=1}^{N_b}\sum_{i=1}^{N_b} |u_{nn}(t,x_i;\boldsymbol{\theta}) - g(x_i)|^2, \quad J_i(\theta) = \sum_{i=1}^{N_i} |u_{nn}(0,x_i;\boldsymbol{\theta}) - u_0(x_i)|^2.$$

- To avoid an extra loss for the BC and initial conditions, we use:

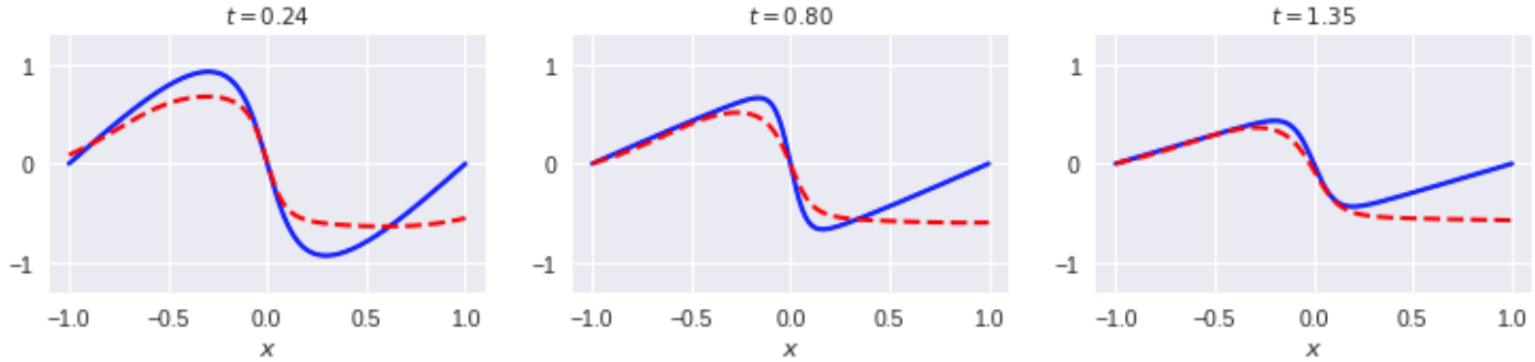
$$\bar{u}_\theta(t,x) = u_0(x) + t(\phi(x) * u_\theta(x)),$$

with $\phi(x) = g(x)$ on the boundary and something inside.

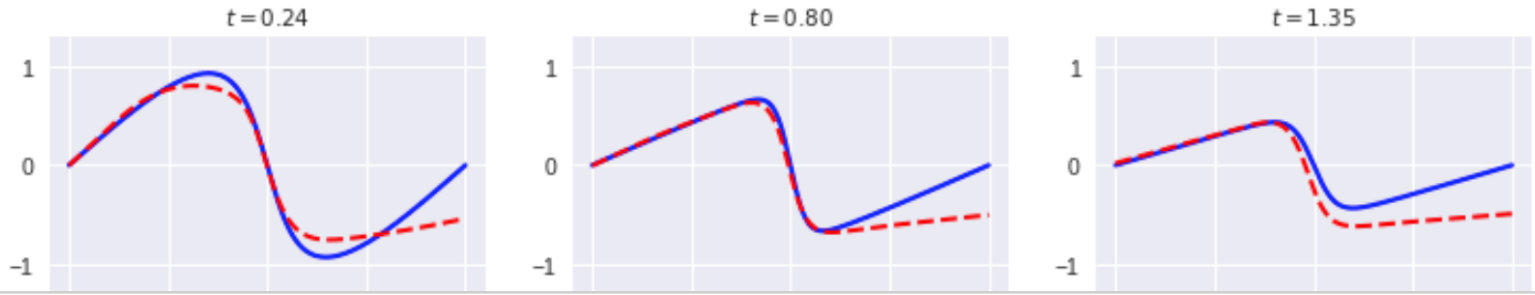
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.1}{\pi}$. 10000 pts, medium-sized NN.
- beginning of training

```
iter =  200
loss = 0.0774
L2 error: 4.7142e-01
```



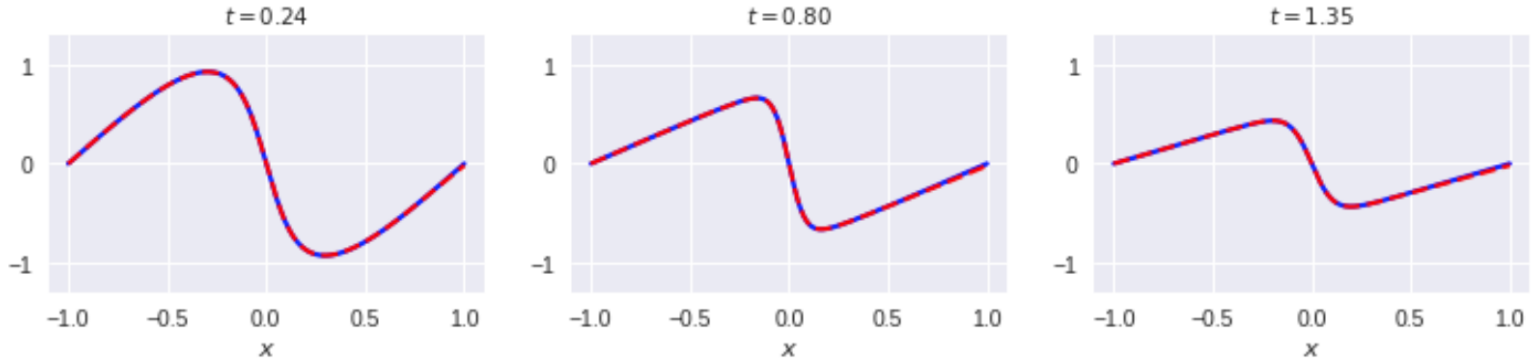

```
iter =  400
loss = 0.0318
L2 error: 4.0850e-01
```



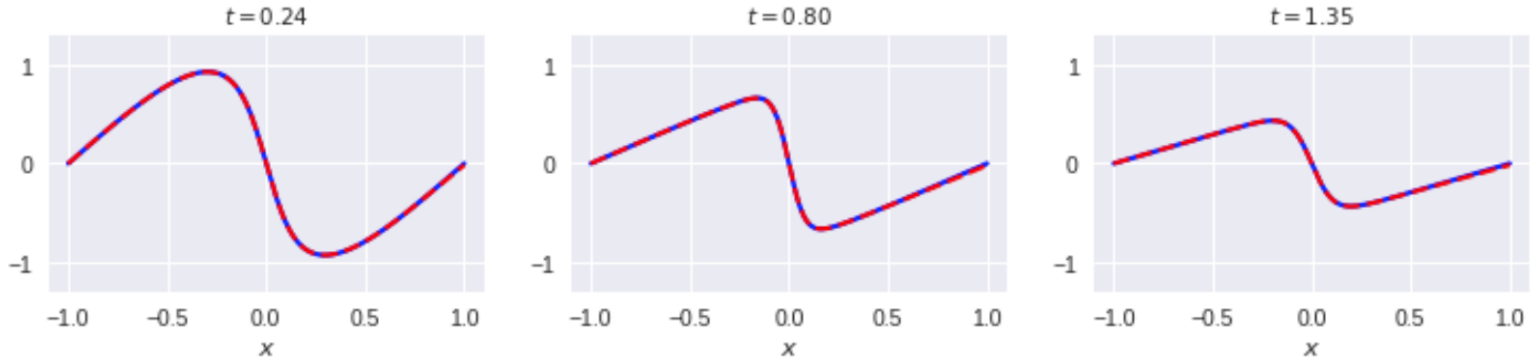
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.1}{\pi}$. 10000 pts, medium-sized NN.
- middle of training

```
iter =  2000
loss = 0.0000
L2 error: 9.0829e-03
```



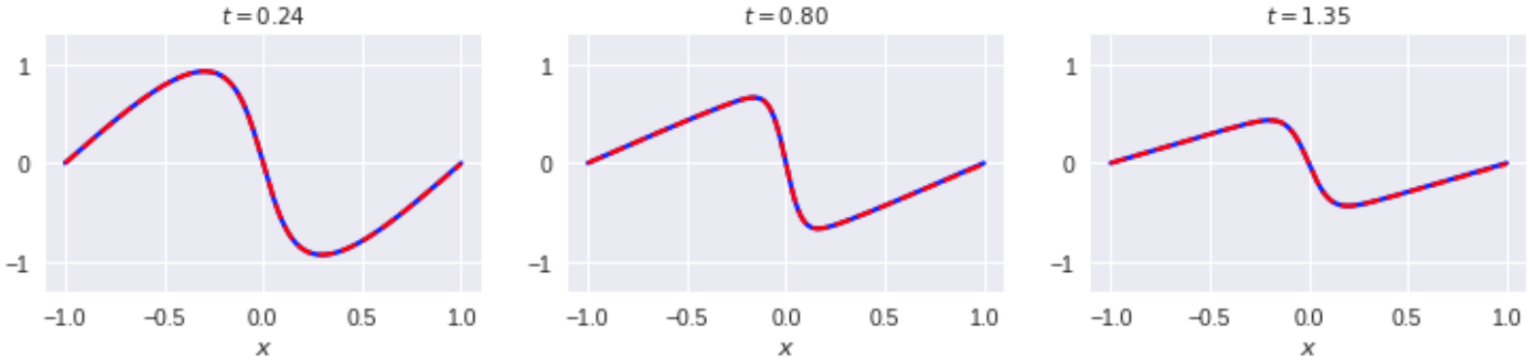

```
iter =  2200
loss = 0.0000
L2 error: 8.2614e-03
```



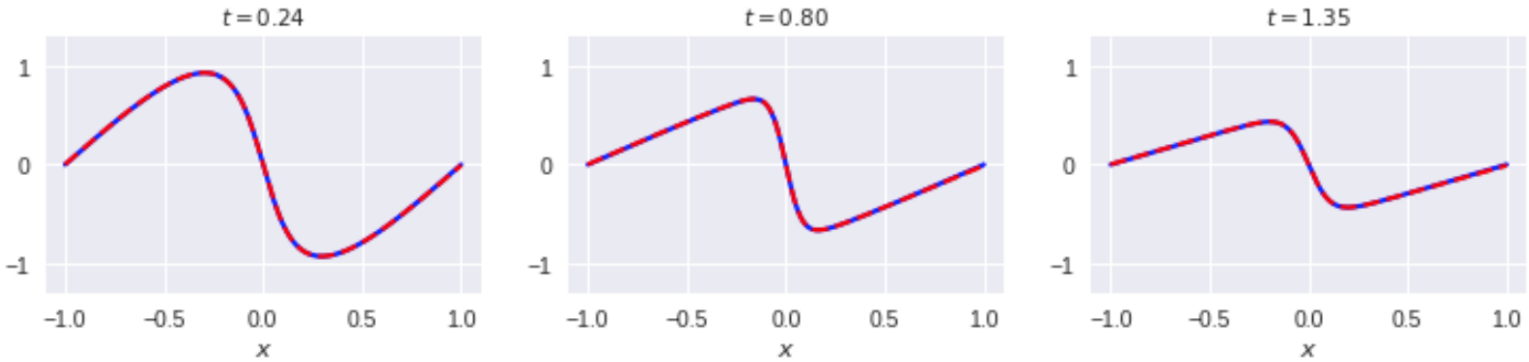
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.1}{\pi}$. 10000 pts, medium-sized NN.
- end of training

```
iter =  4800
loss = 0.0000
L2 error: 4.6718e-03
```



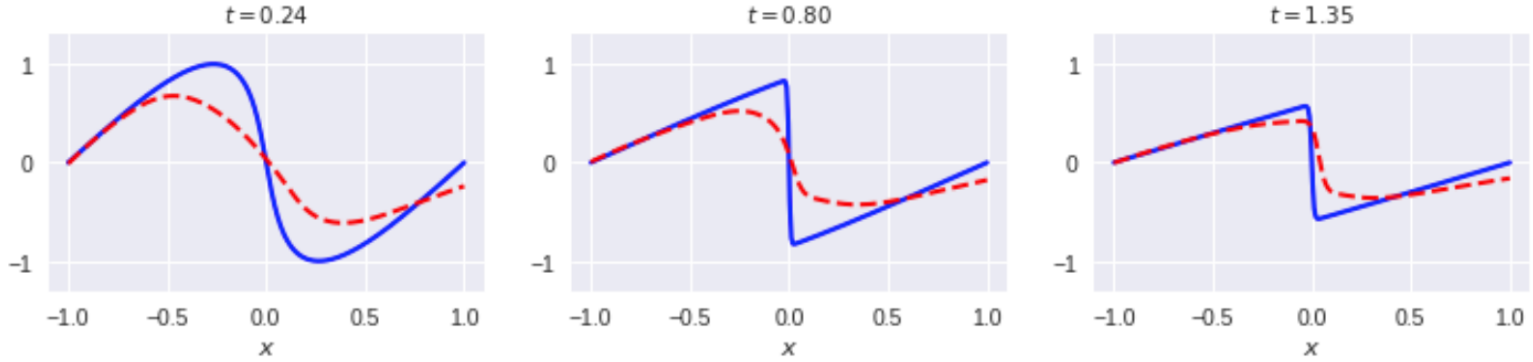

```
iter =  5000
loss = 0.0000
L2 error: 4.7307e-03
```



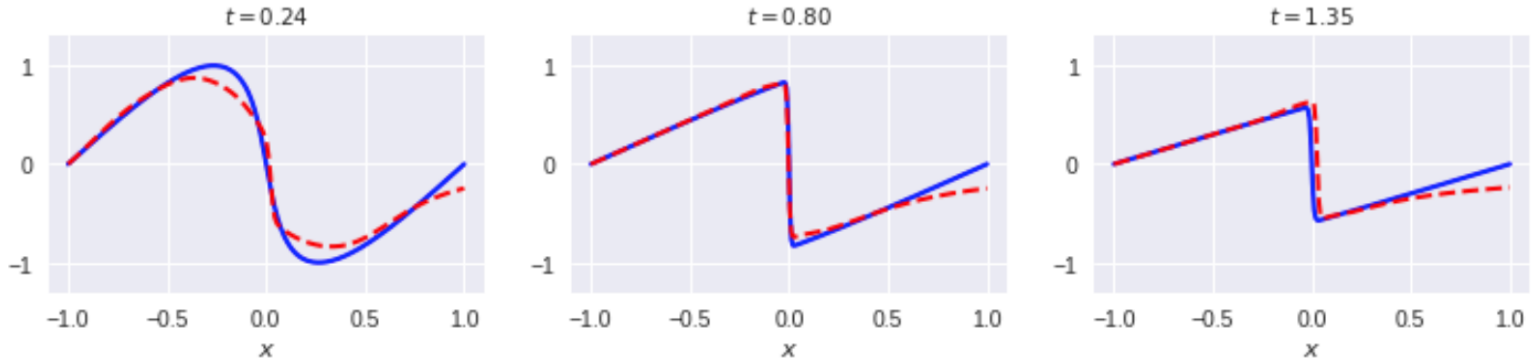
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.01}{\pi}$. 10000 pts, medium NN.
- beginning of training

```
iter =  600
loss = 0.0885
L2 error: 4.3601e-01
```



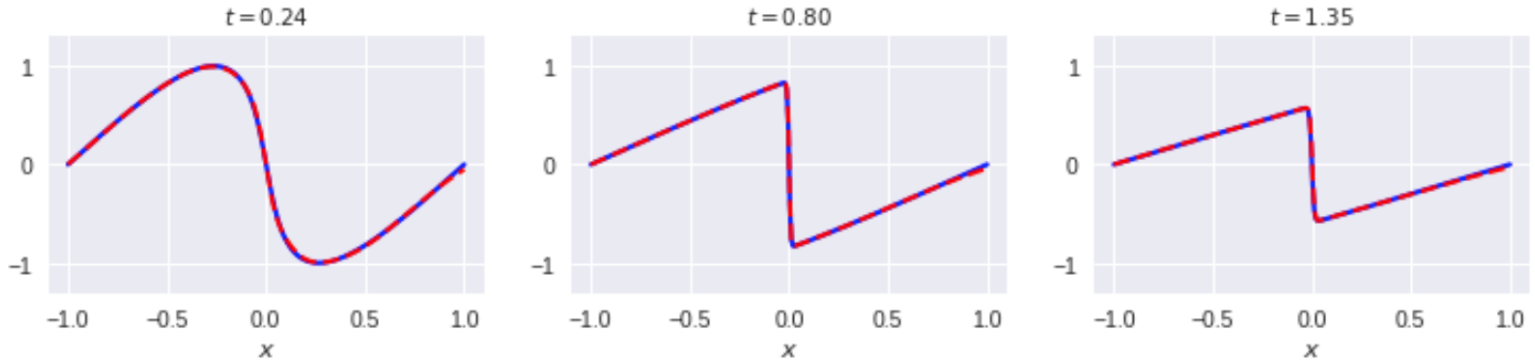

```
iter =  800
loss = 0.0233
L2 error: 2.0901e-01
```



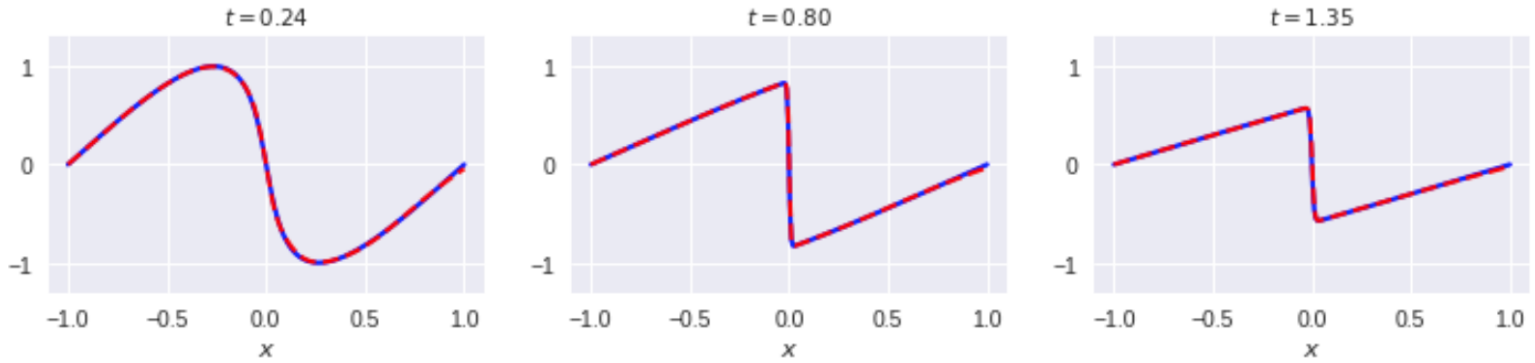
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.01}{\pi}$. 10000 pts, medium NN.
- middle of training

```
iter =  2000
loss = 0.0003
L2 error: 1.8053e-02
```



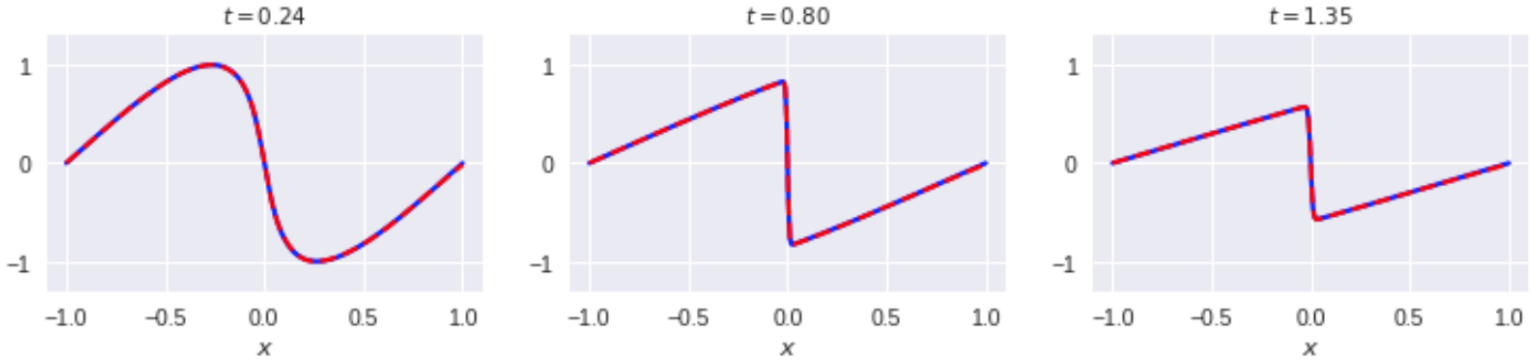

```
iter =  2200
loss = 0.0002
L2 error: 1.7773e-02
```



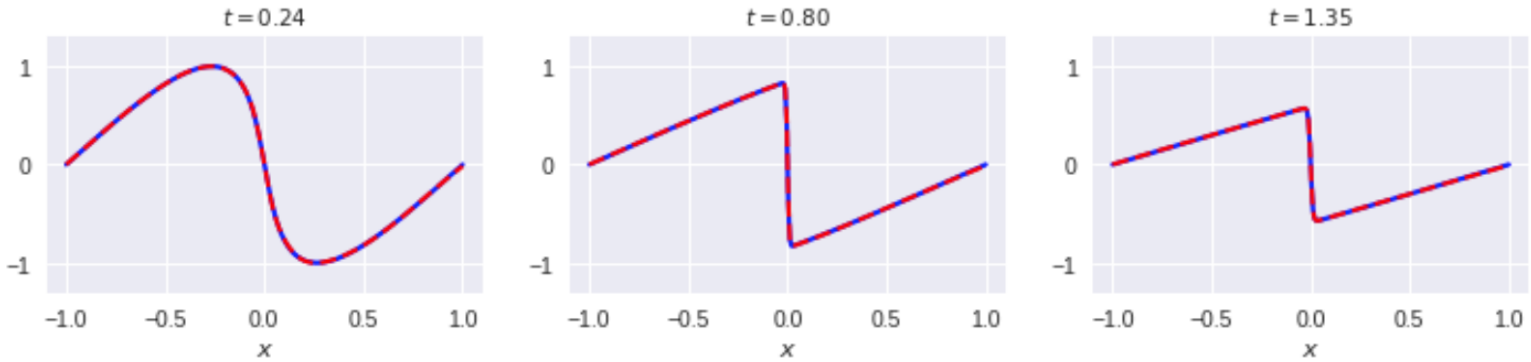
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.01}{\pi}$. 10000 pts, medium NN.
- end of training

```
iter =  4800
loss = 0.0001
L2 error: 5.9728e-03
```



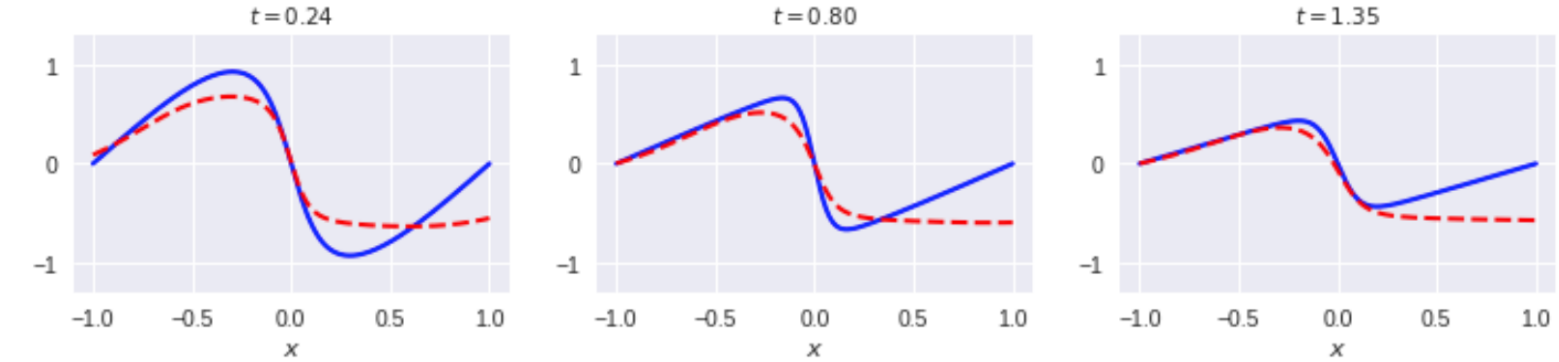

```
iter =  5000
loss = 0.0001
L2 error: 5.2593e-03
```



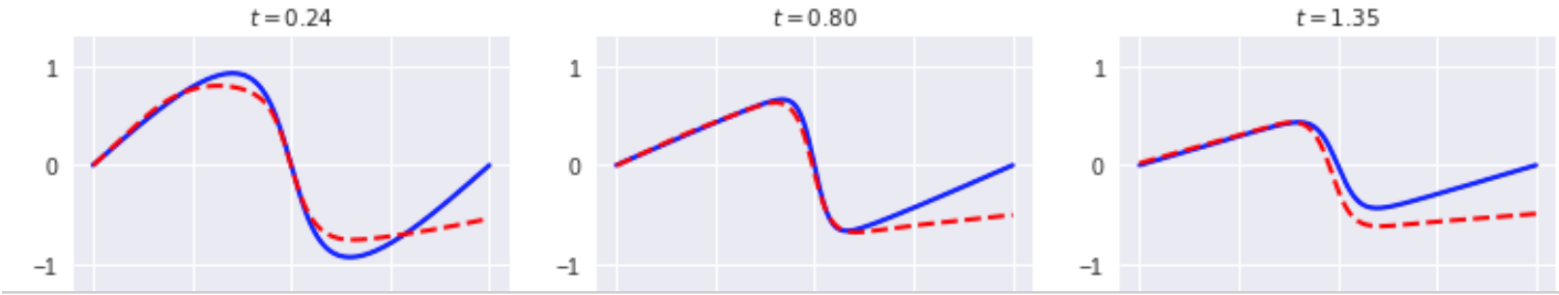
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.002}{\pi}$. 10000 pts, medium NN.
- beginning of training

```
iter =  200
loss = 0.0774
L2 error: 4.7142e-01
```



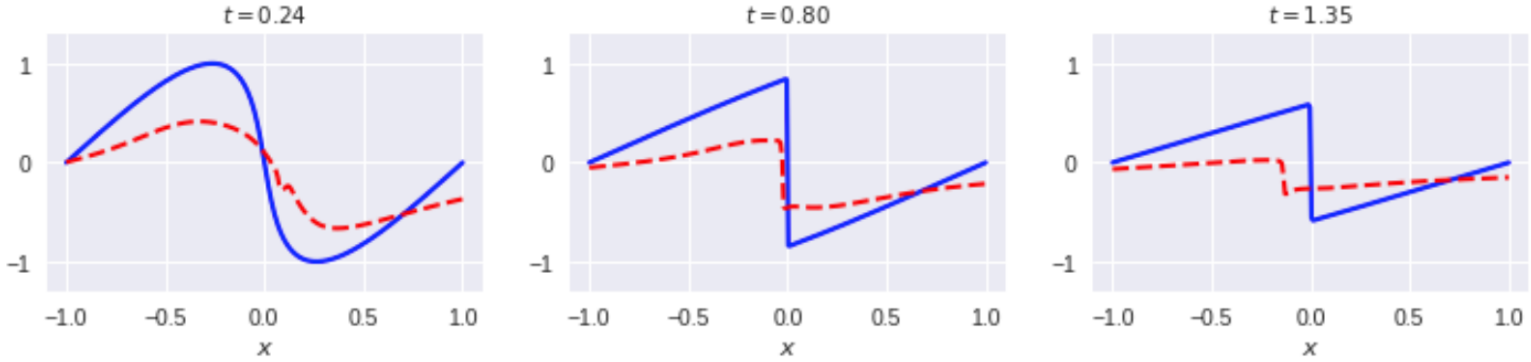

```
iter =  400
loss = 0.0318
L2 error: 4.0850e-01
```



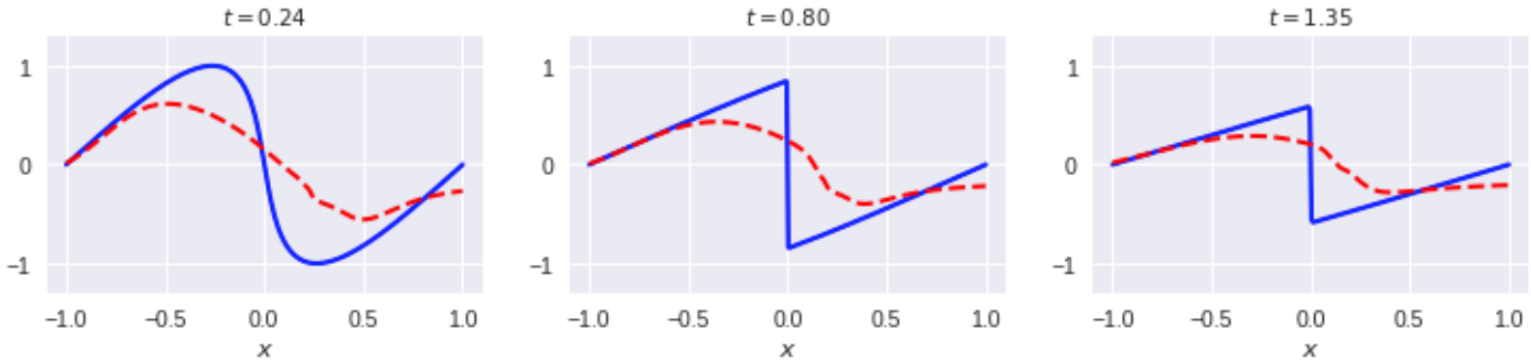
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.002}{\pi}$. 10000 pts, medium NN.
- middle of training

```
iter =  2000
loss = 0.2076
L2 error: 6.2666e-01
```



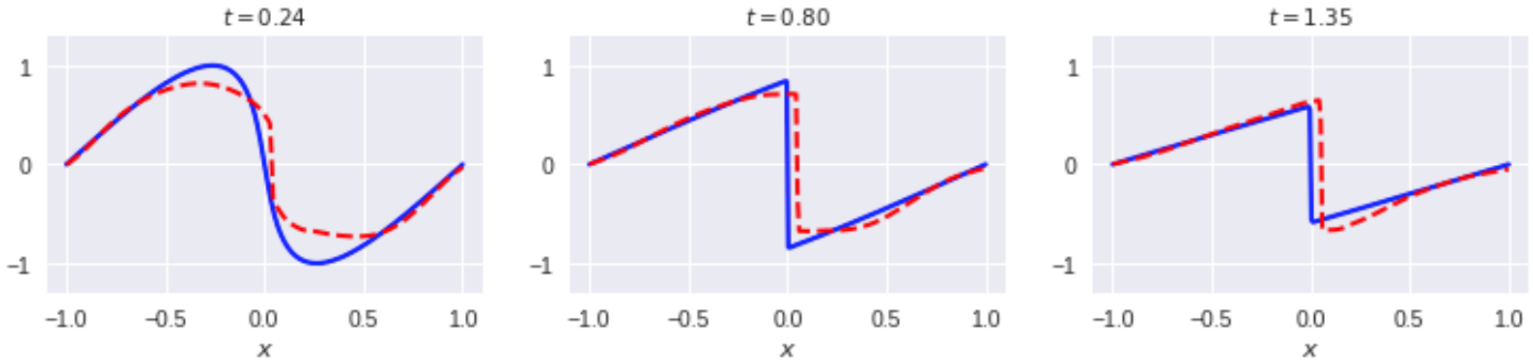

```
iter =  2200
loss = 0.1361
L2 error: 6.0138e-01
```



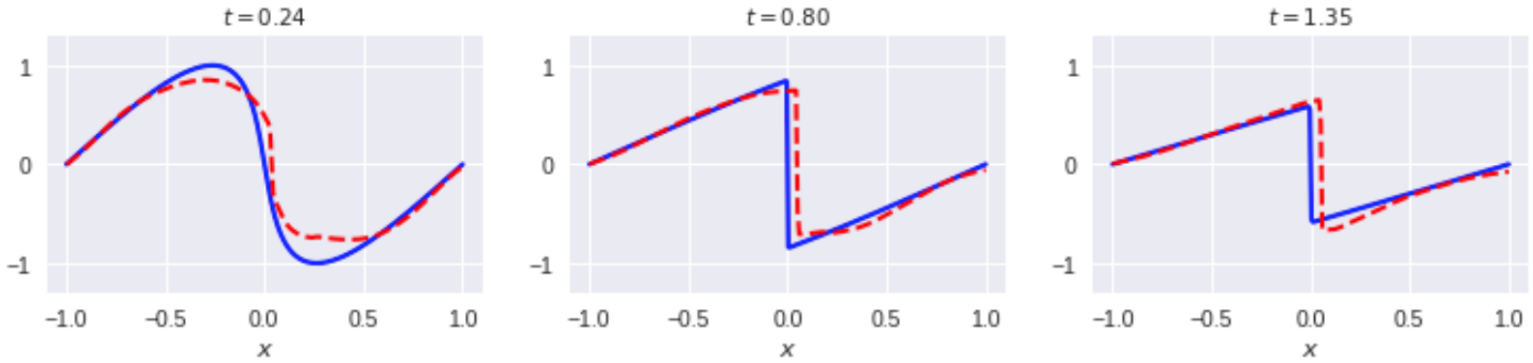
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.002}{\pi}$. 10000 pts, medium NN.
- end of training

```
iter =  4800
loss = 0.0272
L2 error: 4.0909e-01
```



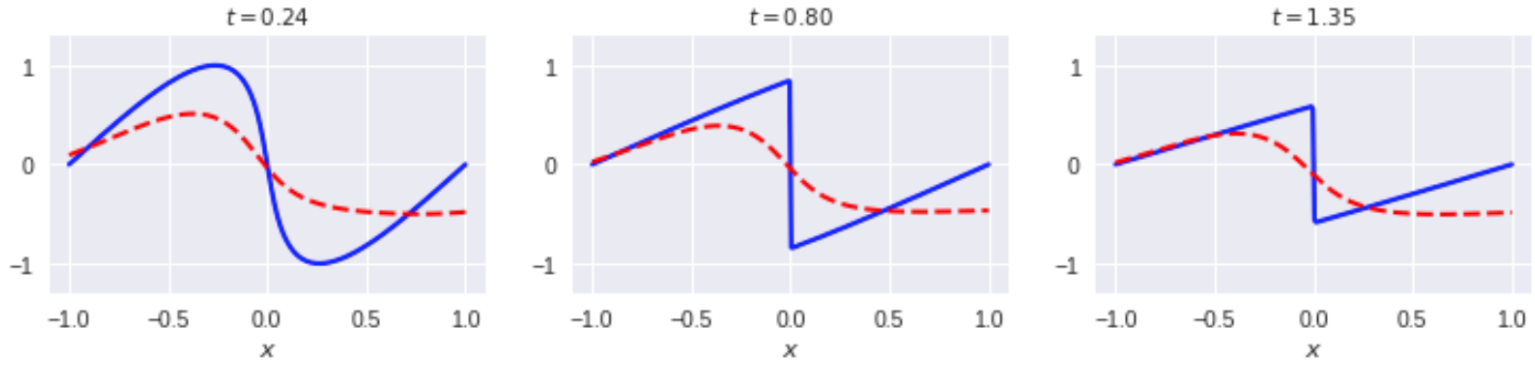

```
iter =  5000
loss = 0.0212
L2 error: 4.0300e-01
```



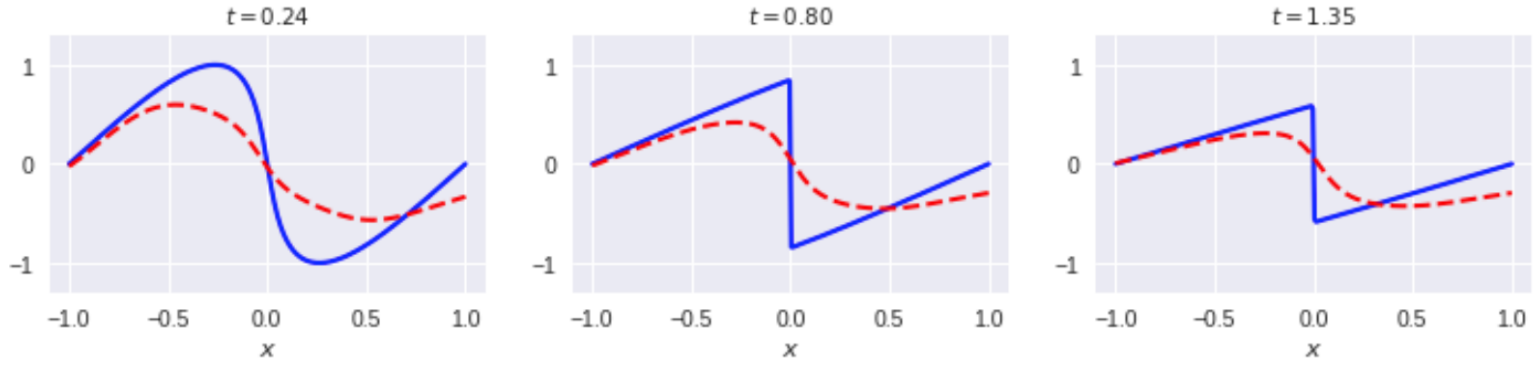
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.002}{\pi}$. 20000 pts, medium NN.
- beginning of training

```
iter =  200
loss = 0.1495
L2 error: 6.1471e-01
```



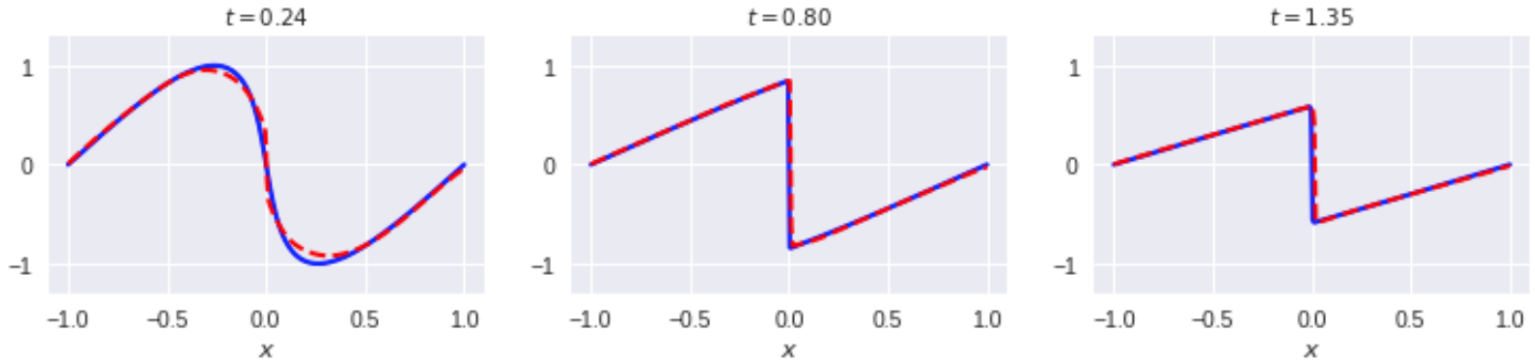

```
iter =  400
loss = 0.1170
L2 error: 5.2688e-01
```



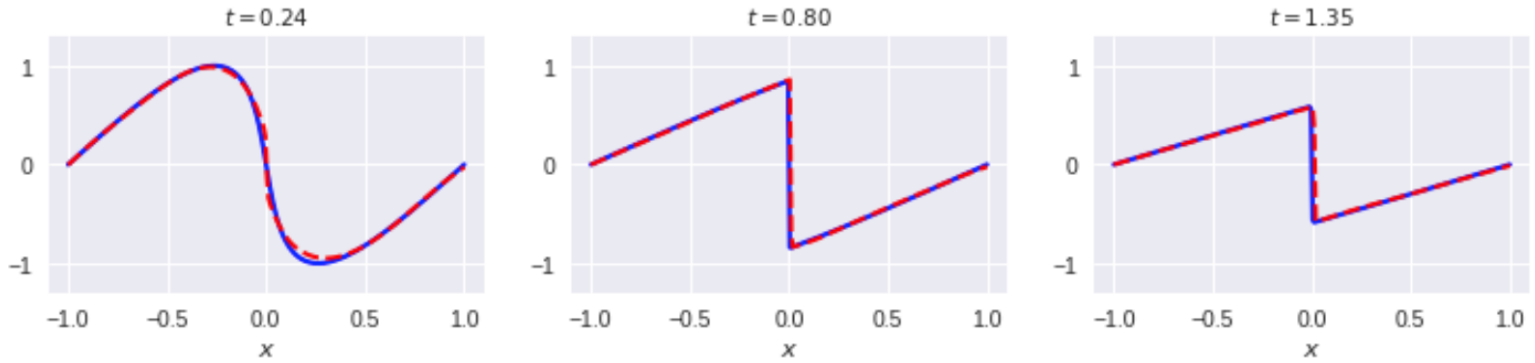
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.002}{\pi}$. 20000 pts, medium NN.
- middle of training

```
iter =  2000
loss = 0.0040
L2 error: 1.7457e-01
```



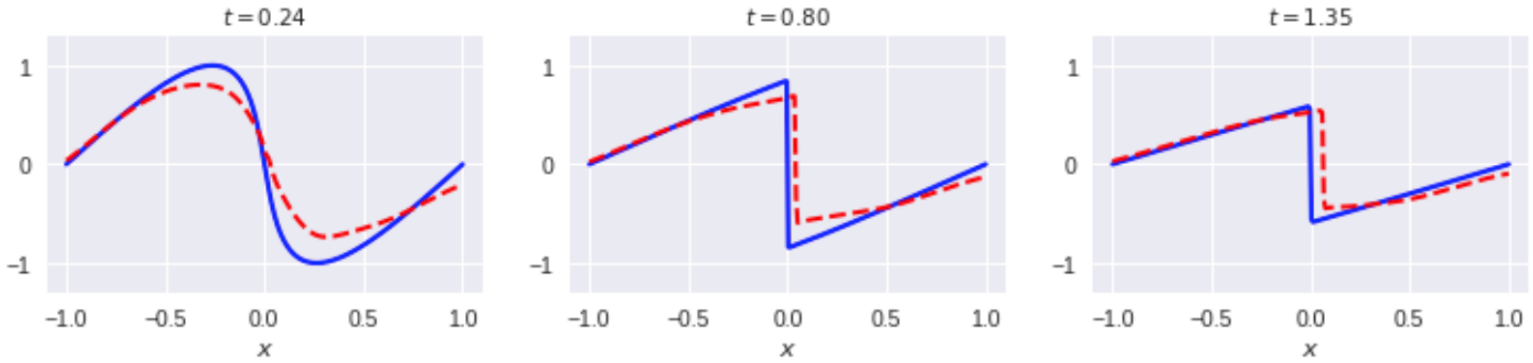

```
iter =  2200
loss = 0.0024
L2 error: 1.6838e-01
```



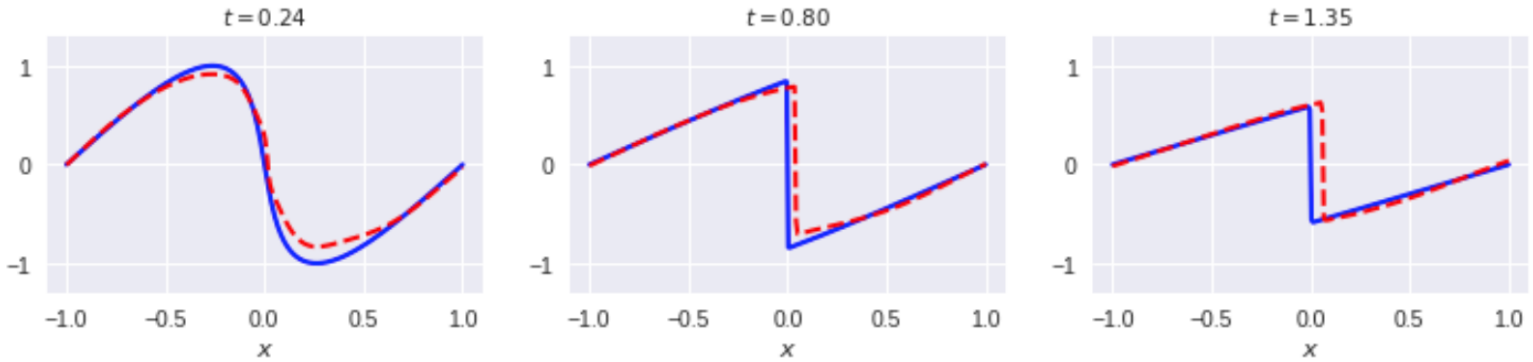
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.002}{\pi}$. 20000 pts, medium NN.
- end of training

```
iter =  4800
loss = 0.0395
L2 error: 3.9314e-01
```



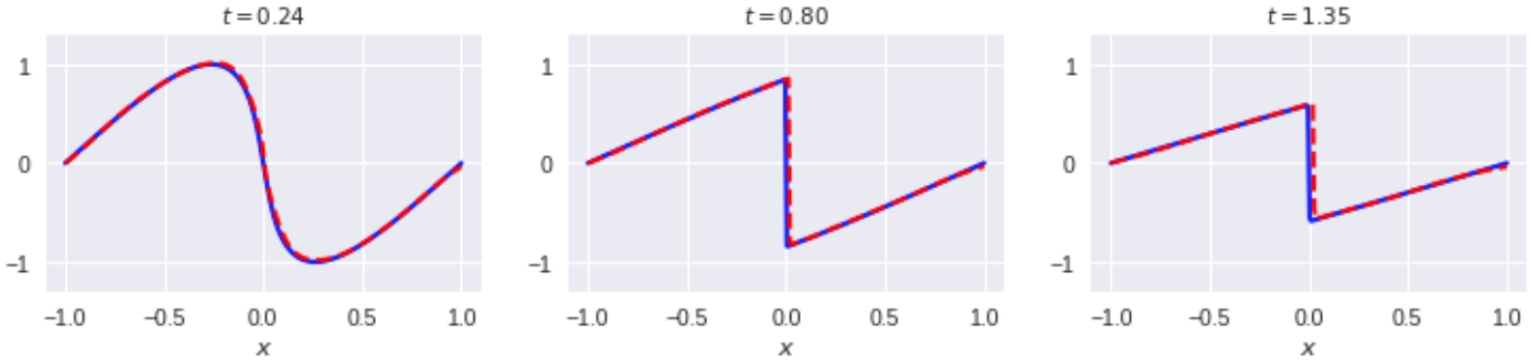

```
iter =  5000
loss = 0.0133
L2 error: 3.6761e-01
```



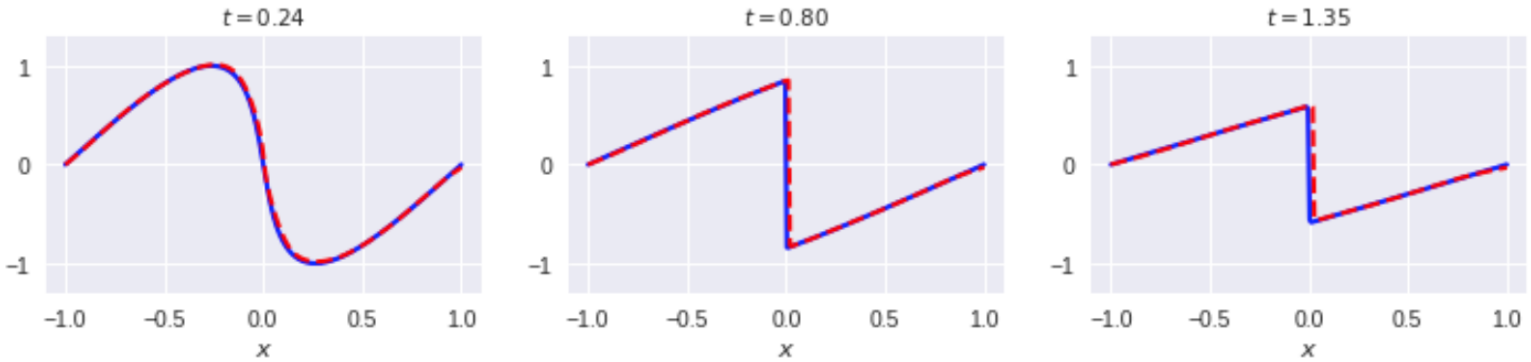
Example: Burgers equation

- Application: Burgers equation $\partial_t \rho + \partial_x \left(\frac{\rho^2}{2} \right) = \nu \partial_{xx} \rho$.
- Solving for different values of the μ parameters:
- $\nu = \frac{0.002}{\pi}$. 40000 pts, larger NN.
- end of training

```
iter =  4800
loss = 0.0006
L2 error: 2.3011e-01
```

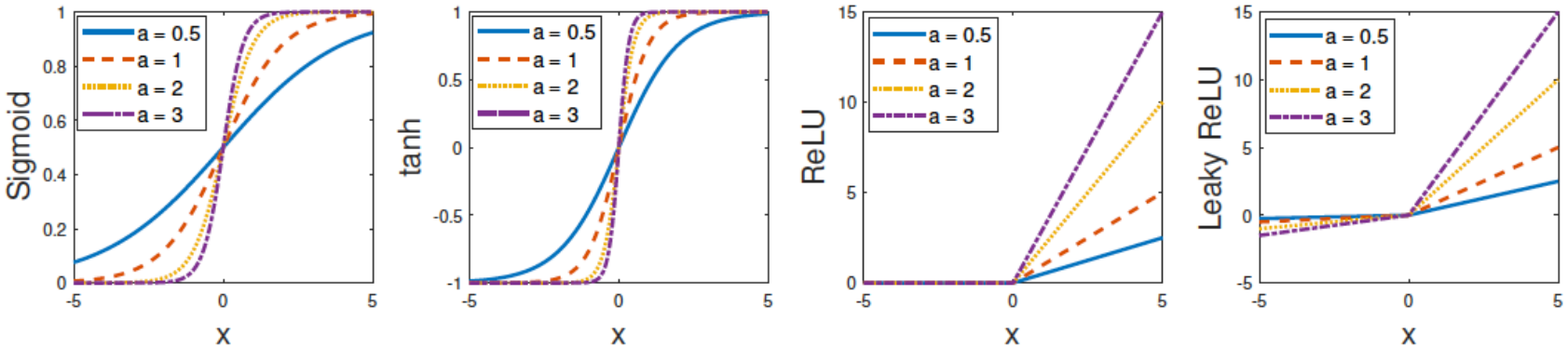



```
iter =  5000
loss = 0.0004
L2 error: 2.2456e-01
```



Adaptive Activation function

- To increase the ability of the PINNs to represent data, we can use learnable activation functions.
  - Adaptive relu: $\sigma_a(x) = \max(0, ax)$
  - Adaptive leaky Relu: $\sigma_a(x) = \max(0, ax) - \nu \max(0, -ax)$
  - Adaptive hyperbolic tangent: $\sigma_a(x) = \frac{e^{ax} - e^{-ax}}{e^{ax} + e^{-ax}}$
  - Adaptive sigmoid: $\sigma_a(x) = \frac{1}{1+e^{-ax}}$ where we learn a.
- So we can write the loss as $L(\theta, a)$, and compute the gradient with respect to a.

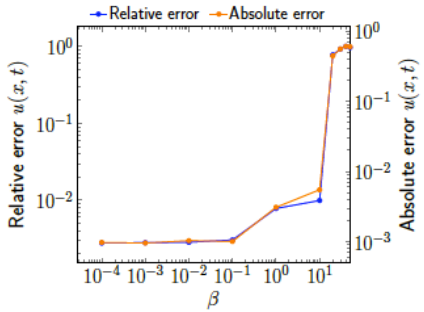


- In general, we use smooth activation functions.

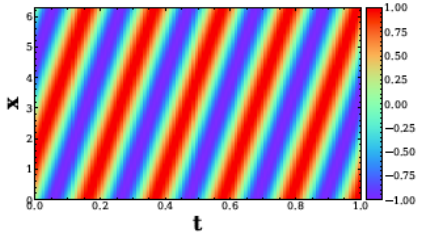
Biases of PINNs and other training methods

- PINNs can suffer from biases, which complexifies the training.
- Large coefficient $\beta \gg 1$ in:

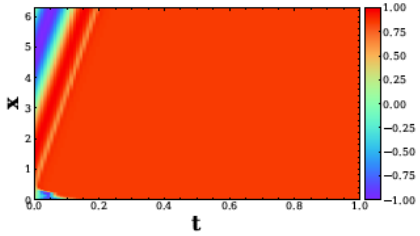
$$\partial_t \rho + \beta \partial_x \rho = 0$$



(a) *Error for different* β

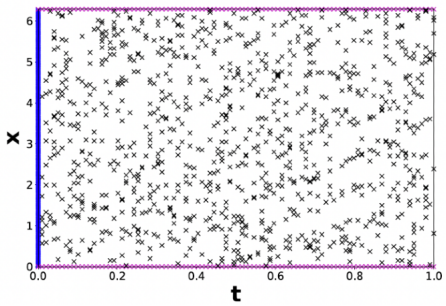


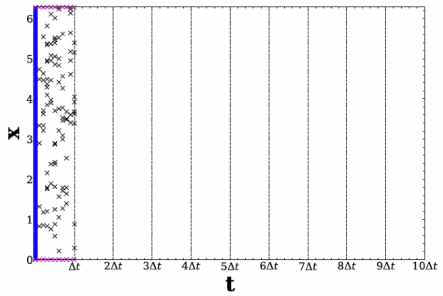
(b) *Exact solution for* $\beta = 30$

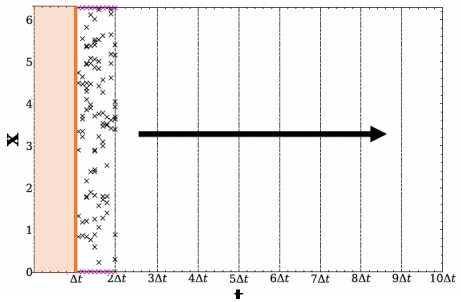


(c) *PINN solution for* $\beta = 30$

  - when β is too large, the residual loss dominates compared to the initial condition loss, and the network learns a constant.
  - **Solution**: progressively increase β. Hard constraints for the initial data?
- **Time bias**: the last time steps are learned before the first. There are different ways to introduce causality in the training. Example:





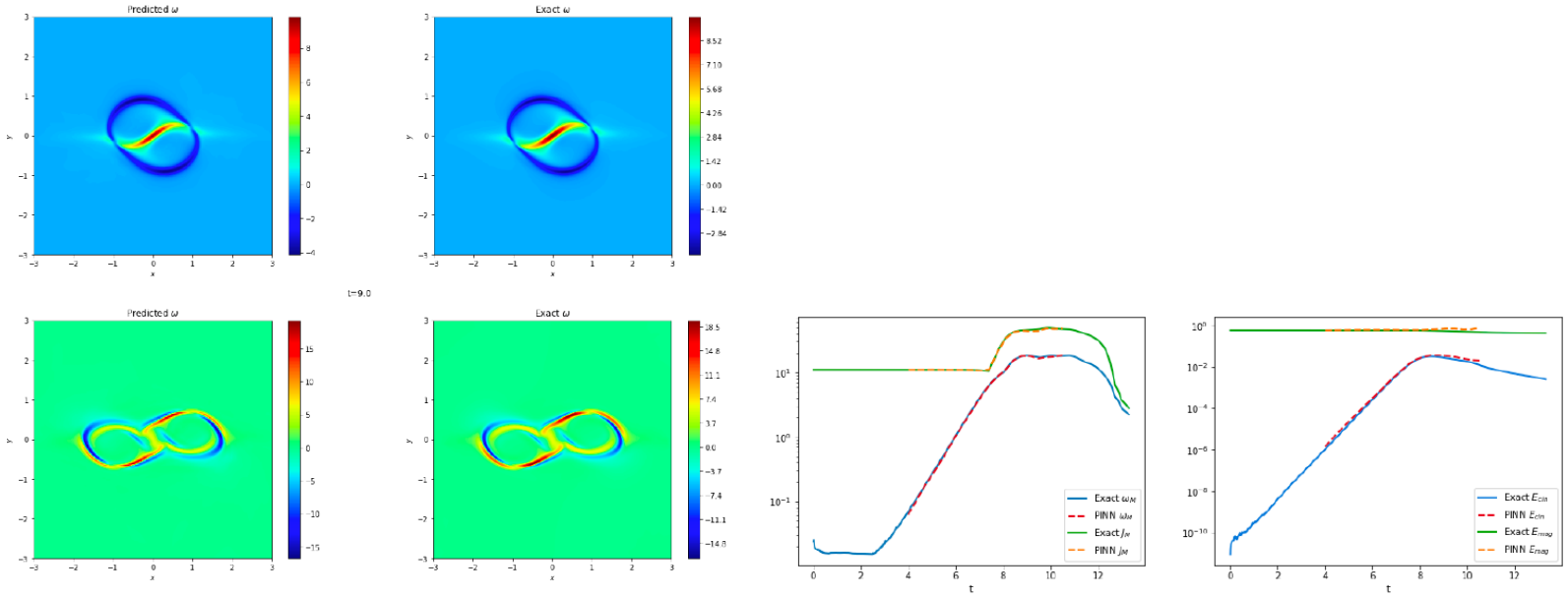


Example: Reduced MagnetoHydroDynamics (MHD)

- 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.
- prediction of ω (left), evolution of the energy (right)



- Since the problem is multi-scale, training is very complicated, and we requires 10 data points in time to get an accurate description of the instability. It took a long time to find the good parameters.

Sampling

- **Importance sampling**: classical approach to reduced Monte Carlo errors. We sample with another law, and we expect a small variance.
- We define

$$R(x,t) = \partial_t u_{nn}(t,x;\theta) - \mathcal{F}(u_{nn}(t,x;\theta), \partial_x u_{nn}(t,x;\theta), \partial_{xx} u_{nn}(t,x;\theta)).$$

We want to minimize:

$$\mathcal{J}(\theta) = \int_0^T \int_\Omega |R(x,t)|^2 dxdt$$

We can modify the sampling law

$$\mathcal{J}(\theta) = \int_0^T \int_\Omega R(x,t) dxdt = \int_0^T \int_\Omega \frac{R(x,t)}{p(t,x)} p(t,x) dxdt \approx \sum_{i=1}^N \frac{R(x_i,t_i)}{p(t_i,x_i)},$$

with $p(x,t)$ a probability law and (x_i, t_n) sampled with p.

- **Idea**: fit a network $p(x,t;\theta_p)$ to mimic $r(t,x)$: thus, points are sampled where the residue is large, which reduces the error.
- We use specific neural networks for probability laws.

PINNs variant I

- Many ideas coming from classical numerical methods can be combined with PINNs.
- **First example**: VPINNs - hp-VPINNs
- In finite differences, we solve, for example:

$$-\Delta u(x) = f(x).$$

- In finite elements, we solve the weak/ultra weak form:

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx, \quad -\int_\Omega u \Delta v dx + \int_{\partial\Omega} u \nabla v \cdot \mathbf{n} dx = \int_\Omega f v dx$$

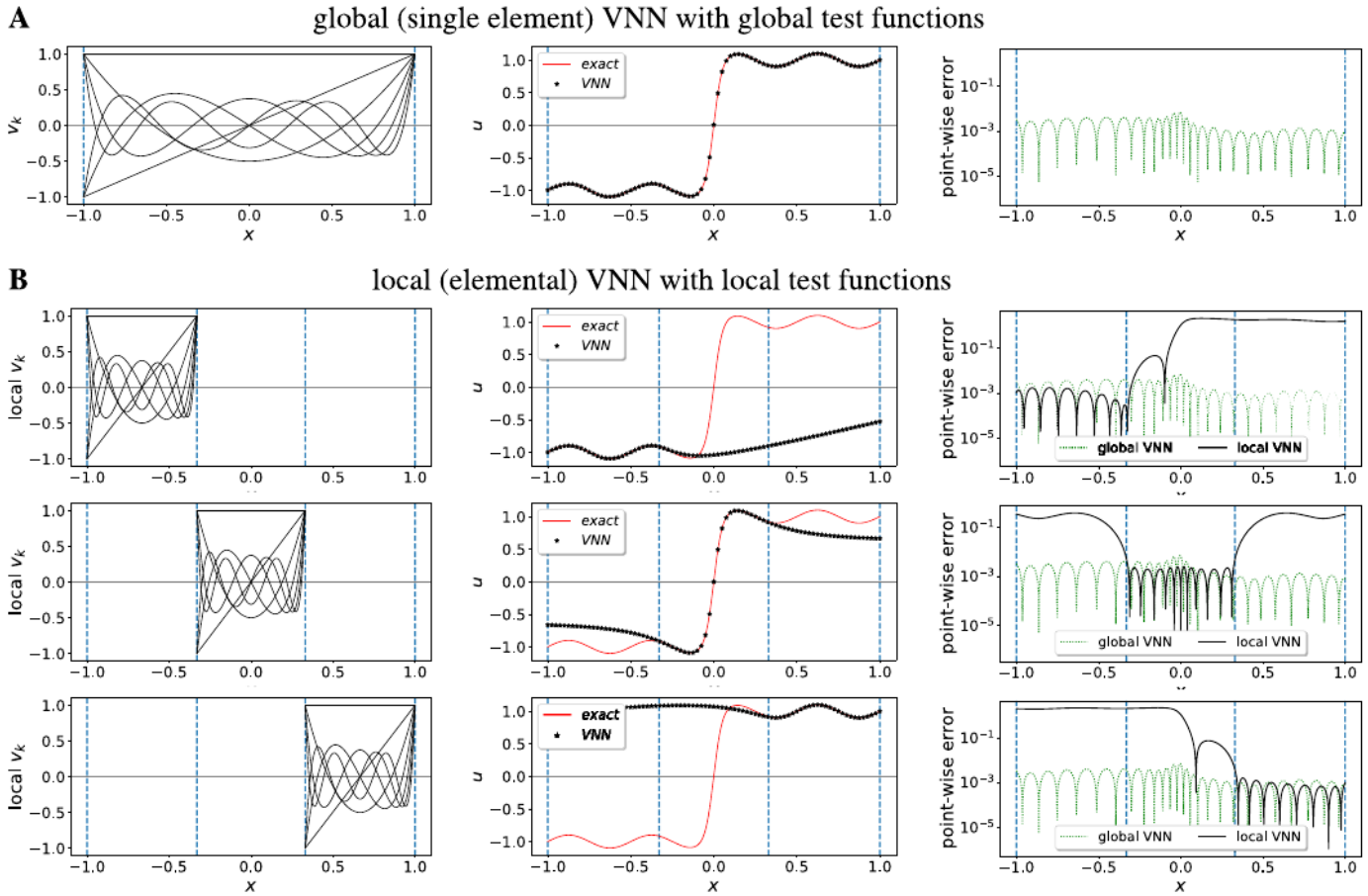
  with $v(x)$ a compactly supported smooth function.
- **Idea of VPINNs**: minimizing a residue using the weak or ultra weak form.
- The new optimization problem becomes, for example

$$\min_\theta \sum_{k=1}^{K} \int_\Omega (\nabla u_{nn}(x;\theta) \cdot \nabla v_k(x) - f(x) v_k(x)) dx$$

  with $v_k(x)$ usual smooth functions, such as polynomials or trigonometric functions.
- **Why it is interesting**: the computation of the first and second derivatives of the network is a time-consuming process, which slows the training down.

PINNs variant II

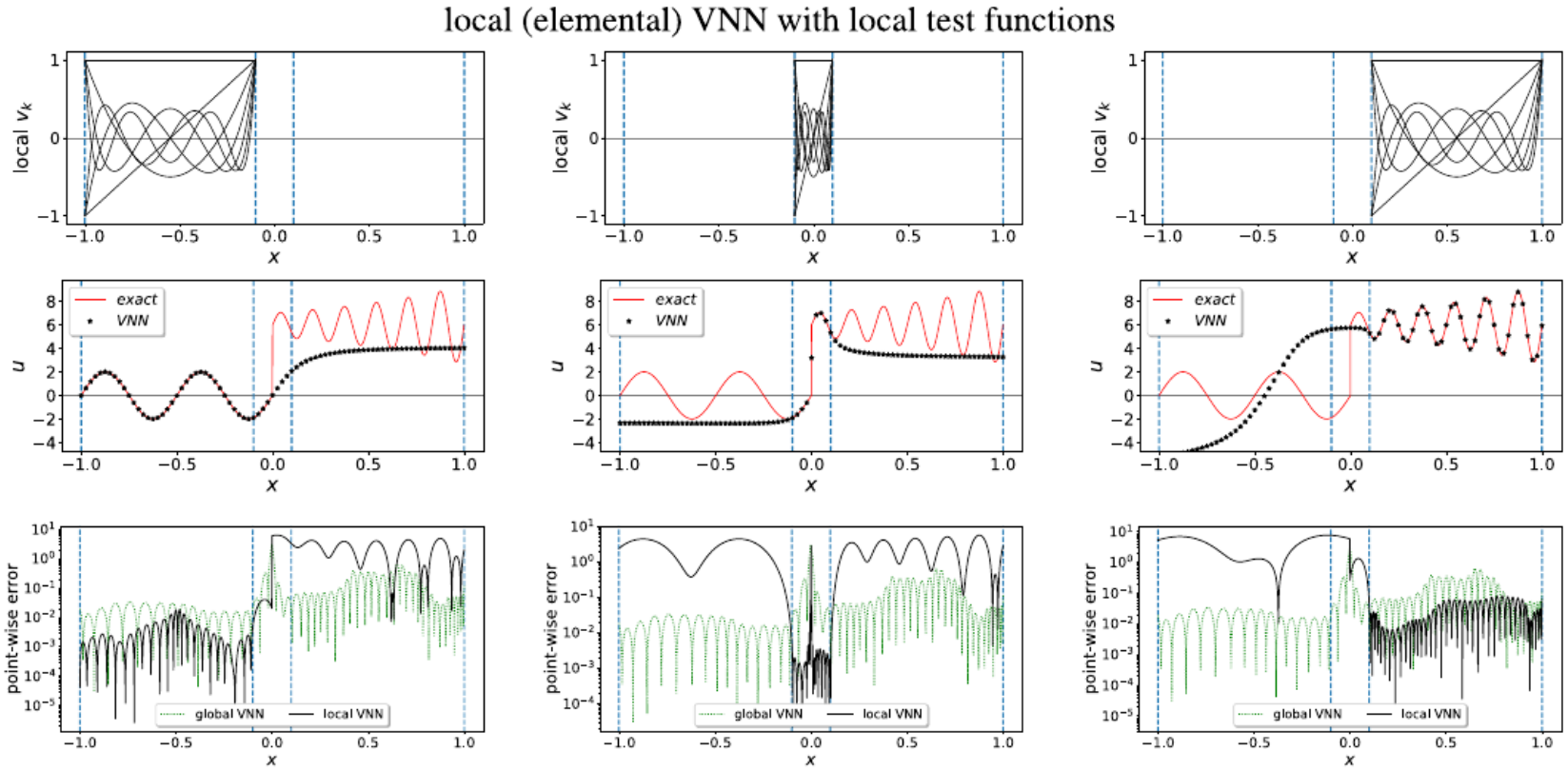
- **hp-VPINNs** enable domain decomposition. We use one network by subdomain, and the functions $v_k(x)$ are local to the subdomains.
- Example (from the original hp-VPINNs paper):



- Many other variants for more specific problems.

PINNs variant II

- **hp-VPINNs** enable domain decomposition. We use one network by subdomain, and the functions $v_k(x)$ are local to the subdomains.
- Example (from the original hp-VPINNs paper):



- Many other variants for more specific problems.

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 PDEs parameterized by some $\boldsymbol{\mu}$.
- The neural network becomes $u_{nn}(t, x, \boldsymbol{\mu}; \theta)$.

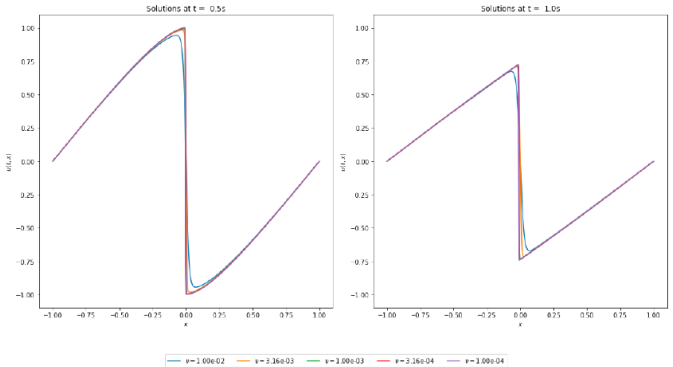
New Optimization problem for PINN's

$$\min_{\theta} J_r(\theta) + \ldots, , \quad \text{with}$$

$$J_r(\theta) = \int_{V_\mu} \int_0^T \int_\Omega \left\| \partial_t u_{nn} - \mathcal{L}\left(u_{nn}(t, x, \boldsymbol{\mu}), \partial_x u_{nn}(t, x, \boldsymbol{\mu}), \partial_{xx} u_{nn}(t, x, \boldsymbol{\mu})\right) \right\|_2^2 dxdt$$

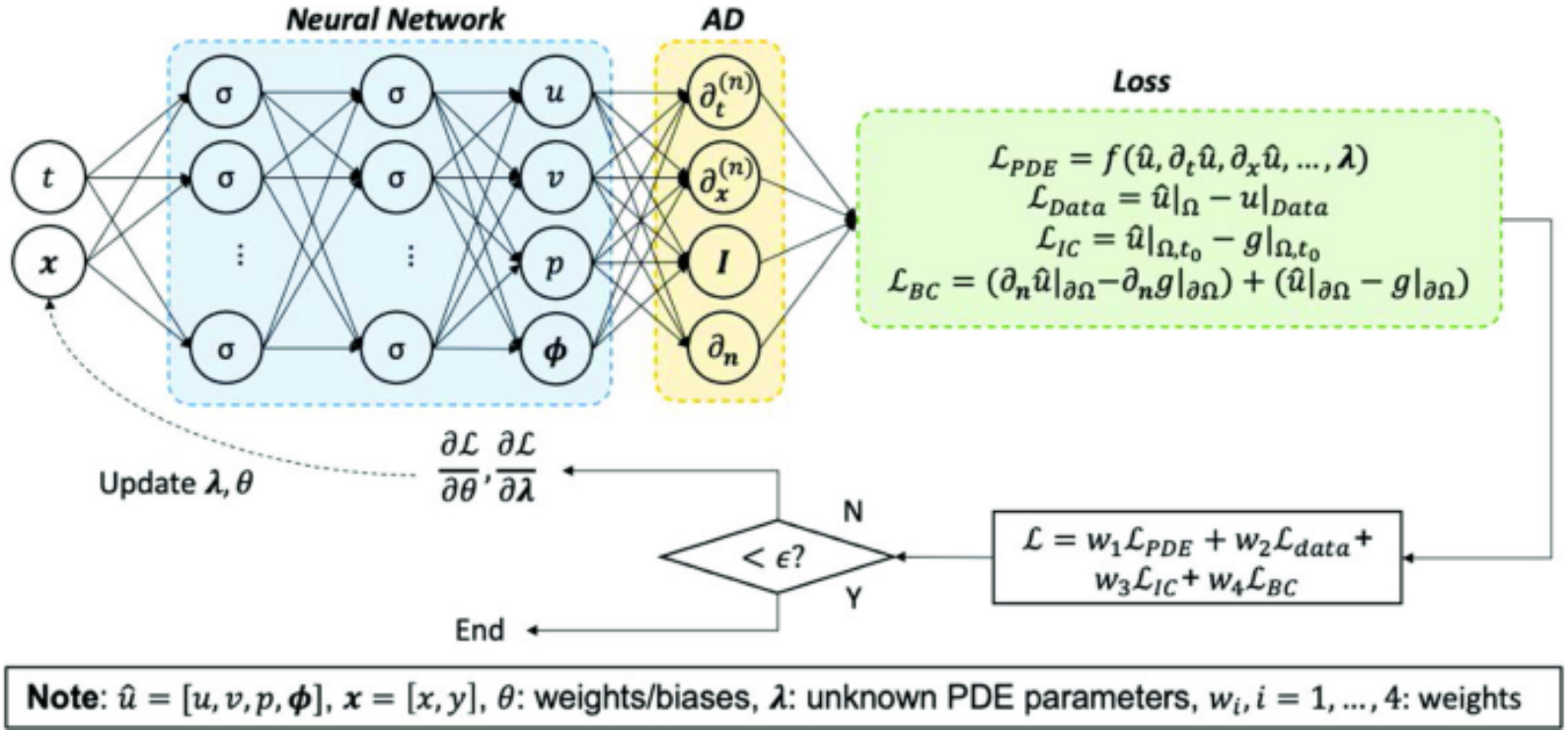
with V_μ a subspace of the parameters $\boldsymbol{\mu}$.

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



- Training for $\mu = 10^{-4}$: 2h. Training for the full viscosity subset: 2h.

Full algorithm for PINNs



Drawbacks and Advantages

- PINNs are less accurate and slower than classical solvers, and it is difficult to choose their hyper-parameters.
- PINNs are mesh-less and largely independent of the dimension. Thus, they are mostly interesting for parametric PDEs or PDEs in large dimension.

PINNs for optima control/inverse problem

- Example of invert problem:

$$\min_{d(x)} J(d) = \min_{d(x)} \int_\Omega \mid P(u(x)) - z_{data} \mid^2 + \lambda \int_\Omega \mid d(x) \mid^2$$

with

$$-\nabla \cdot (d(x)\nabla u) = f(x)$$

- **Aim**: find the media of propagation knowing a part of the solution u.
- **Classical approach**: we write a dual PDE, we solve the PDE and the dual to obtain $\nabla_d J_d$ and we make gradient descent.
- We need to solve many direct and dual problems.

Inverse problem with PINNs

It is easy: we take two networks $u_\theta(x)$ and $d_{\theta(x)}$ and we solve

$$\min_\theta \left(\sum_{i=1}^N J(d(x_i)) + \alpha \sum_{i=1}^N \mid \nabla \cdot (d(x_i)\nabla u(x_i)) + f(x_i) \mid^2 \right)$$

- Key point: choose α.

Operator Learning

Operator learning

- We consider the following problem:

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

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

Objective

Approximate G^+ by a neural network on a subspace of the data where the results do not depend of the mesh resolution of the inputs/output functions.

Problem

We construct a neural network $G_\theta^+(\boldsymbol{\mu}(t,x))$, which minimizes $\mathcal{J}(\theta) = \mathcal{J}_1(\theta) + \mathcal{J}_2(\theta)$, with

$$\mathcal{J}_1(\theta) = \int_{\boldsymbol{V}_\mu} \int_\Omega \int_0^T \| G_\theta^+(t,x,\boldsymbol{\mu}_h(x,t)) - \boldsymbol{u}(x,t) \|_2^2 d\boldsymbol{\mu} dx dt$$

and

$$\mathcal{J}_2(\theta) = \int_{\boldsymbol{V}_\mu} \int_\Omega \int_0^T \| G_{\boldsymbol{\mu}}(G_\theta^+(t,x,\boldsymbol{\mu}_h(x,t))) \|_2^2 d\boldsymbol{\mu} dx dt,$$

where the integrals are approximated by MC.

Latent space Neural Operator I

- We use a simple example to avoid complex notation:

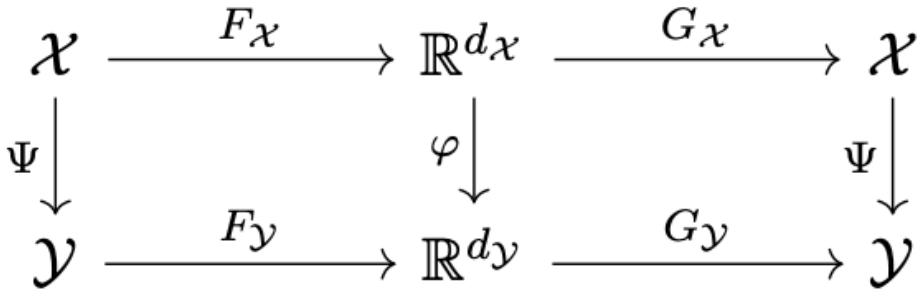
$$-\partial_{xx} u(x) = f(x).$$

- Goal: fit $G^+ : f(x) \to u(x)$.

Latent space NO

We define an encoder $\mathcal{E}_\theta(x) : \mathcal{X} \to \mathbb{R}^{d_x}$ with $\mathcal{X}$ a Hilbert function space, a decoder $\mathcal{D}_\theta(x) : \mathbb{R}^{d_y} \to \mathcal{Y}$ and $\mathcal{A}_\theta : \mathbb{R}^{d_x} \to \mathbb{R}^{d_y}$. A Latent space NO is given by:

$$G^+ = \mathcal{D}_\theta \circ \mathcal{A}_\theta \circ \mathcal{E}_\theta$$



- **Goal**: The decoder $\mathcal{D}_\theta$ should be resolution-invariant; if possible, so should the encoder $\mathcal{E}_\theta$. Another goal is to obtain mesh invariance.

Latent space Neural Operator II

Example 1: DeepONet

- the encoder is $\mathcal{E}_\theta : (f(x_1), \dots, f(x_n)) \mapsto (\alpha_1, \dots, \alpha_{d_x})$ with classical NN
- $\mathcal{A}_\theta : (\alpha_1, \dots, \alpha_{d_x}) \mapsto \beta_1, \dots, \beta_{d_y}$ with classical NN
- the decoder is $\mathcal{D}_\theta : \beta_1, \dots, \beta_{d_y} \mapsto \mathcal{Y}$ with

$$\mathcal{D}_\theta(x) = \sum_{i=1}^{d_y} \beta_i \phi_i(x),$$

with a basis ϕ_i given by a network

$$b_\theta(x) = (\phi_1, \dots, \phi_{d_y}).$$

The outputs mesh and resolution invariant. Inputs on a given point cloud.

Example 2: Nomad

Same as DeepONet, except for the decoder, which becomes $\mathcal{D}_\theta : \beta_1, \dots, \beta_{d_y} \mapsto \mathcal{Y}$, with

$$\mathcal{D}_\theta(x, \beta_1, \dots, \beta_{d_y})$$

a Neural Network. It is a parametric PINNs where the parameters are found by another network (Encoder and Approximator).

Spectral Neural operator I

- Neural networks mapping functions leads to so-called neural operators (N. Kovachki, Z. Li et al 2021).
- **How to construct neural operators**?
- Example:

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

- 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.

Interesting framework

In the formalism proposed by N. Kovachki, Z. Li et al, the key point is to add some non-locality in the layers.

- **Methods using non-locality**: FNO (Z. Li et al 20), WNO (Tripura et al 22), Laplacian NO (Chen et al), Graph kernel Operator, Multipole GNO (N. Kovachki et al 20), Laplace NO (G. Chen et al 23), Non Local NO (Z. Li et al 23), etc.

Spectral Neural operator II

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

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

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.

- **Key point**: we will learn the linear part, as well as the kernel k.
- How to perform this learning in practice?

Spectral Neural operator III

Lifting and projection layers

1. An extrapolation layer P to increase the size of the feature space:

$$[(v_1(x), \dots . v_{d_0}(x))] = P_\theta(\mu(x))$$

with P_θ a FNC.

2. A projector layer Q to decrease the size of the feature space:

$$[u(x)] = Q_\theta((v_1(x), \dots . v_{d_L}(x)))$$

with Q_θ a FNC.

Full neural operator

A neural operator is given by the following composition of layers:

$$u(x) = Q \circ F_L \circ \dots . F_1 \circ F_0 \circ P(\mu(x))$$

- Many different neural operators correspond to different discretizations of the kernel layer.

Neural operator: Fourier NO

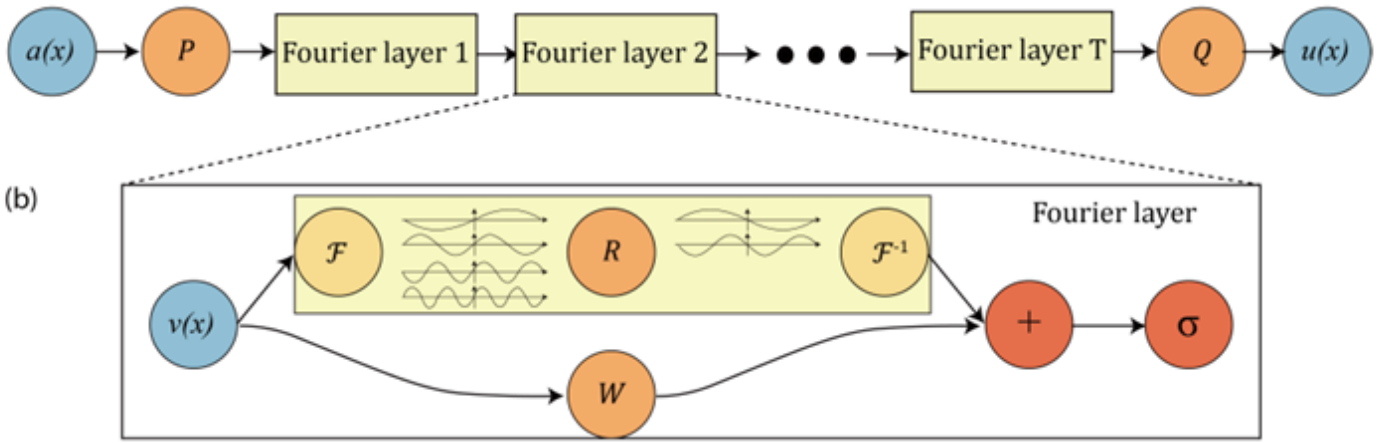
Fourier Neural Network (FNO)

The FNOs use neural operator layers with an integral kernel:

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

where $\mathcal{F}$ is the Fourier transform, and with R_θ 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.

Neural operator: other spectral approaches

General spectral layer

We approximate

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

by

$$\mathcal{K}(v)(x) = \sum_{i=1}^{N} \langle \theta_i v, \phi_i(x) \rangle_{L^2} \phi_i(x),$$

with $\theta_i \in \mathbb{R}^{n_v, n_v}$ and $\phi_i(x)$ a basis function.

- It is equivalent to transforming to a spectral domain, mixing the components of the function and changing the coefficients in the spectral domain.
- **Choice of basis**: Fourier (FNO), Wavelet (WNO), Tchebychev (Spectral NO), Prony Series (Laplace NO).

General idea

This type of NO combines local spatial transformation (component mixing) and local transformations in the spectral domain.

Application to numerical methods

Remark

The accuracy of PINNs or NO is insufficient for many applications. How to combine them with classical approaches?

Idea

Use parametric PINNs/NO to capture large families of solutions, and use the network as a prior of the solution in the numerical methods.

- We assume that we can quickly produce a prior $u_{nn}(x, \boldsymbol{\mu}; \theta)$, which is a parametric family of (approximate) solutions associated to the elliptic equation:

$$G(\boldsymbol{U}, \partial_x \boldsymbol{U}, \partial_{xx} \boldsymbol{U}, \boldsymbol{\mu}) = f(x).$$

- This prior is given by a PINN or by a NO.

Enhanced basis with PINNs

- The discontinuous Galerkin/finite element method relies on a local representation:

$$u_{|\Omega_k} = \sum_{i=1}^{q} \alpha_i \phi_i(x),$$

  with $\phi_i(x)$ a polynomial. The convergence rate is in Δx^q.
- FE assumes the continuity between the local representations at the interface between cells.

Idea

Enhance the basis using the prior. For example:

1. Additive modal version

$$(\phi_1(x), \dots, \phi_q(x)) = [u_{nn}(x, \boldsymbol{\mu}; \theta), 1, \dots \frac{(x - x_k)^{q-1}}{(q-1)!}]$$

2. Multiplicative modal version

$$(\phi_1(x), \dots, \phi_q(x)) = [u_{nn}(x, \boldsymbol{\mu}; \theta), (x - x_k)u_{nn}(x, \boldsymbol{\mu}; \theta), \dots \frac{(x - x_k)^{q}}{q!}u_{nn}(x, \boldsymbol{\mu}; \theta)]$$

3. Multiplicative nodal version

$$(\phi_1(x), \dots, \phi_q(x)) = [u_{nn}(x, \boldsymbol{\mu}; \theta)L(x, x_1), \dots, u_{nn}(x, \boldsymbol{\mu}; \theta)L(x, x_q)],$$

   with $L(x, x_i)$ the Lagrange Polynomial which satisfies $L(x_j, x_i) = \delta_{ij}$.

We can prove the convergence of such modified bases.

Application to elliptic problems

- First test:

$$-\partial_{xx} u = \alpha \sin(2\pi x) + \beta \sin(4\pi x) + \gamma \sin(8\pi x)$$

We train with $(a, b, c) \in [0, 1]^3$ and test with $(a, b, c) \in [0, 1.2]^3$.

Results with 20 cells:

Data set	inside training set	outside training set	inside/outside
Average gain vs FE	101	28.4	76.8
Average gain vs PINNS	6.7	7	6.8

- Second test:

$$v\partial_x u - \frac{v}{Pe}\partial_{xx} u = r.$$

We train with $r \in [1.5, 2.0]$, $v \in [1.5, 2.0]$ and $Pe \in [10, 120]$,
and we test with $r \in [1.5, 2.2]$, $v \in [1.5, 2.2]$ and $Pe \in [10, 150]$.

Results with 20 cells:

Data set:	inside training set	outside training set	inside/outside
Average gain	110	25	81.6

- Current work with Inria Pau: 2D extension, convergence theory, extension to time problems.

Euler-Poisson system in spherical geometry

- We consider the Euler-Poisson system in spherical geometry

$$\begin{cases} \partial_t \rho + \partial_r q = -\frac{2}{r} q, \\ \partial_t q + \partial_r \left(\frac{q^2}{\rho} + p \right) = -\frac{2}{r} \frac{q^2}{\rho} - \rho \partial_r \phi, \\ \partial_t E + \partial_r \left(\frac{q}{\rho}(E+p) \right) = -\frac{2}{r} \frac{q}{\rho}(E+p) - q \partial_r \phi, \\ \frac{1}{r^2} \partial_{rr}(r^2 \phi) = 4\pi G \rho, \end{cases}$$

- The steady solutions at rest are given by

$$q = 0; \qquad \partial_r p + \rho \partial_r \phi = 0; \qquad \partial_{rr}(r^2 \phi) = 4\pi r^2 G \rho.$$

- We consider a polytropic pressure law $p(\rho; \kappa, \gamma) = \kappa \rho^\gamma$ such that the steady solutions satisfy

$$\frac{d}{dr} \left(r^2 \kappa \gamma \rho^{\gamma - 2} \frac{d\rho}{dr} \right) = 4\pi r^2 G \rho,$$

- To simulate flow around a steady solution, we need a scheme that is very accurate when approximating the steady solution.

Euler-Poisson in spherical geometry: steady solution

Training takes about 10 minutes on an old GPU, with **no data**, only the PINN loss. This time, we have two parameters, κ and γ.

We take a quadrature of degree $n_Q = n_G + 1$.

Results for the **polytropic pressure law**

pts	error_{ϕ}^{h}	order	error_{ϕ}^{q}	order	error_{ϕ}^{E}	order	$\text{error}_{\bar{\phi}}^{h}$	order	gain	$\text{error}_{\bar{\phi}}^{q}$	order	gain	$\text{error}_{\bar{\phi}}^{E}$	order	gain
10	1.90e-01	—	1.84e-02	—	4.88e-01	—	5.84e-04	—	326.34	6.32e-03	—	2.92	1.46e-03	—	333.51
20	6.78e-02	1.49	7.60e-03	1.28	1.71e-01	1.51	2.73e-04	1.10	248.20	1.67e-03	1.92	4.55	6.84e-04	1.10	250.74
40	2.41e-02	1.49	2.93e-03	1.37	6.07e-02	1.50	1.01e-04	1.43	237.53	3.75e-04	2.15	7.80	2.54e-04	1.43	238.71
80	8.55e-03	1.50	1.16e-03	1.34	2.15e-02	1.50	3.64e-05	1.48	234.68	8.15e-05	2.20	14.23	9.12e-05	1.48	236.10
160	3.03e-03	1.50	4.64e-04	1.32	7.58e-03	1.51	1.17e-05	1.63	257.14	1.60e-05	2.35	28.97	2.94e-05	1.63	257.38

(a) errors with a one-element basis, $n_G = 1$

Euler-Poisson in spherical geometry: steady solution

Training takes about 10 minutes on an old GPU, with **no data**, only the PINN loss. This time, we have two parameters, κ and γ.

We take a quadrature of degree $n_Q = n_G + 1$.

Results for the **polytropic pressure law**

pts	error_{ϕ}^{h}	order	error_{ϕ}^{q}	order	error_{ϕ}^{E}	order	$\text{error}_{\bar{\phi}}^{h}$	order	gain	$\text{error}_{\bar{\phi}}^{q}$	order	gain	$\text{error}_{\bar{\phi}}^{E}$	order	gain
10	3.72e-03	—	5.34e-03	—	6.49e-03	—	3.74e-05	—	99.38	4.70e-05	—	113.63	9.19e-05	—	70.67
20	6.59e-04	2.50	1.21e-03	2.14	1.21e-03	2.42	7.00e-06	2.42	94.19	1.28e-05	1.87	94.14	1.68e-05	2.45	72.07
40	1.17e-04	2.49	2.27e-04	2.41	2.21e-04	2.45	1.27e-06	2.45	91.93	2.56e-06	2.33	88.59	3.07e-06	2.45	71.84
80	2.06e-05	2.51	4.05e-05	2.49	3.86e-05	2.52	2.24e-07	2.51	92.05	4.70e-07	2.45	86.03	5.45e-07	2.50	70.86
160	3.64e-06	2.51	7.15e-06	2.50	6.56e-06	2.56	3.90e-08	2.52	93.17	8.27e-08	2.51	86.41	9.50e-08	2.52	69.08

(b) errors with a two-element basis, $n_G = 2$

Euler-Poisson in spherical geometry: steady solution

Training takes about 10 minutes on an old GPU, with **no data**, only the PINN loss. This time, we have two parameters, κ and γ.

We take a quadrature of degree $n_Q = n_G + 1$.

Results for the **polytropic pressure law**

pts	error_{ϕ}^{h}	order	error_{ϕ}^{q}	order	error_{ϕ}^{E}	order	$\text{error}_{\bar{\phi}}^{h}$	order	gain	$\text{error}_{\bar{\phi}}^{q}$	order	gain	$\text{error}_{\bar{\phi}}^{E}$	order	gain
10	7.92e-06	—	5.39e-06	—	3.25e-04	—	3.68e-06	—	2.15	3.16e-06	—	1.71	8.16e-06	—	39.81
20	6.96e-07	3.51	9.10e-07	2.57	3.39e-05	3.26	3.60e-07	3.36	1.93	6.02e-07	2.39	1.51	7.41e-07	3.46	45.79
40	6.03e-08	3.53	9.46e-08	3.27	3.21e-06	3.40	3.26e-08	3.47	1.85	5.64e-08	3.42	1.68	7.74e-08	3.26	41.47
80	5.31e-09	3.51	7.97e-09	3.57	2.84e-07	3.50	2.98e-09	3.45	1.78	5.07e-09	3.47	1.57	7.09e-09	3.45	40.15
160	4.81e-10	3.46	7.26e-10	3.46	2.51e-08	3.50	2.74e-10	3.45	1.76	4.61e-10	3.46	1.57	6.46e-10	3.46	39.00

(c) errors with a three-element basis, $n_G = 3$

Euler-Poisson in spherical geometry: steady solution

Training takes about 10 minutes on an old GPU, with **no data**, only the PINN loss. This time, we have two parameters, κ and γ.

We take a quadrature of degree $n_Q = n_G + 1$.

Results for the **polytropic pressure law**

Statistics: gain with respect to the parameter space (from top to bottom: $n_G = 1, n_G = 2, n_G = 3$)

	min. gain	avg. gain	max. gain
ρ	22.21	412.57	6080.00
q	40.90	411.13	5384.43
E	22.25	411.40	6014.11

	min. gain	avg. gain	max. gain
ρ	6.57	154.29	1249.70
q	7.47	180.19	1317.09
E	6.14	110.27	627.65

	min. gain	avg. gain	max. gain
ρ	0.17	12.80	102.00
q	0.20	14.12	109.50
E	3.69	48.66	433.81

Conclusion

Conclusion

Classical Numerical methods

We use a parametric function that is linear with respect to the parameters, plug it in the equation, and solve for the parameters.

PINNs

We use a parametric function that is nonlinear with respect to the parameters, plug it in the equation, and replace algebraic equations with optimization problems. NO generalize PINNs.

Hybridation

These PINNs/NO are not very accurate, but become interesting for large-dimensional problems. We showed a promising approach obtained by hybridizing the ML and the classical methods.

Advert :)

in our Inria team, we have regularly PhD and Post-doc position on these thematics.