DG schemes for hyperbolic sytems with source terms, enhanced by neural networks

E. Franck[1,2], Victor Michel-Dansac[1,2], Laurent Navoret[1,2]

Shark-FV conference 2023

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

Outline

Introduction

Nonlinar conservation laws and WB schemes

- We consider the following type of models (like everybody here):

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

- We are interested by the simulation of flows such as:

$$\partial_x \mathbf{F}(\mathbf{U}) = \mathbb{S}(\mathbf{U}) + \varepsilon \mathbf{P}(t, x)$$

Numerical difficulties

We consider a scheme of order q. For a equilibrium $\partial_x \mathbf{F}(\mathbf{U}) = \mathbf{S}(\mathbf{U})$ we have

$$\partial_x \mathbf{F}(\mathbf{U}_h) = \mathbb{S}(\mathbf{U}_h) + C\Delta x^q \mathbf{Q}_h(t, x).$$

if $\varepsilon < C\Delta x^q$ our scheme will not correctly capture perturbed flows.

WB and A-WB schemes

For a equilibrium $\partial_x \mathbf{F}(\mathbf{U}) = \mathbf{S}(\mathbf{U})$, a Well-Balanced scheme is such that $\partial_x \mathbf{F}(\mathbf{U}_h) = \mathbf{S}(\mathbf{U}_h)$, and an Approximately Well-Balanced scheme is such that

$$\partial_x \mathbf{F}(\mathbf{U}_h) = \mathbf{S}(\mathbf{U}_h) + C_2\Delta x^{q_2} \mathbf{Q}_h(t, x)$$

with $q_2 > q$ or $C_2 \ll C$.

- WB et A-WB make it possible to capture these perturbed flows.

Non-exhaustive state of the art

Many people have been working on approximately and exactly well-balanced schemes, including many in the audience!

For instance, we mention:

- Exactly or approximately well-balanced schemes for the shallow water equations:
  - Audusse, Bouchut, ...
  - Bermúdez, Vázquez, ...
  - Berthon, Chalons, Desveaux, Michel-Dansac,...
  - Clain, Figueiredo, ...
  - the Málaga group: Castro, Parés, ...
  - Noelle, Shu, Xing, ...
- Exactly or approximately well-balanced schemes for the Euler equations:
  - Franck, Mendoza, ...
  - Käppeli, Mishra, ...
  - Thomann, Klingenberg, Puppo, ...
- Exactly or approximately well-balanced schemes for other systems:
  - Busto, Dumbser, Gaburro, ...
  - Chertock, Kurganov, ...

DG schemes

- We recall quickly the Discontinuous Galerkin method.

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

$$\int_{\Omega_j} \partial_t \mathbf{U} \phi dx + \int_{\Omega_j} \partial_x \mathbf{F}(\mathbf{U}) \phi dx = \int_{\Omega_j} \mathbf{S}(\mathbf{U}) \phi dx$$

- In each cell we consider a discrete vectorial polynomial space: $V_h = Span(\phi_1(x), ...\phi_q(x))$ and we use

$$\mathbf{U}_{|\Omega_j}(t,x) = \sum_{i=1}^{q} \alpha_i(t)\phi_i(x) \in V_h$$

and

$$\phi = \phi_1, ...\phi = \phi_q$$

- We obtain a matrix-vector system of size $q \times q$

$$\boxed{\mathcal{M}\partial_t \boldsymbol{\alpha}(t) + \mathcal{K}(\boldsymbol{\alpha}(t)) = \mathcal{S}(\boldsymbol{\alpha}(t))}$$

with $\mathcal{S}, \mathcal{K} \in \mathbb{R}^q$ and $\mathcal{M} \in \mathbb{R}^{q \times q}$

DG scheme with prior on the equilibrium

Main idea

- We consider a family of equilibria
$$\mathbf{U}_{eq}(x;\boldsymbol{\mu})$$
indexed by some parameters $\boldsymbol{\mu}$.
- We assuming that we are able to produce an approximation of this equilibrium family, called the prior:
$$\mathbf{U}_{\theta}(x;\boldsymbol{\mu})$$

Idea

Incorporate this prior in the local basis to obtain an A-WB scheme with a much higher accuracy around the equilibrium.

Questions

- What is the potential new basis (next slide)?
- Can we ensure the convergence of the new scheme (this section)?
- How to construct this prior (next section)?

Proposed basis

Idea

Introduce the prior on the equilibrium into the DG basis, to generate non-polynomial basis.

Basis with multiplicative prior

$$V_h^1 = \text{Span}\left(\mathbf{U}_\theta(x;\boldsymbol{\mu}), \mathbf{U}_\theta(x;\boldsymbol{\mu})(x-x_j), ..., \mathbf{U}_\theta(x;\boldsymbol{\mu})\frac{(x-x_j)^k}{k!} \right)$$

Basis with additive prior

- Solution 1:

$$V_h^2 = \text{Span}\left(\mathbf{U}_\theta(x;\boldsymbol{\mu}), 1, ..., \frac{(x-x_j)^{k-1}}{(k-1)!} \right)$$

- Solution 2:

$$V_h^3 = \text{Span}\left(\mathbf{U}_\theta(x;\boldsymbol{\mu}), (x-x_j), ..., \frac{(x-x_j)^k}{(k)!} \right)$$

- Does DG converge with non-polynomial bases?

Convergence within the Yuan-Shu framework I

anShu06 L. Yuan and C.-W. Shu: *Discontinuous Galerkin method based on non-polynomial approximation spaces*, JCP 2006.

Main result I of [YuanShu06]

We consider a basis $(v_1, ..v_K)$ of the space V_h. If there are constant a_{ik} and b_i **independant of the size of the cell** Δx_j, and if we have

$$|v_i(x) - \sum_{k=1}^{K} a_{ik}(x - x_i)^k| \leq b_i(\Delta x_j)^{K+1} \quad (1)$$

then for any function $u \in H^{K+1}(\Omega_j)$, there exists $v_h \in V_h$ and

$$|v_h - u_h| \leq C|u|_{H^{K+1}(\Omega_j)}(\Delta x_j)^{K+\frac{1}{2}}$$

Main result II of [YuanShu06]

With the first result, we can prove the convergence (with additional steps) of the DG scheme using the V_h basis.

Convergence with the Yuan-Shu framework II

Result in the scalar case

We assume that $\mathbf{u}_\theta(x;\boldsymbol{\mu}) \in C^p(\Omega)$ with $p \geq K+1$. Then, the previously proposed bases satisfy the assumption of [YuanShu06], and the DG scheme converges.

- Example of proof for V_h^1.
- Since the neural network is $C^{K+1}(\mathbb{R})$, we can write a Taylor series, to obtain:

$$u_\theta(x) = u(x_j) + (x - x_j)u_\theta^{'} + ... + \frac{u^{(K)}(x_j)}{K!}(x - x_j)^K + \frac{u^{(K+1)}(c)}{(K+1)!}$$

with $c \in [x_j, x]$. We then get:

$$\begin{pmatrix} u_\theta(x) \\ u_\theta(x)(x-x_j) \\ ... \\ u_\theta(x)(x-x_j)^K \end{pmatrix} = \underbrace{\begin{pmatrix} u_\theta(x_j) & u_\theta^{'}(x_j) & ... & \frac{u_\theta^{(K)}(x_j)}{K!} \\ 0 & u_\theta(x_j) & ... & \frac{u_\theta^{(K-1)}(x_j)}{(K-1)!} \\ ... & & & \\ 0 & 0 & ... & u_\theta(x_j) \end{pmatrix}}_{A} \begin{pmatrix} 1 \\ (x-x_j) \\ ... \\ (x-x_j)^K \end{pmatrix} + \underbrace{\begin{pmatrix} \frac{u_\theta^{K+1}(c)}{(K+1)!} \\ \frac{u_\theta^{K}(c)}{(K)!} \\ ... \\ 1 \end{pmatrix}}_{b}$$

- It easy to see that the matrix A, its inverse A^{-1}, and the vector b are independent from Δx_j. Therefore, the assumption is verified.

Specific estimate I

- **Problem**: the previous approach does not give the expected gain associated with these bases.
- We propose to compute a specific estimate using these bases.
- **Current work**: we obtained the projector estimate for the basis V_h^1.
- **Following**: finish the global proof (we do not expect complications) and do the same for other basis.

First lemma

We consider a basis $(v_1, ..v_K)$ of the space V_h^1 and assume that $u_\theta(x;\boldsymbol{\mu}) \in C^p(\Omega)$

$$u_\theta(x;\boldsymbol{\mu})^2 > \alpha_0, \quad \forall x \in \Omega$$

For any function $u \in H^{K+1}(\Omega_j)$, the L^2 projector on V_h^1, $P_h(u) \in V_h^1$, satisfies

$$|u - P_h(u)| \leq C \left| \frac{u(x)}{u_\theta(x,\boldsymbol{\mu})} \right|_{H^{K+1}(\Omega_j)} (\Delta x_j)^{K+\frac{1}{2}} |u_\theta(x,\boldsymbol{\mu})|$$

- Key point: the pointwise interpolation and L^2 projector associated with this basis can be rewritten as the classical one applied at $\frac{u(x)}{u_\theta(x,\boldsymbol{\mu})}$ and multiplied by $u_\theta(x,\boldsymbol{\mu})$

Specific estimate II

Second lemma

Under the same assumptions than the previous lemma, for the basis $V_h^{mod,3}$, we obtain that for any function $u \in H^{K+1}(\Omega)$

$$|u - P_h(u)|_{L^2(\Omega)} \leq C \left| \frac{u(x)}{u_\theta(x, \boldsymbol{\mu})} \right|_{H^{K+\frac{1}{2}}(\Omega)} (\Delta x)^{K+1} \parallel u_\theta(x, \boldsymbol{\mu}) \parallel_\infty$$

- **Idea of proof**:
  - sum the previous result on the cells
  - use that the prior is $C^p(\Omega)$ to bound $|u_\theta(x, \boldsymbol{\mu})|$ locally in each cell
  - use the assumption $C_1 \Delta x \leq \Delta x_j \leq C_2 \Delta x$
- **Following**: use this estimation to prove the convergence
- do the same for the other bases

Key point

- If the prior is good, then $\left| \frac{u(x)}{u_\theta(x, \boldsymbol{\mu})} \right|_{H^{K+\frac{1}{2}}(\Omega)}$ is small, and so is the error.
- For other bases, we expect an error in $\left| u(x) - u_\theta(x, \boldsymbol{\mu}) \right|_{H^{K+\frac{1}{2}}(\Omega)}$

Prior and PINNs

Supervised machine learning and classical models

- Supervised learning: construct models $f : \mathbb{R}^d \to \mathbb{R}^m$ like

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

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

$$\min_\theta \sum_i^n L(f(\mathbf{x}_i), \mathbf{y}_i),$$

with L a loss (cost) function.

- **Linear regression**: We choose

$$f_\theta = A\mathbf{x} + \mathbf{b}$$

with $\theta = (A, \mathbf{b})$, $A \in \mathbb{R}^{m,d}$ and $b \in \mathbb{R}^m$.

- **polynomial regression**: We choose

$$f_\theta = \sum_{i=1}^{dq} \theta_i P_i(\mathbf{x})$$

with q the order of the polynomial. Huge number of coefficients for large dimensions.

- **kernel regression**: theory using Hilbert space with reproducing kernel.
- In all the case, we obtain convex optimization.

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 $\sigma(\cdot)$ a nonlinear function applied component by component.

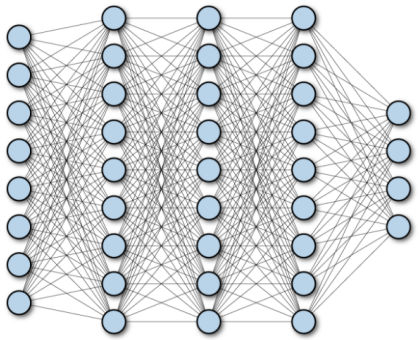
Neural network

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

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

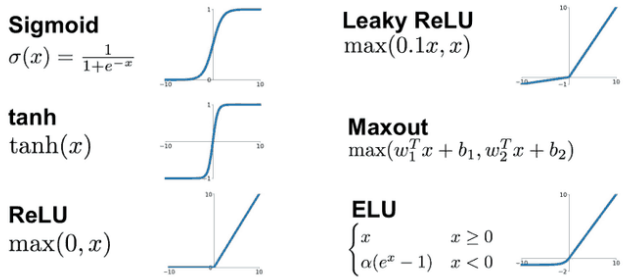
with θ the trainable parameters composed of all the matrices $A_{l,l+1}$ and biases $\mathbf{b}_l$.

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

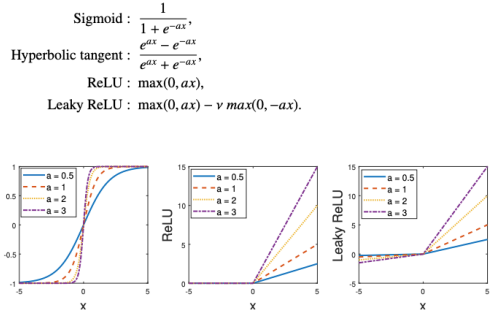


Activation functions and gradients

- The local nonlinear functions are called activation functions.
- Exemple (site MonCoachdata):



- It is possible to use adaptive activation functions (whose parameters are also learned).



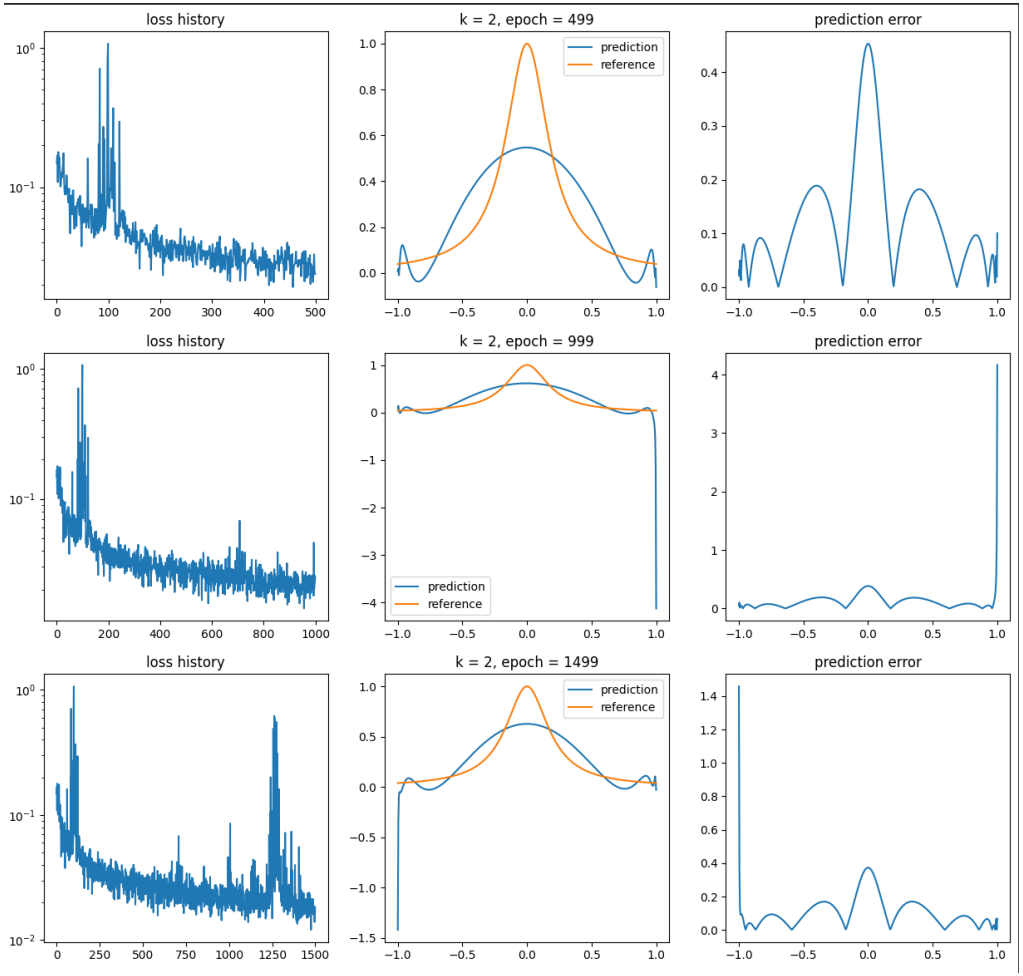
Key point

According to the activation function, the neural network are $C^p(\mathbb{R}^d)$ functions

- We train the Neural network with a gradient-type approach. This generates non-convex optimization problems.

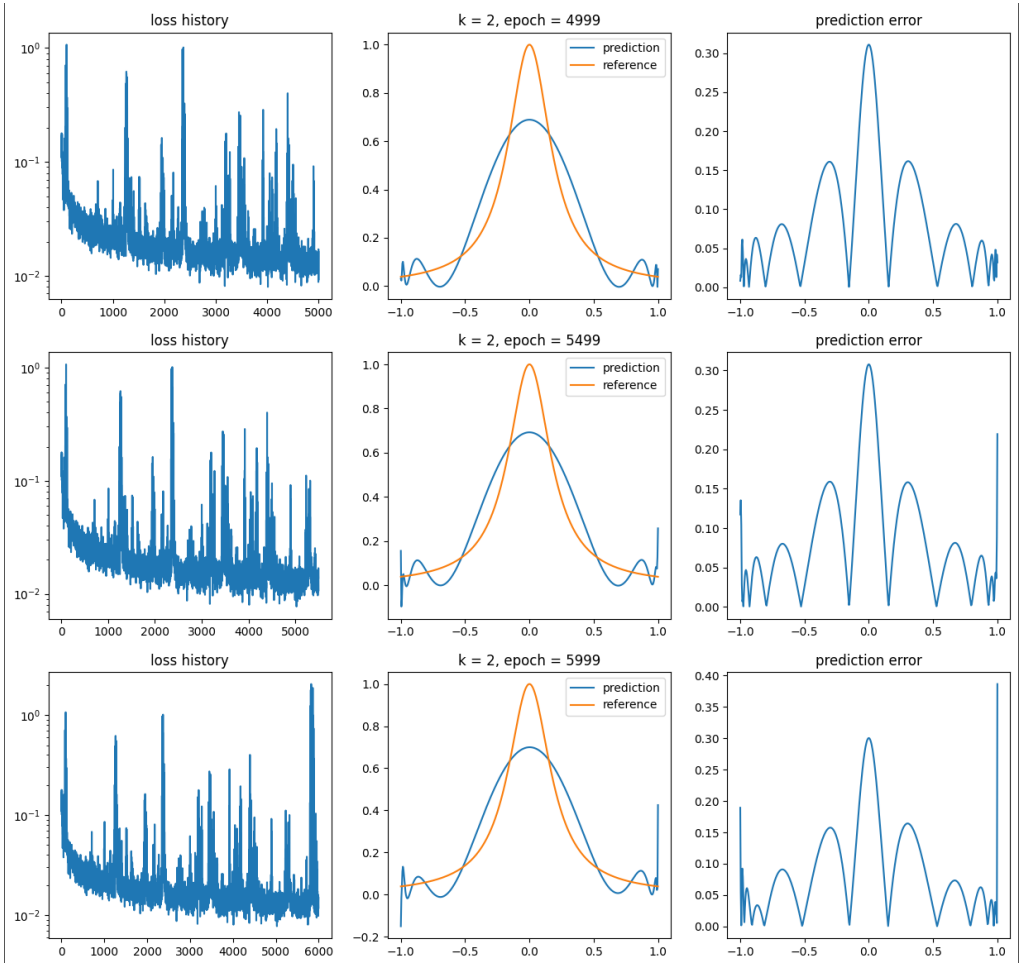
Neural network vs Polynomial

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



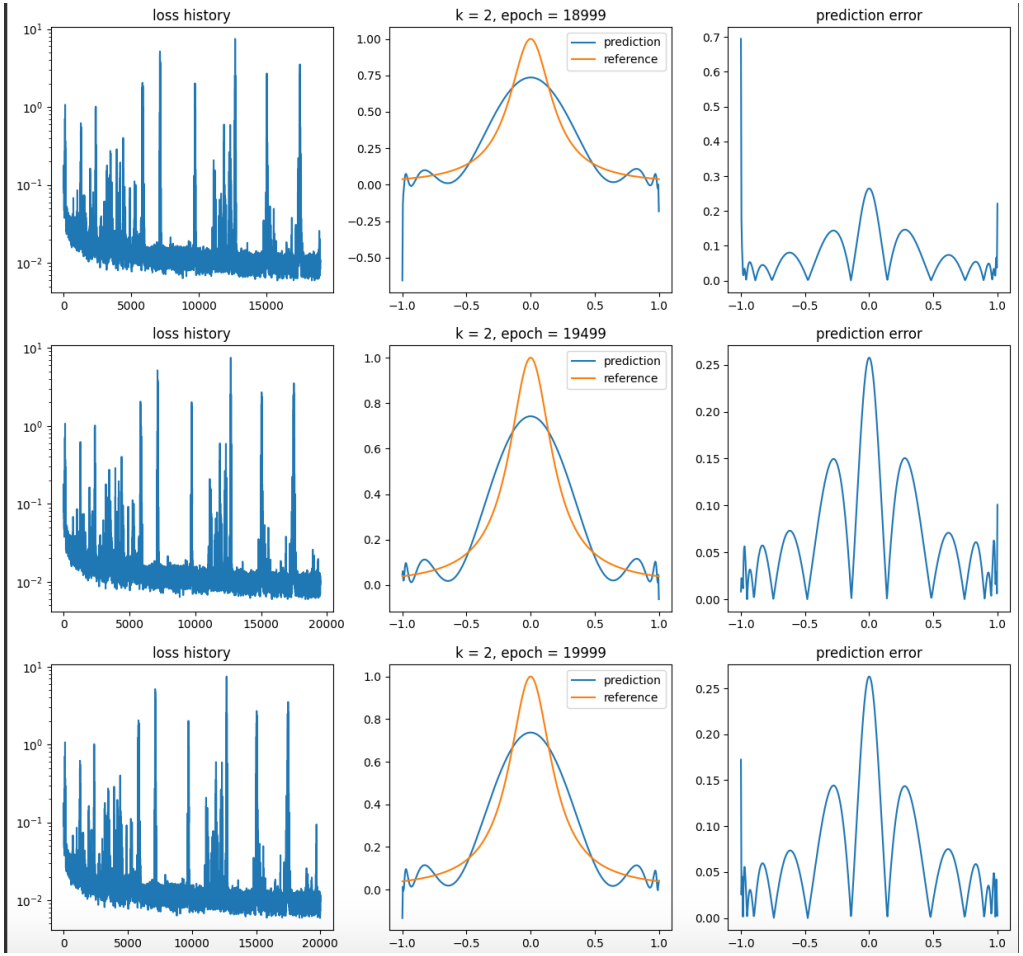
Neural network vs Polynomial

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



Neural network vs Polynomial

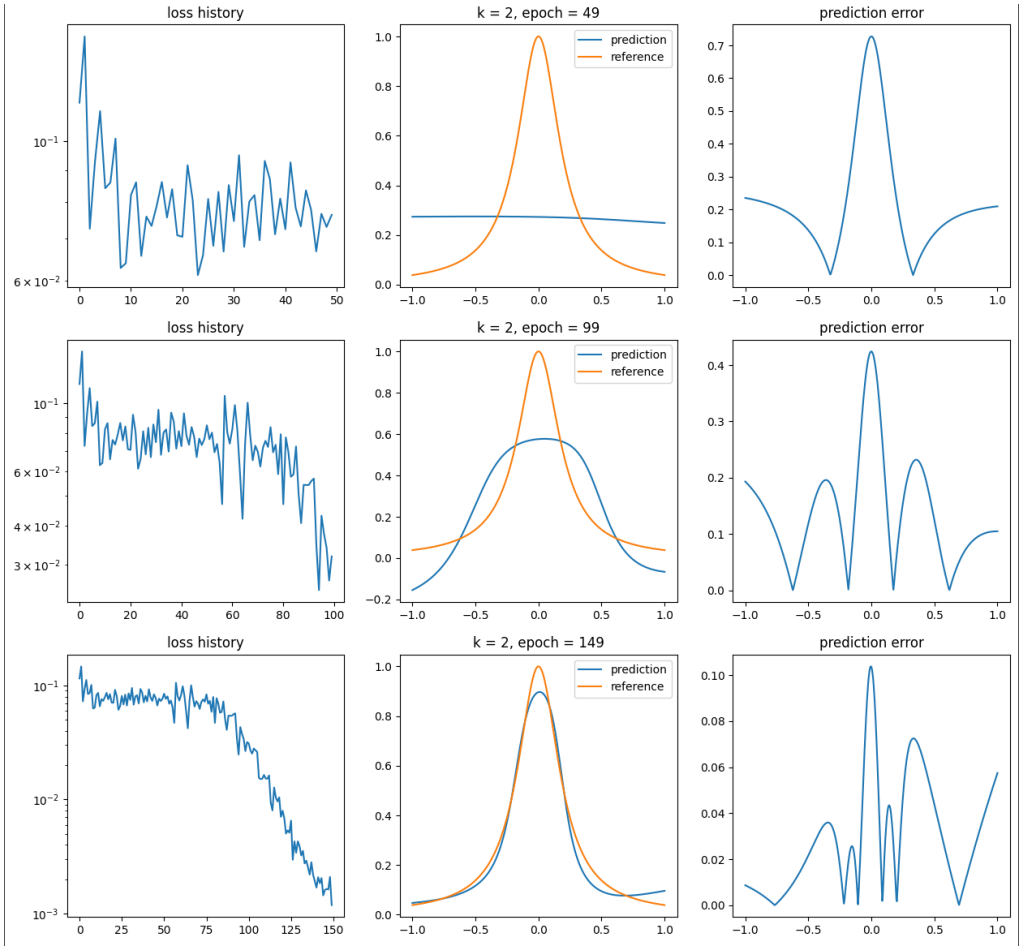
- We compare over-parametrized NN and polynomial regression on the Runge function.
- 120 data and approximately 800 parameters in each model.



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

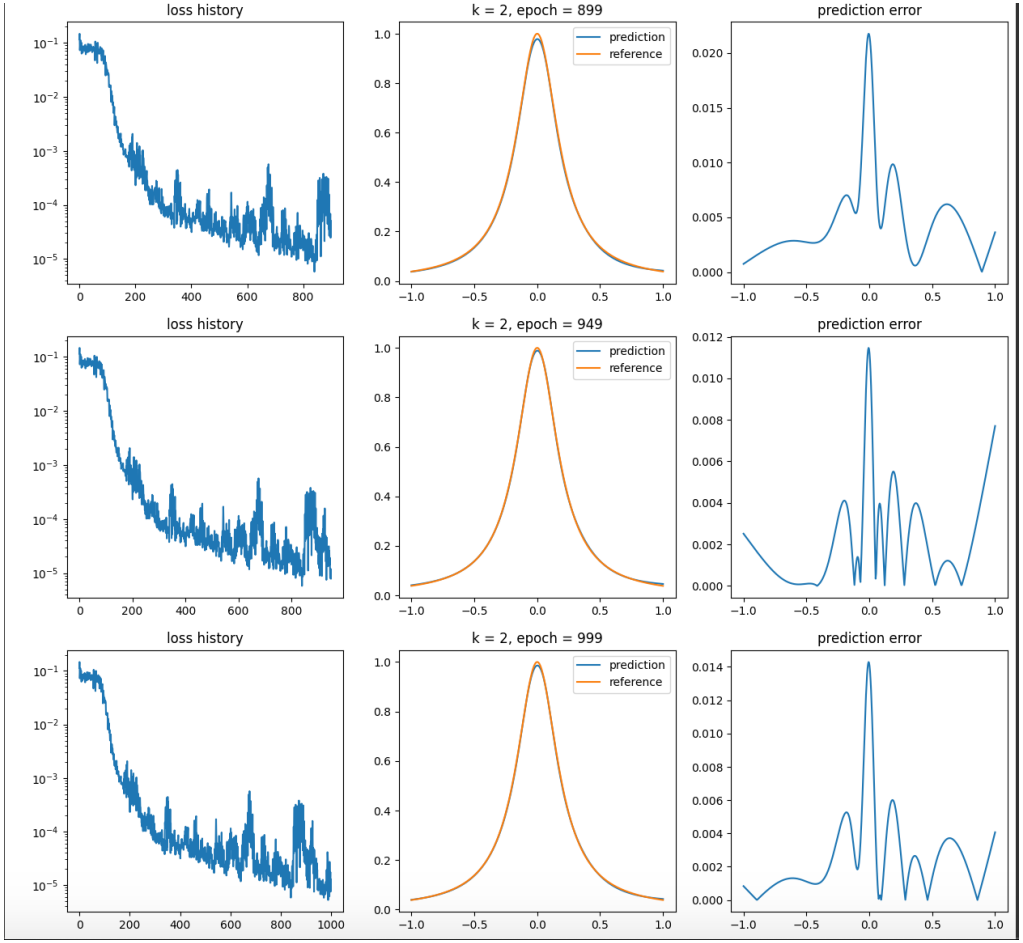
Neural network vs Polynomial

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



Neural network vs Polynomial

- We 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. This property seems to be very helpful for the generalization.

PINN's I

- We solve PDEs of the form:

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

  with the parameters

$$\boldsymbol{\mu} = (\boldsymbol{\alpha}, \boldsymbol{\beta})$$

- The first idea comes from the remark that neural networks are smooth functions of the inputs. Since their derivatives are easily computable by automatic differentiation, ANNs are possible objects to approximate PDE solution.
- A PINNs is a neural network with inputs (t, x), denoted by $\mathbf{U}_\theta(t, x)$.

Basic approach

If we have data $\boldsymbol{U}_i^n$ approximating the solution at the points (x_i, t_n), we will learn the weights of the NN by minimizing the loss:

$$\min_\theta J_{data}(\theta) = \min_\theta \sum_{n=1}^{N_{data}} \sum_{i=1}^{N_{data}} |\boldsymbol{U}_\theta(t_n, x_i) - \boldsymbol{U}_i^n|_2^2$$

How can we do that without data or with few data??

PINNs approach

Since we can differentiate the NN, we compute the PDE residual and check to what extent it is a solution of the PDE. **Main idea**: Learn using this property.

- We define the residual:

$$R(t,x) = |\partial_t \boldsymbol{U}_\theta - \mathcal{N}(\boldsymbol{U}_\theta, \partial_x \boldsymbol{U}_\theta, \partial_{xx} \boldsymbol{U}_\theta, \boldsymbol{\beta})|$$

- To learn $\mathbf{U}_\theta(t,x)$, we minimize:

$$\min_\theta J_{data}(\theta) + J_r(\theta) + J_b(\theta) + J_i(\theta)$$

with

$$J_r(\theta) = \int_0^T \int_\Omega |R(t,x)|_2^2 dxdt$$

and

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

Question

How to compute the integrals of the residuals?

How to compute the integrals of the residuals?

- Quadrature rule. Limited for large domains and small dimension
- Quadrature rule + mesh. Requires a grid and limited to small dimensions.
- Monte-Carlo approach. Slow convergence but no mesh and no dimension issues.

- 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 allows to computie 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 have

$$\left| \frac{1}{N} \sum_{i=1}^{N} f(x_i) - \int_\Omega f(x)dx \right| = O\left(\frac{Var(\mathcal{U}_\Omega)}{\sqrt{N}} \right)$$

  with x_i points sampled uniformly on Ω.

PINN's III

- Applying the MC method to the PINNs loss, we obtain the following minimization problem:

Final PINNs minimization

$$\min_{\theta} J_{data}(\theta) + J_r(\theta) + J_b(\theta) + J_i(\theta),$$

with

$$J_r(\theta) = \sum_{n=1}^{N} \sum_{i=1}^{N} |R(t_n, x_i)|_2^2$$

with (t_n, x_i) sampled uniformly and

$$J_b(\theta) = \sum_{n=1}^{N_b} \sum_{i=1}^{N_b} |\mathbf{U}_\theta(t_n, x_i) - g(x_i)|_2^2, \quad J_i(\theta) = \sum_{i=1}^{N_i} \mathbf{U}_\theta(0, x_i) - \mathbf{U}_0(x_i)|_2^2$$

- These loss functions can be interpreted as a regularization of classical learning which uses data.
- To avoid loss for the BC and initial condition, we use:

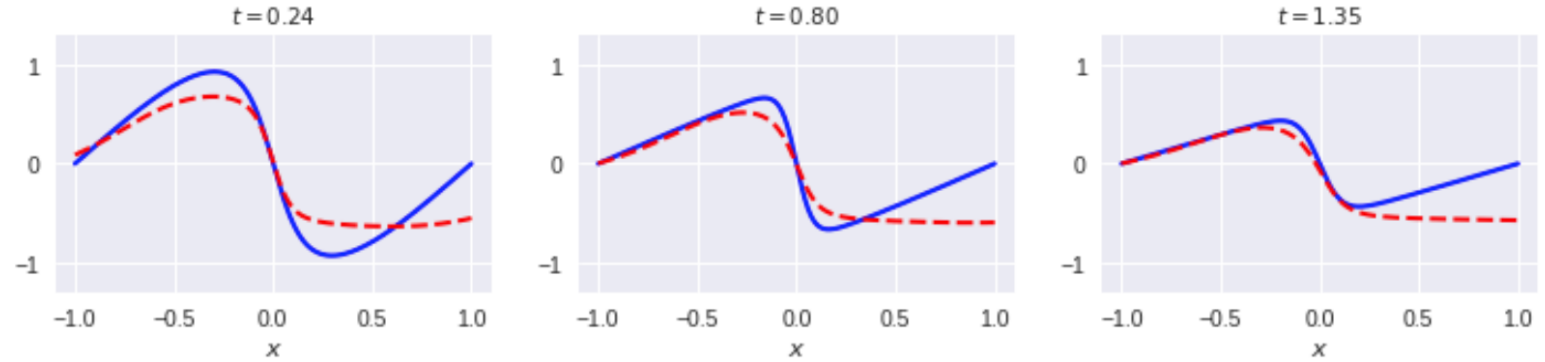
$$\bar{u}_\theta(t, x) = u_0(x) + t(\phi(x) * u_\theta(x))$$

with $\phi(x) = g(x)$ on the boundary and some value within the domain.

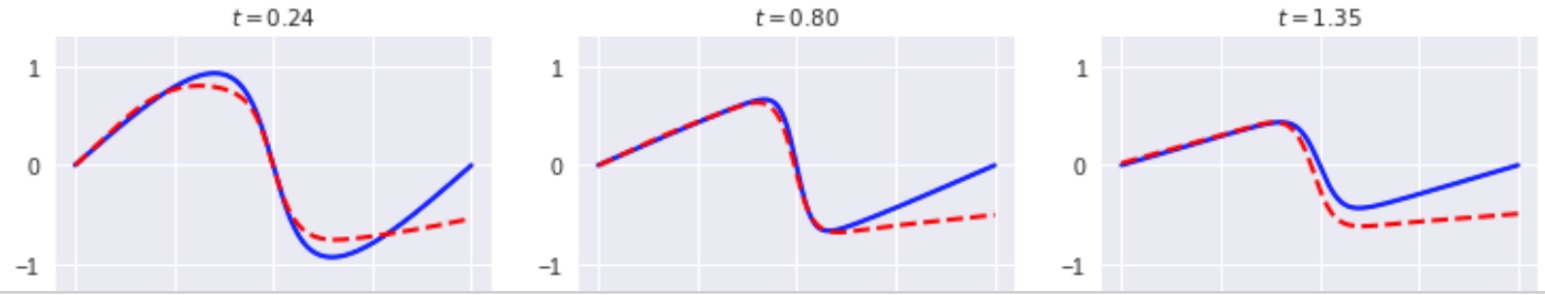
Example: Burgers equation

- 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
```



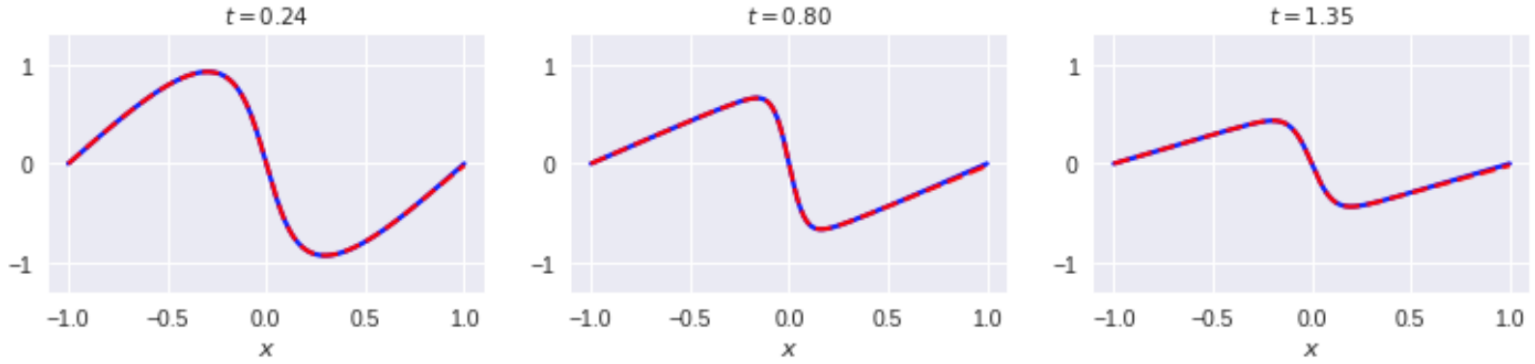

```
iter =  400
loss = 0.0318
L2 error: 4.0850e-01
```



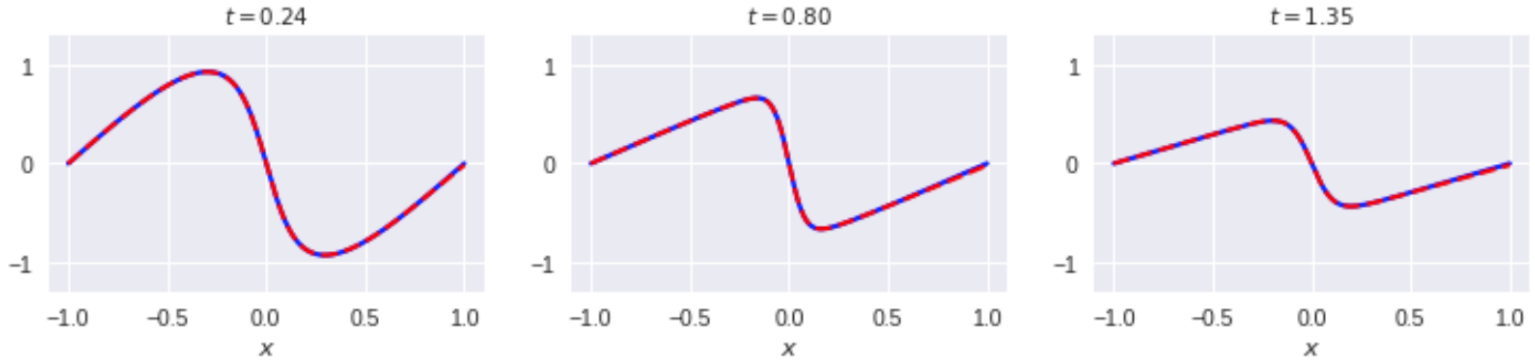
Example: Burgers equation

- 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
```



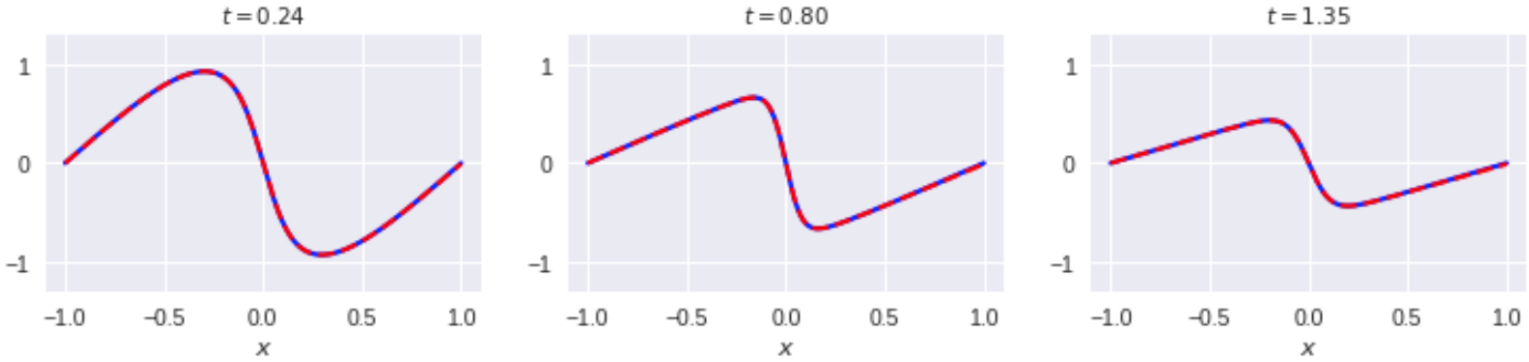

```
iter =  2200
loss = 0.0000
L2 error: 8.2614e-03
```



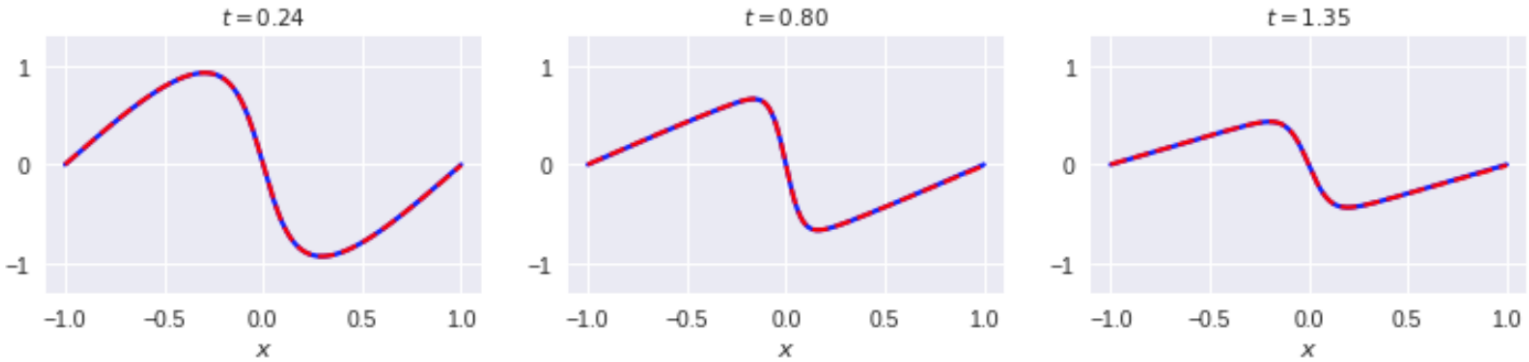
Example: Burgers equation

- 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
```



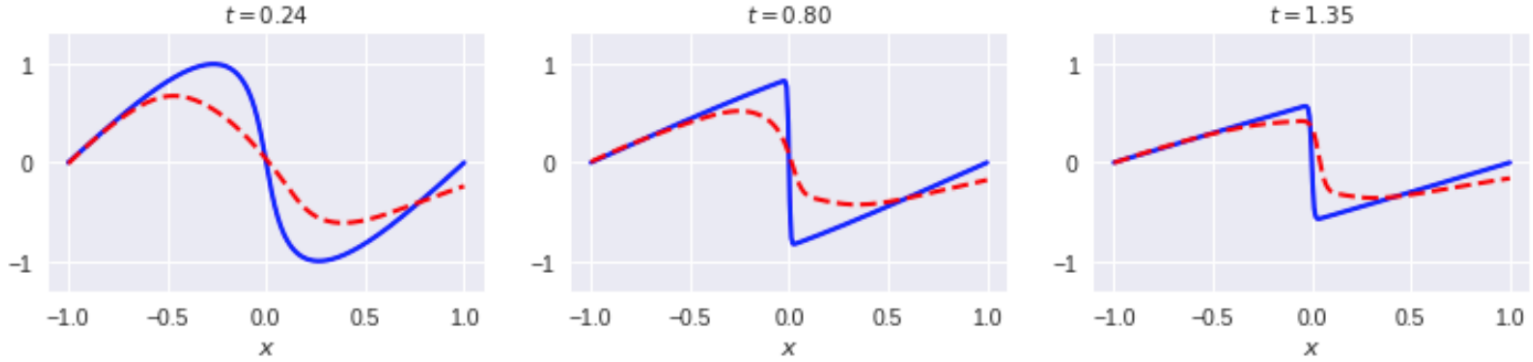

```
iter =  5000
loss = 0.0000
L2 error: 4.7307e-03
```



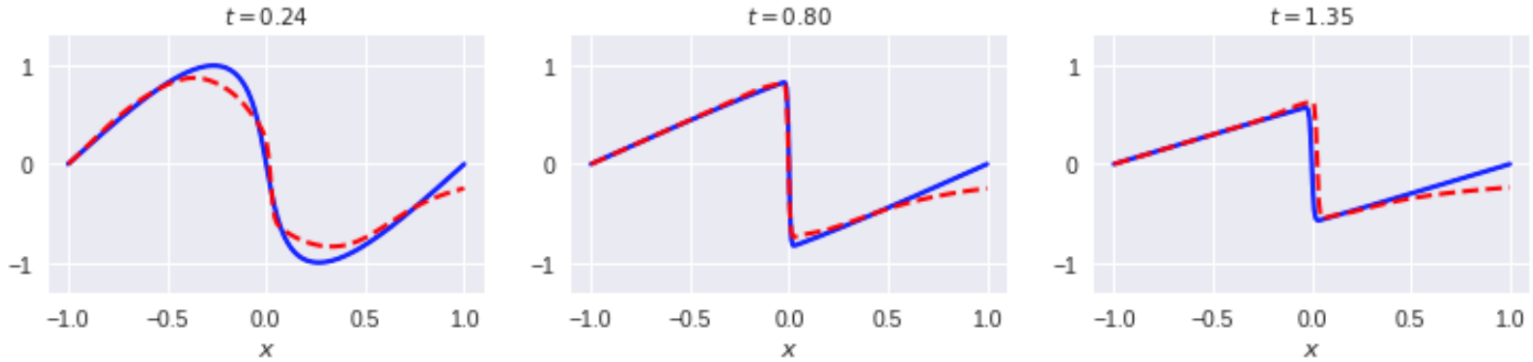
Example: Burgers equation

- 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
```



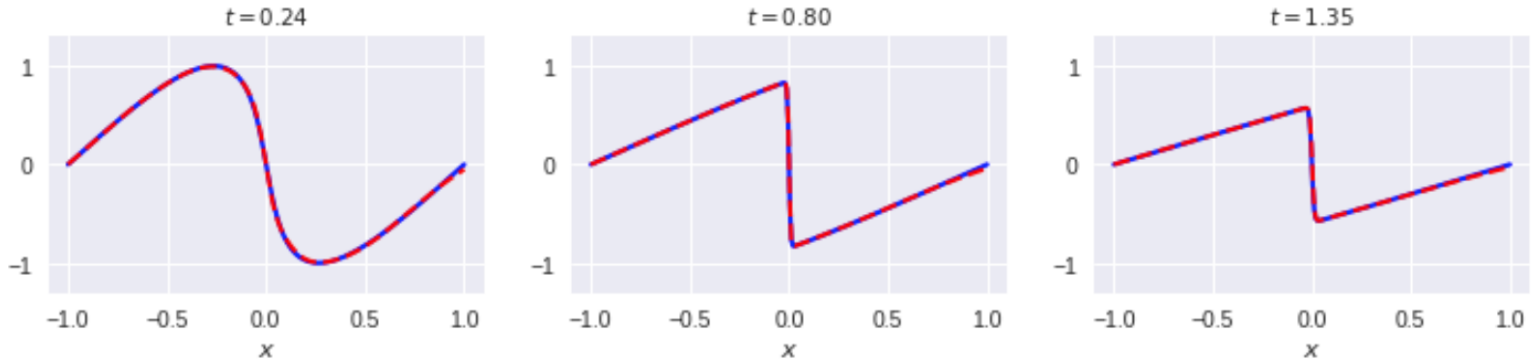

```
iter =  800
loss = 0.0233
L2 error: 2.0901e-01
```



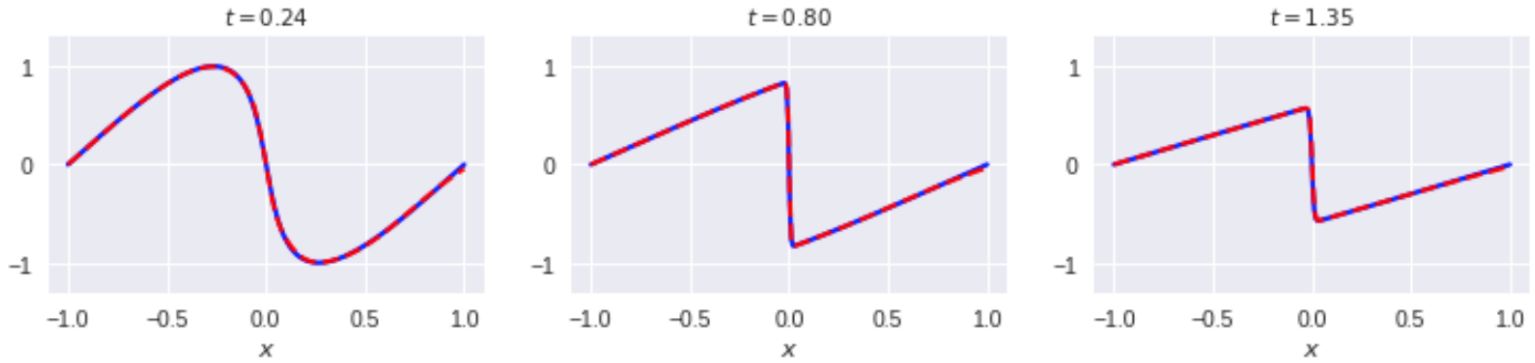
Example: Burgers equation

- 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
```



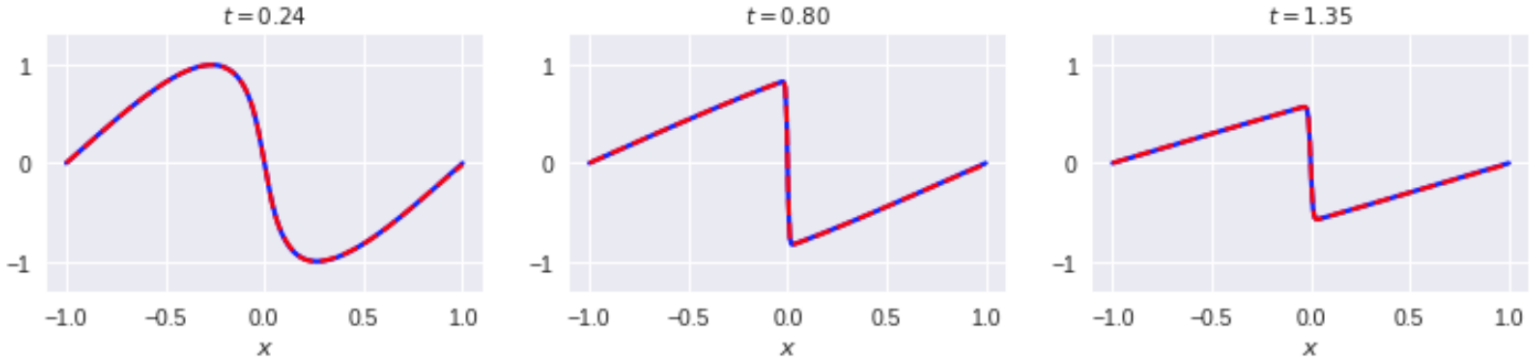

```
iter =  2200
loss = 0.0002
L2 error: 1.7773e-02
```



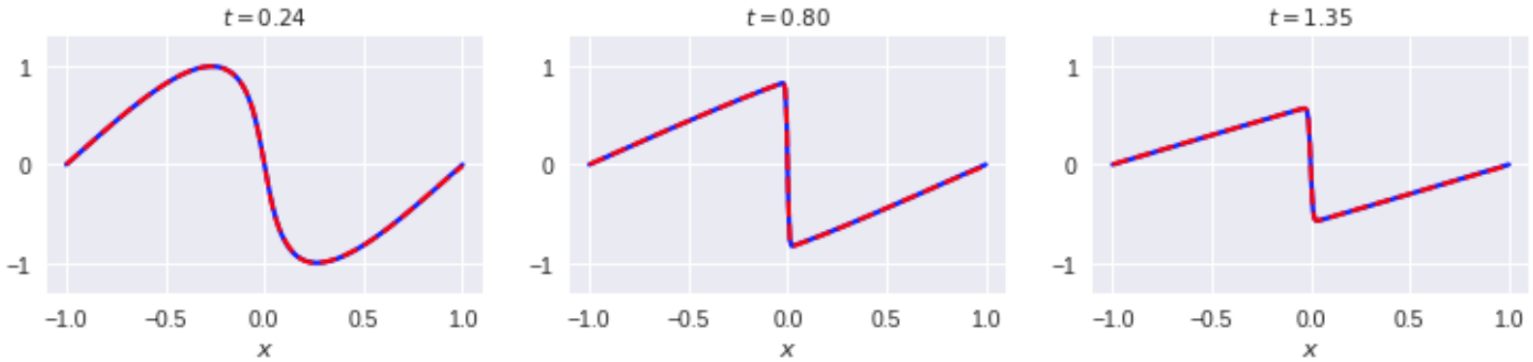
Example: Burgers equation

- 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
```



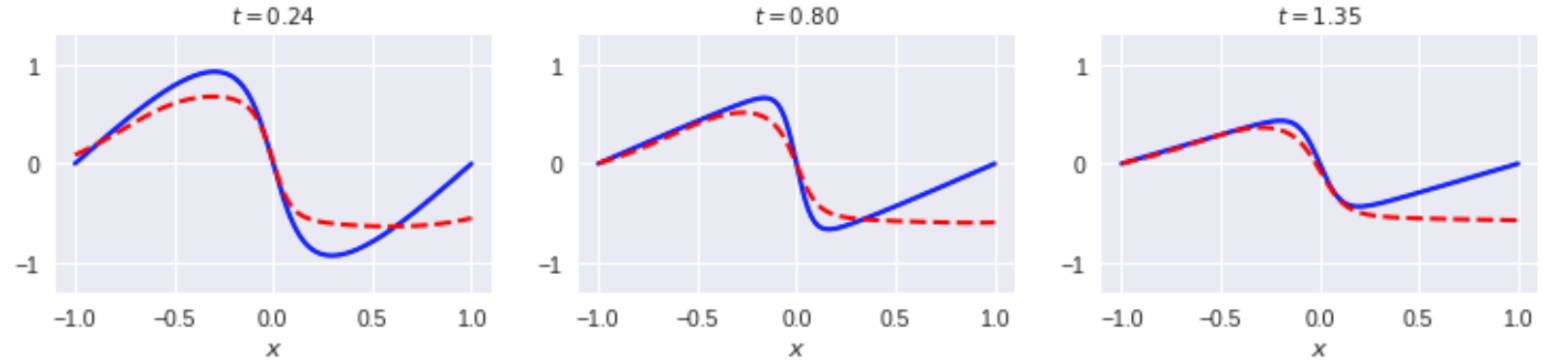

```
iter =  5000
loss = 0.0001
L2 error: 5.2593e-03
```



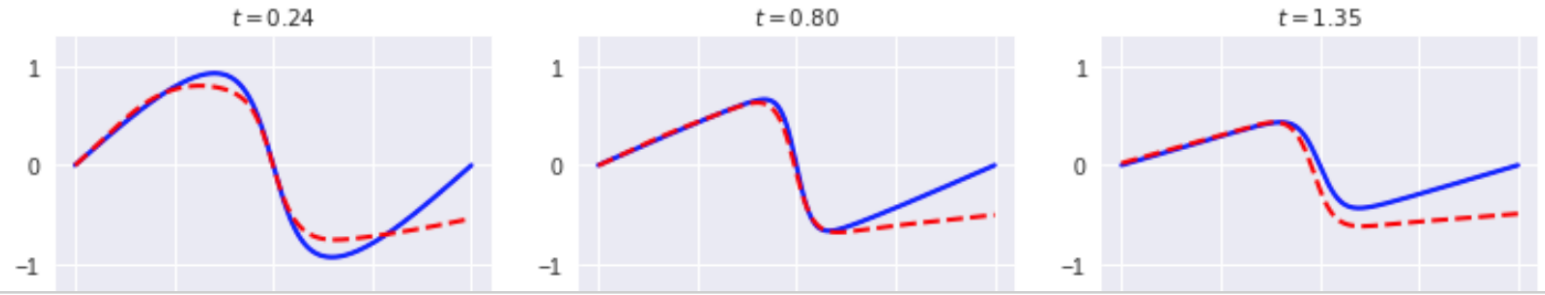
Example: Burgers equation

- 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
```



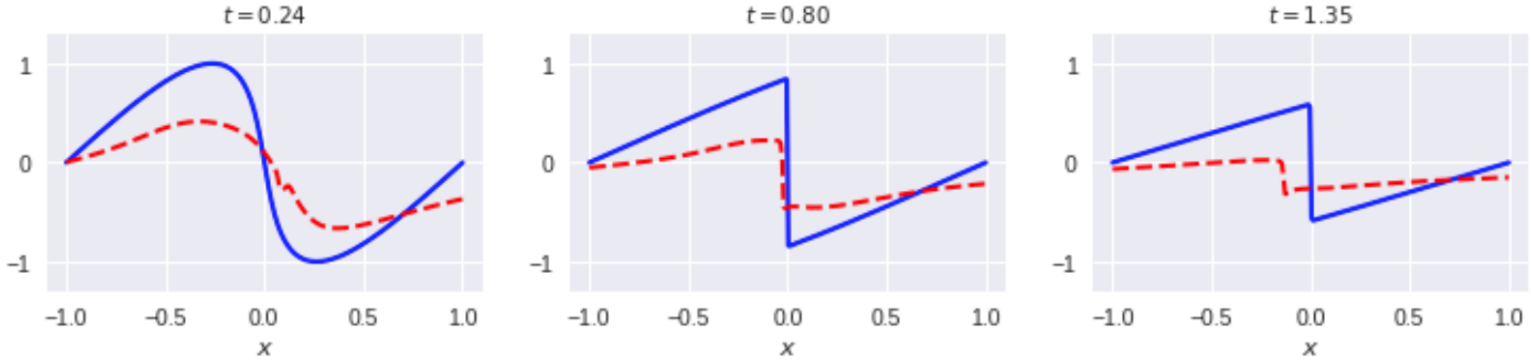

```
iter =  400
loss = 0.0318
L2 error: 4.0850e-01
```



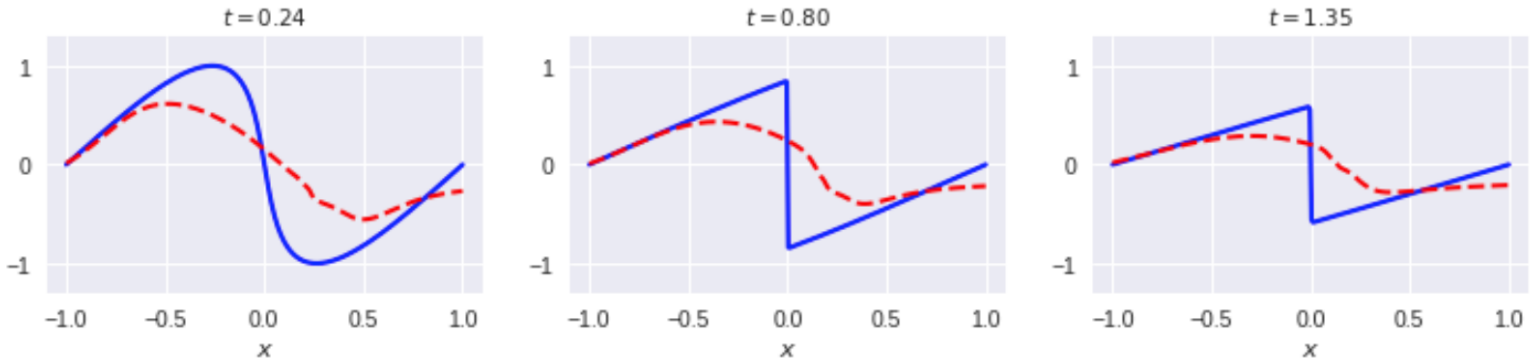
Example: Burgers equation

- 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
```



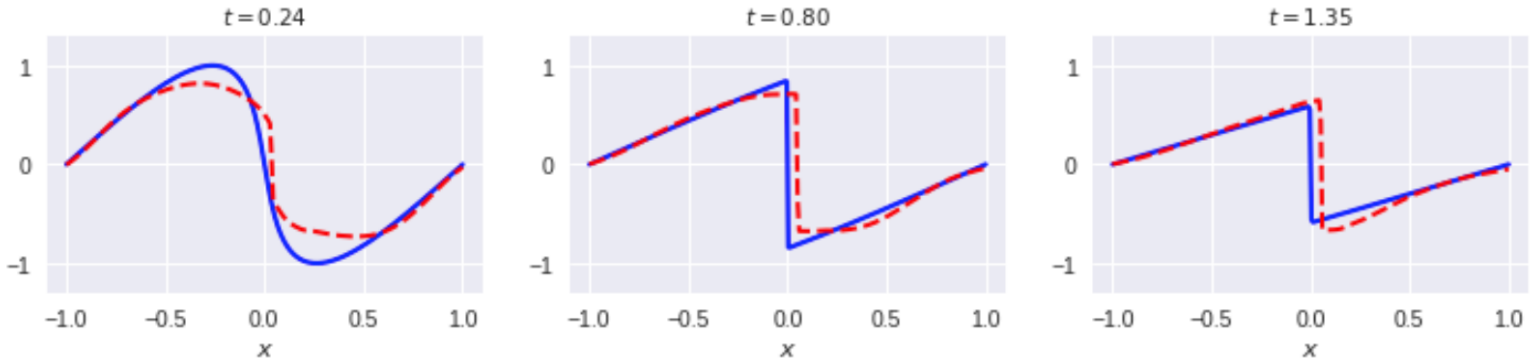

```
iter =  2200
loss = 0.1361
L2 error: 6.0138e-01
```



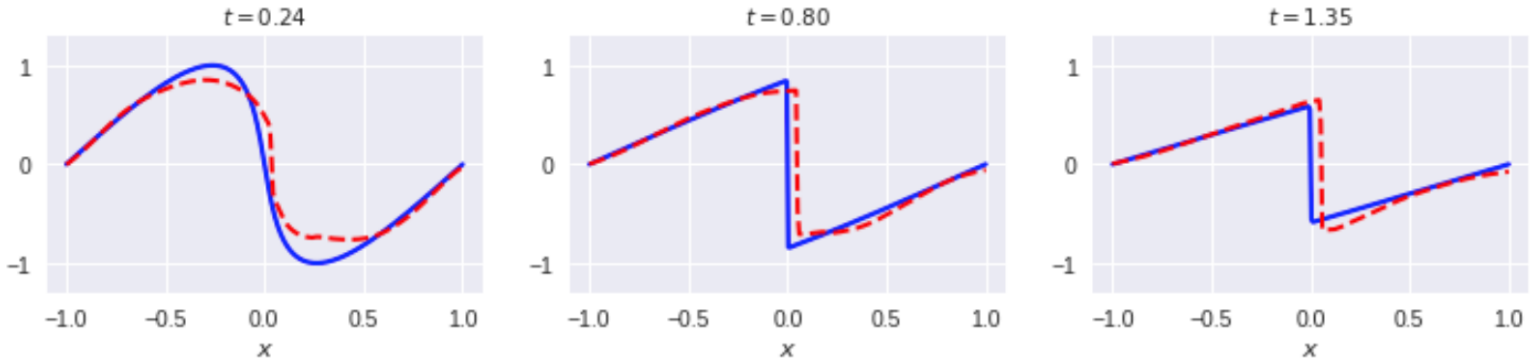
Example: Burgers equation

- 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
```



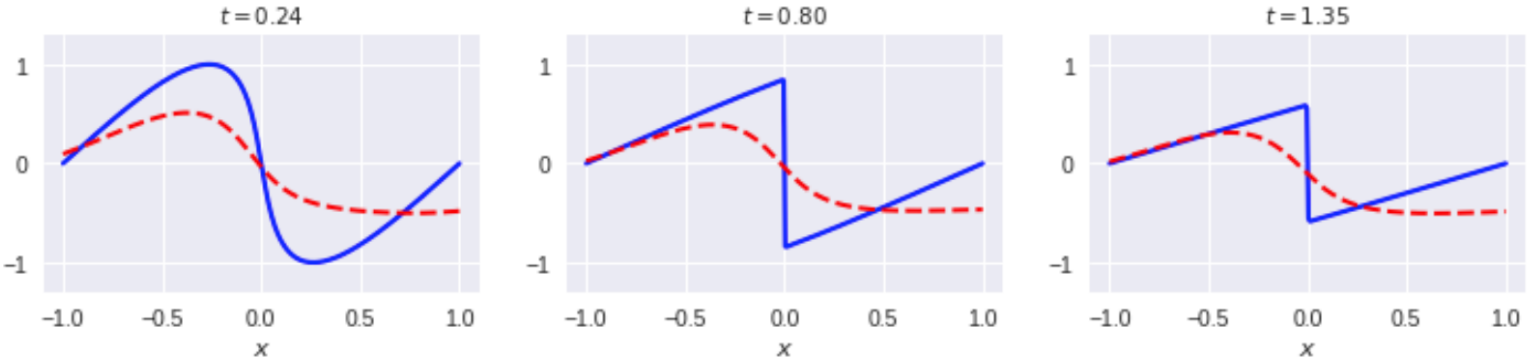

```
iter =  5000
loss = 0.0212
L2 error: 4.0300e-01
```



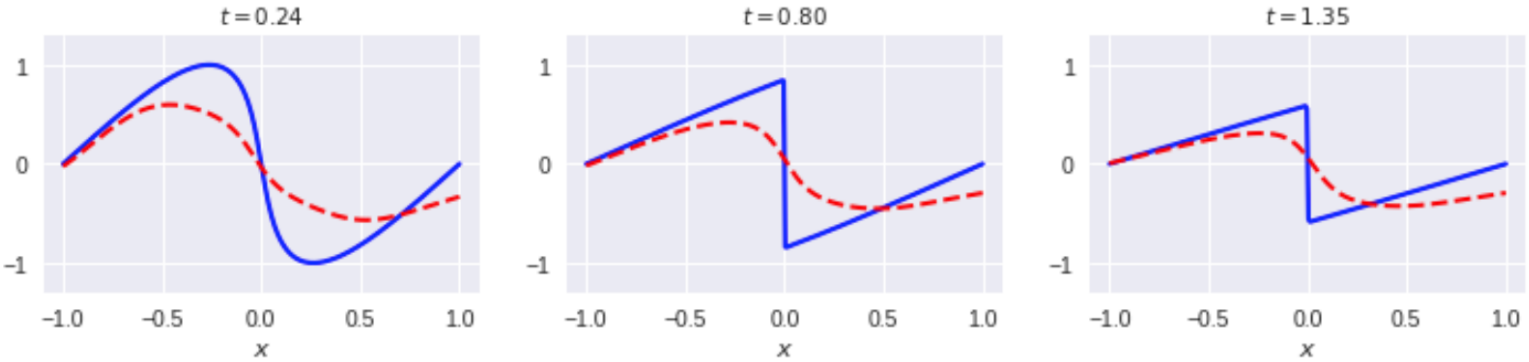
Example: Burgers equation

- 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
```



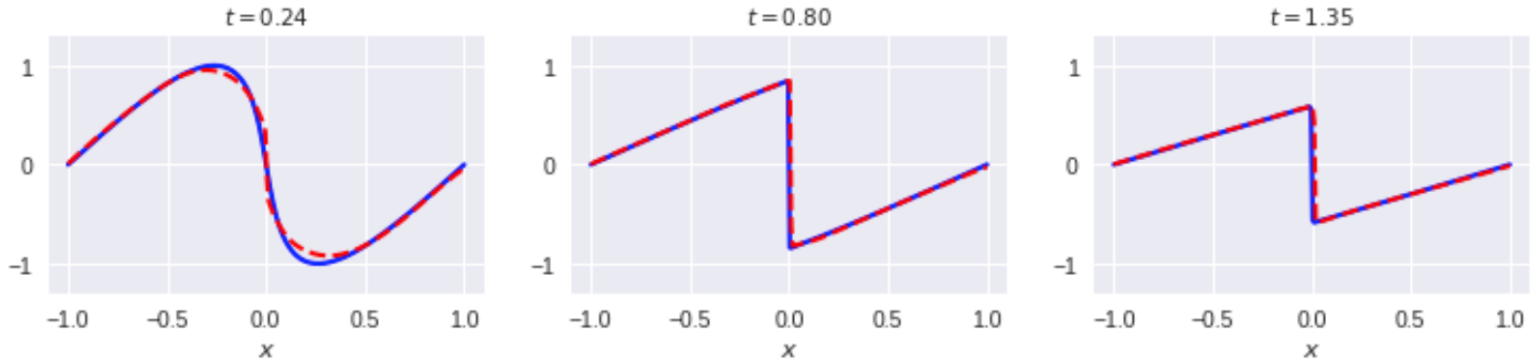

```
iter =  400
loss = 0.1170
L2 error: 5.2688e-01
```



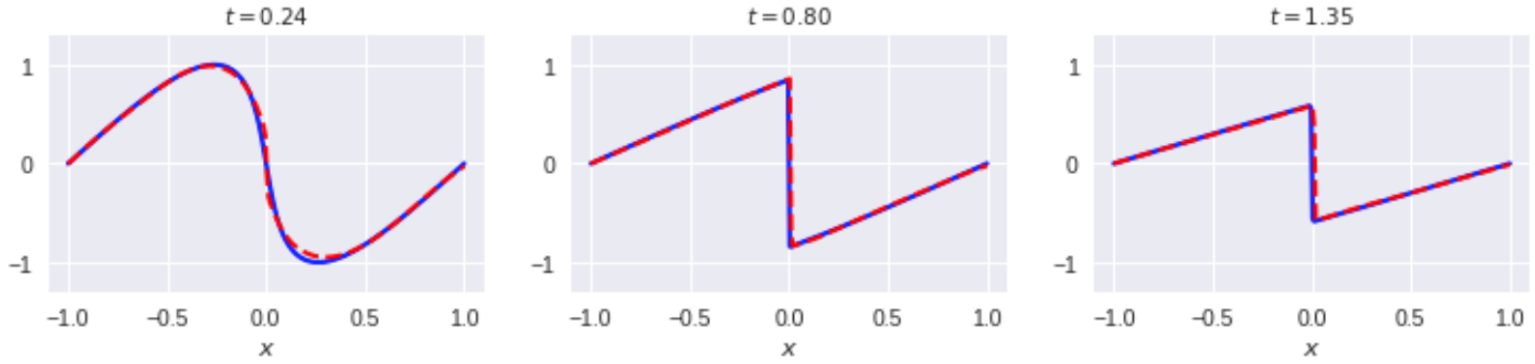
Example: Burgers equation

- 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
```



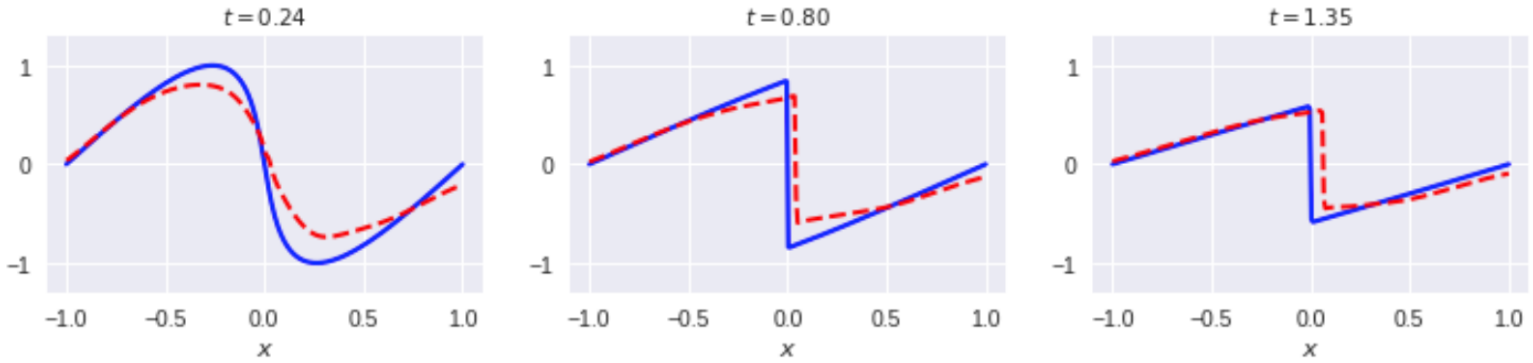

```
iter =  2200
loss = 0.0024
L2 error: 1.6838e-01
```



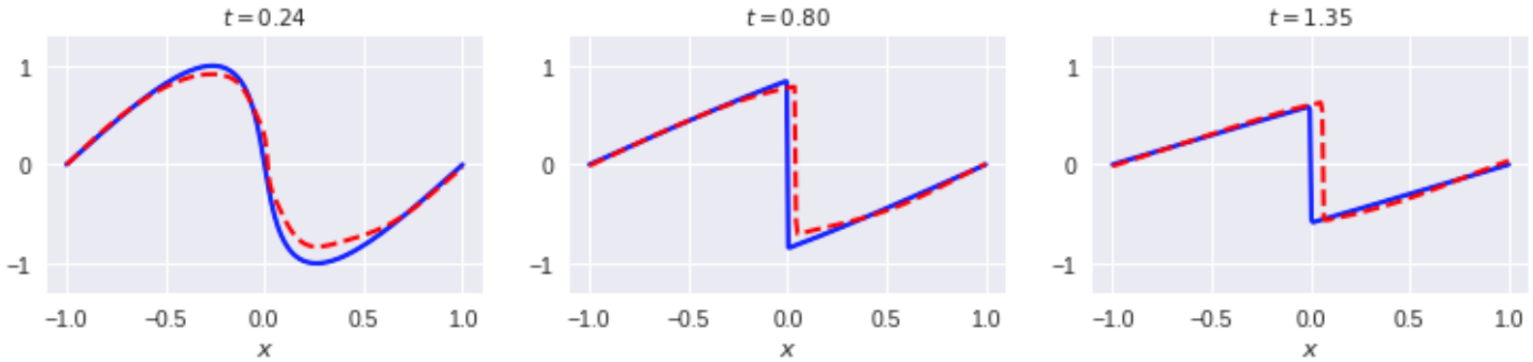
Example: Burgers equation

- 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
```



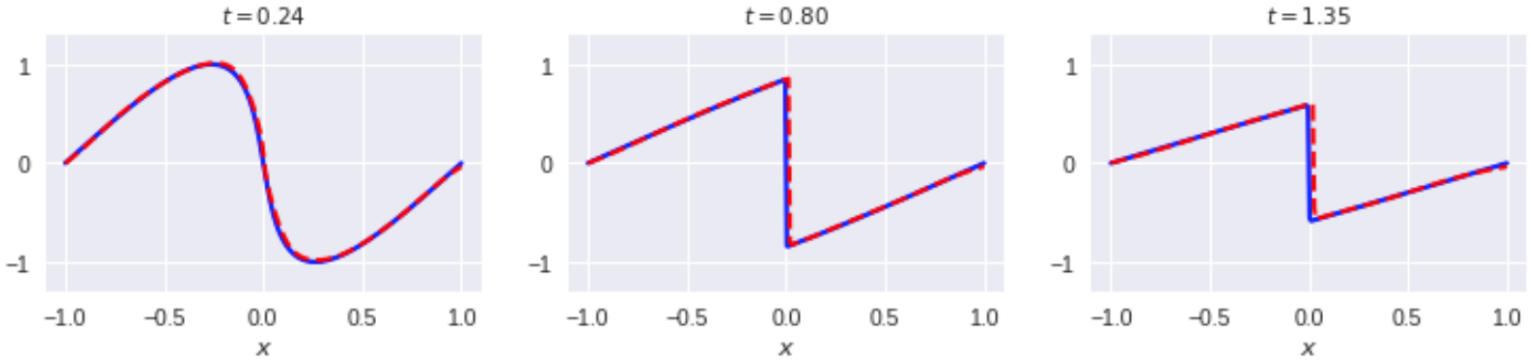

```
iter =  5000
loss = 0.0133
L2 error: 3.6761e-01
```



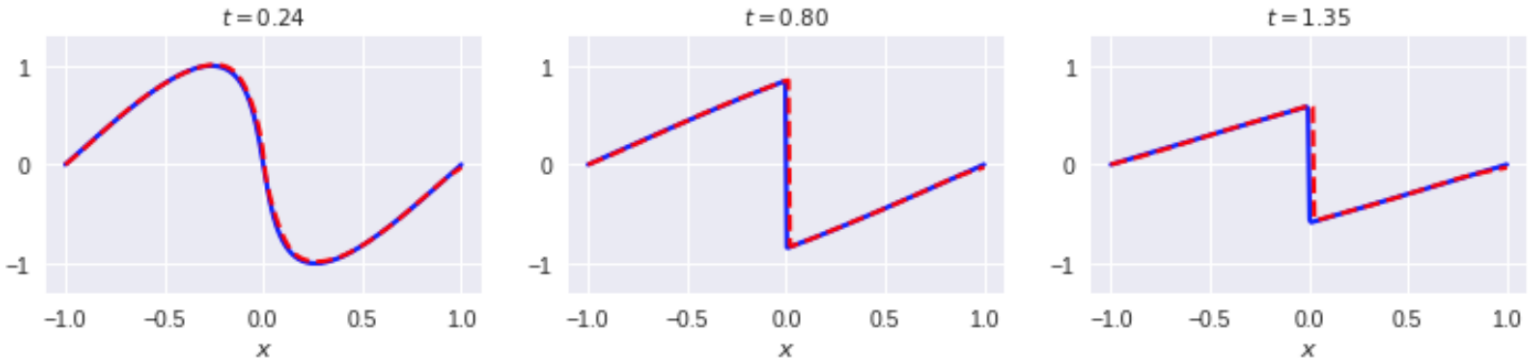
Example: Burgers equation

- 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
```



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 for parametric PINN's

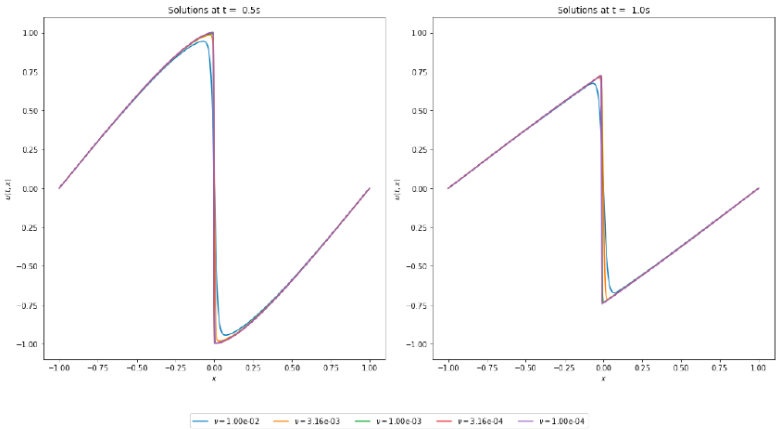
$$\min_\theta J_r(\theta) + \dots$$

with

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

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

- Application to the Burgers equations with many viscosities $[10^{-2}, 10^{-4}]$ (training: 2h). Same time for the smaller viscosity.



Numerical results

Linear advection equation

In all the numerical experiments, we use the V_h^3 basis. Results are similar with the other bases.

We first consider the first-order advection equation

$$\begin{cases} \partial_t u + \partial_x u = s(u;\mu), \\ u(t=0,x) = u_0(x), \end{cases}$$

with the following parameterized source term and initial condition

- $s(u;\alpha,\beta) = \alpha u + \beta u^2$;
- $u_0(x) = \varepsilon + u_{\text{eq}}(x;\alpha,\beta,\upsilon)$, with the steady solution u_{eq} depending on α, β and an additional parameter υ.

Hence, we have three parameters: $0.5 \leq \alpha \leq 1$, $0.5 \leq \beta \leq 1$, $0.1 \leq \upsilon \leq 0.2$

We propose three experiments: approximate

- a steady solution,
- a perturbed steady solution,
- an unsteady solution.

Training the PINN takes about 10 minutes on an old GPU, with **no data**, only the PINN loss.

Last minute remark: the errors are not well computed. Perhaps constant change (for all methods) and a little bit the order.

Linear advection equation: steady solution

In this case, $\varepsilon = 0$ in the initial condition, so we approximate the steady solution itself.

We compute the error between the exact and approximate solutions, for polynomial bases with $n_G \in \{1, 2, 3, 4\}$ elements, and with or without PINN prior.

We take a quadrature of degree $n_Q = \max(3, n_G + 1)$.

pts	error$_\phi$	order	error$_{\bar{\phi}}$	order	gain
10	7.42e-02	—	3.89e-04	—	190.66
20	2.64e-02	1.49	1.45e-04	1.42	181.76
40	9.29e-03	1.51	5.23e-05	1.47	177.55
80	3.27e-03	1.50	1.89e-05	1.46	172.63
160	1.18e-03	1.47	6.95e-06	1.45	170.09

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

Linear advection equation: steady solution

In this case, $\varepsilon = 0$ in the initial condition, so we approximate the steady solution itself.

We compute the error between the exact and approximate solutions, for polynomial bases with $n_G \in \{1, 2, 3, 4\}$ elements, and with or without PINN prior.

We take a quadrature of degree $n_Q = \max(3, n_G + 1)$.

pts	error$_\phi$	order	error$_{\bar{\phi}}$	order	gain
10	1.80e-03	—	1.09e-05	—	164.69
20	3.20e-04	2.50	1.93e-06	2.51	165.75
40	5.51e-05	2.54	3.33e-07	2.53	165.27
80	9.41e-06	2.55	5.64e-08	2.56	166.77
160	1.80e-06	2.38	1.08e-08	2.38	166.83

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

Linear advection equation: steady solution

In this case, $\varepsilon = 0$ in the initial condition, so we approximate the steady solution itself.

We compute the error between the exact and approximate solutions, for polynomial bases with $n_G \in \{1, 2, 3, 4\}$ elements, and with or without PINN prior.

We take a quadrature of degree $n_Q = \max(3, n_G + 1)$.

pts	error$_\phi$	order	error$_{\bar{\phi}}$	order	gain
10	2.23e-05	—	9.34e-07	—	23.94
20	2.02e-06	3.46	8.80e-08	3.41	23.01
40	1.75e-07	3.53	7.41e-09	3.57	23.60
80	1.45e-08	3.59	6.29e-10	3.56	23.14
160	1.46e-09	3.32	6.35e-11	3.31	22.99

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

Linear advection equation: steady solution

In this case, $\varepsilon = 0$ in the initial condition, so we approximate the steady solution itself.

We compute the error between the exact and approximate solutions, for polynomial bases with $n_G \in \{1, 2, 3, 4\}$ elements, and with or without PINN prior.

We take a quadrature of degree $n_Q = \max(3, n_G + 1)$.

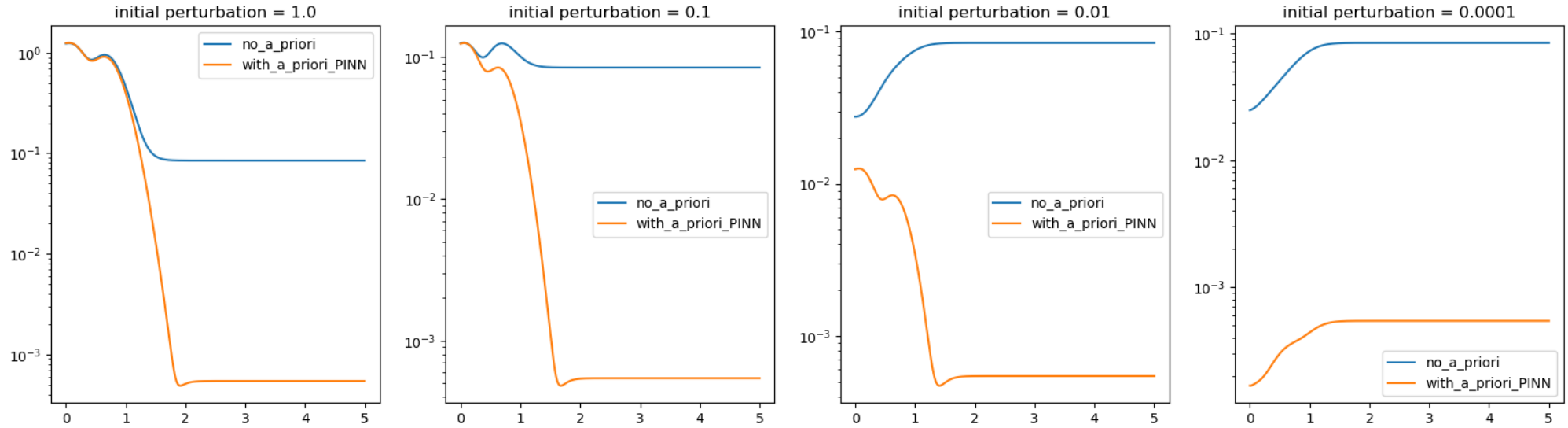
pts	error_ϕ	order	$\text{error}_{\bar{\phi}}$	order	gain
10	2.81e-07	—	6.49e-08	—	4.33
20	1.26e-08	4.48	3.02e-09	4.42	4.17
40	5.72e-10	4.46	1.32e-10	4.52	4.34
80	2.31e-11	4.63	5.40e-12	4.61	4.29
160	1.21e-12	4.25	2.77e-13	4.29	4.40

(d) errors with a four-element basis, $n_G = 4$

Linear advection equation: perturbed steady solution

We now study the effect of nonzero values of ε in the initial condition: we take $\varepsilon \in \{10^{-4}, 10^{-2}, 10^{-1}, 1\}$ and 20 discretization cells.

We represent the error, over time, between the approximate and exact solutions.

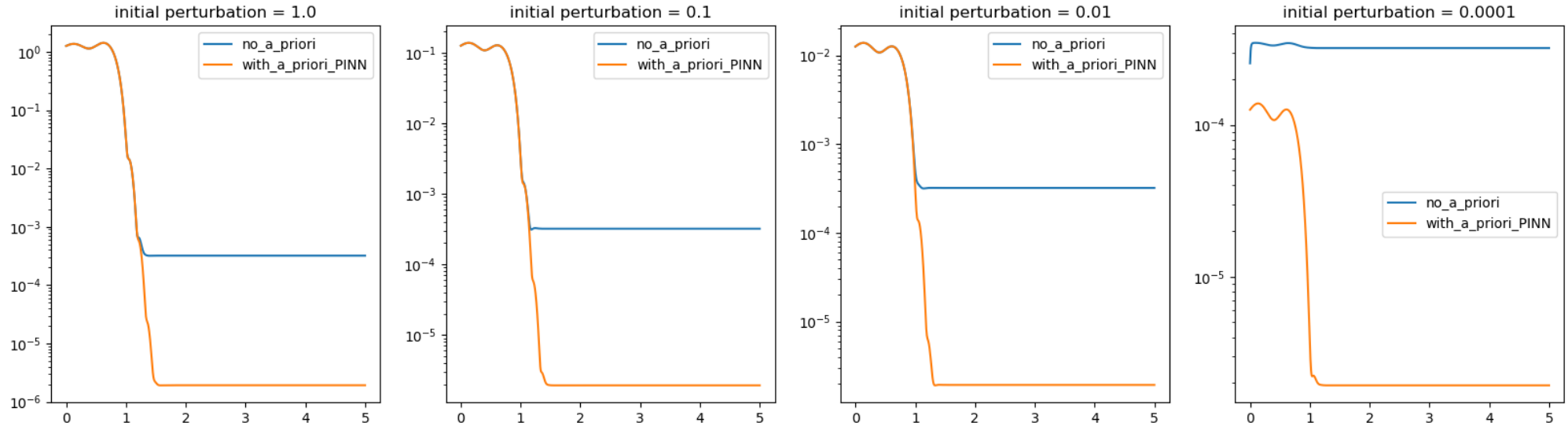


(a) errors with a one-element basis, $n_G = 1$

Linear advection equation: perturbed steady solution

We now study the effect of nonzero values of ε in the initial condition: we take $\varepsilon \in \{10^{-4}, 10^{-2}, 10^{-1}, 1\}$ and 20 discretization cells.

We represent the error, over time, between the approximate and exact solutions.

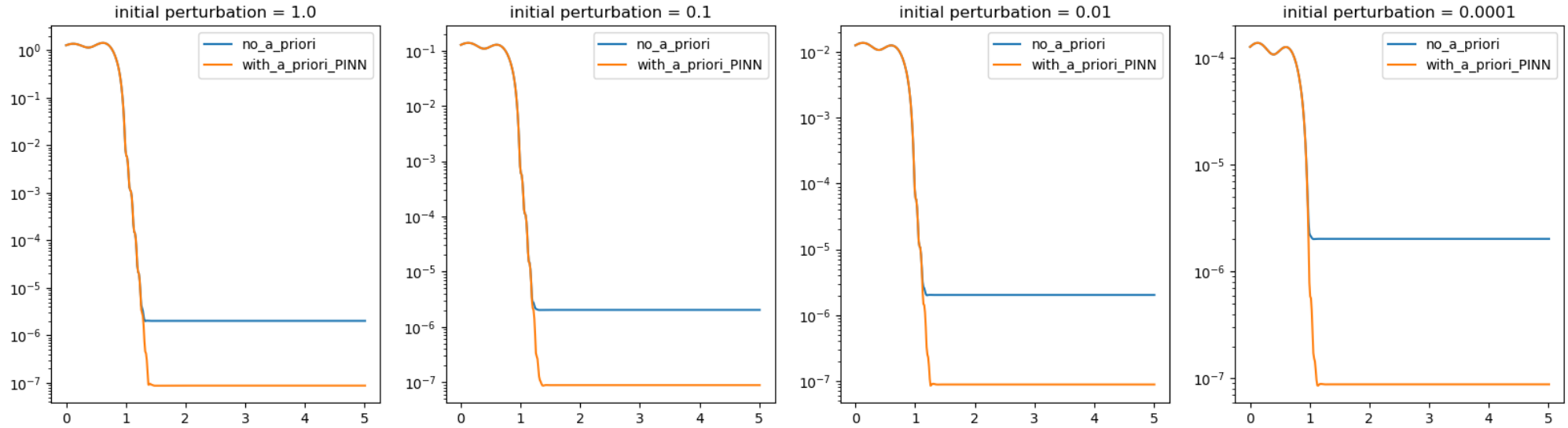


(b) errors with a two-element basis, $n_G = 2$

Linear advection equation: perturbed steady solution

We now study the effect of nonzero values of ε in the initial condition: we take $\varepsilon \in \{10^{-4}, 10^{-2}, 10^{-1}, 1\}$ and 20 discretization cells.

We represent the error, over time, between the approximate and exact solutions.

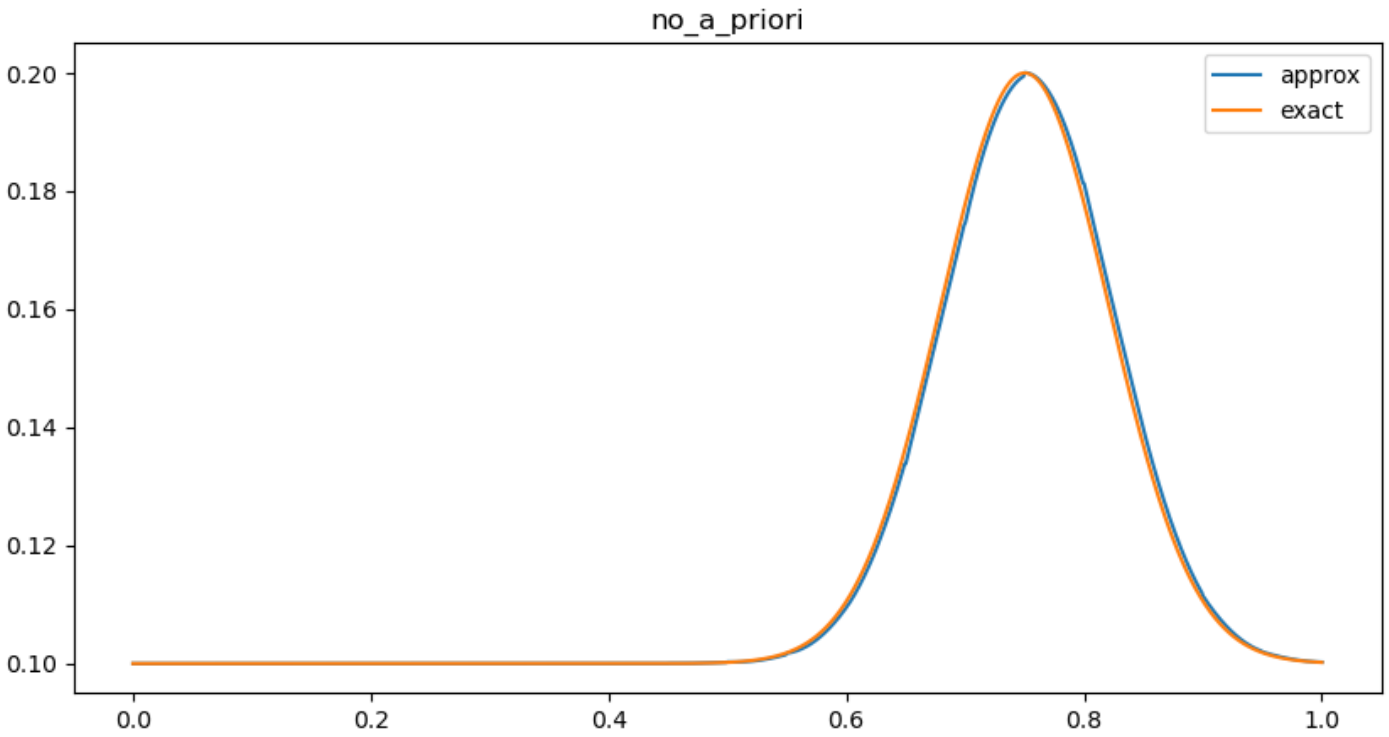


(c) errors with a three-element basis, $n_G = 3$

Linear advection equation: unsteady solution

Lastly, we perform the approximation of an unsteady solution with the two bases (with and without prior), to show that using the enhanced basis does not decrease approximation performance on unsteady solutions. The source term is zero in this case.

In this case, we take $n_G = 3$ and 20 discretization cells.

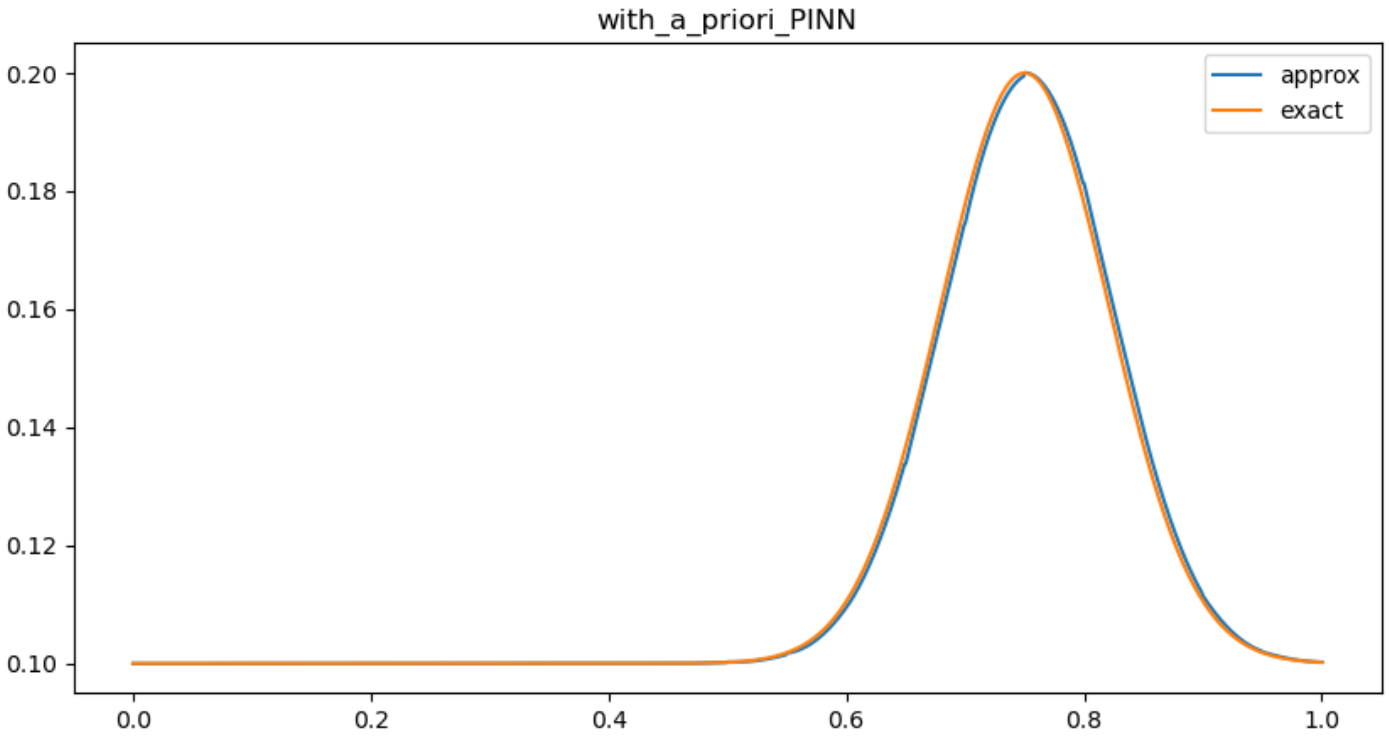


(a) without prior; error is 8.874×10^{-3}

Linear advection equation: unsteady solution

Lastly, we perform the approximation of an unsteady solution with the two bases (with and without prior), to show that using the enhanced basis does not decrease approximation performance on unsteady solutions. The source term is zero in this case.

In this case, we take $n_G = 3$ and 20 discretization cells.



(b) with prior; error is 8.874×10^{-3}, the same as without prior

Shallow water equations

We consider the shallow water system with topography

$$\begin{cases} \partial_t h + \partial_x q = 0, \\ \partial_t q + \partial_x \left(\dfrac{q^2}{h} + \dfrac{1}{2} g h^2 \right) = -gh\partial_x Z. \end{cases}$$

The smooth moving steady solutions are given by

$$q = \text{cst} = q_0; \qquad \frac{q^2}{2h^2} + g(h + Z) = \text{cst} = B_0.$$

The space domain is $(0, 1)$.

We consider two topography functions, which depend on two parameters α and β:

- $Z_g(x; \alpha, \beta) = \beta\omega(\alpha x)$, with ω a Gaussian bump function;
- $Z_c(x; \alpha, \beta) = \beta\omega_0(\alpha x)$, with ω_c a compactly supported Gaussian bump function.

Then, the steady solution $h(x; \alpha, \beta, h_0, B_0)$ depends on the two topography parameters α and β, as well as on the two steady flow parameters B_0 and q_0.

Shallow water equations: steady solution

Training takes about 10 minutes on an old GPU, with **no data**, only the PINN loss.

Results for the non-compactly-supported topography Z_g.

We use a quadrature of degree $n_Q = n_G + 2$.

pts	error^h_ϕ	order	error^q_ϕ	order	$\text{error}^h_{\frac{}{\phi}}$	order	gain	$\text{error}^q_{\frac{}{\phi}}$	order	gain
10	1.88e-01	—	4.34e-01	—	3.76e-04	—	499.42	1.74e-03	—	248.54
20	7.64e-02	1.30	2.35e-01	0.89	1.52e-04	1.31	501.88	4.52e-04	1.95	519.64
40	3.11e-02	1.30	1.04e-01	1.17	6.86e-05	1.15	453.42	2.18e-04	1.05	478.66
80	1.20e-02	1.37	4.18e-02	1.32	2.64e-05	1.38	456.98	9.33e-05	1.23	448.72
160	4.54e-03	1.41	1.58e-02	1.40	9.80e-06	1.43	463.99	3.59e-05	1.38	439.97

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

Shallow water equations: steady solution

Training takes about 10 minutes on an old GPU, with **no data**, only the PINN loss.

Results for the non-compactly-supported topography Z_g.

We use a quadrature of degree $n_Q = n_G + 2$.

pts	error^h_ϕ	order	error^q_ϕ	order	$\text{error}^h_{\vec{\phi}}$	order	gain	$\text{error}^q_{\vec{\phi}}$	order	gain
10	1.78e-02	—	5.43e-02	—	9.56e-05	—	186.55	1.14e-04	—	475.78
20	2.96e-03	2.59	8.13e-03	2.74	2.24e-05	2.09	131.94	6.82e-05	0.74	119.17
40	5.18e-04	2.52	1.36e-03	2.58	3.75e-06	2.58	137.95	1.01e-05	2.76	134.75
80	9.20e-05	2.49	2.40e-04	2.50	6.61e-07	2.51	139.19	1.72e-06	2.55	139.46
160	1.62e-05	2.50	4.24e-05	2.50	1.15e-07	2.51	140.32	2.99e-07	2.53	141.88

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

Shallow water equations: steady solution

Training takes about 10 minutes on an old GPU, with **no data**, only the PINN loss.

Results for the non-compactly-supported topography Z_g.

We use a quadrature of degree $n_Q = n_G + 2$.

pts	error^h_ϕ	order	error^q_ϕ	order	$\text{error}^h_{\bar{\phi}}$	order	gain	$\text{error}^q_{\bar{\phi}}$	order	gain
10	2.52e-03	—	6.21e-03	—	3.84e-05	—	65.83	9.04e-05	—	68.79
20	2.55e-04	3.31	7.78e-04	3.00	3.35e-06	3.52	76.16	8.40e-06	3.43	92.62
40	2.69e-05	3.25	8.23e-05	3.24	3.71e-07	3.18	72.52	1.04e-06	3.01	79.06
80	2.30e-06	3.55	7.67e-06	3.42	3.34e-08	3.47	68.84	1.09e-07	3.25	70.08
160	1.92e-07	3.58	6.91e-07	3.47	2.90e-09	3.53	66.39	1.03e-08	3.41	66.94

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

Shallow water equations: steady solution

Training takes about 10 minutes on an old GPU, with **no data**, only the PINN loss.

Results for the compactly-supported topography Z_c.

We use a quadrature of degree $n_Q = n_G + 7$. This is needed because of the large values of the derivatives of the topography (and therefore of the steady solution).

pts	error_{ϕ}^{h}	order	error_{ϕ}^{q}	order	$\text{error}_{\overline{\phi}}^{h}$	order	gain	$\text{error}_{\overline{\phi}}^{q}$	order	gain
10	1.792e-01	—	8.177e-01	—	5.504e-03	—	32.56	2.985e-02	—	27.39
20	1.133e-01	0.66	2.713e-01	1.59	9.514e-05	5.85	1190.37	6.583e-04	5.50	412.15
40	4.009e-02	1.50	1.516e-01	0.84	5.018e-05	0.92	798.84	1.714e-04	1.94	884.49
80	1.709e-02	1.23	5.929e-02	1.35	2.068e-05	1.28	826.30	6.971e-05	1.30	850.44
160	6.612e-03	1.37	2.290e-02	1.37	8.079e-06	1.36	818.45	2.708e-05	1.36	845.65

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

Shallow water equations: steady solution

Training takes about 10 minutes on an old GPU, with **no data**, only the PINN loss.

Results for the compactly-supported topography Z_c.

We use a quadrature of degree $n_Q = n_G + 7$. This is needed because of the large values of the derivatives of the topography (and therefore of the steady solution).

pts	error_{ϕ}^{h}	order	error_{ϕ}^{q}	order	$\text{error}_{\overline{\phi}}^{h}$	order	gain	$\text{error}_{\overline{\phi}}^{q}$	order	gain
10	1.391e-01	—	3.094e-01	—	5.614e-03	—	24.78	1.626e-02	—	19.03
20	3.101e-02	2.17	7.001e-02	2.14	2.836e-05	7.63	1093.61	7.243e-05	7.81	966.54
40	4.043e-03	2.94	9.297e-03	2.91	2.502e-06	3.50	1615.52	5.953e-06	3.60	1561.70
80	3.919e-04	3.37	1.222e-03	2.93	6.434e-07	1.96	609.06	2.171e-06	1.46	562.93
160	6.011e-05	2.70	1.883e-04	2.70	9.796e-08	2.72	613.60	3.177e-07	2.77	592.73

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

Shallow water equations: steady solution

Training takes about 10 minutes on an old GPU, with **no data**, only the PINN loss.

Results for the compactly-supported topography Z_c.

We use a quadrature of degree $n_Q = n_G + 7$. This is needed because of the large values of the derivatives of the topography (and therefore of the steady solution).

pts	error^h_ϕ	order	error^q_ϕ	order	$\text{error}^h_{\overline{\phi}}$	order	gain	$\text{error}^q_{\overline{\phi}}$	order	gain
10	7.003e-02	—	1.518e-01	—	3.523e-03	—	19.87	1.017e-02	—	14.92
20	9.161e-03	2.93	1.771e-02	3.10	2.876e-05	6.94	318.56	7.913e-05	7.01	223.80
40	6.838e-04	3.74	1.371e-03	3.69	1.080e-06	4.73	633.16	3.014e-06	4.71	454.79
80	4.575e-05	3.90	1.309e-04	3.39	8.268e-08	3.71	553.32	2.236e-07	3.75	585.59
160	4.385e-06	3.38	1.474e-05	3.15	9.817e-09	3.07	446.71	3.035e-08	2.88	485.69

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

Euler-Poisson system in spherical geometry

We consider the Euler-Poisson system in spherical geometry

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

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 two cases:

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

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

- a temperature-based pressure law $p(\rho; \kappa, \alpha) = \kappa \rho T_\alpha$ such that the steady solutions satisfy

$$\frac{d}{dr} \left(r^2 \kappa \frac{T_\alpha}{\rho} \frac{d\rho}{dr} \right) + \frac{d}{dr} \left(r^2 \kappa \frac{dT_\alpha}{dr} \right) = 4\pi r^2 G \rho;$$

  in practice, we take $T_\alpha(r) = e^{-\alpha r}$.

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^h_ϕ	order	error^q_ϕ	order	error^E_ϕ	order	$\text{error}^h_{\bar{\phi}}$	order	gain	$\text{error}^q_{\bar{\phi}}$	order	gain	$\text{error}^E_{\bar{\phi}}$	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

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 α.

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

Results for the **temperature-dependent 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	1.86e-01	—	1.99e-02	—	2.93e-01	—	1.12e-03	—	166.83	8.37e-03	—	2.38	1.26e-03	—	232.61
20	6.61e-02	1.50	8.36e-03	1.25	1.03e-01	1.50	4.22e-04	1.40	156.30	2.09e-03	2.00	4.00	5.62e-04	1.17	184.25
40	2.34e-02	1.50	3.19e-03	1.39	3.66e-02	1.50	1.55e-04	1.44	150.66	4.33e-04	2.27	7.36	2.12e-04	1.40	172.22
80	8.30e-03	1.50	1.34e-03	1.25	1.30e-02	1.49	5.89e-05	1.40	140.87	9.69e-05	2.16	13.85	8.65e-05	1.30	150.39
160	2.94e-03	1.50	5.33e-04	1.33	4.61e-03	1.50	2.04e-05	1.53	144.08	1.91e-05	2.34	27.81	2.92e-05	1.56	157.46

(a) errors with a oe-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 α.

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

Results for the **temperature-dependent 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	5.43e-03	—	8.96e-03	—	4.55e-03	—	8.89e-05	—	61.04	1.12e-04	—	79.89	9.09e-05	—	50.12
20	9.60e-04	2.50	2.29e-03	1.96	8.14e-04	2.48	1.70e-05	2.39	56.43	3.40e-05	1.72	67.55	1.83e-05	2.31	44.42
40	1.71e-04	2.48	4.70e-04	2.29	1.47e-04	2.47	3.19e-06	2.41	53.81	7.52e-06	2.18	62.55	3.73e-06	2.30	39.42
80	3.03e-05	2.50	8.73e-05	2.43	2.56e-05	2.52	5.72e-07	2.48	52.99	1.45e-06	2.37	59.85	7.11e-07	2.39	36.00
160	5.32e-06	2.51	1.54e-05	2.50	4.42e-06	2.53	9.90e-08	2.53	53.76	2.59e-07	2.49	59.55	1.18e-07	2.58	37.32

(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 α.

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

Results for the **temperature-dependent 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	1.37e-04	—	1.41e-04	—	1.54e-04	—	1.06e-05	—	12.95	1.15e-05	—	12.28	1.02e-05	—	15.07
20	1.09e-05	3.65	1.84e-05	2.94	2.06e-05	2.90	1.20e-06	3.14	9.13	1.71e-06	2.75	10.76	1.73e-06	2.56	11.91
40	9.94e-07	3.47	1.39e-06	3.73	2.09e-06	3.31	1.17e-07	3.35	8.45	1.52e-07	3.50	9.17	2.22e-07	2.96	9.39
80	8.71e-08	3.51	1.22e-07	3.51	1.79e-07	3.55	1.09e-08	3.43	7.97	1.29e-08	3.56	9.49	2.05e-08	3.44	8.71
160	7.69e-09	3.50	1.07e-08	3.51	1.56e-08	3.52	1.10e-09	3.31	6.99	1.21e-09	3.41	8.83	1.89e-09	3.44	8.23

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

2D shallow water system

We consider the 2D shallow water equations

$$\begin{cases} \partial_t h + \nabla \cdot q = 0 \\ \partial_t q + \nabla \cdot \left(\dfrac{q \otimes q}{h} + \dfrac{1}{2} g h^2 \right) = -gh \nabla Z(x, y; \alpha, r_0) \end{cases}$$

We define the following compactly supported bump function:

$$\Omega(x, y; \alpha, r_0) = \begin{cases} \alpha \exp\left(\dfrac{-1}{\left(1 - \dfrac{r^2}{r_0^2}\right)^3} \right) & \text{if } r < r_0, \\ 0 & \text{otherwise,} \end{cases}$$

and we take $Z(x, y; \alpha, r_0) = \Omega(x, y; \alpha, r_0)$.

The steady solution is a **vortex**, whose amplitude and radius depend on α, r_0 and an additional parameter Γ: this time, we have three parameters, in addition to x and y.

2D shallow water system: steady solution

Training takes about 20 minutes on an old GPU, with the PINN loss **supplemented with data**.

We need a high-quadrature, of degree $n_Q = 14$, because of the large derivatives of the compactly supported smooth bump function.

pts	error_{ϕ}^{h}	order	$\text{error}_{\phi}^{q_x}$	order	$\text{error}_{\phi}^{q_y}$	order	$\text{error}_{\bar{\phi}}^{h}$	order	gain	$\text{error}_{\bar{\phi}}^{q_x}$	order	gain	$\text{error}_{\bar{\phi}}^{q_y}$	order	gain
20	1.91e-01	—	1.13e+00	—	1.13e+00	—	2.31e-03	—	82.79	1.02e-03	—	1116.93	1.01e-03	—	1119.33
40	4.72e-02	2.02	2.76e-01	2.04	2.76e-01	2.04	5.85e-04	1.98	80.64	2.30e-04	2.15	1199.70	2.22e-04	2.19	1242.66
80	1.16e-02	2.02	6.71e-02	2.04	6.71e-02	2.04	1.46e-04	2.00	79.77	5.72e-05	2.01	1173.39	5.52e-05	2.01	1216.72
160	2.90e-03	2.00	1.68e-02	1.99	1.68e-02	1.99	3.66e-05	2.00	79.45	1.43e-05	2.00	1178.29	1.38e-05	2.00	1222.59

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

2D shallow water system: steady solution

Training takes about 20 minutes on an old GPU, with the PINN loss **supplemented with data**.

We need a high-quadrature, of degree $n_Q = 14$, because of the large derivatives of the compactly supported smooth bump function.

pts	error$^h_\phi$	order	error$^{q_x}_\phi$	order	error$^{q_y}_\phi$	order	error$^h_{\bar{\phi}}$	order	gain	error$^{q_x}_{\bar{\phi}}$	order	gain	error$^{q_y}_{\bar{\phi}}$	order	gain
20	2.32e-02	—	2.10e-01	—	2.10e-01	—	2.59e-04	—	89.71	5.49e-04	—	382.67	5.73e-04	—	367.32
40	3.60e-03	2.69	2.86e-02	2.88	2.86e-02	2.88	3.15e-05	3.04	114.33	4.24e-05	3.70	675.67	4.30e-05	3.73	665.36
80	5.28e-04	2.77	3.56e-03	3.01	3.57e-03	3.01	3.95e-06	2.99	133.61	6.07e-06	2.80	587.71	6.16e-06	2.80	578.89
160	7.02e-05	2.91	4.63e-04	2.94	4.63e-04	2.94	4.96e-07	2.99	141.49	7.90e-07	2.94	586.16	8.02e-07	2.94	577.49

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

2D shallow water system: steady solution

Training takes about 20 minutes on an old GPU, with the PINN loss **supplemented with data**.

We need a high-quadrature, of degree $n_Q = 14$, because of the large derivatives of the compactly supported smooth bump function.

pts	error$^{h}_{\phi}$	order	error$^{q_x}_{\phi}$	order	error$^{q_y}_{\phi}$	order	error$^{h}_{\bar{\phi}}$	order	gain	error$^{q_x}_{\bar{\phi}}$	order	gain	error$^{q_y}_{\bar{\phi}}$	order	gain
20	5.17e-03	—	6.05e-02	—	6.05e-02	—	3.05e-04	—	16.97	1.63e-03	—	37.11	1.60e-03	—	37.72
40	4.32e-04	3.58	4.35e-03	3.80	4.34e-03	3.80	2.07e-06	7.20	208.24	4.35e-06	8.55	999.02	4.47e-06	8.49	969.66
80	2.87e-05	3.91	2.73e-04	3.99	2.73e-04	3.99	1.30e-07	3.99	220.41	2.84e-07	3.94	961.63	2.89e-07	3.95	942.16
160	1.72e-06	4.06	1.81e-05	3.91	1.81e-05	3.91	8.17e-09	3.99	210.88	1.59e-08	4.15	1136.57	1.62e-08	4.16	1117.87

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

Future work and extension

Operator learning

- Parametric PINNs make it possible to approximate a family of PDE solutions, but we need a parametrisation (e.g. of the topography).
- **More general**: Operator learning
- Principle: we consider

$$-\partial_x(\alpha(x)\partial_x u(x)) = f(x)$$

- Formaly there exist a operator $G^+ : \mathcal{H}^2 \to \mathcal{H}$, with $\mathcal{H}$ an Hilbert space, defined by

$$G^+(\alpha(x), f(x)) \to u(x)$$

Neural operator

Construct a neural network G^+_θ, approximation of G^+, where the result and, if possible, the input, do not depend on the mesh resolution.

Next

- this week: construct Neural operator which takes the topography a input, and gives the equilibrium for the 2D Shallow Water equations.
- 2-year post-doc position in Strabourg to improve neural operators for time-dependent hyperbolic systems.

MHD equilibrium

- For Tokamaks and stellarators, we solve flows around equilbrium.
- General equilibrium:

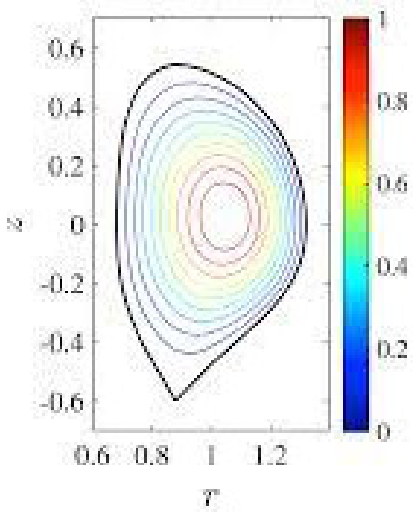
$$\nabla P = \mathbb{J} \times \mathbb{B}$$

with $\mu_0 \mathbb{J} = \nabla \times \mathbb{B}$.

- Tokamak equilibrum

$$-\Delta^* \psi(R, Z) = -\mu_0 R^2 \frac{dP(\psi, \boldsymbol{\mu}_1)}{d\psi} - \frac{1}{2} \frac{dF(\psi, \boldsymbol{\mu}_2)}{d\psi}$$

with $\boldsymbol{\mu}_1$, $\boldsymbol{\mu}_2$ some parameters.



Aim

Solve the equilibrium family with PINNs/Neural operator, and couple with a DG scheme, or another efficient scheme.

Time-dependent and asymptotic problems

- Coming back to time-dependant problems:

$$\partial_t u + a\partial_x u = u^2$$

- If the prior is perfect in time and space and the quadrature also the spatial part is exact but it change nothing for the time part.
- The modification only acts on the spatial error.

Next

- Using DG in time or other approaches, we wish to obtain a very accurate approximation in time around a family of space-time solution.
- Example: a compressible scheme which would be more accurate around a family of incompressible flows.

Next

- Direct optimization of the basis functions to minimize the error on some solutions

Conclusion

PINNs

Physics-Informed neural networks and neural operators are a good way to compute and store large families of solutions.

Enhanced DG

Using this prior in the basis, we significantly increase the accuracy around these families of steady states (including steady solution with no analytical expression).

Questions

Can we obtain entropy stability ?

General

Neural networks are interesting for PDEs but we do not have the same guarantees of classical numerical methods. Using the NN as a "predictor/preconditioner" for the numerical method, we hope to gain in CPU time, accuracy, and retain good convergence properties.

Thank you for your attention!

Finite Volumes for Complex Applications 10 (**FVCA10**),
in Strasbourg, 30/10/2023 – 03/11/2023