Réseaux de neurones et modèles physiques basés sur des équations différentielles

Hugo Gangloff

MIA PARIS-SACLAY

Statistiques au sommet de Rochebrune, 25 - 29 mars 2024

Introduction

Physics Informed Neural Networks (PINNs)

- an hybrid machine-learning approach
- parametric approximation of the solution using neural networks
- exact computations of the derivatives
- both an **alternative solver** and a flexible framework for **injecting physical knowledge** in statistical models (forward & inverse problems)

PINNs is a rapidly growing field

- First mention of the idea in (Dissanayake et al. 1994)
- **Seminal work:** PINNs for PDEs (Raissi et al. 2017; Raissi et al. 2019) and since then many models have been proposed
- Theoretical analysis (S. Wang, Teng, et al. 2020; Doumèche et al. 2023)
- Recent reviews (Karniadakis et al. 2021; Cuomo et al. 2022; S. Wang, Sankaran, et al. 2023)
- First benchmark in (Hao, Yao, et al. 2023)
- Two major Python libraries: `DeepXDE` (L. Lu et al. 2021) and `Nvidia Modulus` (`https://developer.nvidia.com/modulus`)

Outline

→ *Some of the slides have been written by Nicolas Jouvin*
→ *All the experimental results come from our package `jinns`*

Basics

Partial Differential Equations

In all generality, a PDE with solution u is defined by a space domain $\Omega \subset \mathbb{R}^d$, a time domain $I = [0, T]$, a differential operator parameterized by θ such that

$$\mathcal{N}_\theta[u](t, x) = 0, \quad \forall t, x \in I \times \Omega$$

with initial condition

$$u(0, \cdot) = u_0(x), \quad \forall x \in \Omega$$

and a boundary condition

$$\mathcal{B}[u](t, \delta x) = f(t, \delta x), \quad \forall t \in I, \forall \delta x \in \partial\Omega$$

Traditional PDE solvers

- mesh-dependent, approximate derivatives
- piecewise approximation of the solution
- theoretical and numerical guarantees
- difficult to account for observations

Illustration: a classical example

Burger's equation in 1D

With $\Omega = [-1, 1]$, and $I = [0, 1]$,

$$\begin{cases} \frac{\partial}{\partial t} u(t,x) + u(t,x)\frac{\partial}{\partial x} u(t,x) - \theta \frac{\partial^2}{\partial x^2} u(t,x) = 0, \\ u(0,x) = -\sin(\pi x), \\ u(t,-1) = u(t,1) = 0 \end{cases}$$

On the right we plot u for $\theta = 0.01\pi$

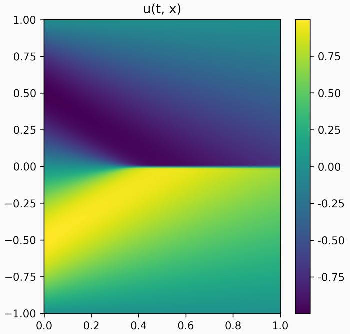


Illustration: a classical example

Burger's equation in 1D

With $\Omega = [-1, 1]$, and $I = [0, 1]$,

$$\begin{cases} \frac{\partial}{\partial t} u(t,x) + u(t,x) \frac{\partial}{\partial x} u(t,x) - \theta \frac{\partial^2}{\partial x^2} u(t,x) = 0, \\ u(0,x) = -\sin(\pi x), \\ u(t,-1) = u(t,1) = 0 \end{cases}$$

On the right we plot u for $\theta = 0.01\pi$

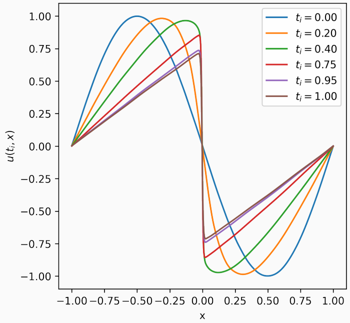


Illustration: ODEs and a classical example

Ordinary Differential Equations: $u(t)$ only, no spatial domain

Generalized Lotka Volterra

$$\frac{\partial}{\partial t}u_i(t) = r_i - \sum_{j\neq i}\alpha_{ij}u_j(t) - \alpha_{i,i}u_i(t) + c_i u_i(t) + \sum_{j\neq i} c_j u_j(t), i \in \{1, 2, 3\}$$

Plot of each solution u_i

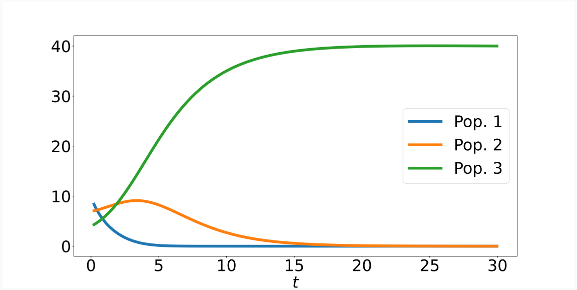
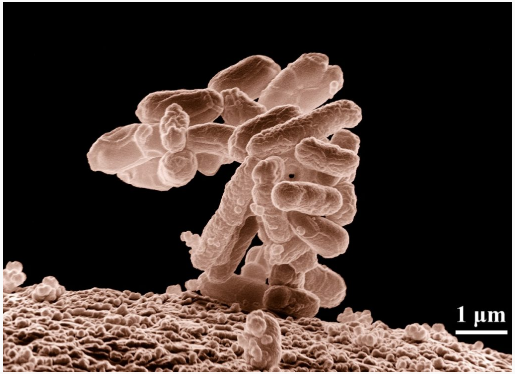
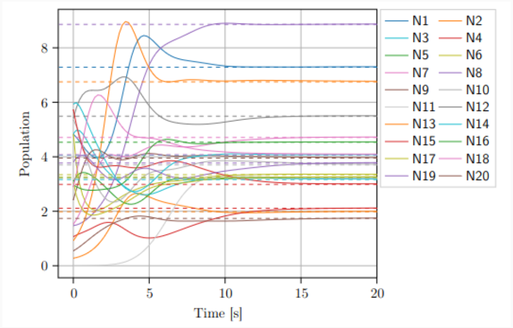


Illustration: ODEs and a classical example





https://www.inserm.fr/dossier/microbiote-intestinal-flore-intestinale/

(Hossie et al. 2024)

Supervised learning

Observe: $\{(x_i, y_i)\}_{i=1}^{n_{obs}}$

Goal: learning $\hat{u}$ such that $y \approx \hat{u}(x)$ on new data

How: Parametric function $u_\nu(x)$ and minimize a *loss* $\mathcal{L}$

$$\hat{u} = u_{\hat{\nu}} \qquad \hat{\nu} = \arg\min_{\nu} \mathcal{L}(\nu; X, Y)$$

The loss can be

- mean-squared error: $\mathcal{L}(\nu) = \sum_i |y_i - u_\nu(x_i)|^2$
- likelihood of some parametric statistical model: $\mathcal{L}(\nu) = -\log p_\nu(X, Y)$

The function u_ν may be

- linear $u_\nu(x) = \nu^\top x$, polynomial, functional basis (splines, etc.)
- or ...

Neural networks

- A neural network u_ν is a composition of L layers
- Each layer is an elementary parametric function composed with σ an *activation function* $u^l_{\nu_l} = \sigma(g^l_{\nu_l})$

Neural networks

- A neural network u_ν is a composition of L layers
- Each layer is an elementary parametric function composed with σ an *activation function* $u^l_{\nu_l} = \sigma(g^l_{\nu_l})$
- Parameters: $\nu = \{\nu_1, \dots, \nu_L\}$.
- A standard combination is an affine g^l and $\sigma = \tanh$ activation:

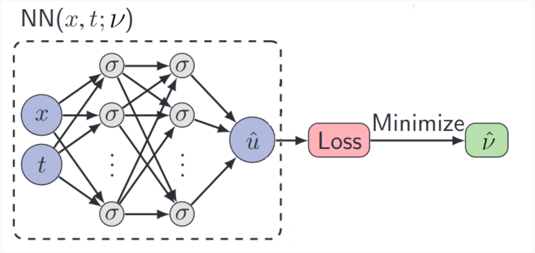
$$u^l_{\nu_l}(x) = \tanh(w_l^\top x + b_l) \text{ with } \nu_l = \{w_l, b_l\}$$

- *Universal approximators*: can approximate many classes of functions with sufficiently large depth or width (Hornik et al. 1989)

Neural networks illustrated

$$\hat{\nu} \in \arg\min_{\nu} \mathcal{L}_{\mathrm{NN}}(\nu) \text{ with } \mathcal{L}_{\mathrm{NN}}(\nu) = \sum_{i=1}^{n_{obs}} |y_i - u_\nu(x_i)|^2$$

→ Highly non-convex and hard to minimize (Lee et al. 2016)



Adapted from L. Lu et al. (2021)

Loss optimization

Stochastic Gradient Descent

In order to train the neural network, we classically perform stochastic gradient descent with *mini-batches* of data. At each step t:

$$\nu^{t+1} = \nu^t - \gamma \sum_{(x_i,y_i)\in D_k} \nabla_\nu \mathcal{L}_{\text{NN}}(\nu, x_i, y_i),$$

where the dataset $\mathcal{D}$ is divided in mini-batches $\mathcal{D} = \{D_1, \dots, D_K\}$

We perform a step over all the mini-batches and call this an *epoch*

Efficient computation of $\nabla\mathcal{L}_{\text{NN}}$ is one of the critical point in deep learning

Forward problems

Introducing physical prior

Constrain $\hat{u}_\nu$ to be solution to a given PDE

Introducing physical prior

Loss function: for a set of equation parameters θ and neural network u_ν

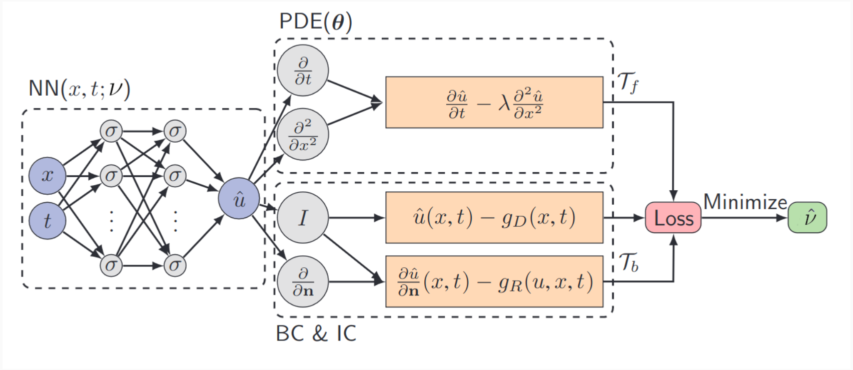
$$\begin{aligned}\mathcal{L}_{\mathrm{PINN}}(\nu,\theta) &:= \sum_{i=1}^{n_x}\sum_{j=1}^{n_t} |\mathcal{N}_\theta[u_\nu](t_j,x_i)|^2 + w_{ic}\sum_{i=1}^{n_x}|u_\nu(0,x_i)-u_0|^2 \\ &\quad + w_{bc}\sum_{j=1}^{n_t}\sum_{k=1}^{n_{bc}}|\mathcal{B}[u_\nu](t_j,\delta x_k)-f(t_j,\delta x_k)|^2 + w_{obs}\sum_{l=1}^{n_{obs}}|u_l-u_\nu(t_l,x_l)|^2_{obs}, \\ &= \underbrace{\mathcal{L}_{dyn} + w_{bc}\mathcal{L}_{bc} + w_{ic}\mathcal{L}_{ic}}_{\text{physical prior}} + \underbrace{w_{obs}\mathcal{L}_{obs}}_{\text{statistical information}}\end{aligned}$$

where

- $\{x_i, t_j, \delta x_k\}_{i,j,k}$ are *collocation points* drawn from $\Omega \times [0,T] \times \partial\Omega$
- $\{((t_l,x_l),u_l)\}_{l=1}^{n_{obs}}$ are noisy observations of $u^\star$ (possibly missing)
- w_{ic}, w_{bc}, w_{obs} are weights balancing the different terms

Introducing physical prior

→ $\mathcal{L}_{\text{PINN}}$ is even more highly non-convex and harder to optimize than $\mathcal{L}_{\text{NN}}$



PDE($\boldsymbol{\theta}$)
NN($x, t; \nu$)
x
t
σ
$\hat{u}$
$\frac{\partial}{\partial t}$
$\frac{\partial^2}{\partial x^2}$
$\frac{\partial \hat{u}}{\partial t} - \lambda \frac{\partial^2 \hat{u}}{\partial x^2}$
$\mathcal{T}_f$
I
$\frac{\partial}{\partial \mathbf{n}}$
$\hat{u}(x,t) - g_D(x,t)$
$\frac{\partial \hat{u}}{\partial \mathbf{n}}(x,t) - g_R(u,x,t)$
$\mathcal{T}_b$
Loss
Minimize
$\hat{\nu}$
BC & IC

Graphical representation of a PINN ($w_{obs} = 0$) adapted from L. Lu et al. (2021)

Forward problem

Goal: for a given set of equation parameters θ, find a parametric function $u_{\hat{\nu}}$

$$\hat{\nu} \in \arg\min_{\nu} \mathcal{L}_{\mathrm{PINN}}(\nu, \theta)$$

We can distinguish 2 situations

1. **PDE solver ($\mathbf{w_{obs}} = \mathbf{0}$)** where PINNs are viewed as an alternative to standard numerical methods
2. **Hybrid-modeling ($\mathbf{w_{obs}} > \mathbf{0}$)** where $\mathcal{L}_{\mathrm{PINN}}$ combines statistical information and physics prior

Forward problem

Goal: for a given set of equation parameters θ, find a parametric function $u_{\hat{\nu}}$

$$\hat{\nu} \in \arg\min_{\nu} \mathcal{L}_{\mathrm{PINN}}(\nu, \theta)$$

- **compute the loss** → compute gradients w.r.t. the NN inputs, e.g.:

$$\sum_{x,t} |\mathcal{N}_\theta[u_\nu](t,x)|^2 = \sum_{t,x} |\frac{\partial u_\nu}{\partial t}(t,x) - \Delta u_\nu(t,x)|^2 = \sum_{t,x} \left| \frac{\partial u_\nu}{\partial t}(t,x) - \frac{\partial^2 u_\nu}{\partial x_1^2}(t,x) - \frac{\partial^2 u_\nu}{\partial x_2^2}(t,x) \right|^2$$

- **stochastic optimization** → compute gradients w.r.t. ν

→ Both tasks rely on **automatic differentiation**

Automatic Differentiation (AD) (Baydin et al. 2018)

- Numerical and exact way to compute the derivatives of a function
- Automatic differentiation $\neq$ symbolic differentiation $\neq$ numerical differentiation
- Particularly suitable for composition of elementary functions (like NNs): leverages chain rule & known derivatives
- **Backpropagation** Goodfellow et al. (2016) is the main AD algorithm
- Implemented in all ML libraries (tensorflow, PyTorch, JAX, etc.)

Automatic Differentiation (AD) (Baydin et al. 2018)

$$u_\nu : x \mapsto f_L \circ f_{L-1} \circ \cdots \circ f_0(x) \text{ with } f_l(x) = \sigma(w_l x + b_l)$$

Let $h_{l+1} \coloneqq f_l(h_l)$, $h_0 \coloneqq x$ and $y \coloneqq h_{L+1} = u_\nu(x)$. We are interested in computing $\nabla_{x_k} y_{k'}, \forall k, k'$. Let us write the chain rule in the vectorial case. We follow the **computational graph**:

$$\underbrace{\nabla_{x_k} y_{k'}}_{1\times 1} = \underbrace{\nabla_{h_{L+1}} y_{k'}}_{1\times \dim y} \times \underbrace{\mathrm{Jac}_{h_L} f_L(h_L)}_{\dim y \times \dim h_L} \times \underbrace{\mathrm{Jac}_{h_{L-1}} f_{L-1}(h_{L-1})}_{\dim h_L \times \dim h_{L-1}} \times \cdots \times \underbrace{\mathrm{Jac}_{h_0} f_0(h_0)}_{\dim h_1 \times \dim x} \times \underbrace{\nabla_{x_k} h_0}_{\dim x \times 1}$$

$\rightarrow$ The same procedure is used for gradients with respect to ν

$\rightarrow$ There are at least 2 ways to parse the chain rule formula...

Forward AD in a neural network

$$\nabla_{x_k} y = \left(\nabla_{h_{L+1}} y \times \left(\mathrm{Jac}_{h_L} f_L(h_L) \times \left(\mathrm{Jac}_{h_{L-1}} f_{L-1}(h_{L-1}) \times \cdots \times \left(\mathrm{Jac}_{h_0} f_0(h_0) \times \nabla_{x_k} h_0 \right) \right) \right) \right)$$

- Right to left successive computations of the type $\underbrace{\mathrm{Jac}_{h_l} f_l(h_l)}_{\mathrm{dim} h_{l+1} \times h_l} \times \underbrace{\nabla_{x_k} h_l}_{\mathrm{dim} h_l \times 1}$
- This elementary operation is called a **Jacobian-Vector Product** in AD
- All the points where we need to evaluate the Jacobians are computed on the fly as we go down the computational graph (here, these are the h_l)

Forward AD in a neural network

$$\nabla_{x_k} y = \left(\nabla_{h_{L+1}} y \times \left(\text{Jac}_{h_L} f_L(h_L) \times \left(\text{Jac}_{h_{L-1}} f_{L-1}(h_{L-1}) \times \cdots \times \left(\text{Jac}_{h_0} f_0(h_0) \times \nabla_{x_k} h_0\right)\right)\right)\right)$$

- JVPs hardcoded in AD libraries for all kinds of f $\rightarrow$ Jacobians are never explicitly computed
- Forward mode enables recovering **one column at a time of the Jacobian** $\nabla_x y$ $\rightarrow$ **most efficient for tall Jacobians**, i.e., differentiation of a function from $\mathbb{R}^{\dim x} \rightarrow \mathbb{R}^{\dim m}$, $\dim y >> \dim x$

Reverse AD in a neural network

$$\nabla_x y_{k'} = \left(\left(\left(\left(\nabla_{h_{L+1}} y_{k'} \times \mathrm{Jac}_{h_L} f_L(h_L)\right) \times \mathrm{Jac}_{h_{L-1}} f_{L-1}(h_{L-1})\right) \times \ldots \times \mathrm{Jac}_{h_0} f_0(h_0)\right) \times \nabla_x h_0\right)$$

- Left to right successive computations of the type $\underbrace{\nabla_{h_{l+1}} y_{k'}}_{\dim 1 \times h_{l+1}} \times \underbrace{\mathrm{Jac}_{h_l} f_l(h_l)}_{\dim h_{l+1} \times h_l}$
- This elementary operation is called a **Vector-Jacobian Product** in AD
- Reverse AD relies on a previous forward pass in the computational graph: we precompute and store all the points at which we will evaluate the Jacobians
- The popular backpropagation algorithm is a reverse AD algorithm

Reverse AD in a neural network

$$\nabla_x y_{k'} = \left(\left(\left(\left(\nabla_{h_{L+1}} y_{k'} \times \mathrm{Jac}_{h_L} f_L(h_L)\right) \times \mathrm{Jac}_{h_{L-1}} f_{L-1}(h_{L-1})\right) \times \ldots \times \mathrm{Jac}_{h_0} f_0(h_0)\right) \times \nabla_x h_0\right)$$

- VJPs hardcoded in AD libraries for all kinds of f $\rightarrow$ Jacobians are never explicitly computed
- Reverse mode enables recovering **one row at a time of the Jacobian** $\nabla_x y$ $\rightarrow$ **most efficient for large Jacobians**, i.e., differentiation of a function from $\mathbb{R}^{\dim x} \rightarrow \mathbb{R}^{\dim y}$, $\dim x >> \dim y$
- The popular backpropagation algorithm is a reverse AD algorithm

Research direction: improving learning

- Importance sampling of the collocation points

$$\int_{\Omega} |\mathcal{N}[u](x)|^2 \,\mathrm{d}x \approx \frac{1}{n} \sum_{i=1}^{n} \frac{1}{q(x_i)} |\mathcal{N}[u](x_i)|^2, \quad x_i \overset{i.i.d.}{\sim} q.$$

$\rightarrow$ adaptive q charges regions of Ω with high residuals (Wu et al. 2023)

- Adaptive weights w_{obs}, w_{ic} and w_{bc} during learning (Xiang et al. 2022)

Research directions: theoretical analysis

- Results in the PDE solver case (Mishra et al. 2022)

- Results in the hybrid modeling case (Doumèche et al. 2023)
  $\rightarrow$Regularization strategies to prevent over-fitting

$$\min_{\nu} \mathcal{L}_{\text{PINN}}(\nu) + \lambda \|\nu\|$$

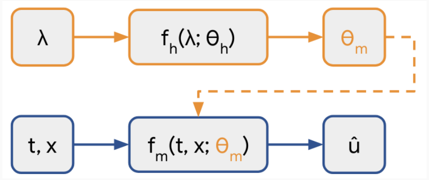
  $\rightarrow$ *Sobolev regularization* of the risk to have u_ν converging to the PDE solution

Research directions: metamodel learning

- Learn a function $\hat{u}_\nu(t, x, \theta)$ such that

$$\forall \theta, \quad \mathcal{N}_\theta[\hat{u}_\nu(\cdot, \cdot, \theta)](t, x) \approx 0$$

- Learn to solve many equations at once $\rightarrow$ evaluation is cheap with NNs
- HyperPINNs (Avila Belbute-Peres et al. 2021) have been proposed for this task



from (Avila Belbute-Peres et al. 2021)

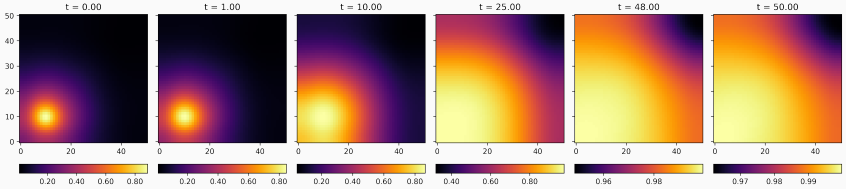
Illustration: metamodel for an advection diffusion PDE

- Consider the following PDE problem

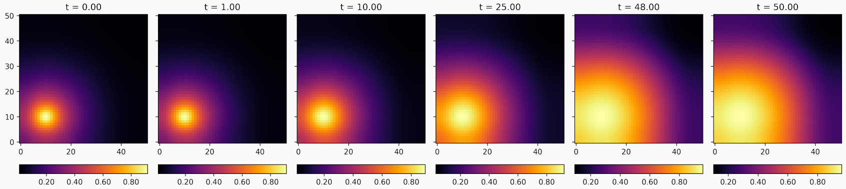
$$\begin{cases} \frac{\partial}{\partial t}u(t,x) = D\Delta u(t,x) + u(t,x)(r - u(t,x)), t \geqslant 0, x \in \Omega, \Omega = [0,50]^2 \\ \frac{\partial u(t,x)}{\partial n}\Big|_{x\in\partial\Omega} = \nabla u(t,x)\cdot n = 0, t \geqslant 0, \text{Neumann condition}, \\ u(0,x) = u_0(x), x \in \Omega \end{cases}$$

- The hyperparameters are D and r
- Train an HyperPINN to learn a function $\hat{u}_\nu$ such that $\forall (D,r) \in [0.05,1] \times [0.05,0.15], \quad \mathcal{N}_{(D,r)}[\hat{u}_\nu(\cdot,\cdot,D,r)](t,x) \approx 0$

Illustration: metamodel for an advection diffusion PDE



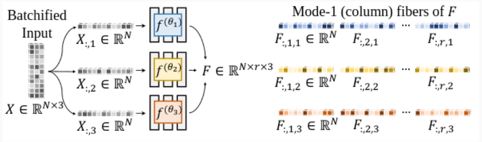
$\hat{u}(t,x)$ *estimated for* $D = 1$, $r = 0.15$



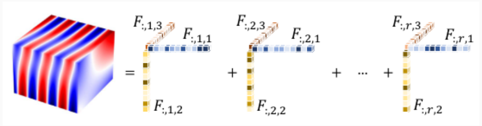
$\hat{u}(t,x)$ *estimated for* $D = 0.05$, $r = 0.05$

Research directions: reducing computational costs

- More efficient computation of high-order derivatives (Bettencourt et al. 2019; R. Li et al. 2024)
- Leveraging forward mode AD with Separable PINNs (Cho et al. 2024)



Batchified Input
$X \in \mathbb{R}^{N\times 3}$
$X_{:,1} \in \mathbb{R}^N$
$X_{:,2} \in \mathbb{R}^N$
$X_{:,3} \in \mathbb{R}^N$
$f^{(\theta_1)}$
$f^{(\theta_2)}$
$f^{(\theta_3)}$
$F \in \mathbb{R}^{N\times r\times 3}$
Mode-1 (column) fibers of F
$F_{:,1,1} \in \mathbb{R}^N$
$F_{:,2,1}$
$F_{:,r,1}$
$F_{:,1,2} \in \mathbb{R}^N$
$F_{:,2,2}$
$F_{:,r,2}$
$F_{:,1,3} \in \mathbb{R}^N$
$F_{:,2,3}$
$F_{:,r,3}$



Inverse problems

Inverse problem

In most applications, we are also interested in **estimating the equation parameters** $\hat{\theta}$ as well as an approximate solution $u_{\hat{\nu}}$. This leads to solve

$$(\hat{\nu}, \hat{\theta}) \in \arg\min_{\nu,\theta} \mathcal{L}_{\mathrm{PINN}}(\nu, \theta)$$

- The nature of the problem suggests an iterative optimization scheme (Raissi et al. 2019)
    1. $\hat{\nu}^{(t+1)} \in \arg\min_{\nu} \mathcal{L}_{\mathrm{PINN}}(\nu, \theta^{(t)})$
    2. $\hat{\theta}^{(t+1)} \in \arg\min_{\theta} \mathcal{L}_{\mathrm{PINN}}(\nu^{(t+1)}, \theta)$

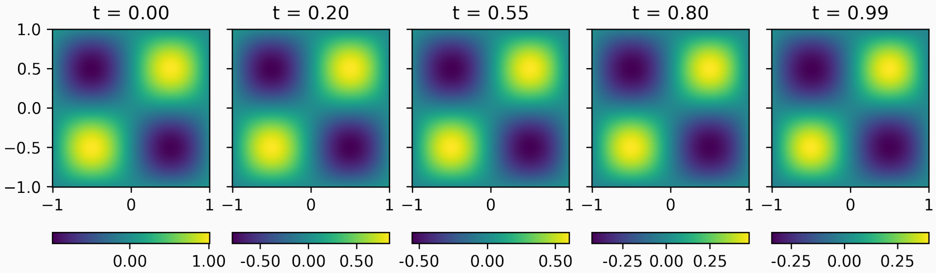
Toy example

- Toy example from the benchmark (Hao, Liu, et al. 2022)
- Consider the following PDE on $\Omega = [0,1]^2$, $I = [0,1]$:

$$\begin{cases} \frac{\partial}{\partial t} u(t,x,y) - \nabla(a(t,x,y)\nabla u(t,x,y)) = f(t,x,y), \\ f(t,x,y) = ((4\pi^2 - 1)\sin \pi x \sin \pi y + \pi^2(2\sin^2 \pi x \sin^2 \pi y - \cos^2 \pi \sin^2 \pi y - \\ \qquad \sin^2 \pi x \cos^2 \pi y))e^{-t}, \text{ (source term).} \end{cases}$$

- Our goal is to learn both $u(t,x,y)$ and $a(t,x,y)$ for all $(x,y) \in \Omega^2, t \in I$
- The diffusion coefficient a is itself modeled by a NN

Toy example



$u(t, x, y)$ as estimated by the PINN

→ This corresponds to the analytical solution: $u(t, x, y) = e^{-t} \sin \pi x \sin \pi y$

Mechanistic-statistical models

General case: $u^\star$ is indirectly involved in some statistical model $Y \mid O$

$$\begin{aligned} &i) \quad u^\star \text{ solution of } \mathrm{PDE}_{\theta^\star}, && \text{(Mecanistic model)} \\ &ii) \quad Y \mid O \sim p(\cdot \mid O, u^\star, \theta^\star). && \text{(Statistical model)} \end{aligned}$$

- In (Roques 2013), the inference for θ is done in the Bayesian context by sampling from the posterior $p(\theta \mid Y) \propto p(Y \mid O, u^\star, \alpha)\pi(\theta)$
- Computing the likelihood or posterior involves $u^\star$ $\rightarrow$ numerous calls a PDE solver
- Can we use PINNs to bypass the need of PDE solvers ?

Mechanistic-statistical models: toy example

- We have observed $\{((t_l, x_l), y_l)\}_{l=1}^{n_{obs}}$ forming the likelihood

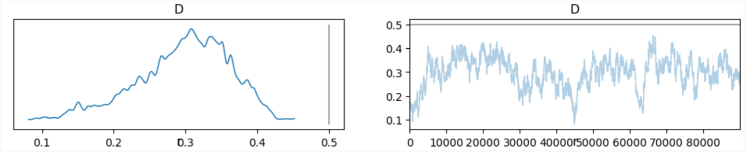
$$\prod_{l=1}^{n_{obs}} p(y_l \mid (t_l, x_l), u^\star, D, r) = \prod_{l=1}^{n_{obs}} \mathcal{N}(y_l; u^\star_{D,r}(t_l, x_l), \sigma^2)$$

- with $\frac{\partial}{\partial t} u^\star_{D,r}(t,x) = D\Delta u^\star_{D,r}(t,x) + u^\star_{D,r}(t,x)(r - u^\star_{D,r}(t,x)), t \geqslant 0, x \in \Omega, \Omega = [0, 50]^2$
- Define the prior $\pi(D, r) \propto \mathbf{1}(0.05 \leqslant D \leqslant 1)\mathbf{1}(0.05 \leqslant r \leqslant 0.15)$
- We want to sample from the posterior $p(D, r|y)$

Mechanistic-statistical models: toy example

- **Traditional MCMC approach** (Roques 2013)

  $\rightarrow$ solve the PDE for each new proposals (D, r)



- **Potential PINN/MCMC approach**

  $\rightarrow$ Make the effort to train PINN

  $\hat{u}_\nu(t, x, D, r), \forall (D, r) \in [0.05, 1] \times [0.05, 0.15]$

  $\rightarrow$ Only need a forward pass in the PINN for each proposal (D, r)

  $\rightarrow$ Orders of magnitude faster after once the network is trained...

Mechanistic-statistical models: in the real world

- Observations from more **complex noise models**
  $\rightarrow$ e.g. count data: observations are $\{((t_l, \omega_l), y_l)\}$ where $\omega_l \subset \Omega$ and the likelihood reads

$$\prod_{l=1}^{n_{obs}} p(y_l \mid (t_l, \omega_l), u^\star, D, r) = \prod_{l=1}^{n_{obs}} \mathcal{P}(y_l; \int_{\omega_l} u^\star_{D,r}(t_l, x) \mathrm{d}x)$$

Mechanistic-statistical models: in the real world

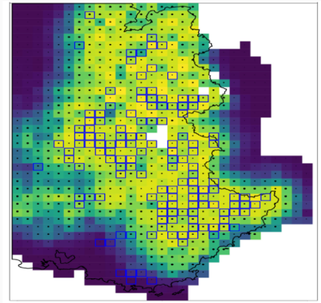
- Often, we have **nested covariates** (Soubeyrand et al. 2014)
  $\rightarrow$ e.g. a spatially varying reproduction rate $r(x)$ which depends on the type of forest covering $c(x)$ (data we have access to) through the logistic link:

$$r(x) = \frac{1}{1 + e^{\theta_0 + \theta_1 c(x) + \theta_2 c(x)^2}}$$

  $\rightarrow$ We want to estimate the vector θ

Mechanistic-statistical models: in the real world

Mechanistic-statistical models: in the real world



From (Louvrier et al. 2020)

jinns

JAX + PINNs = `jinns`

- `jinns` is developped by Nicolas Jouvin (MIA, Paris-Saclay, INRAE) and me
- Past members of the project: Pierre Gloaguen (now at LMBA, Université Bretagne-Sud) and Achille Thin (now data scientist at Genesis)

JAX + PINNs = `jinns`

- `jinns` is developped by Nicolas Jouvin (MIA, Paris-Saclay, INRAE) and me
- Past members of the project: Pierre Gloaguen (now at LMBA, Université Bretagne-Sud) and Achille Thin (now data scientist at Genesis)

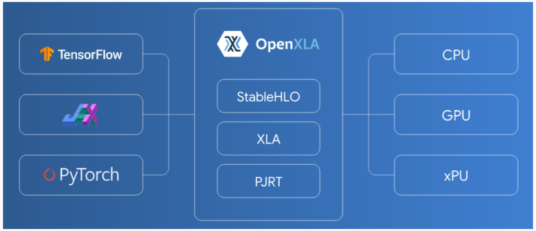
- Modulable to implement your own research ideas with PINNs
- Development is driven towards the resolution of **inverse problems**
- Optimized code thanks to JAX
- Integrates the JAX ecosystem: `diffrax`, `equinox`, `blackjax`, `optax`, ...

`https://pypi.org/project/jinns/`

JAX

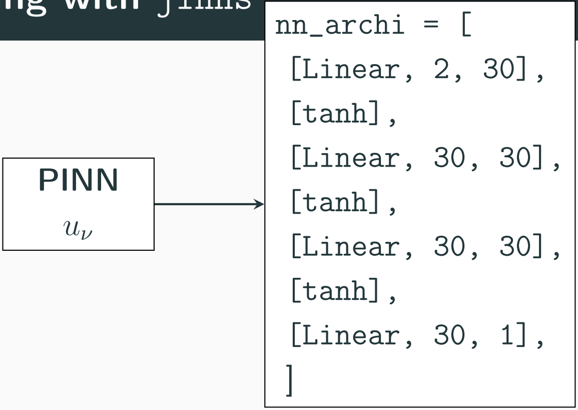
JAX Python library (Bradbury et al. 2018):

- **automatic differentiation**: forward/backward AD, custom JVPs/VJPs, ...
- **code vectorization** and **parallel computing**: `vmap`, `pmap`, *shardings*, ...
- **Just-In-Time compilation**: `jax.numpy` → `jaxpr` → `XLA`



openxla.org

Working with `jinns`



```
nn_archi = [
 [Linear, 2, 30],
 [tanh],
 [Linear, 30, 30],
 [tanh],
 [Linear, 30, 30],
 [tanh],
 [Linear, 30, 1],
 ]
```

First define u_ν

- Helper functions for standard architectures
- Users can implement their own neural network architectures
- PINNs, HyperPINNs, Separable PINNs are implemented

Working with `jinns`

Mesh
Ω, $\partial\Omega$ and I
PINN
u_ν

Then define the space/time domain

- Controls the collocation points on the sets Ω, $\partial\Omega$ and $I = [0, T]$
- `DataGenerator` objects will send batches of collocation points to the loss

Working with `jinns`

PINN

u_ν

Mesh

Ω, $\partial\Omega$ and I

Observations

$\{(o_l, y_l)\}_{l=1}^{n_{obs}}$

Hyperparameters

θ

It is also possible to send to the loss **batches of observations** and/or **batches of hyperparameters**

Working with `jinns`

$\boxed{\mathcal{L}_{dyn}}$ $\boxed{\mathcal{L}_{bc}}$ $\boxed{\mathcal{L}_{ic}}$ $\boxed{\mathcal{L}_{obs}}$ $\boxed{\cdots}$

Mesh
Ω, $\partial\Omega$ and I

PINN
u_ν

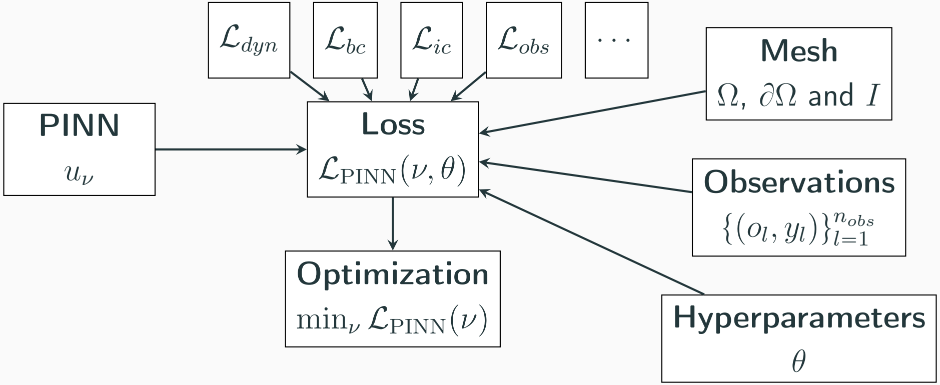
Observations
$\{(o_l, y_l)\}_{l=1}^{n_{obs}}$

Hyperparameters
θ

Finally, define your loss, *i.e.* your PDE problem. A loss is composed of:

- `DynamicLoss` classes implementing $\mathcal{L}_{dyn}$
- Initial and boundary conditions
- Other user-defined constraints (normalization, *Sobolev*, ...)

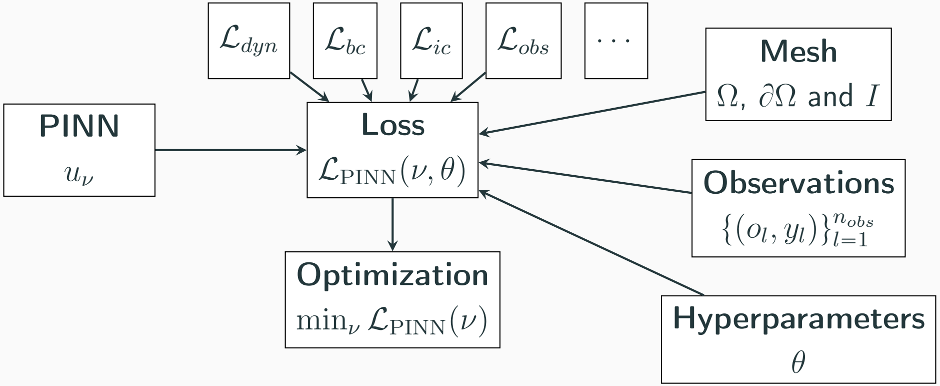
Working with `jinns`


$\mathcal{L}_{dyn}$
$\mathcal{L}_{bc}$
$\mathcal{L}_{ic}$
$\mathcal{L}_{obs}$
$\dots$
Mesh
Ω, $\partial\Omega$ and I
PINN
u_ν
Loss
$\mathcal{L}_{\text{PINN}}(\nu, \theta)$
Observations
$\{(o_l, y_l)\}_{l=1}^{n_{obs}}$
Optimization
$\min_\nu \mathcal{L}_{\text{PINN}}(\nu)$
Hyperparameters
θ

Finally, define your loss, *i.e.* your PDE problem. A loss is composed of:

- `DynamicLoss` classes implementing $\mathcal{L}_{dyn}$
- Initial and boundary conditions
- Other user-defined constraints (normalization, *Sobolev*, ...)

Working with `jinns`

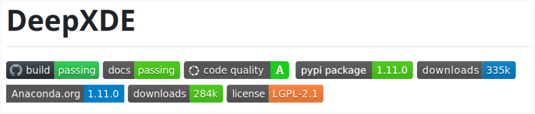

$\mathcal{L}_{dyn}$
$\mathcal{L}_{bc}$
$\mathcal{L}_{ic}$
$\mathcal{L}_{obs}$
$\dots$
PINN
u_ν
Loss
$\mathcal{L}_{\text{PINN}}(\nu, \theta)$
Mesh
Ω, $\partial\Omega$ and I
Observations
$\{(o_l, y_l)\}_{l=1}^{n_{obs}}$
Optimization
$\min_\nu \mathcal{L}_{\text{PINN}}(\nu)$
Hyperparameters
θ

Optimization is carried by the `solve()` function

- Uses `optax` optimizers: several optimization algorithms available
- Combines the PINN, the loss and all the `DataGenerator` objects
- Handles optimization w.r.t. ν and/or θ (forward/inverse problems)

Comparison with DeepXDE

DeepXDE (L. Lu et al. 2021) the most popular library for research with PINNs



DeepXDE

build passing | docs passing | code quality A | pypi package 1.11.0 | downloads 335k
Anaconda.org 1.11.0 | downloads 284k | license LGPL-2.1

- Wider scope than PINNs
- Several backends are being implemented (JAX, tensorflow, pytorch, ...)
- Slower than `jinns`
- No focus on inverse problems (see PINNacle (Hao, Liu, et al. 2022))

Conclusion

Conclusion

Pros:

- Fast by leveraging AD and modern ML libraries
- Flexible framework to incorporate physics prior into statistical learning
- Promising results on many classical problems, offers interesting research directions and new perspectives

Cons:

- Can fail to converge, requires hyper-parameter tuning
- Few theoretical results

$\rightarrow$ **Long-term impact of PINNs is still unclear**

References i

[1] F. de Avila Belbute-Peres, Y.-f. Chen, and F. Sha. *HyperPINN: Learning parameterized differential equations with physics-informed hypernetworks*. 2021. arXiv: `2111.01008 [cs.LG]`.

[2] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. “Automatic differentiation in machine learning: a survey”. In: *Journal of machine learning research* 18.153 (2018), pp. 1–43.

[3] J. Bettencourt, M. J. Johnson, and D. Duvenaud. “Taylor-mode automatic differentiation for higher-order derivatives in JAX”. In: *Program Transformations for ML Workshop at NeurIPS 2019*. 2019.

[4] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: `http://github.com/google/jax`.

[5] J. Cho, S. Nam, H. Yang, S.-B. Yun, Y. Hong, and E. Park. “Separable physics-informed neural networks”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[6] S. Cuomo, V. S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli. “Scientific machine learning through physics–informed neural networks: Where we are and what’s next”. In: *Journal of Scientific Computing* 92.3 (2022), p. 88.

[7] M. Dissanayake and N. Phan-Thien. “Neural-network-based approximations for solving partial differential equations”. In: *communications in Numerical Methods in Engineering* 10.3 (1994), pp. 195–201.

[8] N. Doumèche, G. Biau, and C. Boyer. “Convergence and error analysis of PINNs”. In: *arXiv preprint arXiv:2305.01240* (2023).

References ii

[9] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[10] Z. Hao, S. Liu, Y. Zhang, C. Ying, Y. Feng, H. Su, and J. Zhu. "Physics-informed machine learning: A survey on problems, methods and applications". In: *arXiv preprint arXiv:2211.08064* (2022).

[11] Z. Hao, J. Yao, C. Su, H. Su, Z. Wang, F. Lu, Z. Xia, Y. Zhang, S. Liu, L. Lu, et al. "Pinnacle: A comprehensive benchmark of physics-informed neural networks for solving pdes". In: *arXiv preprint arXiv:2306.08827* (2023).

[12] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[13] P. J. Hossie, B. Laroche, T. Malou, L. Perrin, T. Saigre, and L. Sala. "Simulating interactions in microbial communities through Physics Informed Neural Networks: towards interaction estimation". In: (Feb. 2024). working paper or preprint. URL: https://hal.inrae.fr/hal-04440736.

[14] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang. "Physics-informed machine learning". In: *Nature Reviews Physics* 3.6 (2021), pp. 422–440.

[15] J. D. Lee, M. Simchowitz, M. I. Jordan, and B. Recht. "Gradient descent only converges to minimizers". In: *Conference on learning theory*. PMLR. 2016, pp. 1246–1257.

[16] R. Li, H. Ye, D. Jiang, X. Wen, C. Wang, Z. Li, X. Li, D. He, J. Chen, W. Ren, et al. "A computational framework for neural network-based variational Monte Carlo with Forward Laplacian". In: *Nature Machine Intelligence* (2024), pp. 1–11.

References iii

[17] J. Louvrier, J. Papaix, C. Duchamp, and O. Gimenez. "A mechanistic–statistical species distribution model to explain and forecast wolf (Canis lupus) colonization in South-Eastern France". In: *Spatial Statistics* 36 (2020), p. 100428.

[18] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis. "DeepXDE: A deep learning library for solving differential equations". In: *SIAM review* 63.1 (2021), pp. 208–228.

[19] S. Mishra and R. Molinaro. "Estimates on the generalization error of physics-informed neural networks for approximating a class of inverse problems for PDEs". In: *IMA Journal of Numerical Analysis* 42.2 (2022), pp. 981–1022.

[20] M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational physics* 378 (2019), pp. 686–707.

[21] M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations". In: *arXiv preprint arXiv:1711.10561* (2017).

[22] L. Roques. "Modèles de réaction-diffusion pour l'écologie spatiale: Avec exercices dirigés". In: *Modèles de réaction-diffusion pour l'écologie spatiale* (2013), pp. 1–176.

[23] S. Soubeyrand and L. Roques. "Parameter estimation for reaction-diffusion models of biological invasions". In: *Population ecology* 56 (2014), pp. 427–434.

[24] S. Wang, S. Sankaran, H. Wang, and P. Perdikaris. "An expert's guide to training physics-informed neural networks". In: *arXiv preprint arXiv:2308.08468* (2023).

References iv

[25] S. Wang, Y. Teng, and P. Perdikaris. "Understanding and mitigating gradient pathologies in physics-informed neural networks". In: *arXiv preprint arXiv:2001.04536* (2020).

[26] C. Wu, M. Zhu, Q. Tan, Y. Kartha, and L. Lu. "A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks". In: *Computer Methods in Applied Mechanics and Engineering* 403 (2023), p. 115671.

[27] Z. Xiang, W. Peng, X. Liu, and W. Yao. "Self-adaptive loss balanced Physics-informed neural networks". In: *Neurocomputing* 496 (2022), pp. 11–34.

Jacobian-vector product: example

- Take a simple scalar activation function $f\colon x \mapsto \tanh(x)$. Its associated JVP is a function

$$(x, v) \mapsto \tanh'(x)v = (1 - x^2)v, \forall (x, v) \in \mathbb{R}^2$$

- Take the example of a linear layer function $f\colon x \mapsto wx + b, x \in \mathbb{R}^n, w \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$. Its associated JVP is a function

$$(x, v) \mapsto \frac{\mathsf{d}f}{\mathsf{d}x} v = wv, \forall (x, v) \in (\mathbb{R}^n)^2$$

- Remark that we expressed the JVPs as expression devoid of Jacobians. Composing the JVPs in forward AD is thus efficient