Scientific machine learning: principle

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

Numerical Analysis and PDE seminar of Rennes

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

Outline

Introduction

Maching learning: principle

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

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

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

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

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

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

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

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

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

basic approaches: linear regression

- **Linear regression** for $f : \mathbb{R}^d \to \mathbb{R}$
- We want to solve:

$$\min_\theta \sum_i^n \| f_\theta(\mathbf{x}_i) - y_i \|_2^2$$

with $f_\theta \langle \theta, \mathbf{x} \rangle + b$

- Matricial form for the problem:

$$\min_\theta \| A\theta - Y \|_F^2$$

with $A_i j = x_i^j$ and $Y_i = y_i$.

Solution

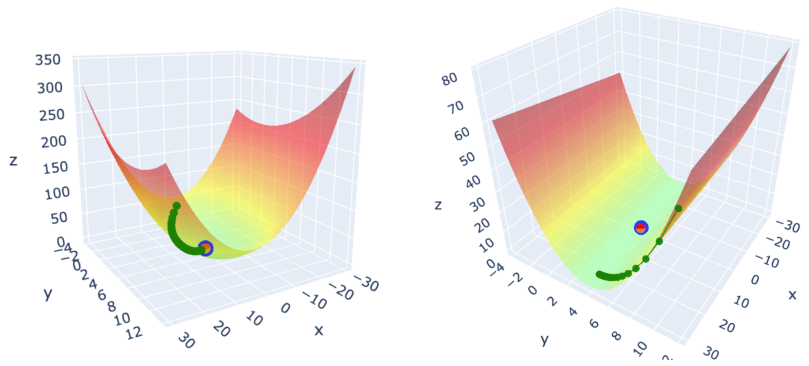
It is a convex problem. The solution is given by the solution of the normal equation

$$A^t A\theta = A^t Y$$

In general unique solution if $n > d$

Overfitting

if $d > n$ non uniqueness of the solution. **Overfitting**: good on the training data, false on test data.



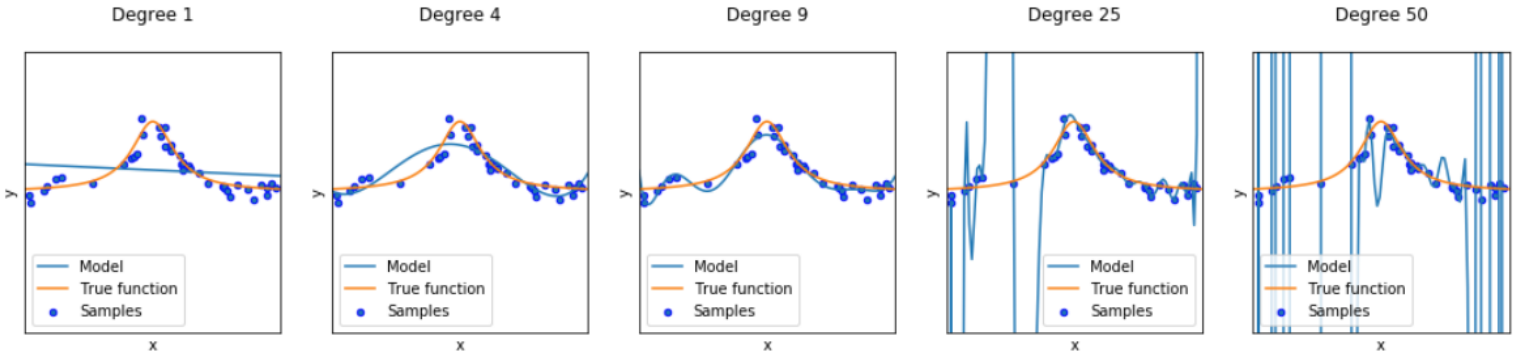
basic approaches: nonlinear regression I

- Nonlinear extension: Polynomial regression

$$\min_\theta \sum_i^n \| f_\theta(\mathbf{x}_i) - y_i \|_2^2$$

with $f_\theta(\mathbf{x}) = \langle \theta, P(\mathbf{x}) \rangle$.

- Example of the Runge function:



Overfitting

In the over-parametrized regime (more parameters than data), we can approximate very well the data and admit poor generalization property for new data

- The over-parametrized regime is common in large dimensions.
- When the polynomial model have too much freedom we obtain oscillatory behavior.
- Another approach: kernel regression (and Gaussian process). Theory based on reproducing kernel Hilbert space.

Reproducing kernel Hilbert space

V metric space. H Hilbert space of real function defined on V. A function $k : V \times V \to \mathbb{R}$ is called reproducing kernel is

- H contains on the functions of the form

$$\forall x \in H, \quad k_x(y) \to k(x, y)$$

- $\forall x \in V, f \in H$ we have:

$$f(x) = \langle f, K_x \rangle_H$$

In this case H is a reproducing kernel Hilbert space (RKHS).

Representation theorem

We have a k the kernel and H_k the associate RKHS. We consider a data set $(x_1, ..., x_n) \in V$ and $(y_1, ..., y_n) \in \mathbb{R}$. We consider a loss $L(x, y) \in \mathbb{R}^2 \to \mathbb{R}$ and $\lambda > 0$. The solution of the problem:

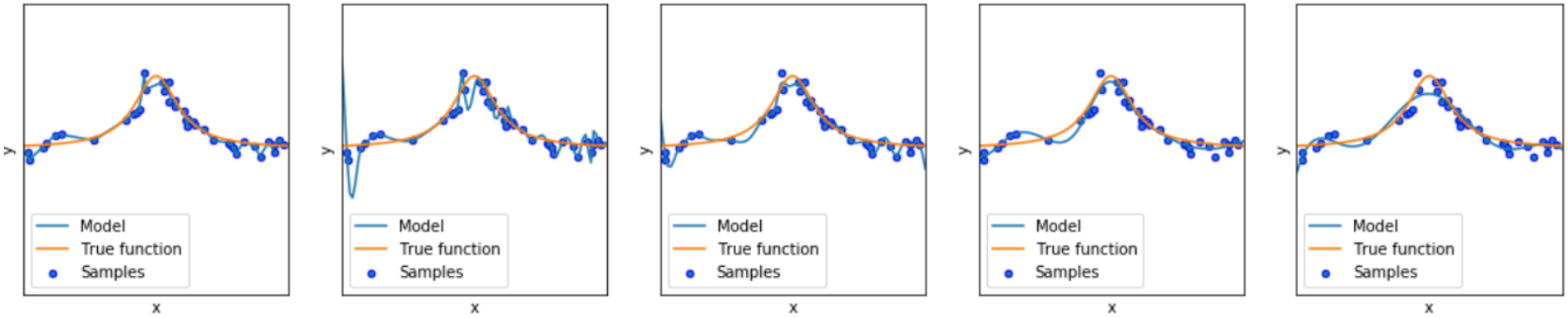
$$\min_{f \in H_k} \left(\sum_{i=1}^{n} L(f(x_i), y) + \lambda \parallel f \parallel_{H_k} \right)$$

is of the form:

$$f(x) = \sum_{i=1}^{n} \alpha_i k(x, x_i)$$

basic approaches: nonlinear regression III

- The theorem gives the optimal form of parametric function in RKHS. We can apply a regression to obtain a mean square problem.
- It works if H is sufficiently large to approximate many functions.
- **Key point**: the Matern kernel parametrized by ν gives all the Sobolev space which are contained in the continuous function space.



- Left to right. $H = H^1$, H^2, H^3, H^6, C^∞.
- The choice of the kernel can be viewed as a regularity prior.
- **Kernel regression** generated functions smoother or lower frequency than polynomial models. Better for generalization.
- Necessary to store the data. Expensive for large data set.

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.

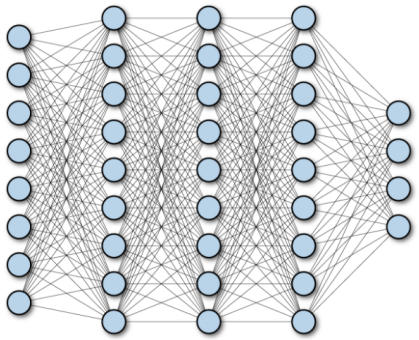
Neural network

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

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

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

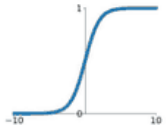
- Fully connected neural network (FCNN): the matrices $A_{l,l+1}$ are dense.



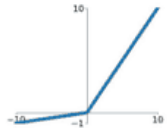
Activation functions

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

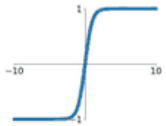
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



Leaky ReLU
$\max(0.1x, x)$

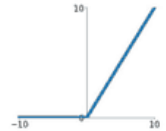


tanh
$\tanh(x)$

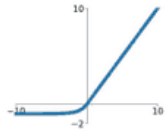


Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$

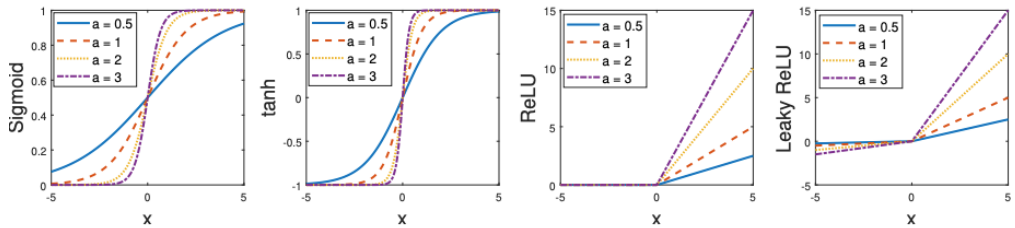


ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



- Adaptive activation functions (we learn the parameter).

$$\begin{aligned} \text{Sigmoid}: & \ \frac{1}{1+e^{-ax}}, \\ \text{Hyperbolic tangent}: & \ \frac{e^{ax}-e^{-ax}}{e^{ax}+e^{-ax}}, \\ \text{ReLU}: & \ \max(0, ax), \\ \text{Leaky ReLU}: & \ \max(0, ax) - \nu \, max(0, -ax). \end{aligned}$$



Training and gradient methods

Key point

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

- We train the Neural network with gradients type approach. It is a non-convex problem.
- Full gradient:

$$\nabla_\theta \left(\sum_i^n \parallel f_\theta(\mathbf{x}_i) - y_i \parallel_2^2 \right) = \sum_i^n \nabla_\theta \parallel f_\theta(\mathbf{x}_i) - y_i \parallel_2^2$$

- If $n >> 1$ the gradient is very costly. To avoid that and add some exploration we use stochastic gradient

$$\sum_i^n \nabla_\theta \parallel f_\theta(\mathbf{x}_i) - y_i \parallel_2^2 \approx \mathbb{E}_{x\sim\mu} \parallel f_\theta(\mathbf{x}) - y \parallel_2^2 \approx \sum_k^m \nabla_\theta \parallel f_\theta(\mathbf{x}_{i(k)}) - y_{i(k)} \parallel_2^2$$

Back-propagation

How compute $\nabla_\theta f_\theta(\mathbf{x})$ with L layers with $f_\theta(\mathbf{x}) = f^n_{\theta_n} \circ ..... \circ f^1_{\theta_1}(\mathbf{x})$ using gradient chain rules we obtain:

$$\nabla_\theta \mathcal{L}(\theta) = \sum_{i=1}^n \nabla_{\mathbf{h}_i} \mathcal{L}(\theta) \frac{\partial f^i_\theta}{\partial \theta}, \quad \text{and} \quad \nabla_{\mathbf{h}_i} \mathcal{L}(\theta) = \nabla_{\mathbf{h}_{i+1}} \mathcal{L}(\theta) \frac{\partial f^{i+1}_\theta}{\partial \theta}$$

Depth, expressivity and stability

Theorem of Cybenko (89)

$\sigma()$ the sigmoid activation.The set of fully connected neural network is dense in $C^0([0,1],\mathbb{R})$.

Theorem of Barron

We consider $f(x) : [0,1]^d \to \mathbb{R}$ such that the Fourier transform satisfy

$$\gamma(f) = \int \| \omega \|_1^2 | \hat{f}(\omega) |^2 \, d\omega < +\infty$$

then there exist an FCN with σ the ReLu function such that

$$\| f - f_\theta \|_{L^2} \leq \frac{3\gamma(f)}{n}$$

- How the number of layers allows increasing the **expressivity** ? Example.

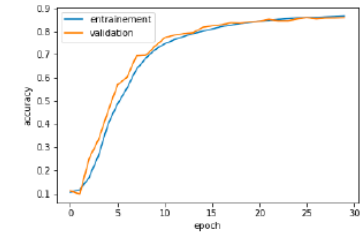
Theorem of B. Després (22)

For a class of polynomial $H(x)$ the set of ReLu neural networks $\boldsymbol{f}_\theta(x)$ at l layers converges with the following estimate

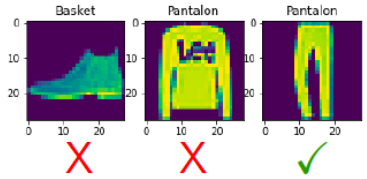
$$\| \boldsymbol{f}_\theta(x) - H(x) \|_{L^\infty} \leq C^l \| H(x) \|_{L^\infty}$$

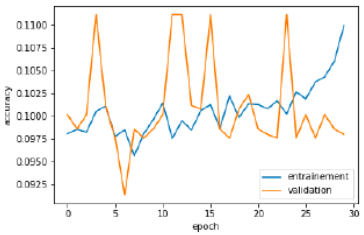
Stability

- Default of stability.
- Four and six layer FCN (30 neuronal by layer) with sigmoid.

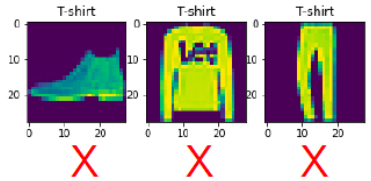


Précision : 87.1%, 86.0%

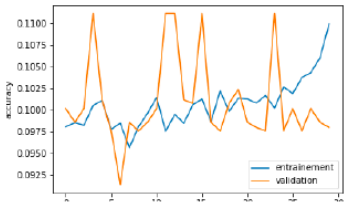




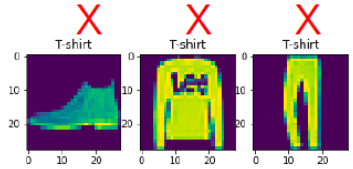
Précision : 10.0%, 9.8%

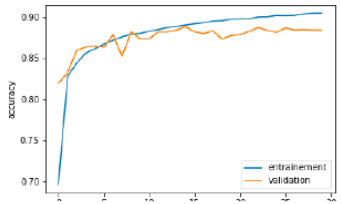


- **Vanishing gradient**: the gradient of some layers falls to zero and blocks learning.
- Replacing sigmoid by Relu:

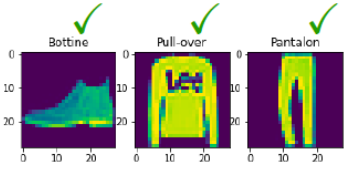


Précision : 10.0%, 9.8%



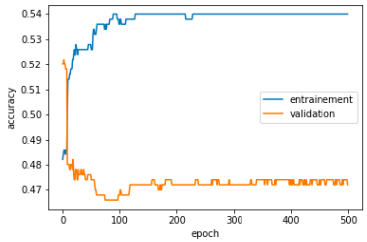


Précision : 91.0 %, 88.4 %

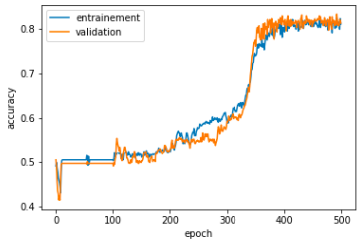


Stability

- Default of stability.
- Other test: tanH (6 layers), Relu (6 layers) and Relu (20 layers)

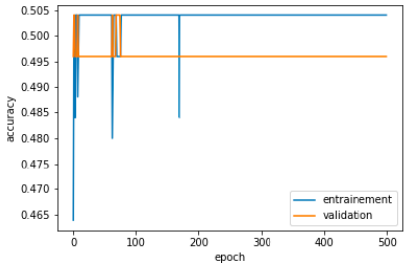


Précision : 54.0%, 47.2%

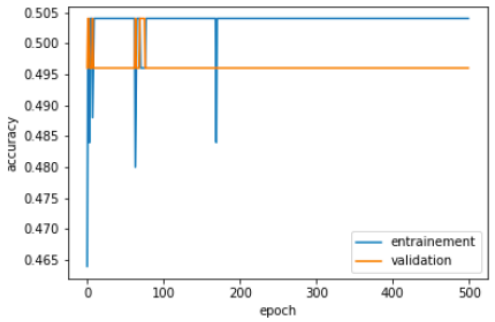


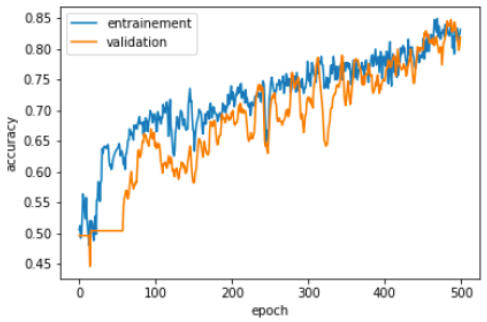
Précision : 81.4%, 81.4%

Précision : 50.4%, 49.6%



- Possible solution: normalize the data at each layer.

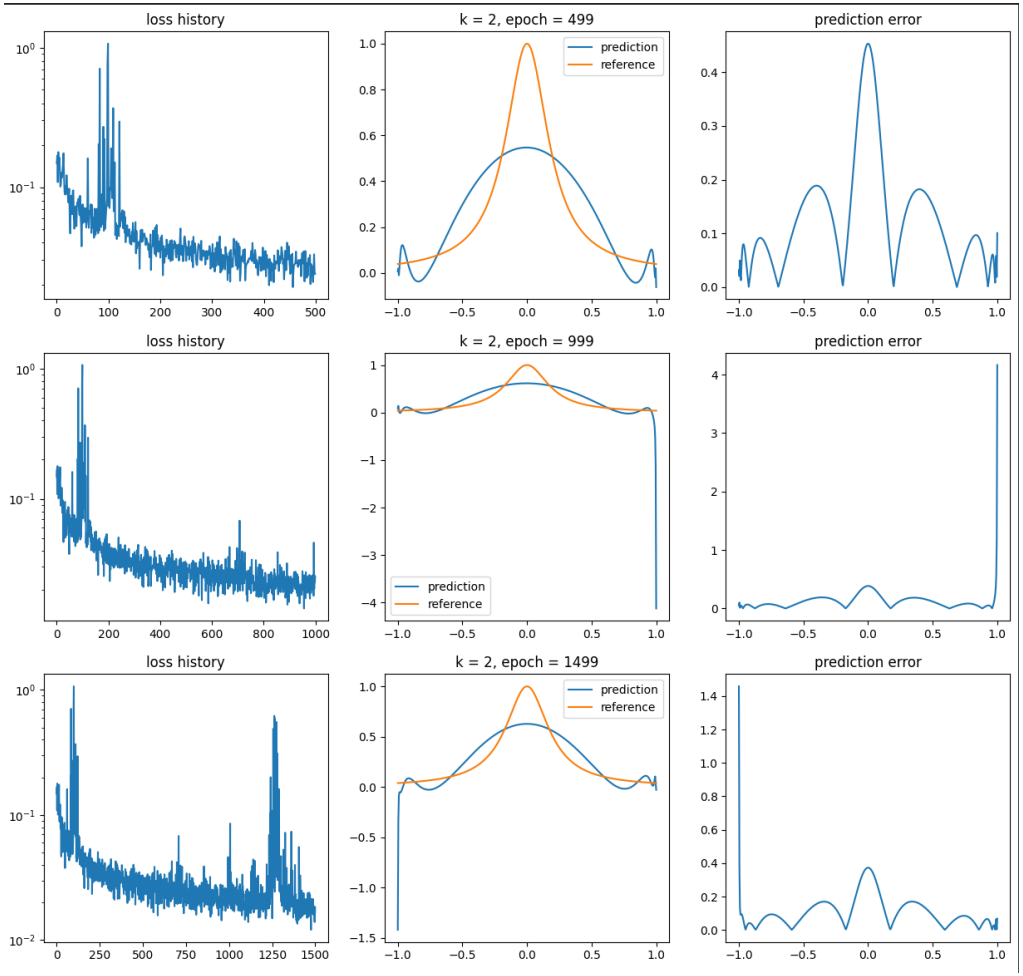




- Comparison of Relu network (20 layers) without and with batch normalization.

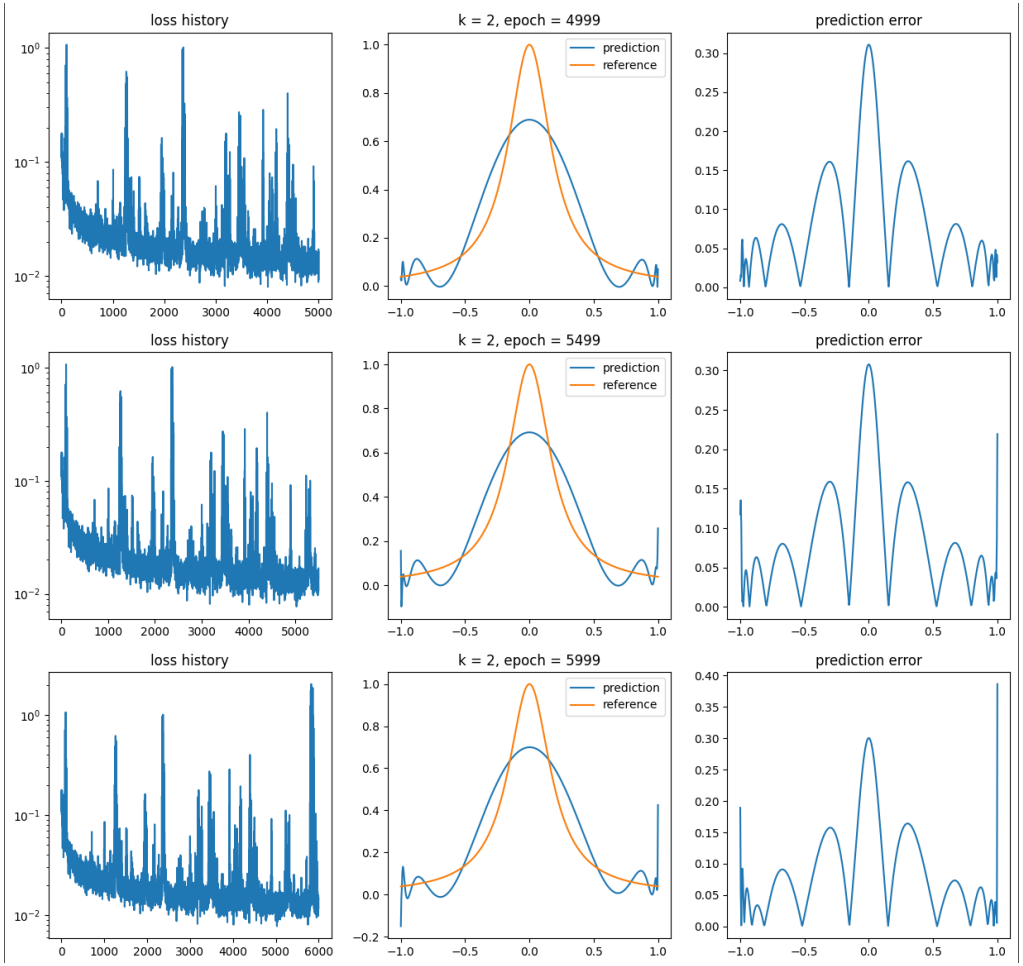
NN vs Polynomial

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



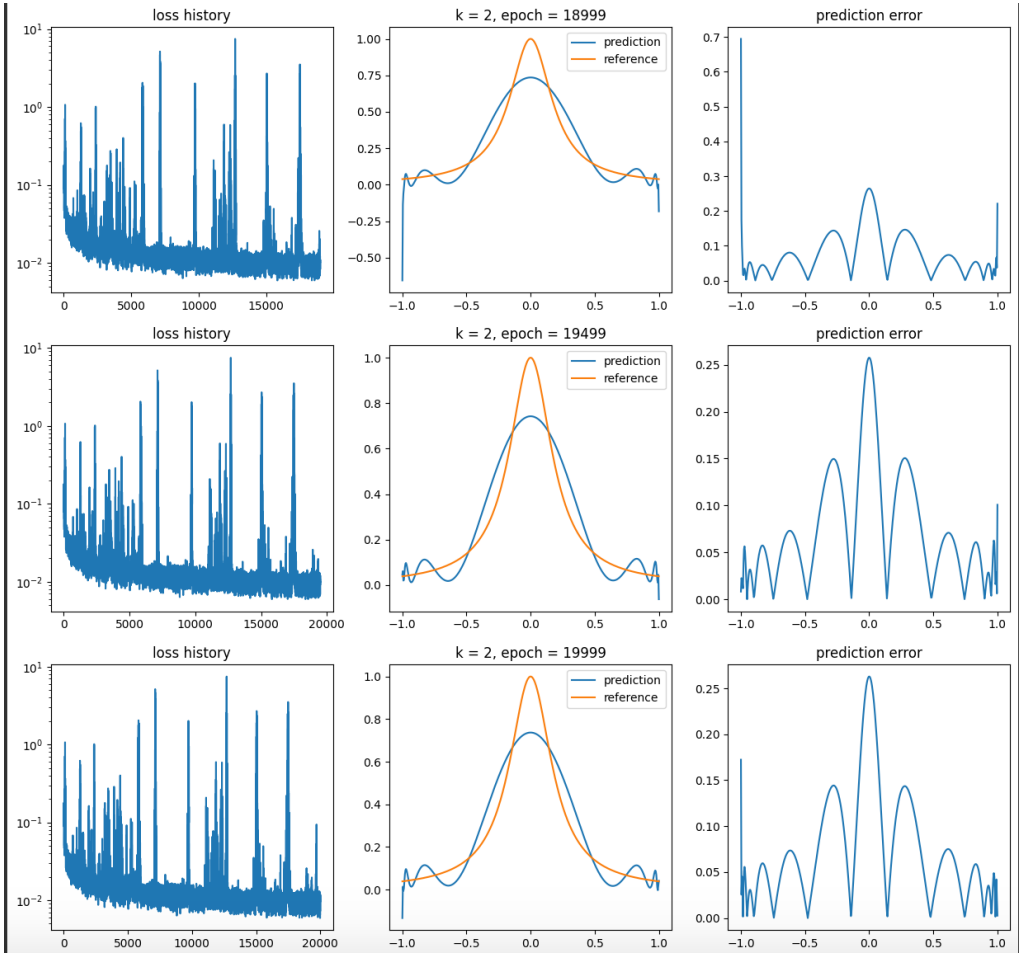
NN vs Polynomial

- 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

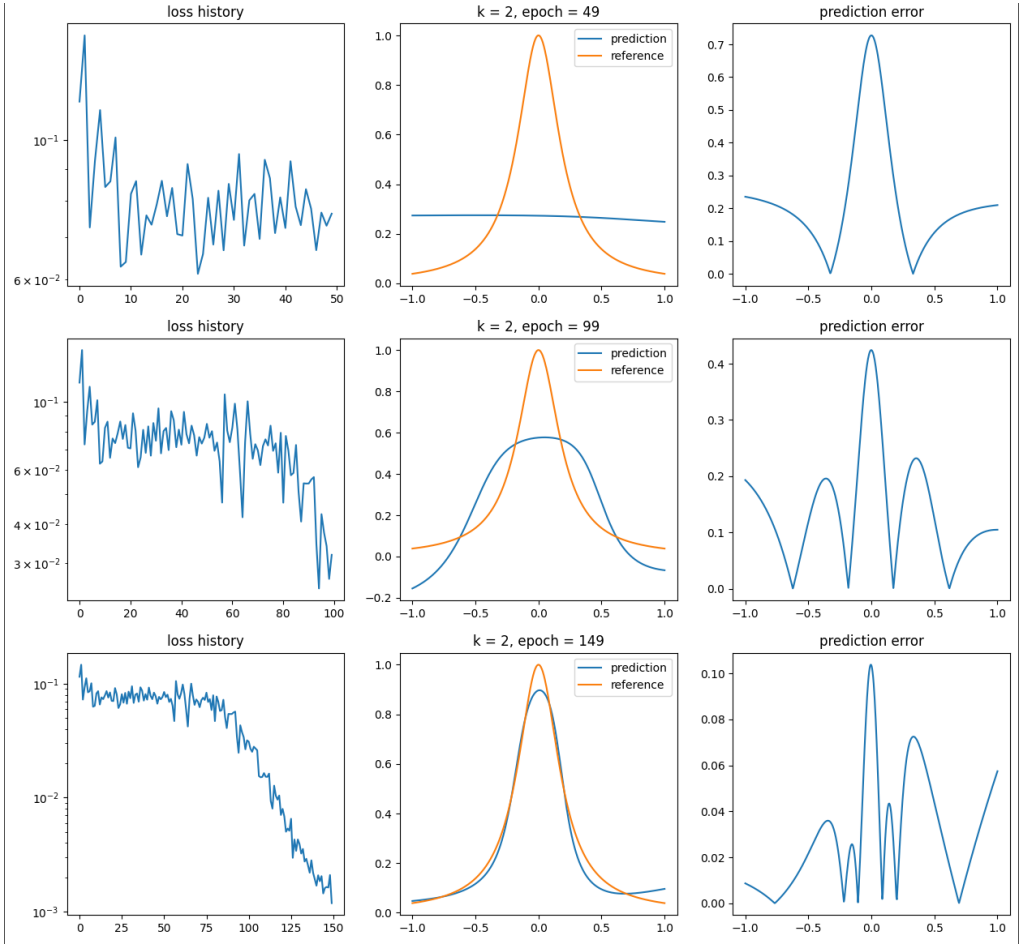
- We will compare over-parametrized NN and polynomial regression on the Runge function.
- 120 data and approximately 800 parameters in each model.



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

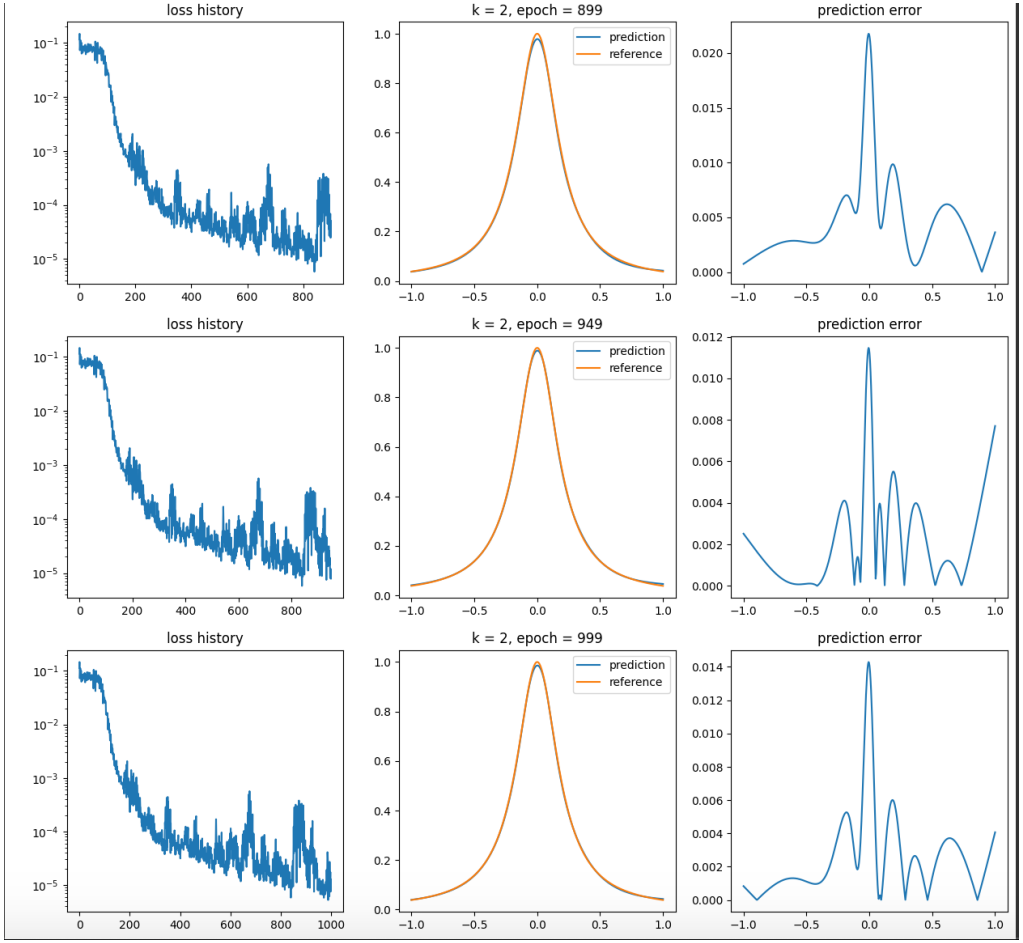
NN vs Polynomial

- 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

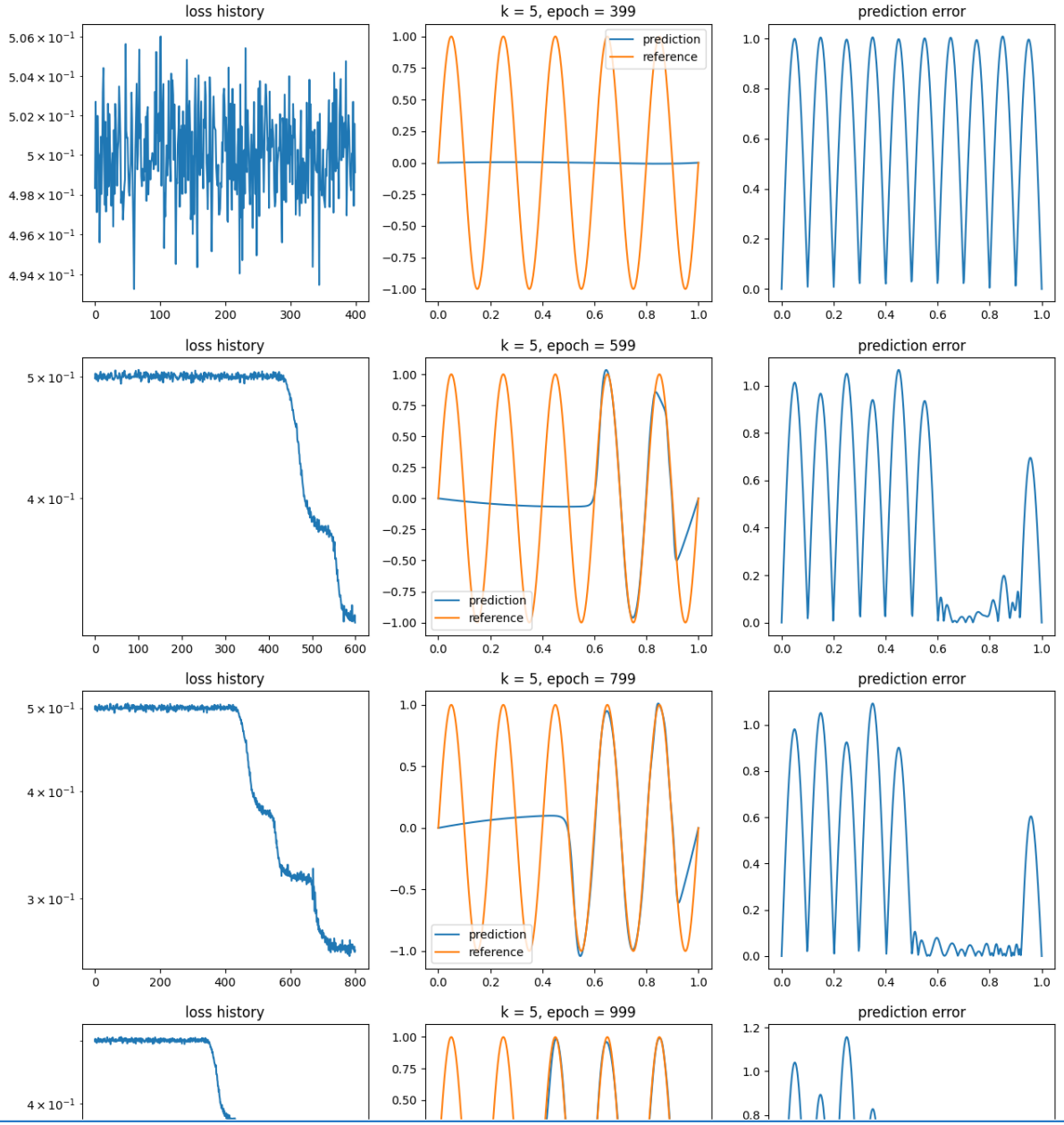
- We will compare over-parametrized NN and polynomial regression on the Runge function.
- 120 data and approximately 800 parameters in each model.



- 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

NN vs Polynomial

- Other example of ANN learning.



Convolutive neural networks I

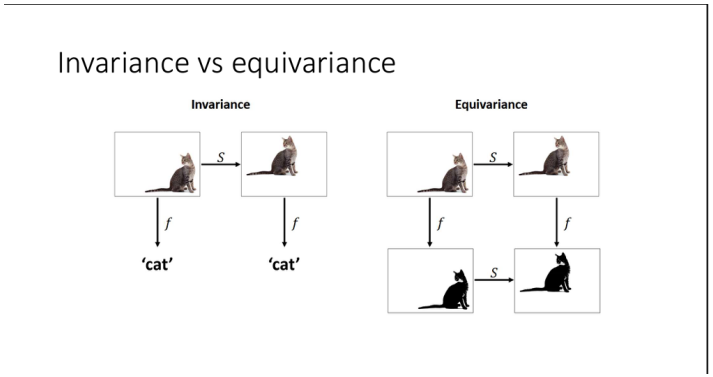
Limites

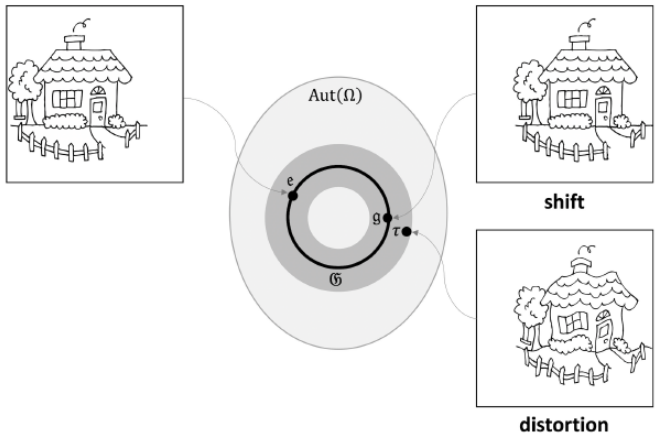
Fully connected NN non-sufficient for large-dimensional problems like image processing or language.

Other neural networks

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

- Example: classify pictures. We want to construct $g(f) : f(x, y) \to s \in [0, 1]$ with $f(x, y)$ a signal (discretized in practice).
- The CNN introduce some priors on the problem in the architecture.
- Priors: **Geometric** (left), **Stability for small deformation** (right).



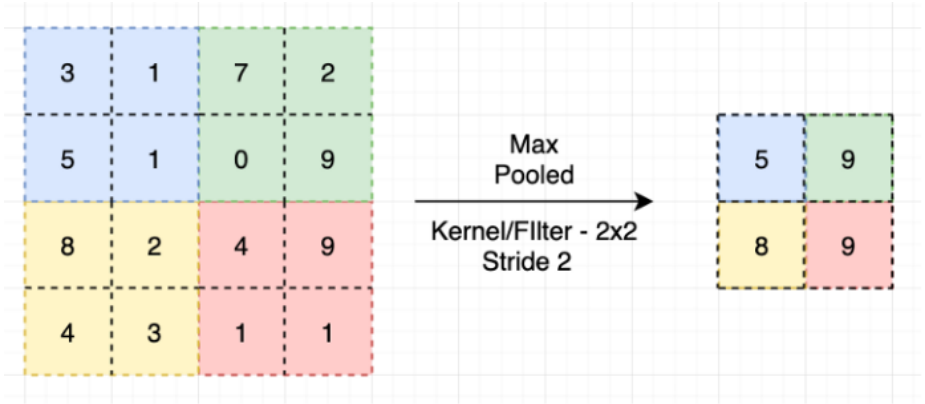


Convolutive neural networks II

- **Encoding geometric prior**:
  - Convolutional layers: the matrices A_i are Toeplitz matrices (shift-invariant).

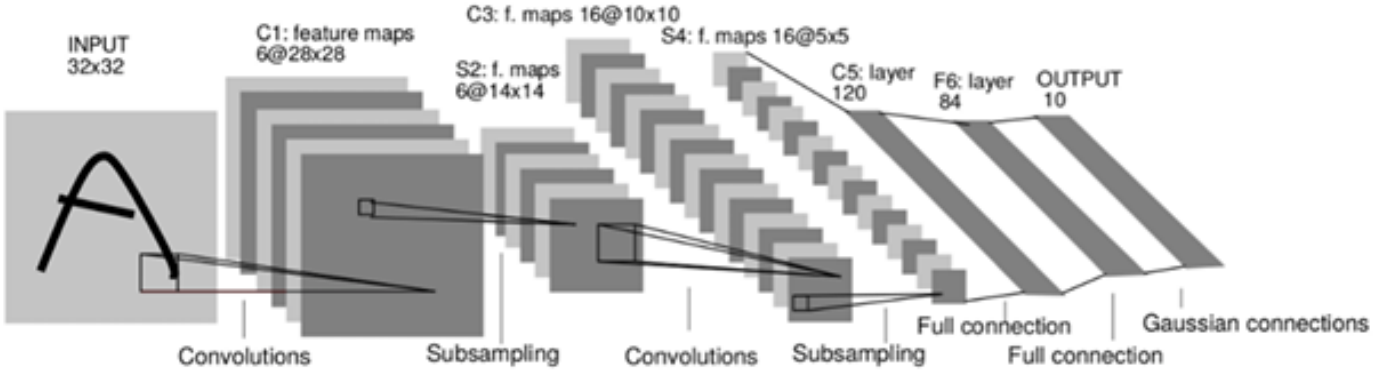
$$A = \begin{pmatrix} a & c & e & 0 \\ b & a & c & e \\ d & b & a & c \\ 0 & d & b & a \end{pmatrix}$$

  - This is equivalent to applying a convolution kernel to the 1D signal.
  - We often apply some convolutions to the signal on each layer, to create several new signals.
- **Encoding stability prior**: use local filters (equivalent to sparse matrix A_i)
- **Last prior**: Multi-scale separation. We can begin by the local scale and move to the global one.
- For that we altern convolutional layers with pooling layers.

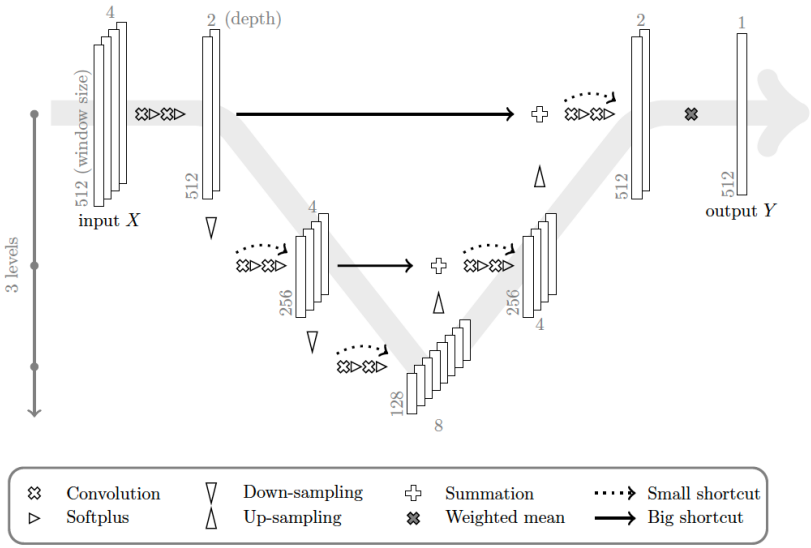


Convolutive neural networks III

- Example of 2D convolutional network for classification:



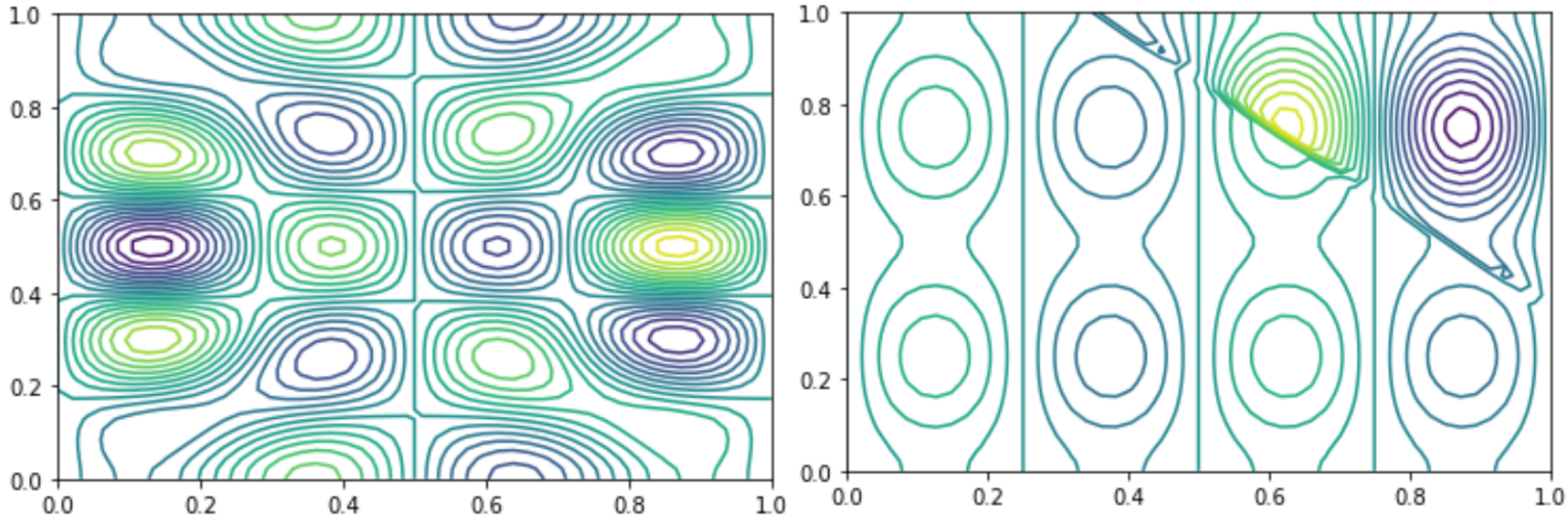
- Example of 2D CNN called Unet for regression:



- Study of the CNN: Mallat (Collège de France lecture)

Convolutive neural networks IV

- Comparison between CNN and FC on signal problems.
- Classify function 48*48 with or without a discontinuity.



- Results for 1 minute training:
  - FC 1: 566000 parameters. Error: $\approx 50\%$ of succes
  - FC 2: 236000 parameters. Error: $\approx 48\%$ of succes
  - CNN: 41000 parameters. Error: $\approx 92\%$ of succes
- Extension of CNN to graph, mesh, manifold ("Geometric Deep Learning" Bronstein and al 22) encoding other geometric prior.
- Recurrent neural network for time series and transformers for language.

Physics-informed Neural Networks

PINN's I

- We solve PDE 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

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

- The first idea comes from the remark that neural networks are smooth functions compared to the inputs.Since the derivative are easily computable by automatic derivative, the ANN are natural objects to approximate PDE solution.
- A PINNs is a neural network which as inputs (t,x), and we note $\mathbf{U}_\theta(t,x)$.

Basic approach

If we have data $\boldsymbol{U}_i^n$ which approximate the solution of the points (x_i, t_n) we will learn the NN 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 do that without data or with few data ??

PINNs approach

Since we can derivate 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:

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

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

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

with

$$J_r(\theta) = \int_0^T \int_\Omega \parallel R(t,x) \parallel_2^2 \, dxdt$$

and

$$J_b(\theta) = \int_0^T \int_{\partial\Omega} \parallel \mathbf{U}_\theta(t,x) - g(x) \parallel_2^2 \, dxdt, \quad J_i(\theta) = \int_\Omega \parallel \mathbf{U}_\theta(0,x) - \mathbf{U}_0(x,\boldsymbol{\alpha}) \parallel_2^2 \, dx$$

Question

How compute the integrals of the residues ?

How compute the integrals of the residues ?

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

- The Monte-Carlo method comes 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 to:

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

  with X_i an random example sampled with the law μ
- It allows computing integral. 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

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

  with x_i points sampled uniformly on Ω.

PINNs III

- Applying the MC method to the PINNs We obtain the following minimization problem:

final PINNs minimization

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

with

$$J_r(\theta) = \sum_{n=1}^{N}\sum_{i=1}^{N} \parallel R(t_n, x_i) \parallel_2^2$$

with (t_n, x_i) sampled uniformly and

$$J_b(\theta) = \sum_{n=1}^{N_b}\sum_{i=1}^{N_b} \parallel \mathbf{U}_\theta(t_n, x_i) - g(x_i) \parallel_2^2, \quad J_i(\theta) = \sum_{i=1}^{N_i} \mathbf{U}_\theta(0, x_i) - \mathbf{U}_0(x_i) \parallel_2^2$$

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

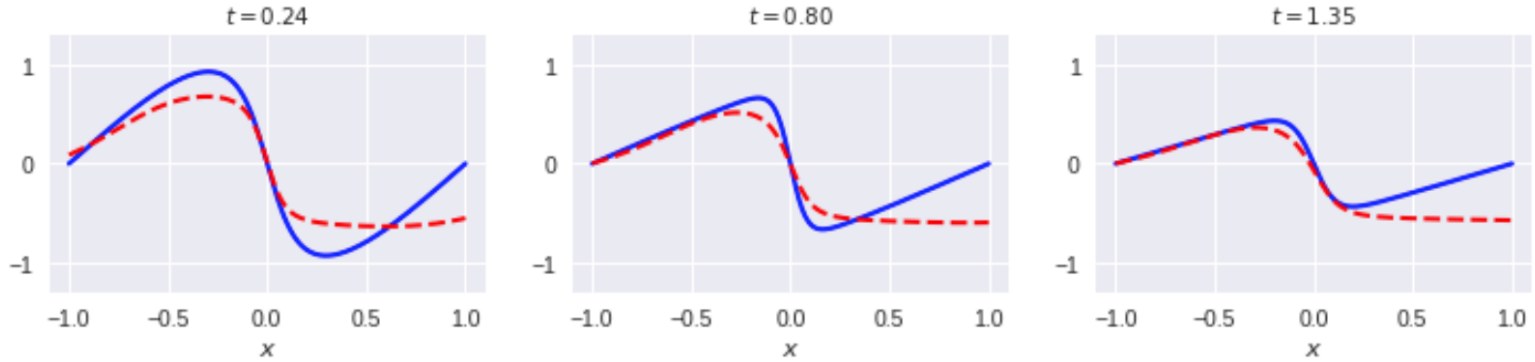
$$\bar{u}_\theta(t, x) = u_0(x) + t(\phi(x) * u_\theta(x))$$

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

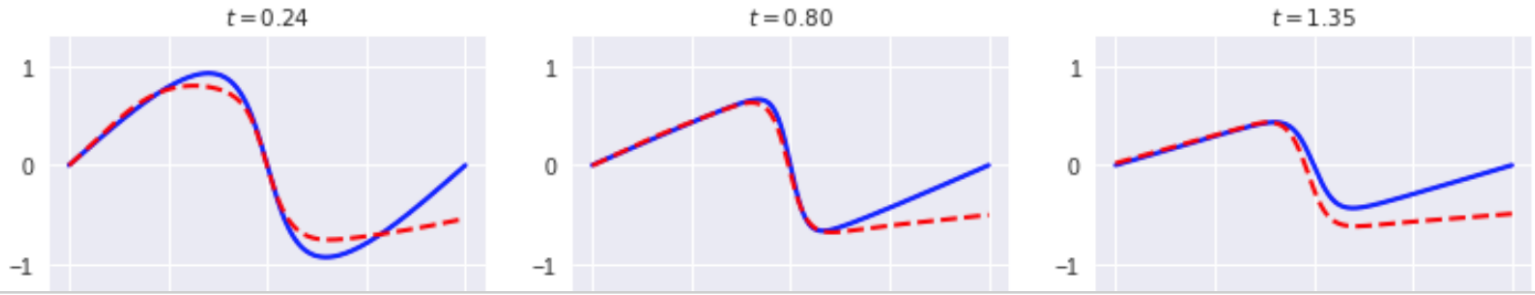
Example: Burgers equation

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

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



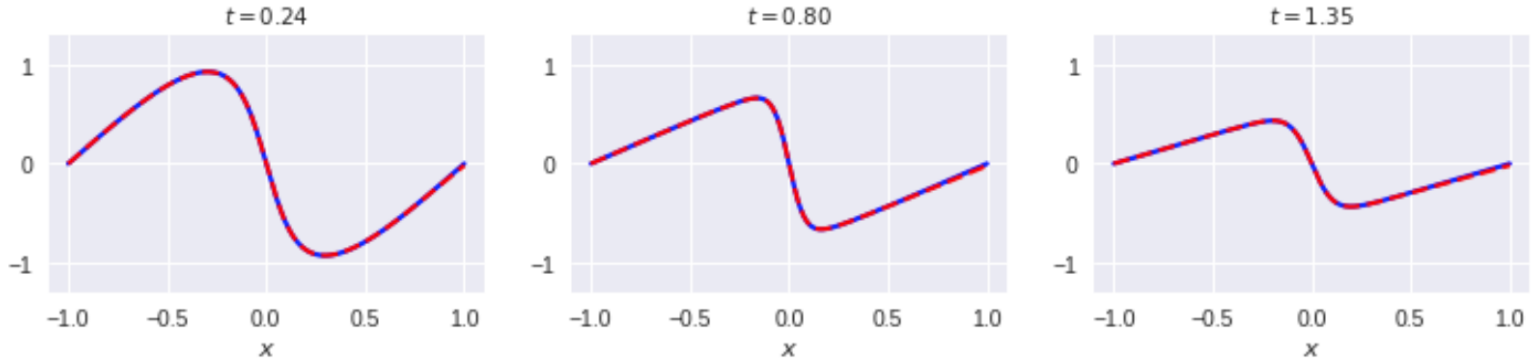

```
iter =  400
loss = 0.0318
L2 error: 4.0850e-01
```



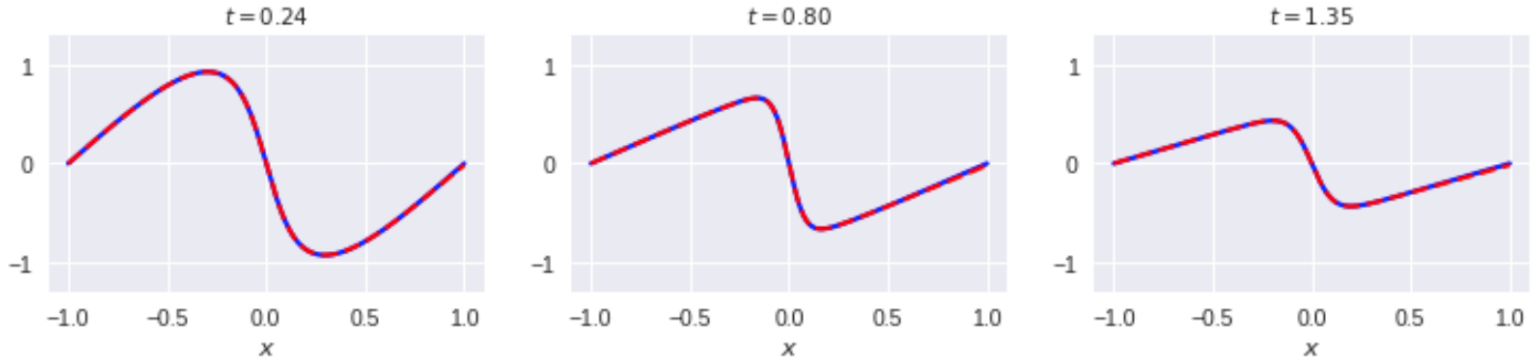
Example: Burgers equation

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

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



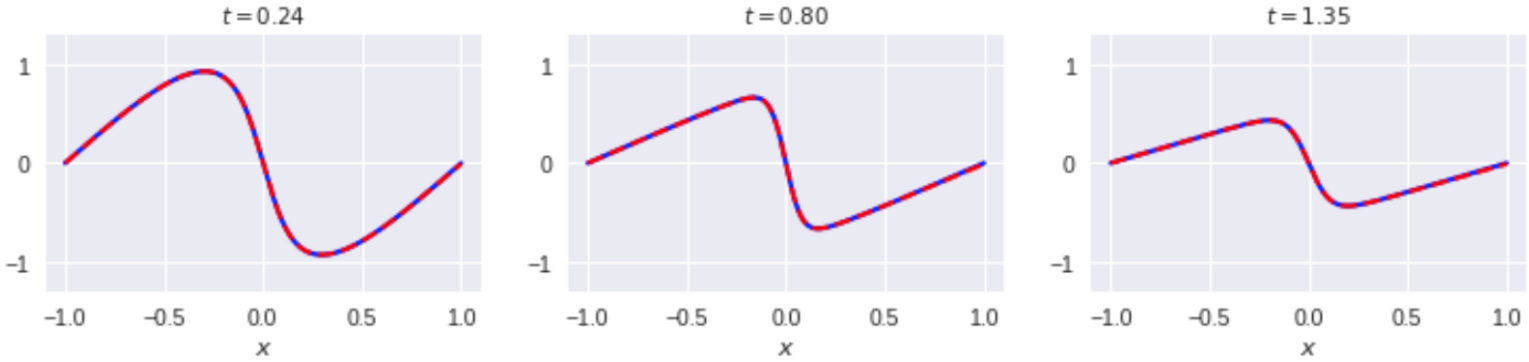

```
iter =  2200
loss = 0.0000
L2 error: 8.2614e-03
```



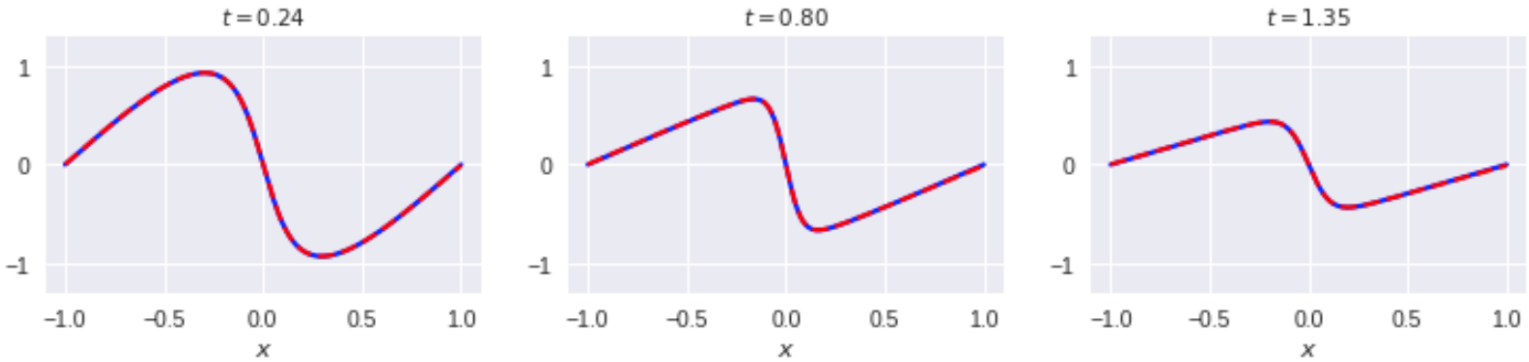
Example: Burgers equation

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

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



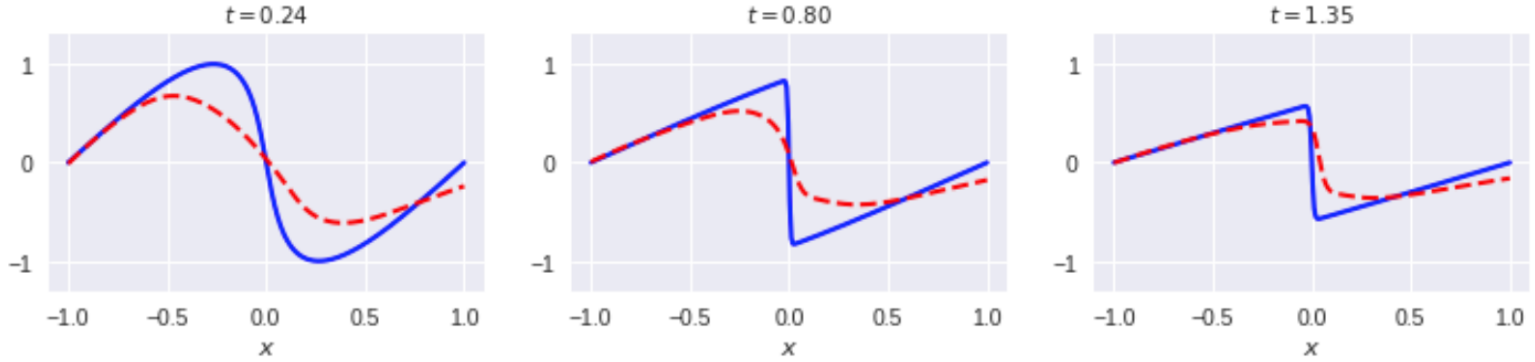

```
iter =  5000
loss = 0.0000
L2 error: 4.7307e-03
```



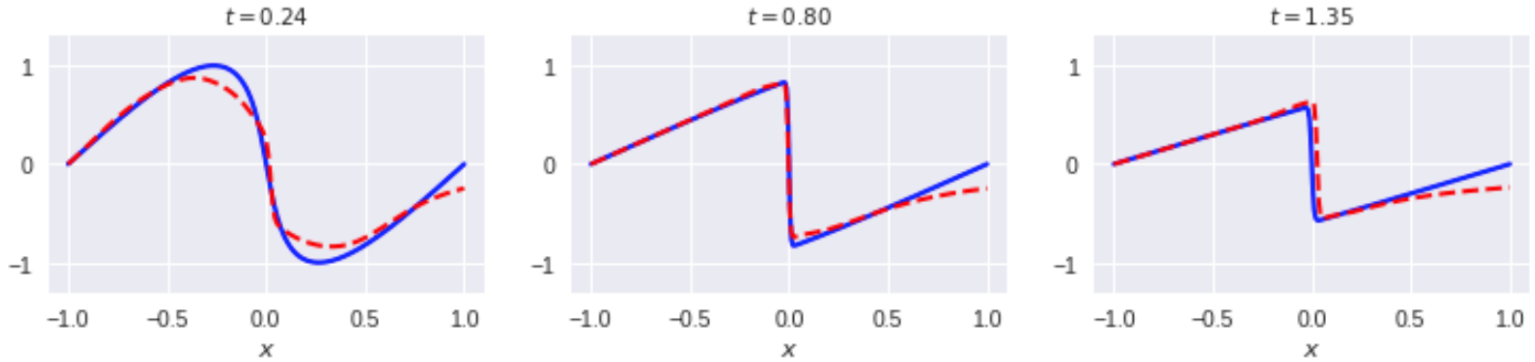
Example: Burgers equation

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

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



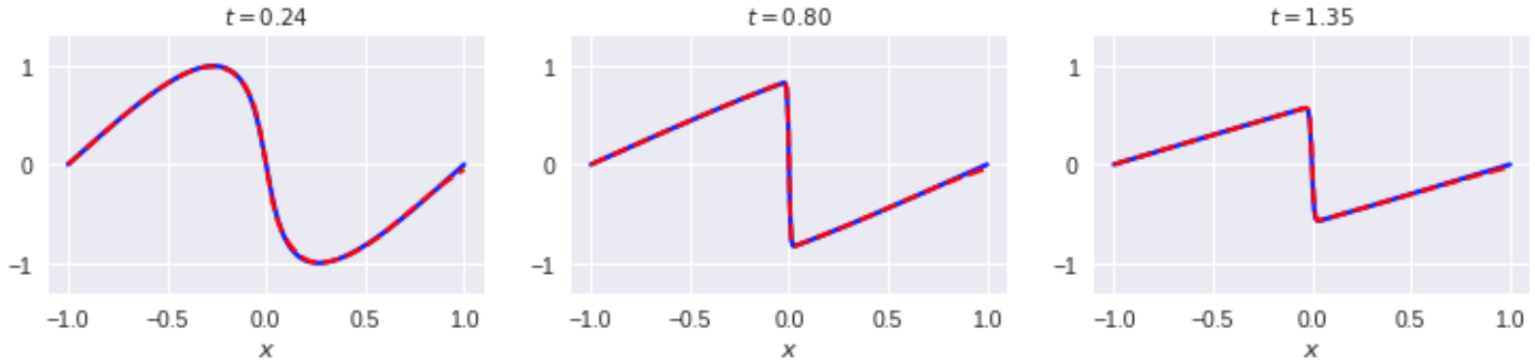

```
iter =  800
loss = 0.0233
L2 error: 2.0901e-01
```



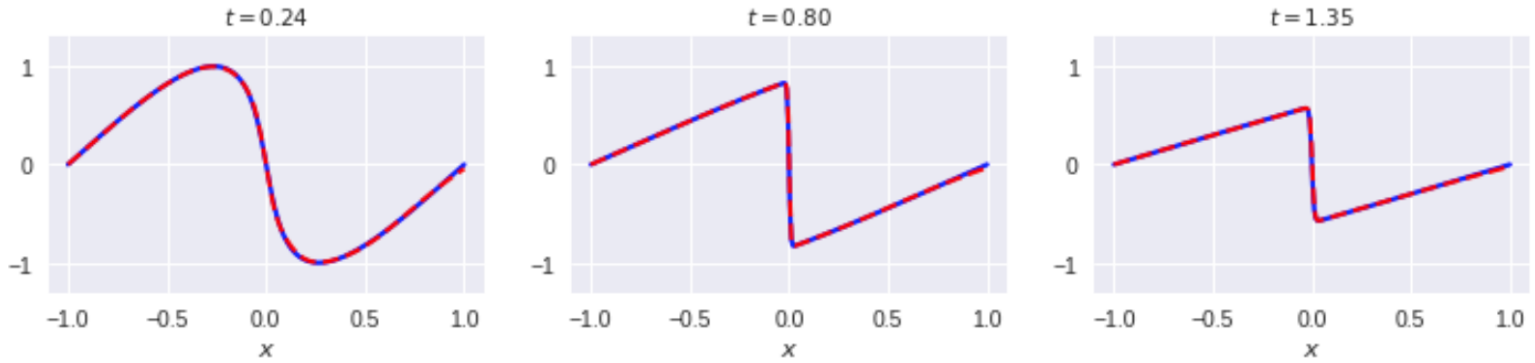
Example: Burgers equation

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

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



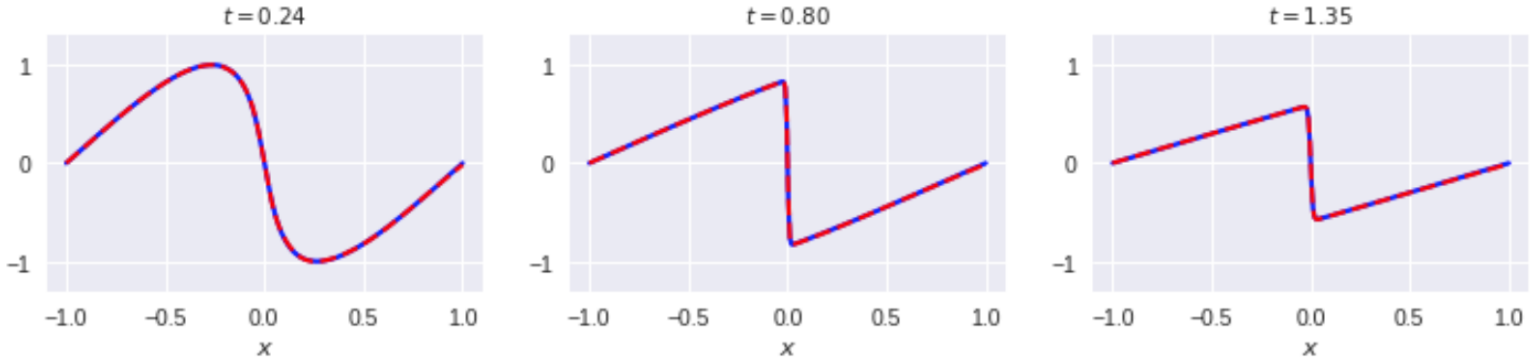

```
iter =  2200
loss = 0.0002
L2 error: 1.7773e-02
```



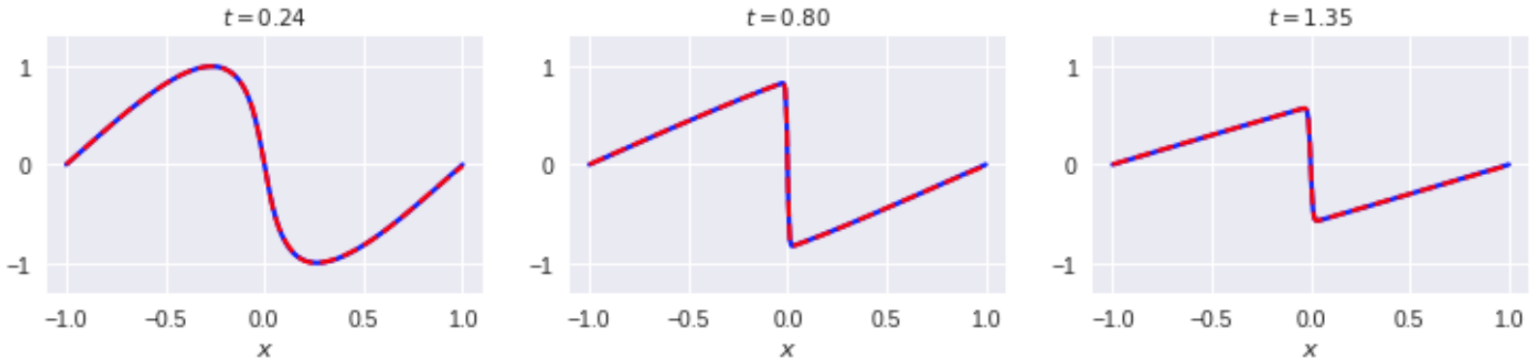
Example: Burgers equation

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

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



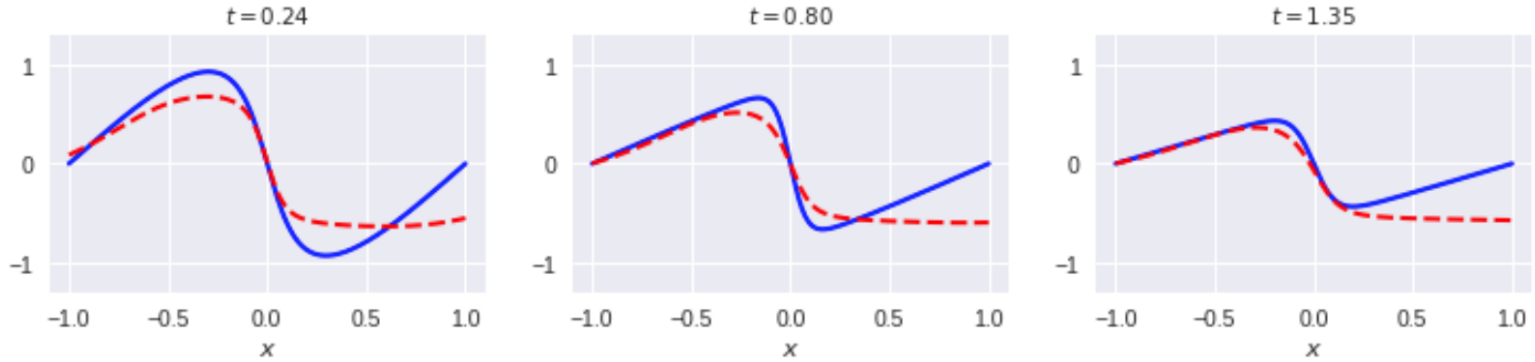

```
iter =  5000
loss = 0.0001
L2 error: 5.2593e-03
```



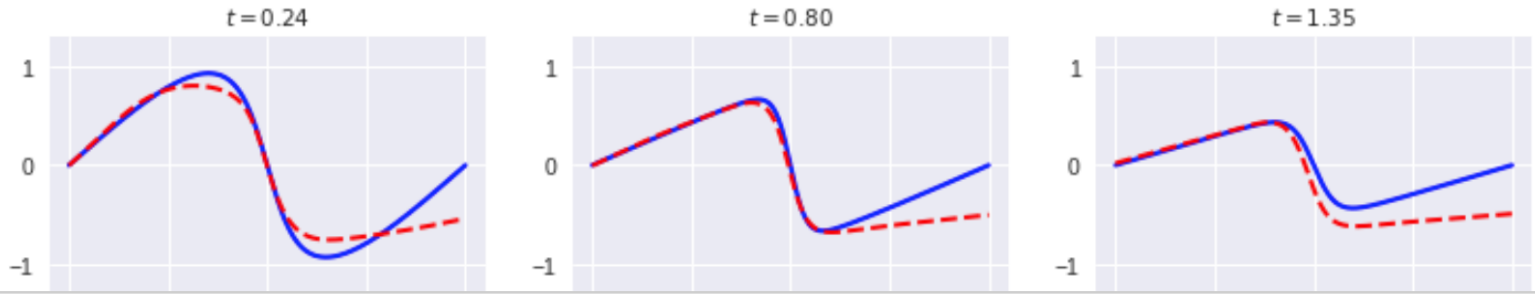
Example: Burgers equation

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

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



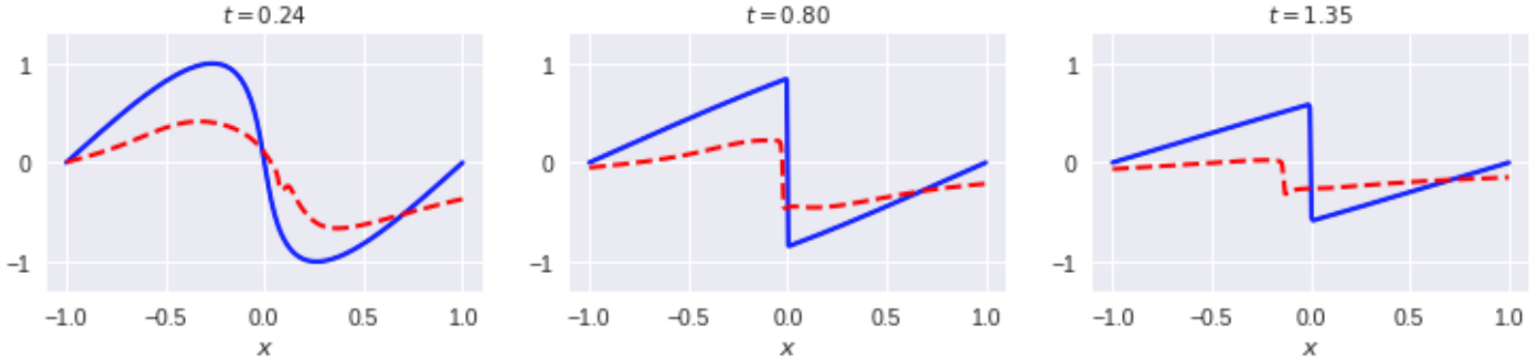

```
iter =  400
loss = 0.0318
L2 error: 4.0850e-01
```



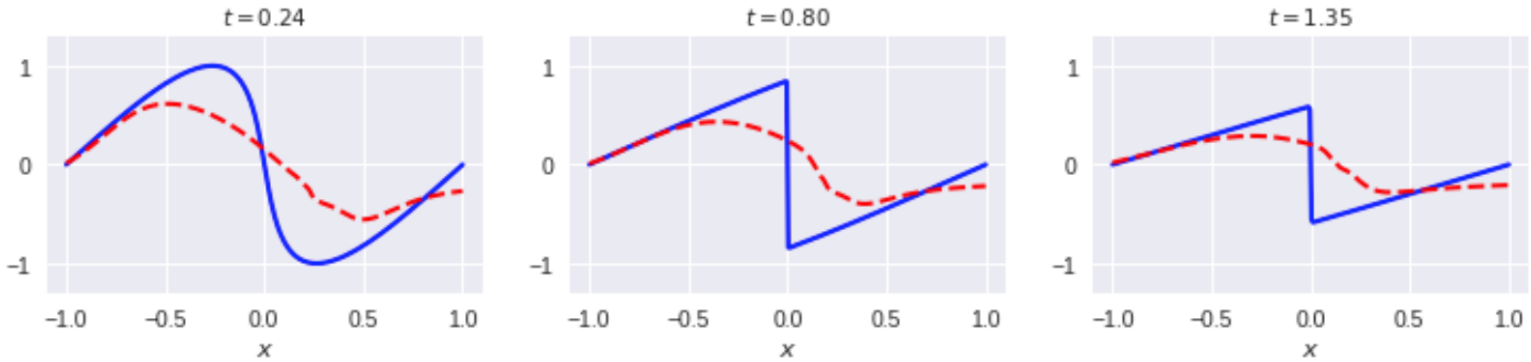
Example: Burgers equation

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

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



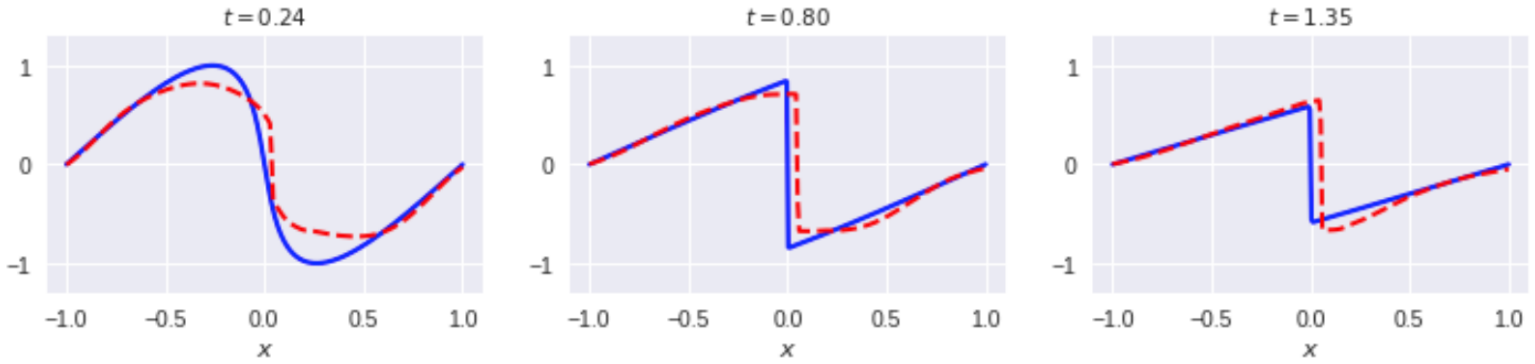

```
iter =  2200
loss = 0.1361
L2 error: 6.0138e-01
```



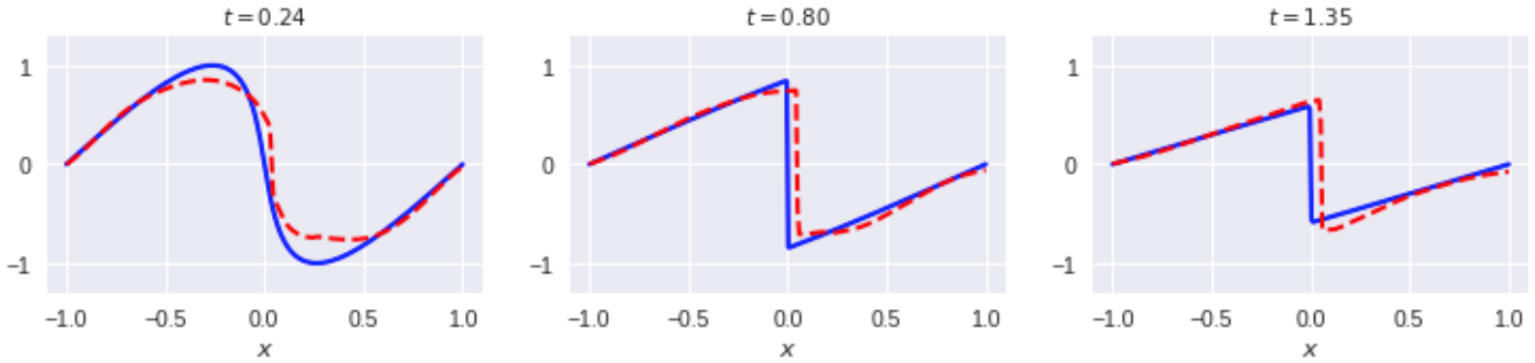
Example: Burgers equation

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

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



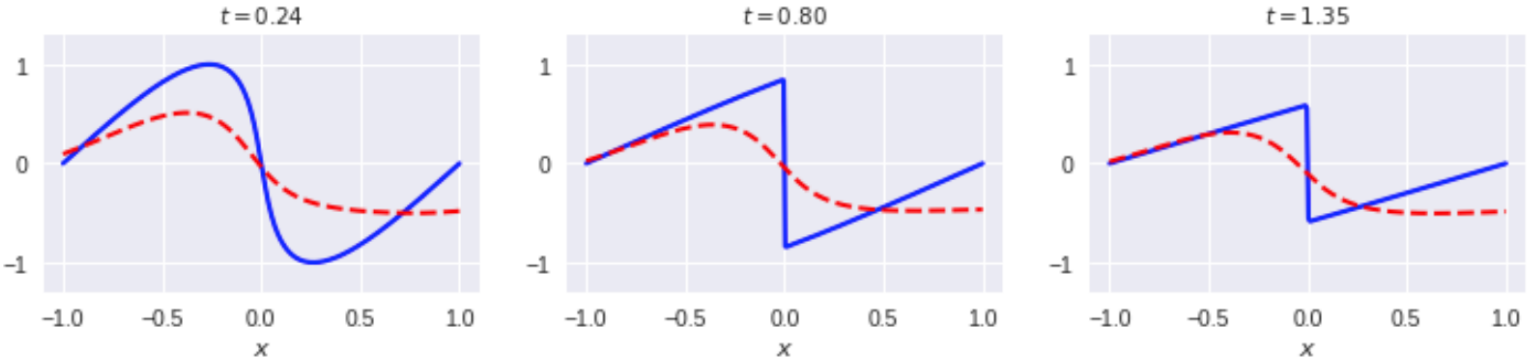

```
iter =  5000
loss = 0.0212
L2 error: 4.0300e-01
```



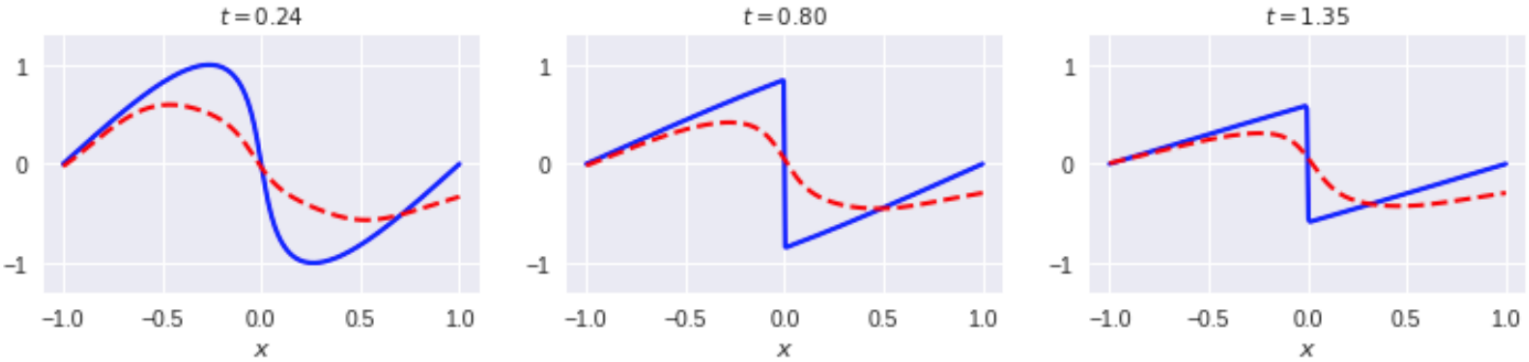
Example: Burgers equation

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

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



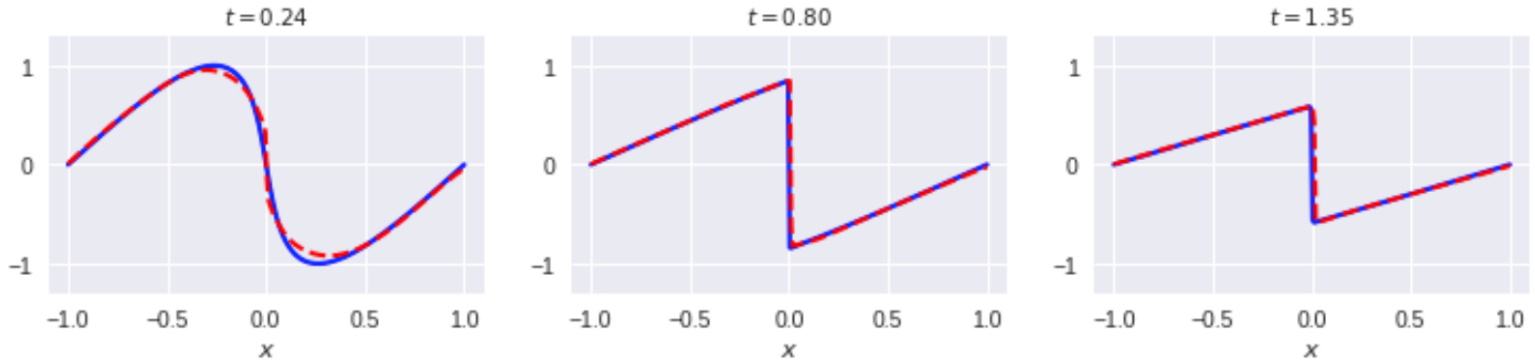

```
iter =  400
loss = 0.1170
L2 error: 5.2688e-01
```



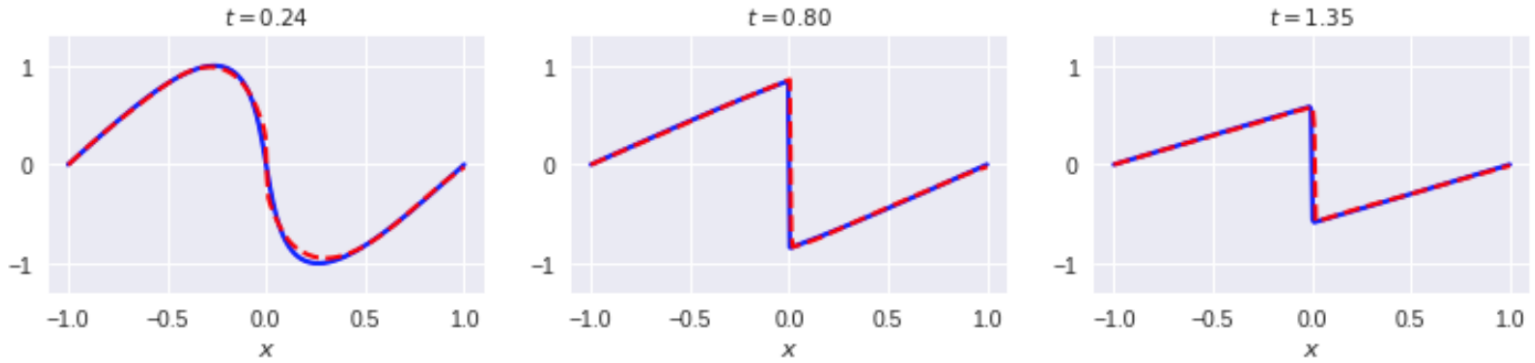
Example: Burgers equation

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

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



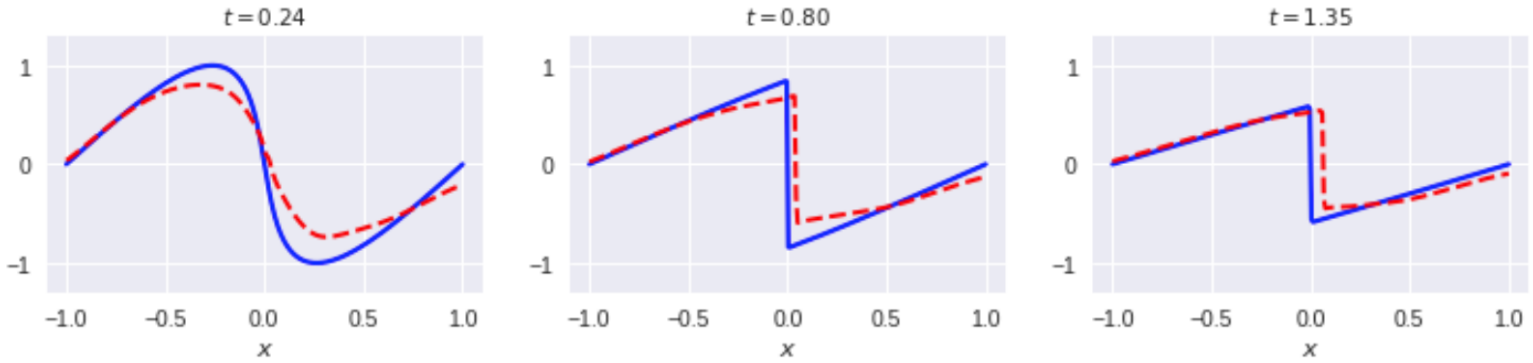

```
iter =  2200
loss = 0.0024
L2 error: 1.6838e-01
```



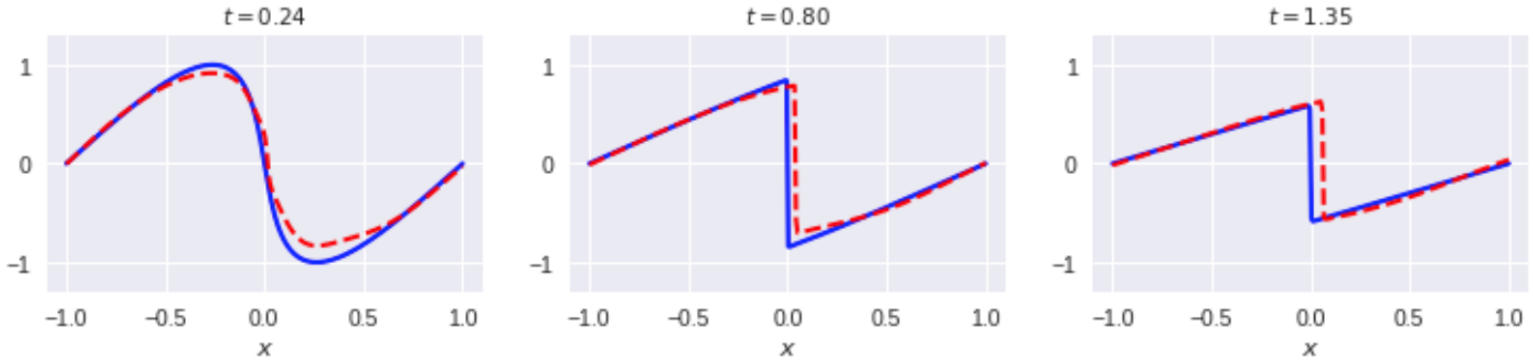
Example: Burgers equation

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

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



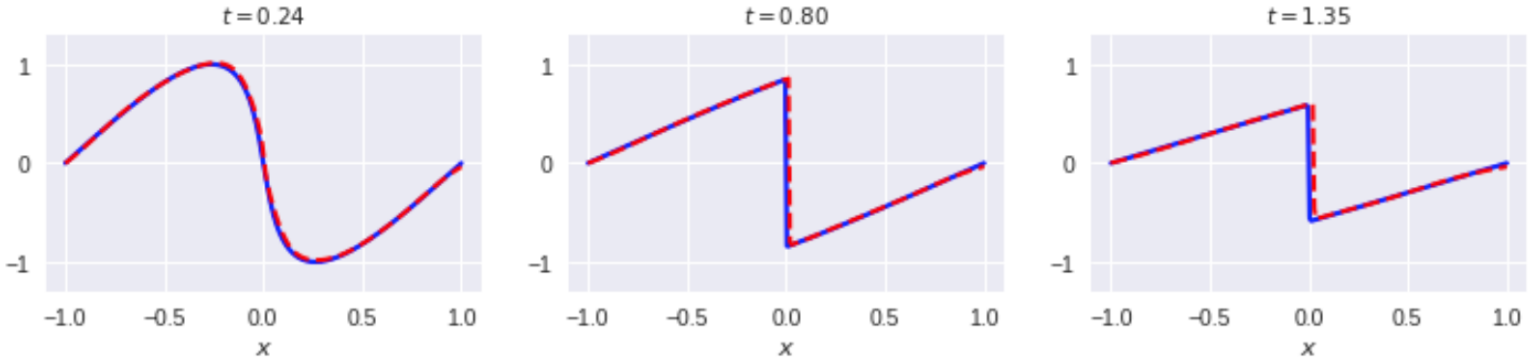

```
iter =  5000
loss = 0.0133
L2 error: 3.6761e-01
```



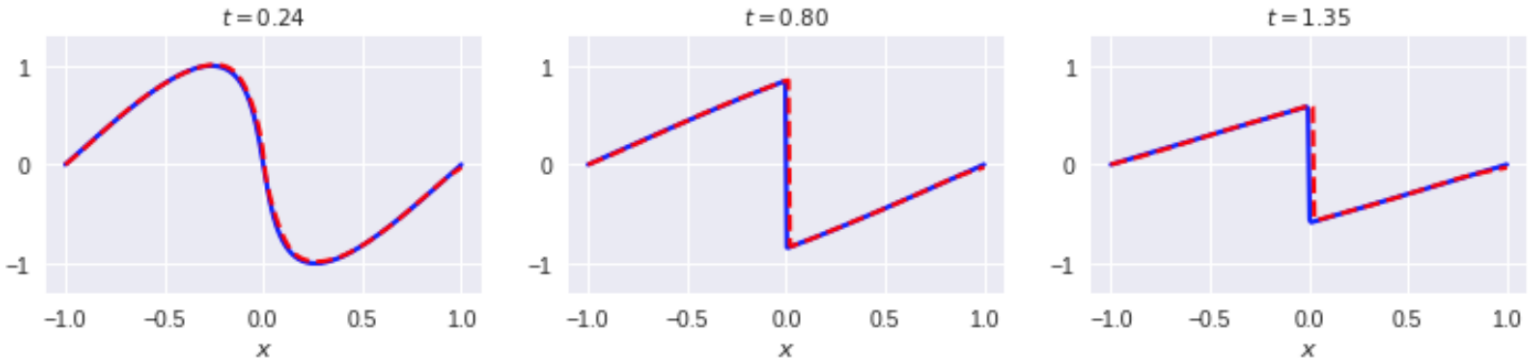
Example: Burgers equation

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

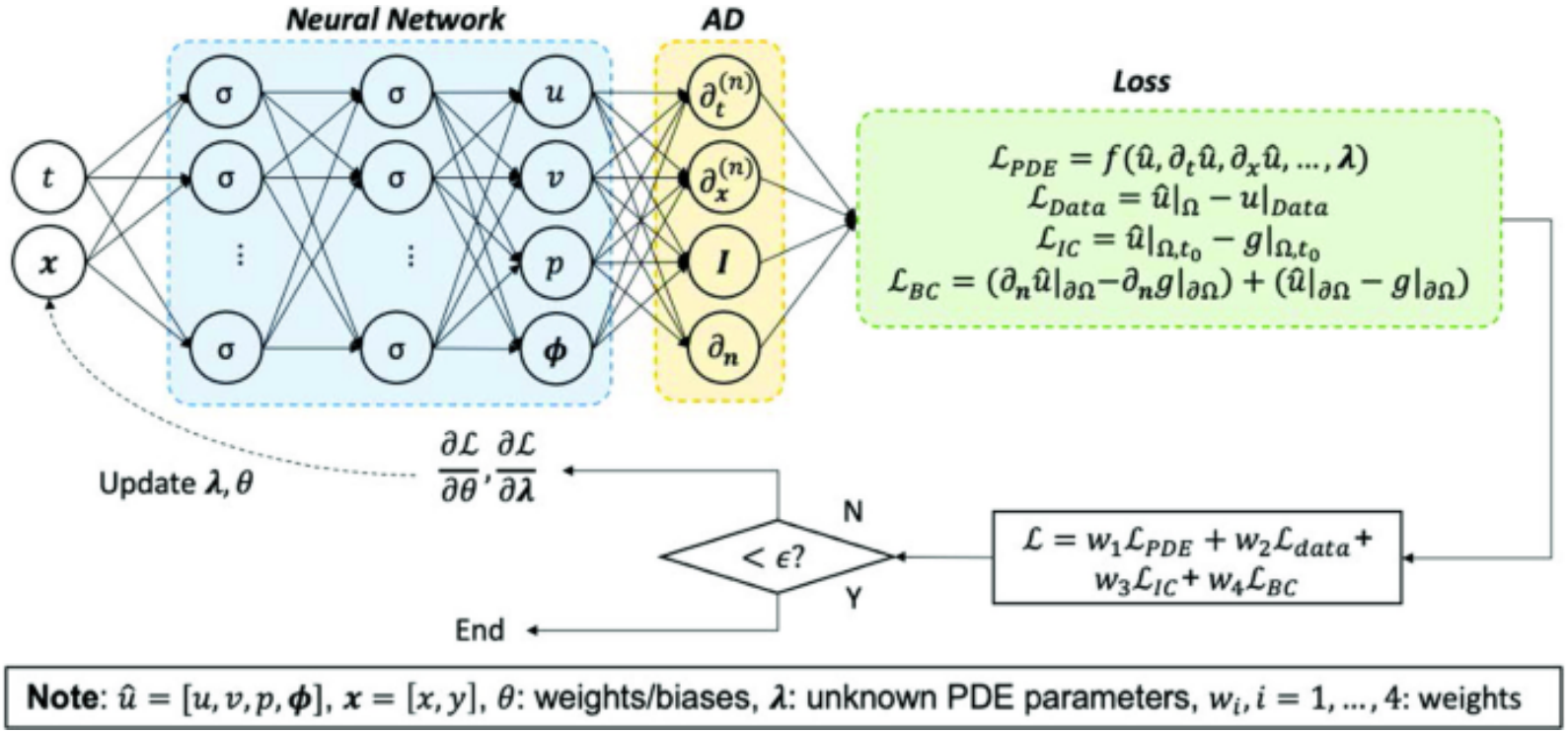
```
iter =  4800
loss = 0.0006
L2 error: 2.3011e-01
```




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



Full algorithm for PINNs



- There exists many variants of PINNs. Variational PINNS, hp-VPINNs which use weak form of the equation. C-PINNs, XPINNs acting like domain decomposition.
- Specific PINNs for hyperbolic problems, nonlocal PDE, stochastic equations, etc.

Defaults and Advantages

- Less accurate and fast than classical solvers. Difficult to choose hyper-parameters.
- mesh-less and mainly independent of the dimension.

Analysis of PINNs, Biais and NTK

- We consider a FCN f_θ with L layers. The gradient descent can be viewed as a discretization to:

$$\frac{d\theta(t)}{dt} = -\frac{1}{N}\sum_{i=1}^{N} \nabla_{\theta(t)} l(f_{\theta(t)}(x_i), y_i)$$

- which is equivalent to

$$\frac{d\theta(t)}{dt} = -\frac{1}{N}\sum_{i=1}^{N} \left(\nabla_{\theta(t)} f_{\theta(t)}(x_i)\right) \left(\nabla_f l(f, y_i)\right)$$

we obtain

$$\frac{df_{\theta(t)}(x)}{dt} = -\frac{1}{N}\sum_{i=1}^{N} \underbrace{\left(\nabla_{\theta(t)} f_{\theta(t)}(x)\right)^T \left(\nabla_{\theta(t)} f_{\theta(t)}(x_i)\right)}_{k_{\theta(t)}(x,y) \text{ called NTK kernel}} \left(\nabla_f l(f, y_i)\right)$$

NTK theorem

If we initialize $\theta(0) \sim \mathcal{N}(0, I)$ and the number of neurons $(n_1, ..., n_L) \to +\infty$ we have: $k_{\theta(0)}(x, y)$ is deterministic and $k_{\theta(t)}(x, y)$ is constant is time.

- So in this limit (called over-parametrized):

$$Y_{\theta(t)} = Y + (Y_{\theta(t)} - Y)e^{-tK_\infty(X,X)}$$

with $Y_{\theta(t)}$ the output of the network and $K_\infty(X, X)$ the NTK, on the data set.

- Study $K_\infty(X, X)$ allows the study the learning speed compared to some parameters.

PINN's and parametric PDEs

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

New Optimization problem of PINN's

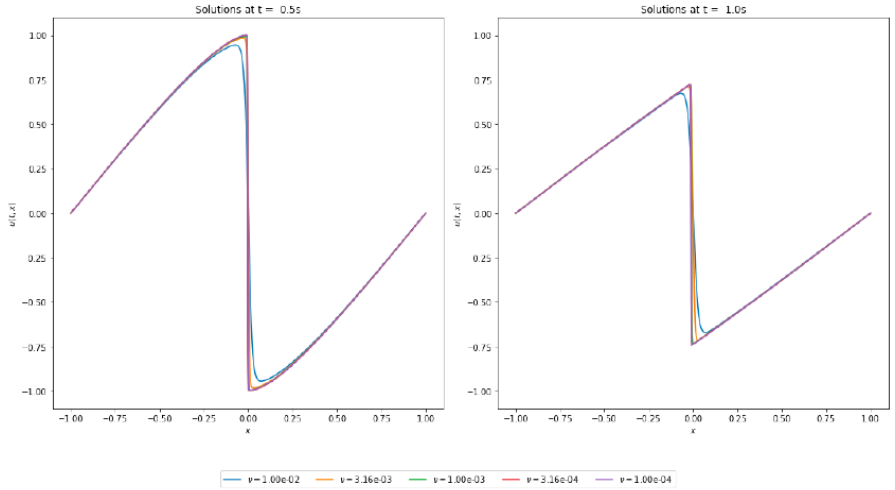
$$\min_\theta J_r(\theta) + \dots$$

with

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

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

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



Operator Learning

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 note $\boldsymbol{\mu}(t,x) = (\alpha(x,t), g(x), u_0(x))$ 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.

First approach: discrete approach

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

$$\mathcal{J}_1(\theta) = \int_{\boldsymbol{\mu}} \sum_{n=1}^{n_T} \| \, G_\theta^+(\boldsymbol{u}_h^n, \boldsymbol{\mu}_h) - \boldsymbol{u}_h^{n+1} \, \|_2^2 \, d\boldsymbol{\mu}$$

and

$$\mathcal{J}_2(\theta) = \int_{\boldsymbol{\mu}} \sum_{n=1}^{n_T} \| \, G_{\boldsymbol{\mu},\Delta t}(G_\theta^+(\boldsymbol{u}_h^n, \boldsymbol{\mu}_h^n), \boldsymbol{u}_h^n) \, \|_2^2 \, d\boldsymbol{\mu}$$

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

Operator learning II

Operator learning

We speak about operator learning if your neural network map data living in a Hilbert functional space.

Second approach: continuous approach

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

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

and

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

where the integrals are approximated by MC.

My understanding

When we speak about construct a map between functions, we want a neural network where the results do not depend of the discretization of the inputs/output functions.

Neural operator

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

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

- The solution is given by

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

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

Interesting framework

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

- **Methods using non-locality**: FNO (Z. Li and al 20), WNO (Tripura and al 22), Laplacian NO (Chen and al), Graph kernel Operator, Multipole GNO (N. Kovachki and al 20), DeepONet (Karniadakis and al 19)
- **Methods using dimension reduction**: PCANet (N. Kovachki and al 19), NOMAD (Perdirakis and al),Meta-AE (Ye and al).

Neural operator I

Integral kernel

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

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

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

Neural operator layer

We call 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 and the kernel k.
- How make that in practice ?

Neural operator II

Lifting and projection layers

- Extrapolation layer P : increase the size of feature space:

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

with P_θ a FNC.

- Projector layer Q : decrease the size of feature space:

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

with Q_θ a FNC.

Full neural operator

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

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

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

Neural operator: Fourier NO

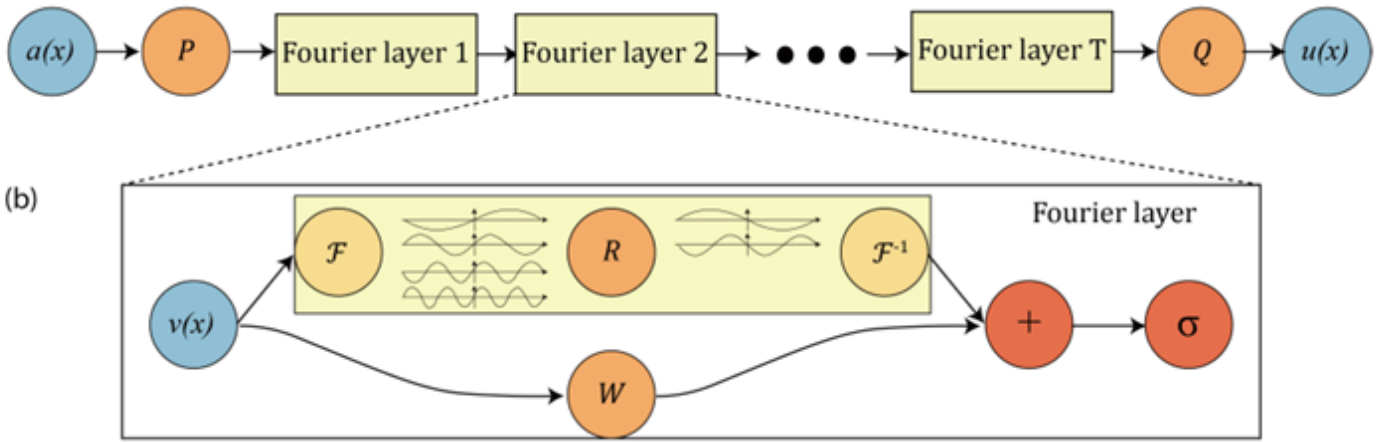
Fourier Neural Network

The FNOs use neural operator layers with an integral kernel:

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

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

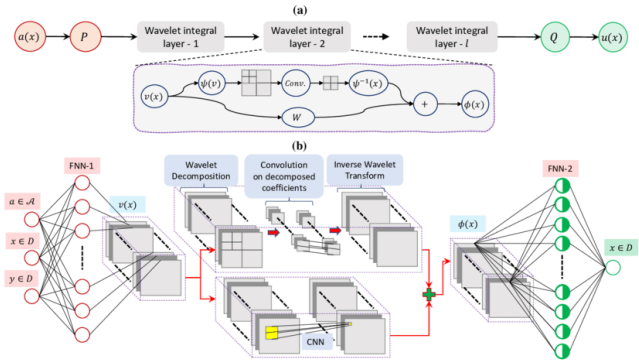
- Principle:



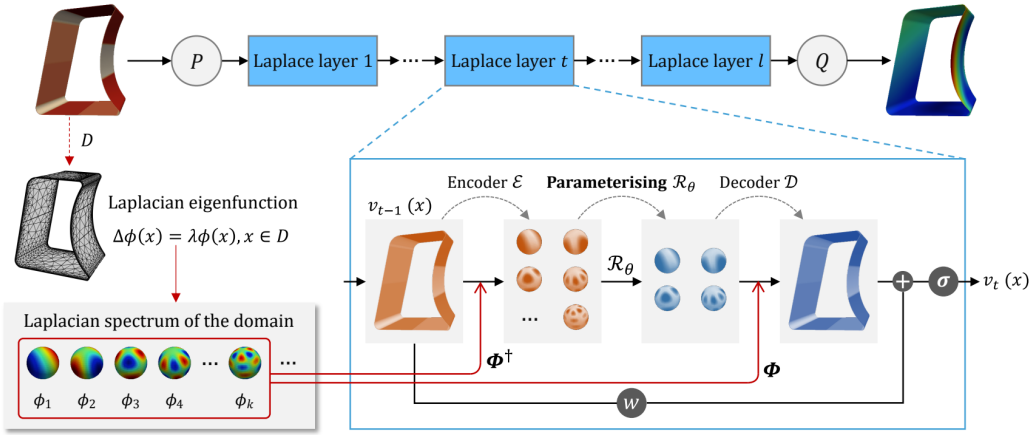
- Contrary to the discrete CNN case, we can change the mesh resolution (it is also possible with CNNs, provided interpolation is used), and we could adapt the approach to unstructured grids.

FNO like neural Operator

- **Wavelet Neural Operator**: we replace the Fourier transform by a wavelet transform (closer to CNN).



- **GFNO** (Li and al): learn mapping between your geometry and a square and use FNO
- **Laplace NO** (Cheng and al 23): extension to FNO on Manifold



- Replace the Fourier basis by the eigenvectors of the Laplace Beltrami.
- The Laplace Beltrami on a mesh/graph associated with the manifold is approximated by the Graph Laplacian (nice theory very used in ML).

Differentiable physics

Principle I

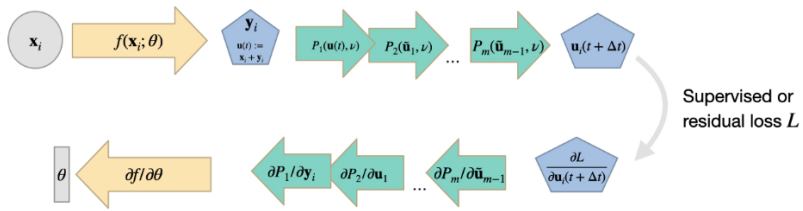
- At the beginning, PINNs/neural operators are used to solve PDEs.
- **Other approach**: supervised learning, Hesthaven et al 2017, 2018, 2020, 2022, B. Després and H. Joudren 2020, R. Loubère et al 2020, etc.
- Differentiable physics, Michael P. Brenner et al 2018, 2020, "Physics-based deep learning",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

Coming back to neural networks

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

Back-propagation and automatic differentiation

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



Principle II

Differentiable physics

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

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

Link with optimal control

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

Drawback

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

Conclusion

Conclusion

Neural networks

The deep neural networks are good candidates to gives smooth approximations using data in large dimensions.

PINNS

PINNS gives a new way to solve PDE. Less efficient and accurate but interesting in large dimensions.

Neural operator

NO can gives good approximation of PDe solutions on semi-large parameters datasets.

Differential physics

Use the huge progress of automatic differentiability to optimize a part of code/scheme.

Next Talk

How to couple these approaches with standard one to obtain guarantees and better efficiency.