IPP INSTITUTE OF PLASMA PHYSICS OF THE CZECH ACADEMY OF SCIENCES

Differentiable Numerical Simulations (Differentiable Physics)

J. Seidl

22.5.2023

MINISTRY OF EDUCATION, YOUTH AND SPORTS

Partial differential equations (PDEs)

1st era - analytical theory and analytical solutions (mid 18th century)

- partial differential equations (PDEs) become core of the physics
- focused on describing world by PDEs, on analytical theory and analytical solutions of PDEs

- 1746/7 - d'Alembert's solution to 1D wave equation followed by many others
- many solution methods: Fourier, Laplace transforms, Sturm-Liouville theory, Green's functions, …
- most focus on linear operators (tractable)

2nd era - computational approximations (mid 20th century)

- finite difference, finite element, spectral techniques, …
- rise of high-performance computing (HPC)
- but solutions to real-world strongly nonlinear, complex, high-dimensional PDE systems often still very demanding

⇩

3rd era - machine learning and optimization methods - starting now!

Examples of emerging techniques

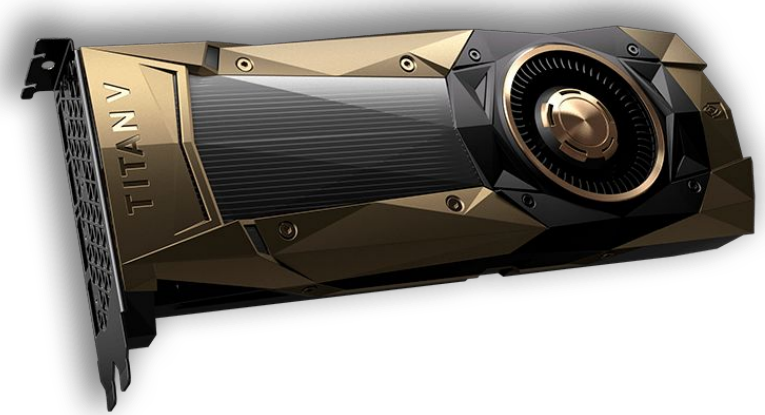
- **building surrogate models**
  - replace classical simulation or experiment that transforms f: $I_{nputs} \rightarrow O_{utput}$ with a fast surrogate model f': $I_{nputs} \rightarrow O_{utputs}$
- **solving inverse problems**
  - solve PDE with known data given elsewhere than at initial or boundary conditions, i.e. f^{-1}: $O_{utputs} \rightarrow I_{nputs}$
  - e.g. Physics Informed Neural Networks (PINNs) or differentiable physics
- **improving traditional numerical methods**
  - in terms of speed and/or accuracy
- **coarse-grained simulations**
  - fast/inaccurate coarse-grained simulation + correction by a model or real data
- **learning (simplified) governing equations from data or complex simulations**
  - e.g. SINDy, PDE-FIND
- **finding suitable coordinate systems** for non-linear PDEs and reduced representations
- **non-linear operator learning**
- …

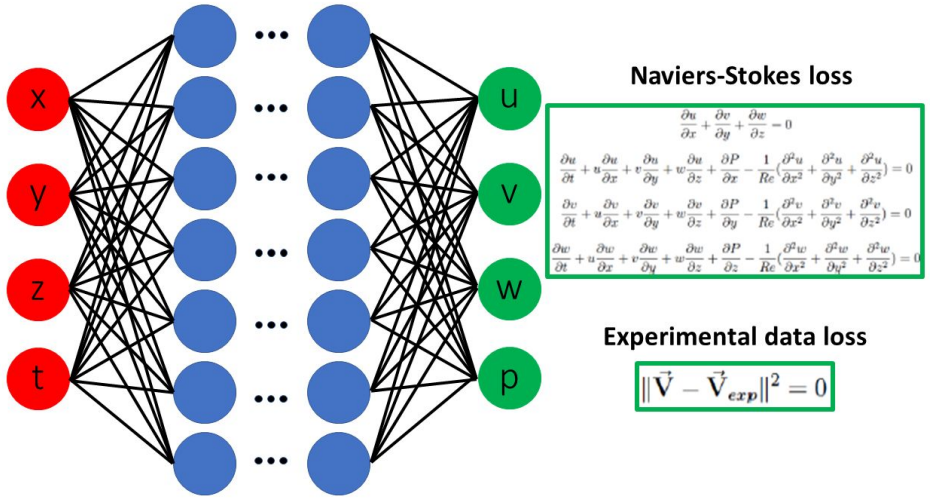
today we'll very briefly touch these parts

for references see e.g. reviews: [1] https://physicsbaseddeeplearning.org [2] Brunton, Kutz (2023), [3] Ramsundar (2021)

Today's topics

- Python for scientific computing and data science
- Speed up your simulations with GPU
- Automatic differentiation of computations
- What are Artificial Neural Networks (NN)
- Differentiable physics - solving inverse problems
- Implicit representation of functions with NN
- Physics Informed Neural Networks - solving PDEs using NN





Naviers-Stokes loss

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}-0$$

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}+w\frac{\partial u}{\partial z}+\frac{\partial P}{\partial x}-\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}\right)=0$$

$$\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}+w\frac{\partial v}{\partial z}+\frac{\partial P}{\partial y}-\frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}+\frac{\partial^2 v}{\partial z^2}\right)=0$$

$$\frac{\partial w}{\partial t}+u\frac{\partial w}{\partial x}+v\frac{\partial w}{\partial y}+w\frac{\partial w}{\partial z}+\frac{\partial P}{\partial z}-\frac{1}{Re}\left(\frac{\partial^2 w}{\partial x^2}+\frac{\partial^2 w}{\partial y^2}+\frac{\partial^2 w}{\partial z^2}\right)=0$$

Experimental data loss

$$\|\vec{V}-\vec{V}_{exp}\|^2=0$$

Choosing programming language

Typical problem of scientific computing: solving (a system of) partial differential equations such as

$$\nabla \cdot \overline{u} = 0$$

$$\rho \frac{D\overline{u}}{Dt} = -\nabla p + \mu \nabla^2 \overline{u} + \rho \overline{F}$$

\+ some data "fixing" the solution

Steps:

- decide on the **methods** that will be used to solve the problem
- decide which **programming language** to use

Most of the high-performance codes in the fusion community are written in C/C++/Fortran, but **is it still the best way?**

C/C++/Fortran/…

- **:(** complex, steep learning curve, slow to develop
- **:)** realtime
- **:)** fast (if you know what you are doing)

high-level languages (e.g. Python)

- **:)** simple and rapid development, powerful feature-rich libraries
- **:(** not suitable for hard real time
- **:|** typically (but not always) somewhat slower

Cost of a simulation

(1) (2) (3)

$$\text{simulation cost} = \text{cost}_{\text{development}} + \text{cost}_{\text{run performance}} + \text{cost}_{\text{usage}}$$

complexity

often hidden / neglected
⇒ limits code usability, manpower costs, introduces mistakes & wrong results

often: performance cost < development & usability cost

C/C++/Fortran/…

- **:(** complex, steep learning curve, slow to develop
- **:)** realtime
- **:) faster** (if you know what you are doing)

we need this (2) …

high-level languages (Python)

- **:) simple and rapid development, powerful feature-rich libraries**
- **:(** not suitable for real time
- **:|** typically (but not always) somewhat slower

… and this (1), (3) at the same time

while it's not easy to simplify the low-level languages, the performance gap is gradually being closed by dedicated python libraries and new high-level languages like Julia or Mojo that are specifically designed for high-performance computations

Python techniques to speed-up the code

- vectorization (numpy, scipy, …)
- optimized algorithms and data structures (scipy, pandas, xarray, NetworkX, JAX, …)
- just-in-time (JIT) compilation (numba, JAX, …)
- parallelization (Dask, JAX, …)
- hardware acceleration with GPU/TPU (JAX, PyTorch, CuPy, …)
- distributed computations (Dask, JAX, Apache Spark, …)
- use Cython, Julia, Mojo, …
- write C/Fortran extension or use existing C/Fortran code under the hood

still high-level programming
+
large ecosystem - no need to DIY
(faster, safer)

different high-level language

use low-level language when you need it,
do the rest at high-level

Thanks to all this, Python is a language #1 for data science (incl. machine learning and AI) and its significance in technical computations steadily grows

Python techniques to speed-up the code

- vectorization (numpy, scipy, …)
- optimized algorithms and data structures (scipy, pandas, xarray, NetworkX, JAX, …)
- just-in-time (JIT) compilation (numba, JAX, …)
- parallelization (Dask, JAX, …)
- hardware acceleration with GPU/TPU (JAX, PyTorch, CuPy, …)
- distributed computations (Dask, JAX, Apache Spark, …)
- use Cython, Julia, Mojo, …
- write C/Fortran extension or use existing C/Fortran code under the hood

see next slides

still high-level programming
+
large ecosystem - no need to DIY
(faster, safer)

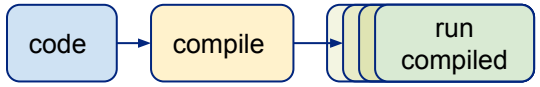
different high-level language

use low-level language when you need it,
do the rest at high-level

Thanks to all this, Python is a language #1 for data science (incl. machine learning and AI) and its significance in technical computations steadily grows

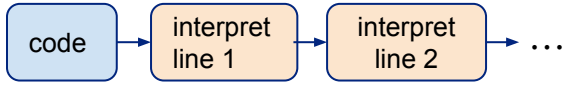
Just-in-time compilation

- C/C++/Fortran are **compiled languages**



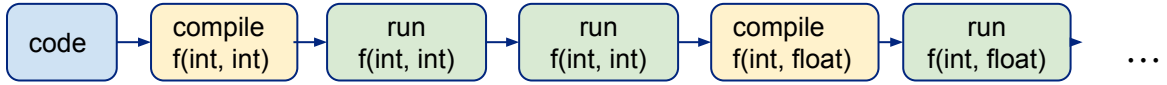
  - explicit compilation step to machine code run ahead of execution
  - compilation to machine code includes optimizations that boost performance in general + for a specific hardware target
  - statically typed → optimization ahead of time often possible

- Python is an **interpreted language**



  - code (converted to bytecode) executed by python interpreter at runtime using Python Virtual Machine; line by line - slow
  - dynamically typed → optimizations often not possible

- **Just-in-time (JIT) compilation** - a golden middle way



  - used by interpreted languages
  - block of code (e.g. function) is compiled to machine code at runtime at the time of first use
    - run-time overhead during the first execution of the code (x caching), but then performance on par with compiled languages
    - the input types and data shapes are known at runtime → performance optimization possible

JIT in Python (an example)

JAX

- developed by Google
- numpy-like interface + re-implementation of many numerical algorithms from scipy
- growing ecosystem of numerical and ML libraries
- based on TensorFlow's XLA (Accelerated Linear Algebra) compiler
- hardware acceleration (GPU/TPU) out of the box
- automatic differentiation capabilities

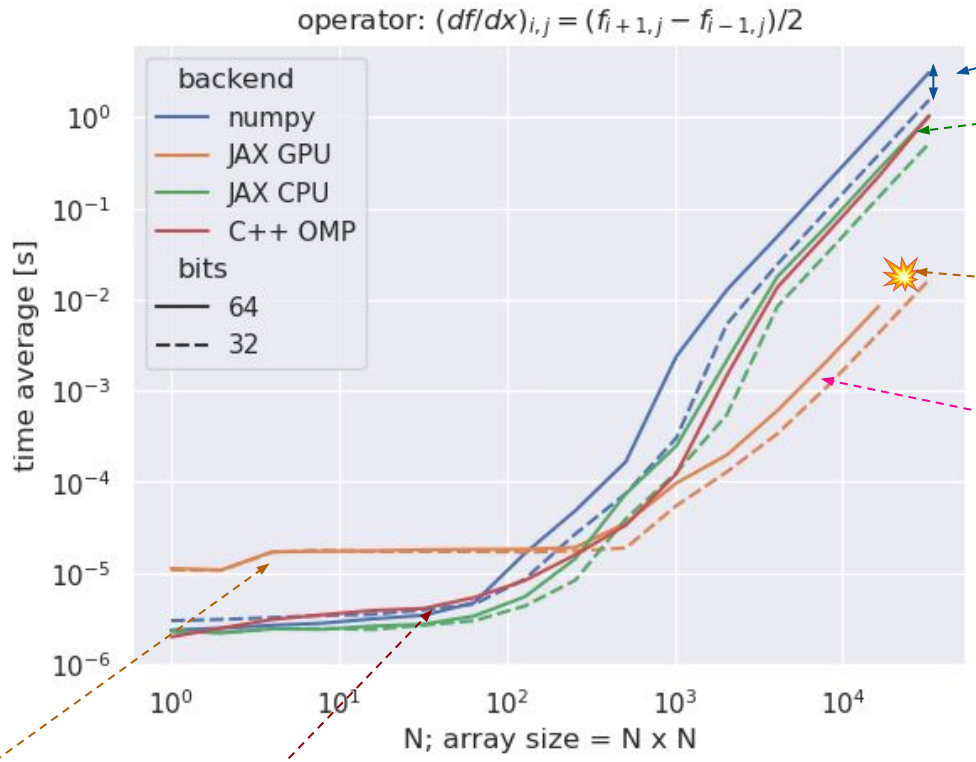
- JAX has some specifics (e.g. specific treatment of conditions and loops), but in general as simple as:

```
# numpy


def central_difference(f):
    return  (f[2:, :] - f[:-2, :]) / 2
```

optimize and run at GPU/TPU (if available)

```
# JAX
import jax

@jax.jit
def central_difference(f):
    return (f[2:, :] - f[:-2, :]) / 2
```

JIT speed-up



would 32-bit precision be sufficient for your application?

JIT can be comparable to C++ code

GPUs can easily run out of memory for large arrays 😦

100x speed-up w/ GPU at the cost of a single line of code! 😍

current WIP project at IPP (J. Seidl, P. Macha):
porting fluid plasma turbulence models to GPU w/ JAX

inefficient number of OMP threads in C++

GPU may have overhead ⇒ pays-off only for larger arrays

Note: this is a simple example, different problems may scale somewhat differently

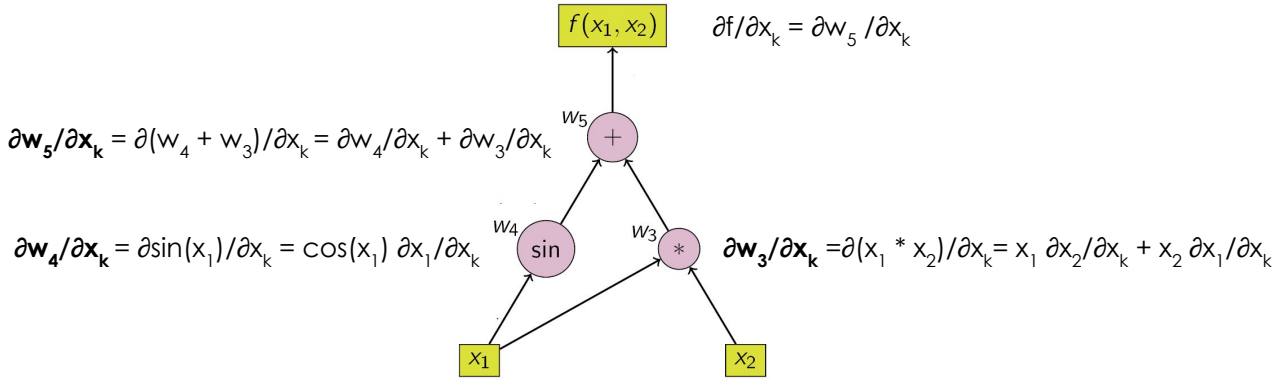
Automatic differentiation (AD)

AD is a technique for computing function derivatives **efficiently** and **accurately** (no discretization) by applying

the chain rule: $\partial(g \circ f)/\partial x|_x = \partial g / \partial f|_{f(x)} * \partial f / \partial x|_x$ (or its generalisation for multivariate functions)

AD works on functions that are (arbitrarily deep) composition of functions with known derivatives, e.g.:

$\mathbf{sin(x_1) + x_1 * x_2}$



AD represents function as a call-graph of elementary functions whose derivatives (Jacobians) are analytically known

equation images taken from stackexchange

AD modes

$$\mathbf{y} = h(g(f(\mathbf{x})))$$

$$\mathbf{a} = f(\mathbf{x}), \quad \mathbf{b} = g(\mathbf{a}), \quad \mathbf{y} = h(\mathbf{b})$$

multiplication of Jacobians

$$\underbrace{\frac{\partial \mathbf{y}}{\partial \mathbf{x}}}_{|\mathbf{y}|\times|\mathbf{x}|} = \underbrace{\frac{\partial h(\mathbf{b})}{\partial \mathbf{b}}}_{|\mathbf{y}|\times|\mathbf{b}|} \underbrace{\frac{\partial g(\mathbf{a})}{\partial \mathbf{a}}}_{|\mathbf{b}|\times|\mathbf{a}|} \underbrace{\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}}_{|\mathbf{a}|\times|\mathbf{x}|}$$

Jacobian matrix sizes:

forward mode

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial h(\mathbf{b})}{\partial \mathbf{b}} \left(\frac{\partial g(\mathbf{a})}{\partial \mathbf{a}} \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right)$$

- # multiplications: |x|·|a|· |b| + |x|·|b|·|y|
- **forward mode better when |y| > |x|** **(# outputs > # inputs)**
- single forward pass evaluating function values & gradients
- Ex: sensitivity analysis of a simulation
  - few simulation input parameters, many outputs on large grid

reverse mode

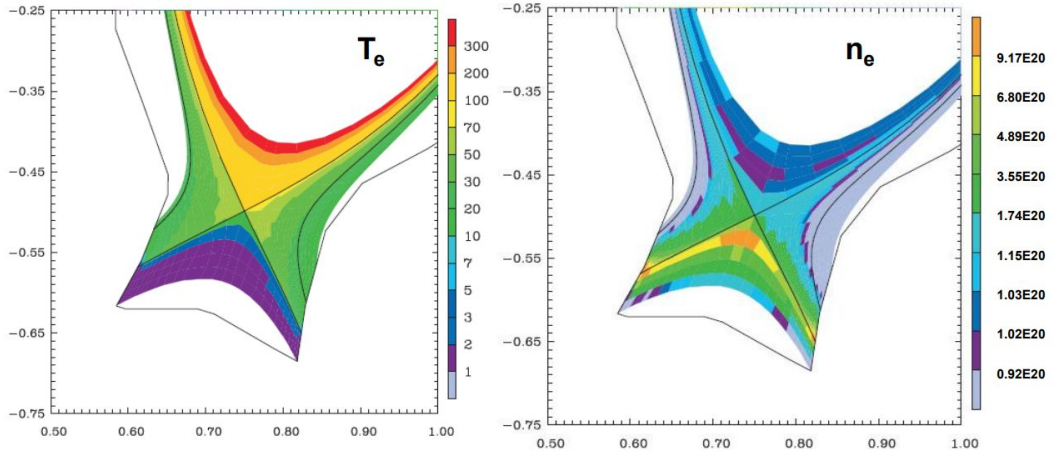
$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \left(\frac{\partial h(\mathbf{b})}{\partial \mathbf{b}} \frac{\partial g(\mathbf{a})}{\partial \mathbf{a}} \right) \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$$

- # multiplications: |y|·|a|·|b| + |y|·|b|·|x|
- **backward mode better when |x| > |y|** **(# inputs > # outputs)**
- 2 passes: forward (function values) & backward (gradients)
  ⇒ memory intensive

Ex: optimization; neural networks
- many features on input (e.g. image pixels; x > 1) and only scalar loss (y=1) at output

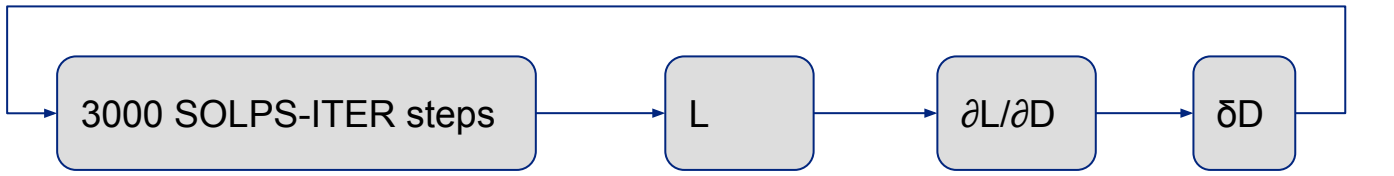
Automatic differentiation in SOLPS-ITER

- SOLPS = workhorse of tokamak edge transport modelling
  - complicated long-running code (days-weeks)
  - lots of free input parameters
  - radial transport described by diffusion coefficients (free par.)
- AD: optimization of diffusion coefficients
  to match exp. profiles or simplified turbulent model



[Carli (2022)]

loss L: weighted MSE of discrepancy from exp data



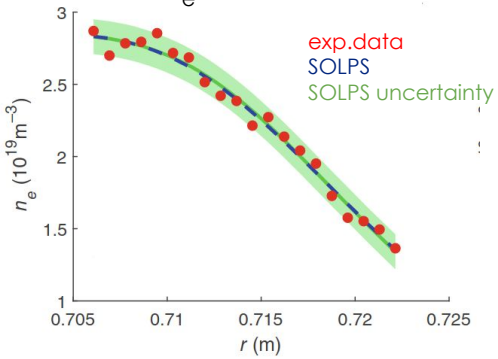
let the simulation converge

model-exp discrepancy

AD through SOLPS-ITER

update value of D

fit to exp n_e profile at midplane



Differentiable Numerical Simulations (Differentiable Physics)

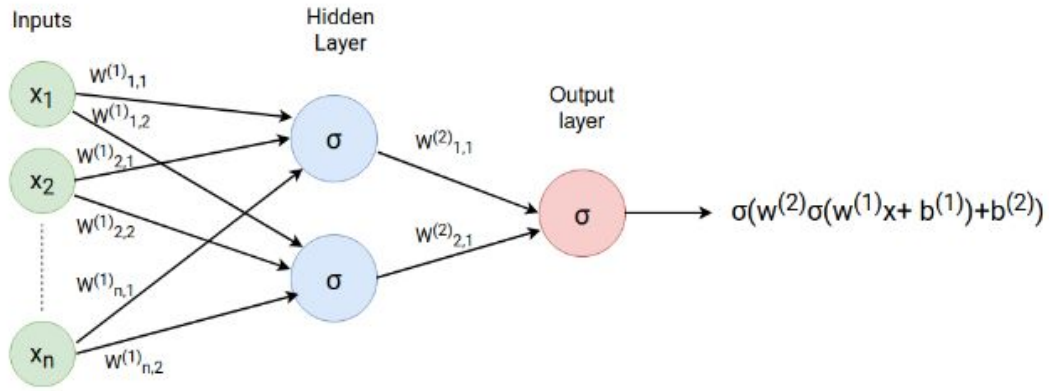
differentiable physics: differentiable phys. models and methods that can compute gradients of outputs wrt inputs and parameters

Neural networks (NN)

Feedforward (FF) Neural Network: $\mathbf{y} = \sigma_i(\mathbf{W_i} \cdot \sigma_{i-1} (\mathbf{W_{i-1}} \cdot \sigma_{i-2} (\dots \cdot \sigma_1(\mathbf{W_1} \cdot \mathbf{x} + \mathbf{b_1})\dots)))$

x … input feature vector
y … network output
$\mathbf{W_i}$ … weight matrix (learnable parameters!)
σ_i … nonlinear activation function

- e.g. ReLU(x) = (x if x > 0 else 0)
  tanh(x)
  …



Universal approximation theorem:
any continuous function can be approximated arbitrarily well by a neural network with at least 1 hidden layer and finite number of weights

Automatic differentiation can be used to compute $\partial y/\partial W_i$ and $\partial y/\partial x$

Gradient Descent

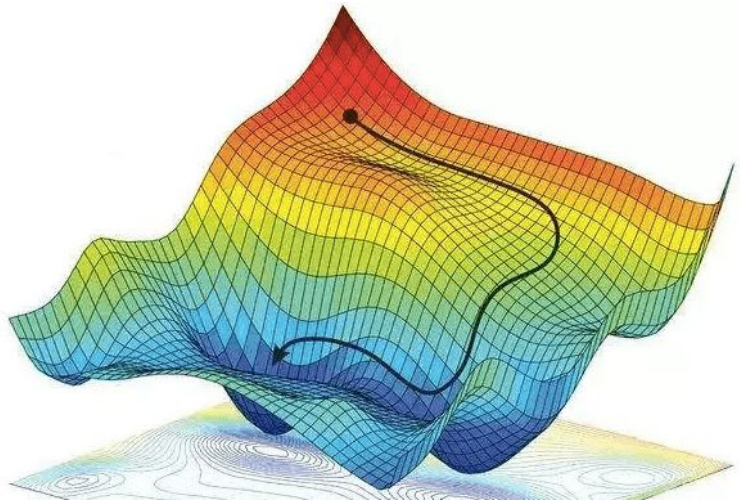
Feedforward (FF) Neural Network $f_{NN}(\mathbf{x}, \mathbf{W})$: $\mathbf{y} = \sigma_i(\mathbf{W_i} \cdot \sigma_{i-1} (\mathbf{W_{i-1}} \cdot \sigma_{i-1} (\dots \cdot \sigma_1(\mathbf{W_1} \cdot \mathbf{x})\dots)))$

Data $\{(\mathbf{x_k}, \mathbf{y_k})\}$: values of an unknown function **f(x)** sampled at points $\mathbf{x_k}$

Task: Find values of the weights $\mathbf{W_i}$ such that $f_{NN}(\mathbf{x}; \mathbf{W})$ will represent the unknown function f(x) as good as possible
as good as possible ⟷ value of a scalar loss function $\mathbf{L(f_{NN}(x_k; W), y_k)}$ is minimal

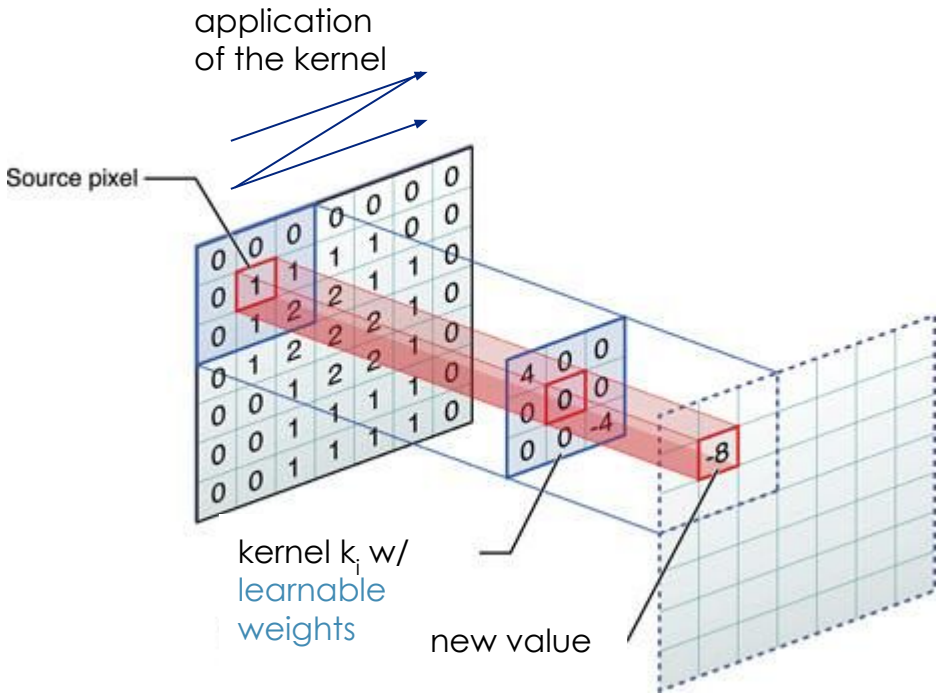
Method: (Stochastic) Gradient Descent

1. Evaluate $y_k = f_{NN}(x_k)$ at some or all $\mathbf{x_k}$
2. Compute $\partial L / \partial w_{ij}$ using reverse mode of AD (backpropagation)
3. Update values of w_{ij} in the direction of best improvement
   $w_{ij} \rightarrow w_{ij}$ - learning_rate $\cdot \partial L / \partial w_{ij}$
4. Repeat until L is sufficiently small or stops decreasing



Convolutional NN (CNN)

Replace matrix multiplication in FF NN by discrete convolution (applied on n-dimensional data)



$$\mathbf{y} = \sigma_i(\, \mathbf{k_i} * \sigma_{i-1}\, (\mathbf{k_{i-1}} * \sigma_{i-1}\, (\, \ldots * \sigma_1(\mathbf{k_1} * \mathbf{x})\ldots)))$$

variant of NN suitable for gridded data with correlated neighbors
- images, simulation grids, …

convolution easily parallelizable ⇒ fast at GPU

Example 1 - hybrid method for solving Poisson equation

$$\Delta\varphi = \omega$$

Poisson equation is a key equation in most high-temperature plasma turbulence simulations.

Poisson equation - classical approach

Poisson equation: $\Delta\varphi = \omega$ (to be computed; ω known)

Laplace stencil:

$\partial^2/\partial y^2$

	1	
1	-4	1
	1	

$\partial^2/\partial x^2$

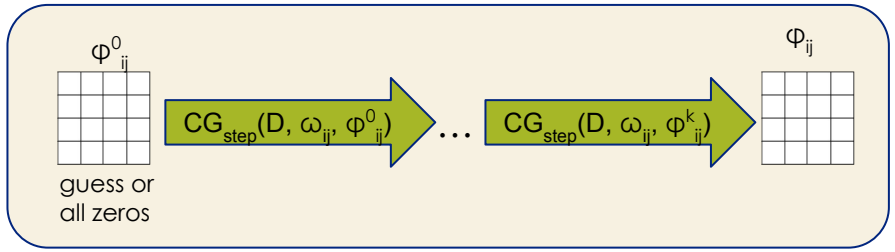
classical approach:

invert $-D =$

$$\begin{bmatrix}
4 & -1 & 0 & 0 & \dots & -1 & 0 & 0 & \dots & & & 0 \\
-1 & 4 & -1 & 0 & 0 & \dots & -1 & 0 & 0 & \dots & & 0 \\
0 & -1 & 4 & -1 & 0 & 0 & \dots & -1 & 0 & 0 & \dots & 0 \\
 & & & \ddots & & & & & \ddots & & & \\
-1 & & & -1 & 4 & -1 & & & & -1 & & \\
0 & -1 & & & -1 & 4 & -1 & & & & -1 & \\
0 & 0 & -1 & & & -1 & 4 & -1 & & & & -1 \\
0 & 0 & 0 & \ddots & & & \ddots & \ddots & \ddots & & & \ddots
\end{bmatrix}
\begin{matrix}
\dots \varphi_{11} \\
\dots \varphi_{12} \\
\dots \varphi_{13} \\
\dots \\
\dots \\
\dots \varphi_{nn-2} \\
\dots \varphi_{nn-1} \\
\dots \varphi_{nn}
\end{matrix}$$

and $\varphi = D^{-1}\omega$

or use iterative solver:



direct inversion typically done for small D, iterative methods for large D

- LU decomposition, Cholesky decomposition, conjugate gradient (CG), ...

Poisson equation as an optimization problem

Poisson equation: $\Delta\varphi = \omega$

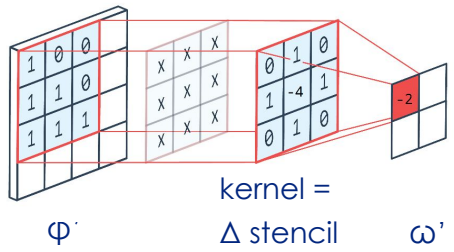
Laplace stencil:

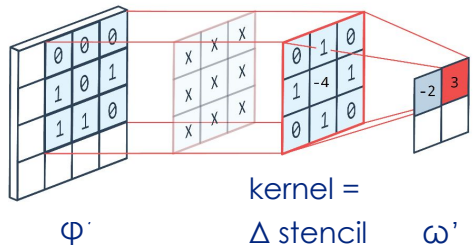
	1	
1	-4	1
	1	

$\partial^2/\partial y^2$ (vertical), $\partial^2/\partial x^2$ (horizontal)

Reformulate as an optimization problem (in this form inefficient!)

1. randomly initialize solution φ
2. apply convolution with Laplace stencil on φ
   → find ω to which the current φ corresponds





3. compute scalar loss: $L = \sum_{ij}(\omega'_{ij} - \omega_{ij})^2 + \alpha \sum_{BC}(\varphi'_{BC} - \varphi_{BC})^2$
4. compute gradients $\partial L/\partial \varphi'_{ij}$
5. optimize the values of φ'_{ij}, e.g by gradient descent, to find φ'_{ij} that will minimize L: $\varphi'_{ij} \rightarrow \varphi'_{ij} - \text{learning\_rate} \cdot \partial L/\partial \varphi'_{ij}$
6. go to (2) and repeat

Poisson equation as an optimization problem

Poisson equation: $\Delta\varphi = \omega$

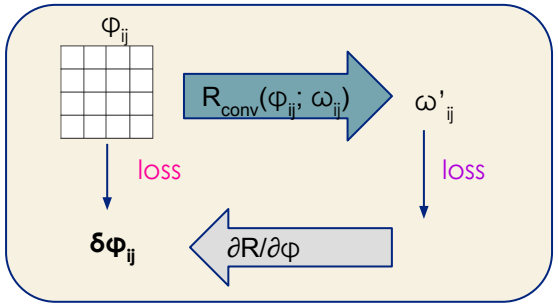
Laplace stencil:

$\partial^2/\partial y^2$

	1	
1	-4	1
	1	

$\partial^2/\partial x^2$

Reformulate as an optimization problem (in this form inefficient!)



$$L = \sum_{ij}(\omega'_{ij} - \omega_{ij})^2 + a \sum_{BC}(\varphi'_{BC} - \varphi_{BC})$$

reconstruction **boundary/initial conditions on φ**

finds only single solution to single ω at a time; GD optimization inefficient

Poisson equation - CNN model of D^{-1}

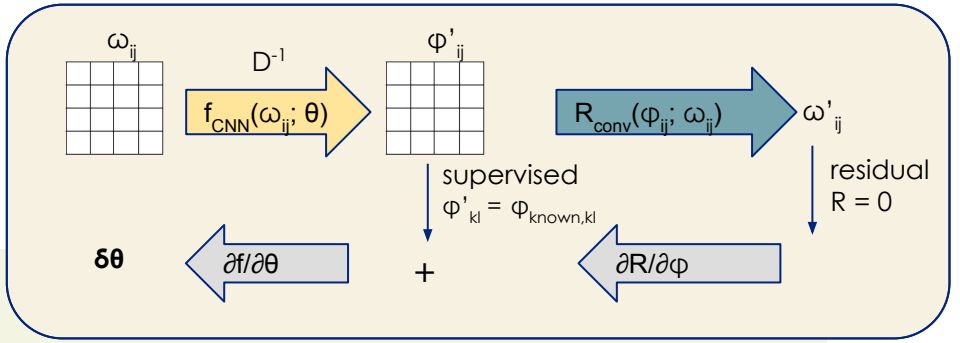
Poisson equation: $\Delta\varphi = \omega$

Laplace stencil:

$\partial^2/\partial y^2$

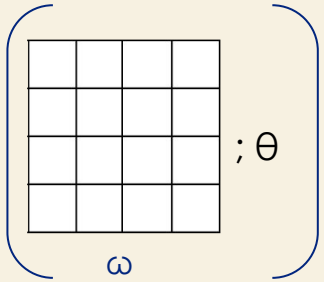
	1	
1	-4	1
	1	

$\partial^2/\partial x^2$



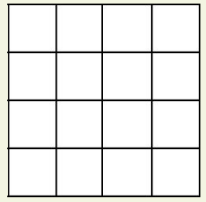
Find a CNN model of D^{-1}:

new step (D^{-1})

f_{CNN}  ; θ

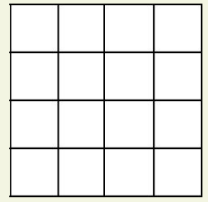
ω

compute gradients as before…



φ'

convolution w/ Laplace stencil



ω'

; $L = \sum_{ij}(\omega'_{ij} - \omega_{ij})^2 + \alpha \sum_{BC}(\varphi'_{BC} - \varphi_{BC})$

⟹ now optimization of parameters θ of the model f_{CNN} using backpropagation (AD)

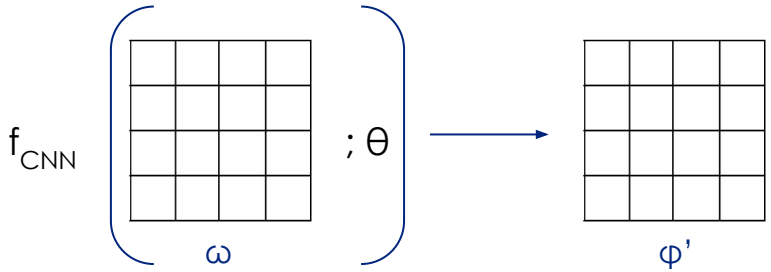
finds a transformation $f_{CNN}(\omega, \theta)$, representing Δ^{-1}, i.e. the whole **class of solutions**

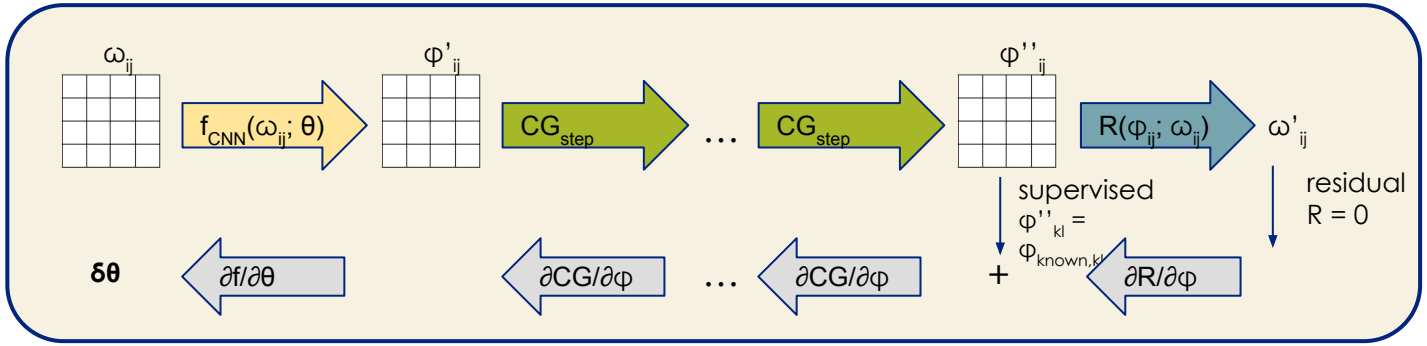
Poisson equation - CNN model of D^{-1} + iterative solver

Poisson equation: $\Delta\varphi = \omega$

Initialize classical solver with CNN model of D^{-1}

1. train f_{CNN} as a proxy of D^{-1} (see prev. slide)





hybrid method combining NN and classical solver

2. use φ' as an initial state for a CG iterative solver computing $\omega = D^{-1}\varphi$
   - good initial guess speeds up convergence of the solver

why to use the method: performance increase

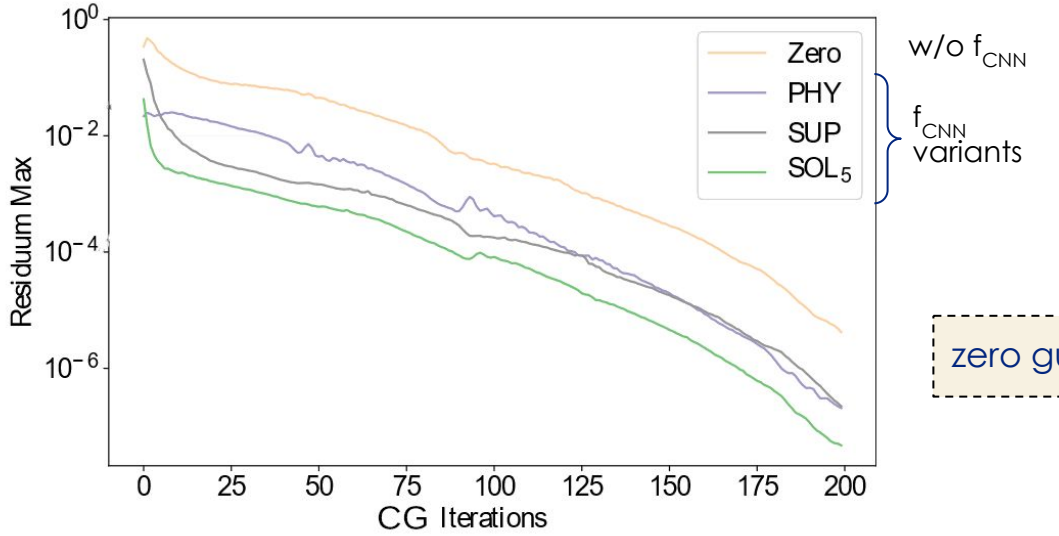
Note: f_{CNN} can be trained separately or together with the attached solver

Poisson equation - model of D^{-1} + iterative solver

Poisson equation: $\Delta\varphi = \omega$

Initialize classical solver with CNN model of D^{-1}

$\Delta\varphi = \nabla p$; $f_{CNN} \rightarrow$ CG solver



zero guess → f_{CNN}: **~60 iterations (30%) less** to reach given precision

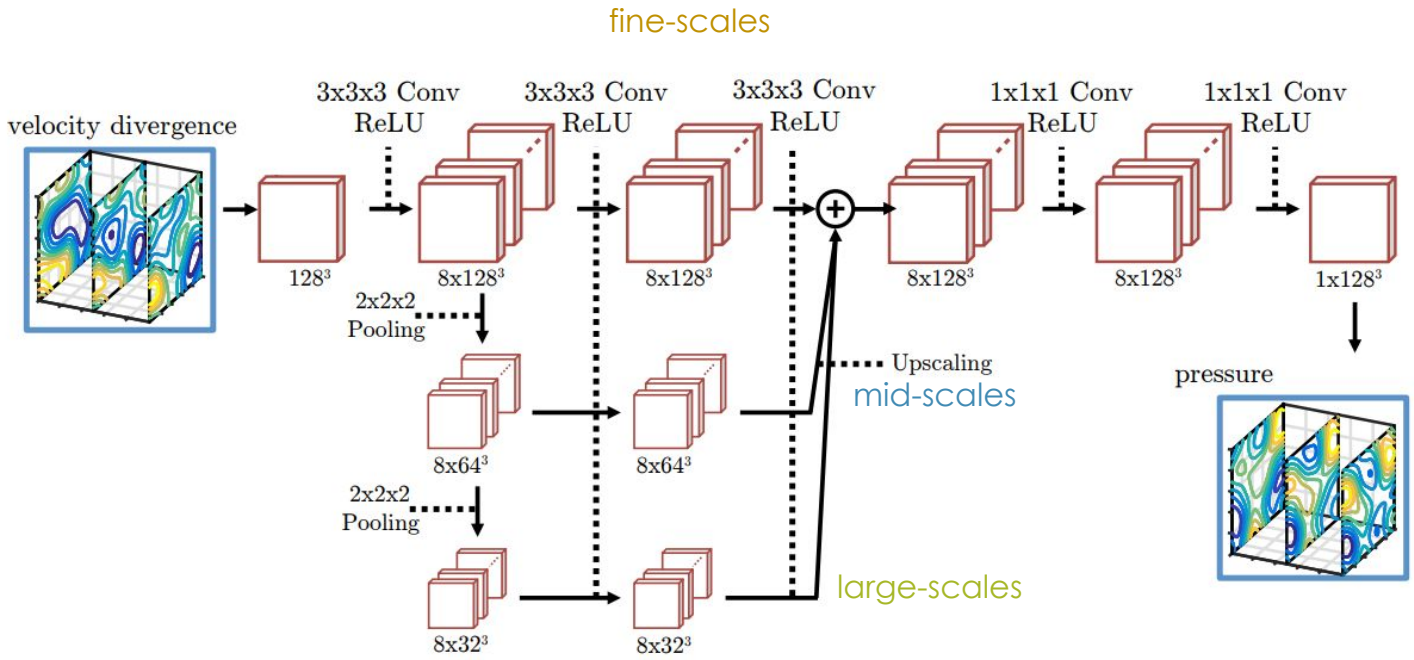
why to use the method: performance increase

[https://github.com/tum-pbs/CG-Solver-in-the-Loop]

Poisson equation - CNN model of D^{-1}

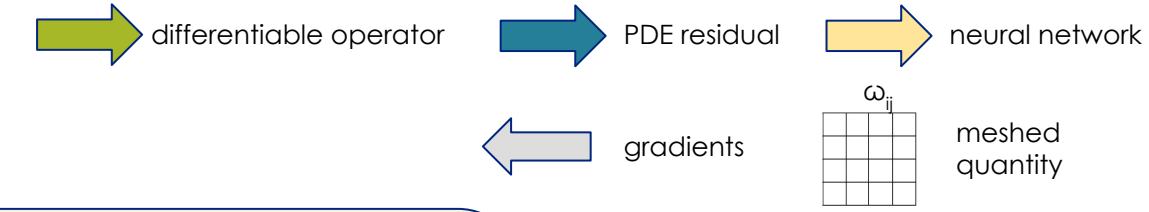
Example f_{CNN} architecture for $\Delta p = \nabla u$ in 3D:

[Tompson 2017]

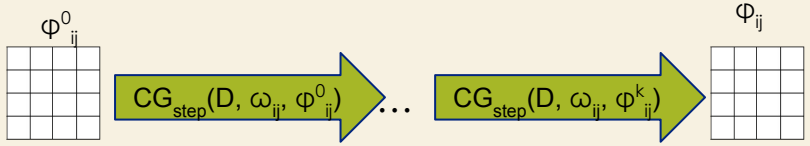


U-net architecture allows long-distance interactions

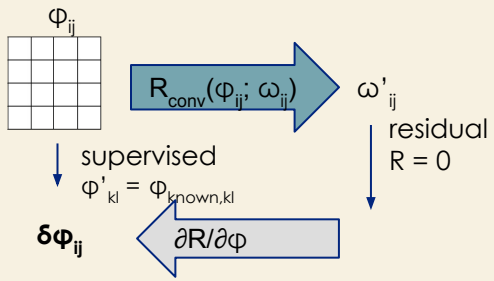
Graphical representation



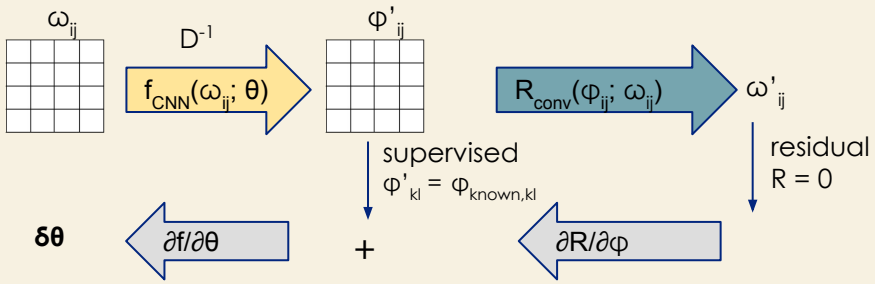
1. classical iterative solver:



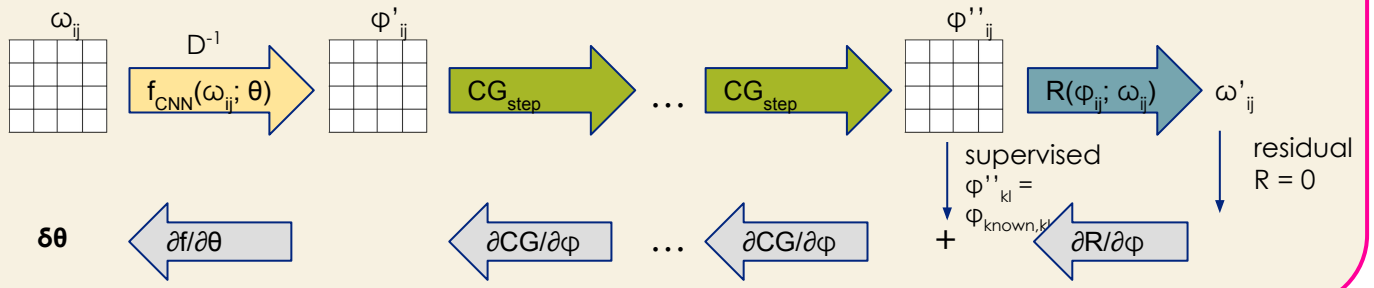
2. direct optimization:



3. NN model of D^{-1}



4. hybrid method - NN model of D^{-1} + iterative optimizer



Example 2 - flexible method for solving Poisson equation

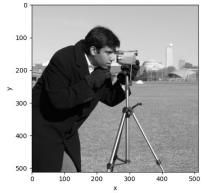
$$\Delta\varphi = \omega$$

Poisson equation is a key equation in most high-temperature plasma turbulence simulations.

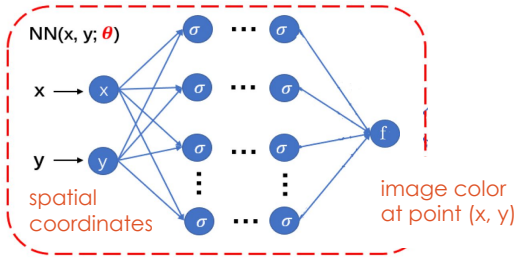
Implicit representation of data by NN

- NNs are typically used as models of data transforming functions
- but NN can be used also as a model of the data itselves / data generating func (implicit representation of the data)
  - implicit = representation of data structure and relations is hidden in the parameters of the NN
  - simplest case: no PDE, just known data

known data



(check e.g. this notebook or [Sitzman 2020])



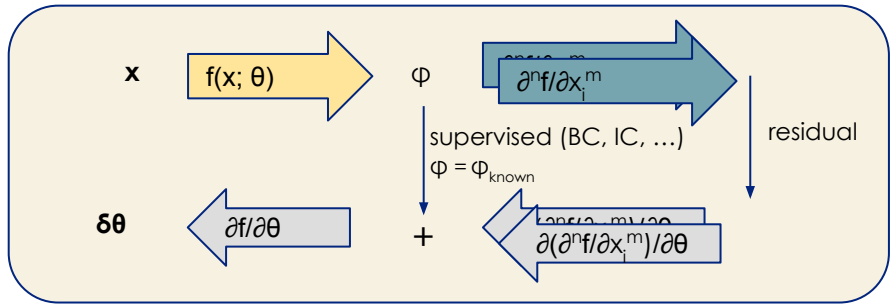
  - PINN: similar principle, but the NN output is also constrained by the PDE

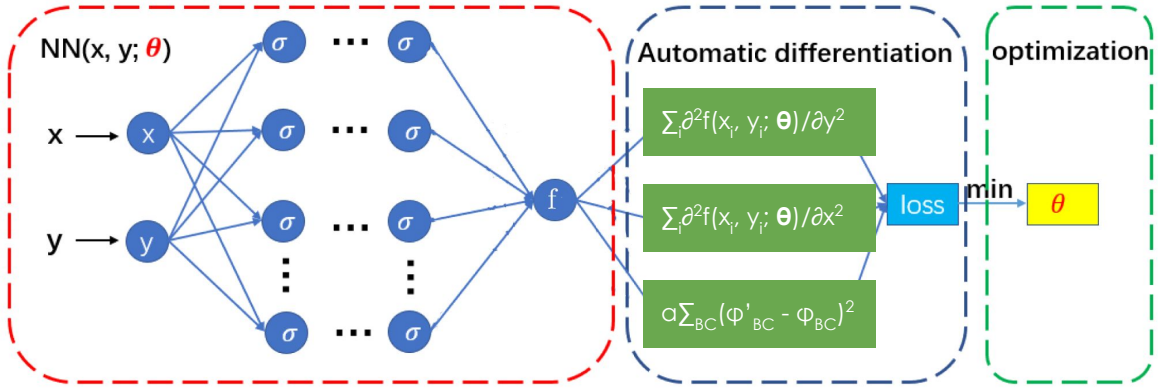
Poisson equation - Physics Informed Neural Network (PINN)

Poisson equation: $\Delta\varphi = \omega$

Find a NN for φ that solves the Poisson equation:

similar to fitting the image (known values of φ at BC), but now adding constraints on the output, given by PDE





now exact derivatives — mesh-free ⇒ anywhere

Loss = residual of PDE + α · distance from data

Q: think what happens when the data are not consistent with the PDE and what is the role of factor α

mesh-free C^{inf} solution with exact derivatives to single ω at a time; simple use, but often **slow/poor convergence**

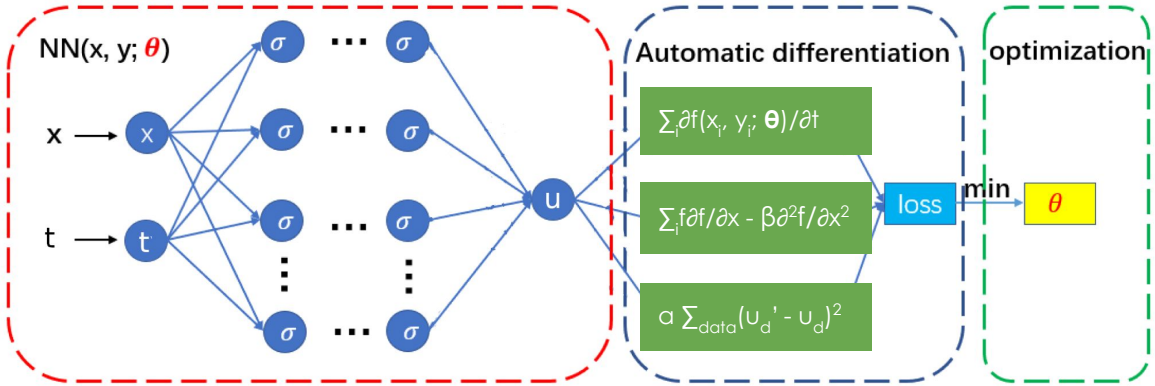
[Raissi 2019]

Example 3 - flexible method for solving time-dependent problem

Time-dependent PDE - PINN

Burgers equation: $u_t + uu_x - \beta u_{xx} = 0$

Full PDE solution implicitly represented by a differentiable NN

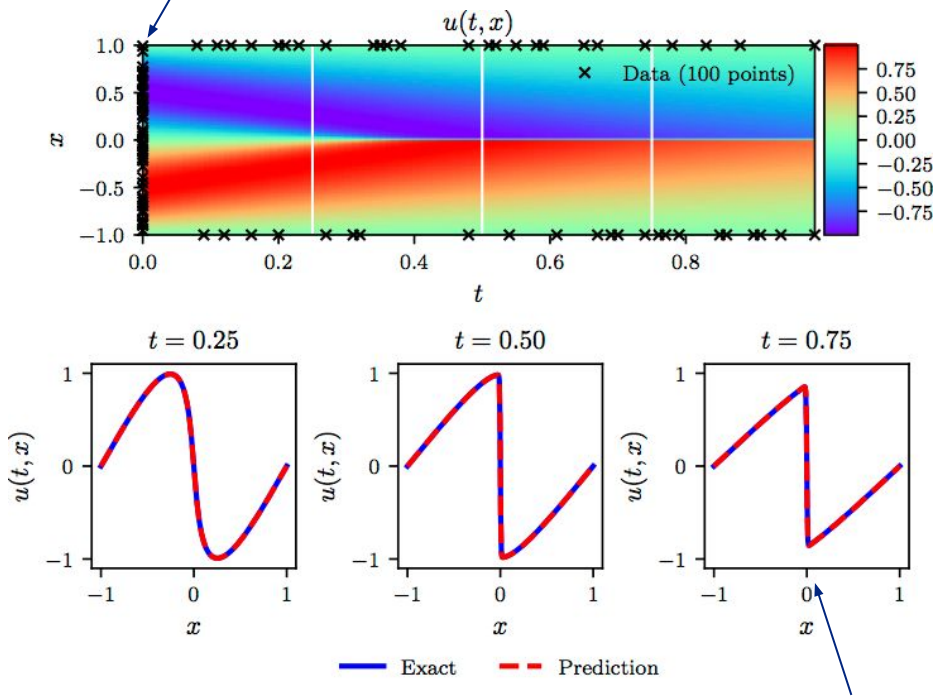


with a good library, basically all you need to do is just to define the PDE to solve:

```
def loss(x, u, params):
    pde = diff(u, 't') + u * diff(u, 'x') - params['beta'] * diff(u, ('x', 'x'))
    return pde(x) ** 2
```

[Raissi (2019)]

this is a forward problem, but the known points can be anywhere (inverse problem)



handles shocks out of the box

why to use the method: easy to use, flexible

combining or extending PDEs is simple

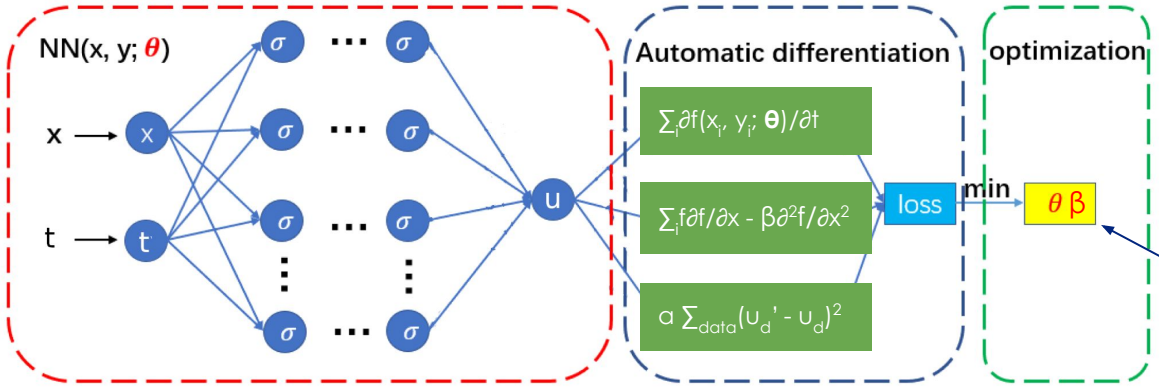
[Raissi (2019)]

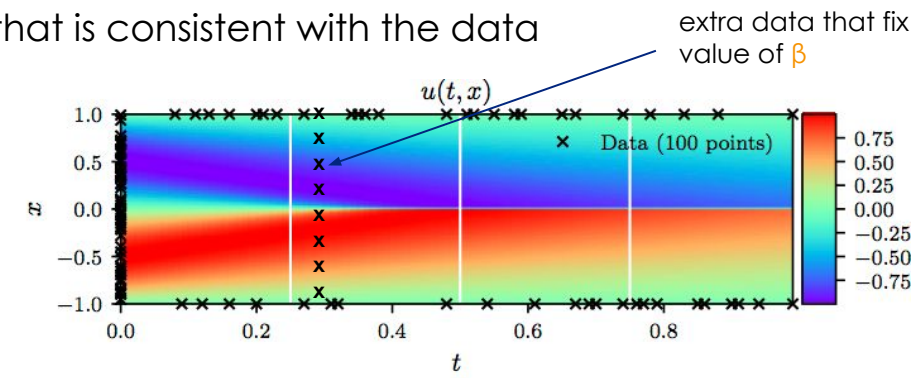
Time-dependent PDE - PINN

Burgers equation: $u_t + uu_x - \beta u_{xx} = 0$

modification: identification of parameter β from data

Assume the data are measured at different times → looking for value of β that is consistent with the data





just treat β as an optimized parameter, i.e. compute $\partial L/\partial \beta$ and optimize jointly with Θ

why to use the method: easy to use, flexible

Example 4 - hybrid method for solving time-dependent problem

[Raissi (2019)]

Time-dependent PDE - standard numerical schemes

PDE: $u(t, x) = N[u(t, x); \boldsymbol{\theta}_{PDE}]$

Standard forward solve: iterative application of an operator P that moves the solution in time $u(t, x) \rightarrow u(t + dt, x)$

$$u(t, x) = P \circ P \circ P \circ P \circ P \circ P \ldots . \circ P(u_0(x); \boldsymbol{\theta}_{PDE})$$

E.g.: Euler scheme: $P: u(t, \mathbf{x}; \boldsymbol{\theta}_{PDE}) \rightarrow u(t+dt, \mathbf{x}; \boldsymbol{\theta}_{PDE}) = u(t, \mathbf{x}; \boldsymbol{\theta}_{PDE}) + dt \cdot N[u(t, \mathbf{x}); \boldsymbol{\theta}_{PDE}]$

With AD, the time-shift operator can be made differentiable to (back) propagate gradients through temporal evolution

- $\partial(u(t_1) - u_{data}(t_1))^2 / \partial\, u(t_2)$ allows finding solution $u(t_2)$ in any time t_2
- $\partial(u(t_1) - u_{data}(t_1))^2 / \partial\, \boldsymbol{\theta}_{PDE}$ allows identification of the value of unknown PDE parameters $\boldsymbol{\theta}_{PDE}$ from the measured data

[Raissi (2019)]

Time-dependent PDE - inverse problem

Burgers equation: $u_t + uu_x - \beta u_{xx} = 0$

Task: from known data at arbitrary time $u_{known}(t_1, x)$ infer initial conditions in time t_0 that generated them:

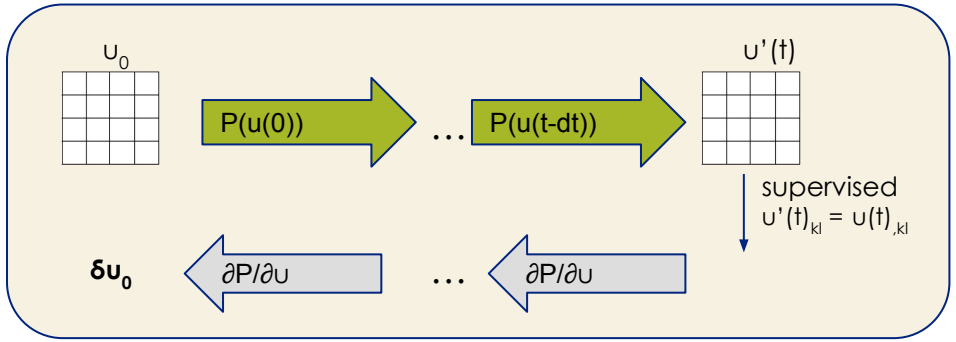
random $\rightarrow u_0$

while L large:

$u_t = P \circ P \circ \ldots \circ P(u_0)$

$L = \sum(u_t - u_{known})^2$

$u_0 \rightarrow u_0 - lr \cdot \partial L / \partial u_0$



Note: the larger the $|t_1 - t_0|$, the harder the optimization (vanishing gradients)

[Raissi (2019)]

Time-dependent PDE - mesh free

Burgers equation: $u_t + uu_x - \beta u_{xx} = 0$ **modification:** generate u_0 by mesh-free NN

Task: from known data $u_{known}(t_1, x)$ infer initial conditions in time t_0 that generated them:

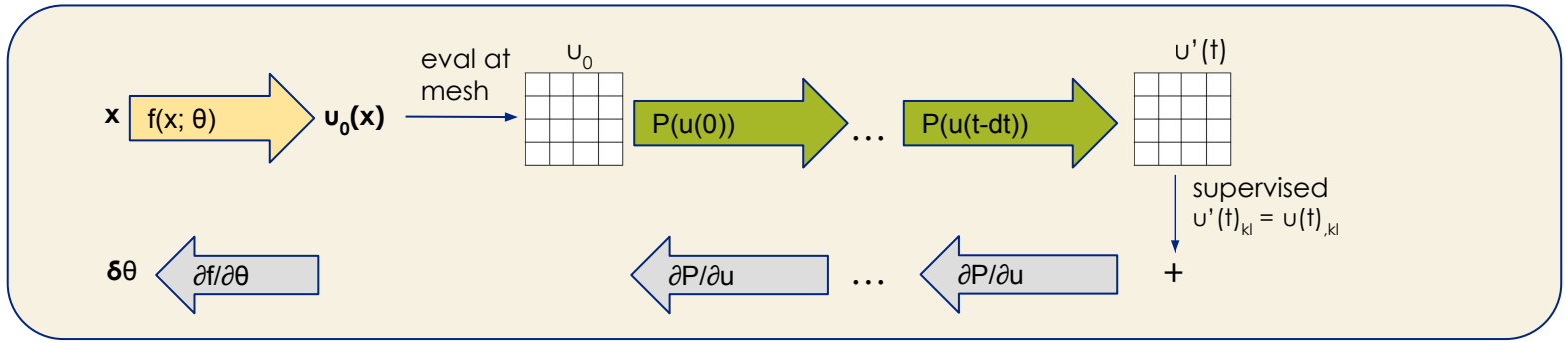
~~random → u_θ~~
random → θ

while L large:

$u_t = P \circ P \circ \ldots \circ P(f_{NN}(x; \theta))$

$L = \sum(u_t - u_{known})^2$

~~$u_\theta \rightarrow u_\theta - lr \cdot \partial L / \partial u_\theta$~~
$\theta \rightarrow \theta - lr \cdot \partial L / \partial \theta$



Note: the larger the $|t_1 - t_0|$, the harder the optimization (vanishing gradients)

[Raissi (2019)]

Time-dependent PDE

Burgers equation: $u_t + uu_x - \beta u_{xx} = 0$ **modification:** find dependence of u_0 on β

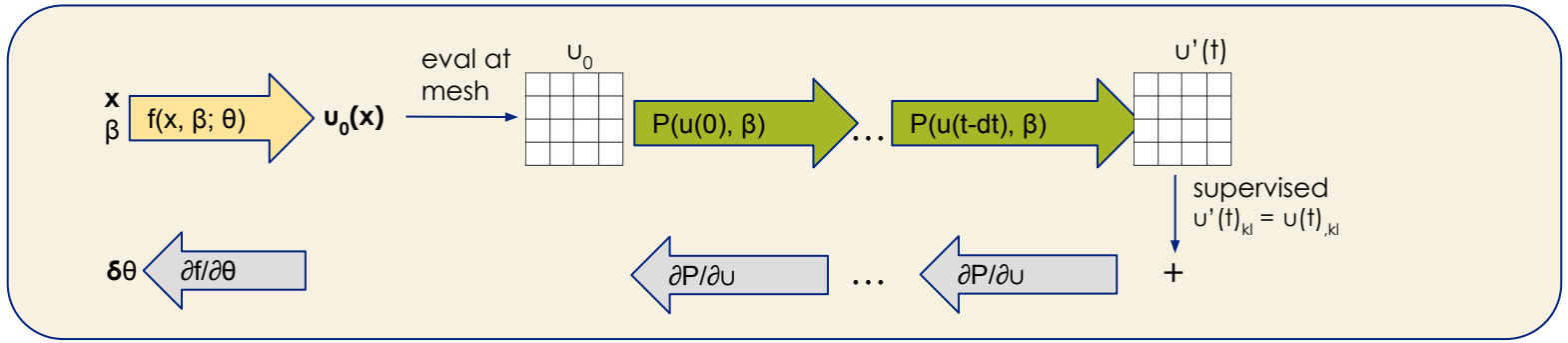
Task: from known data $u_{known}(t_1, x)$ infer initial conditions in time t_0 that generated them for any plausible value of β:

~~random → u_0~~
random → θ

while L large:
random → β
$u_t = P \circ P \circ \ldots \circ P(f_{NN}(x, \beta; \theta))$

$L = \sum(u_t - u_{known})^2$

~~$u_0 \rightarrow u_0 - lr \cdot \partial L / \partial u_0$~~
$\theta \rightarrow \theta - lr \cdot \partial L / \partial \theta$



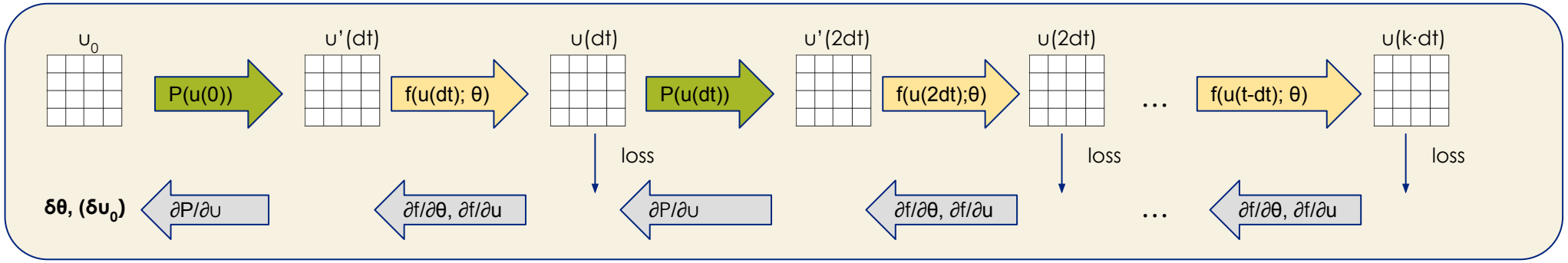
Note: the larger the $|t_1 - t_0|$, the harder the optimization (vanishing gradients)

Hybrid forward solver

PDE: $u(t, x) = N[u(t, x); \boldsymbol{\theta}]$

modification: alternate time shift operator and solution correction by NN

Task: improve accuracy of a standard integration scheme



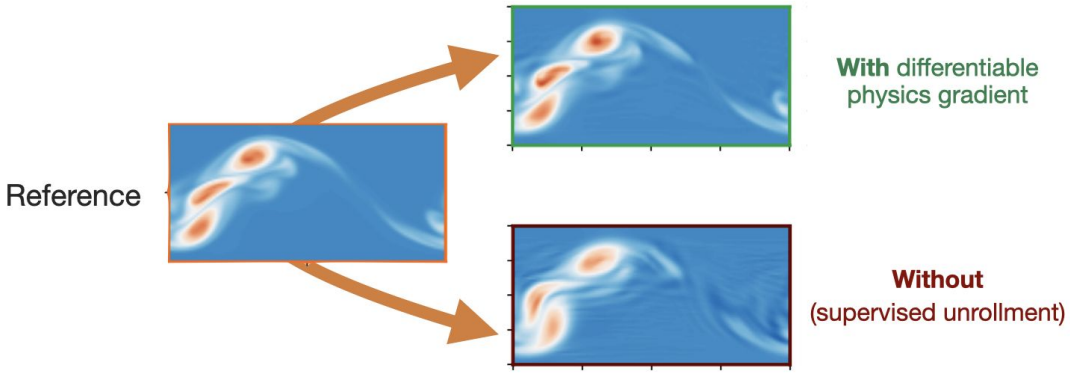
- Each time step is **predicted** by classical method and **corrected** (e.g. conservation laws) by NN
- Since NN acts as a correction of u', often better: $u = u' + f_{NN}(u', \theta)$
- f is a single NN receiving feedback from multiple time steps

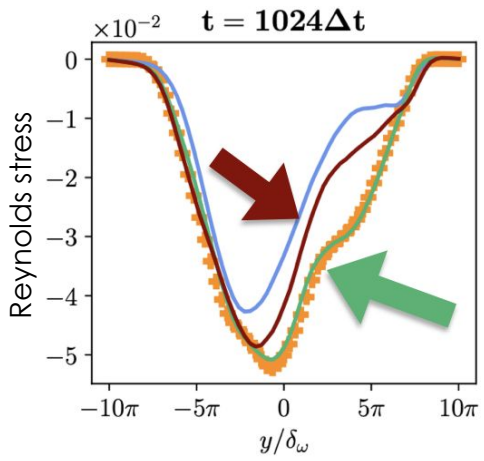
why to use the method: improved precision

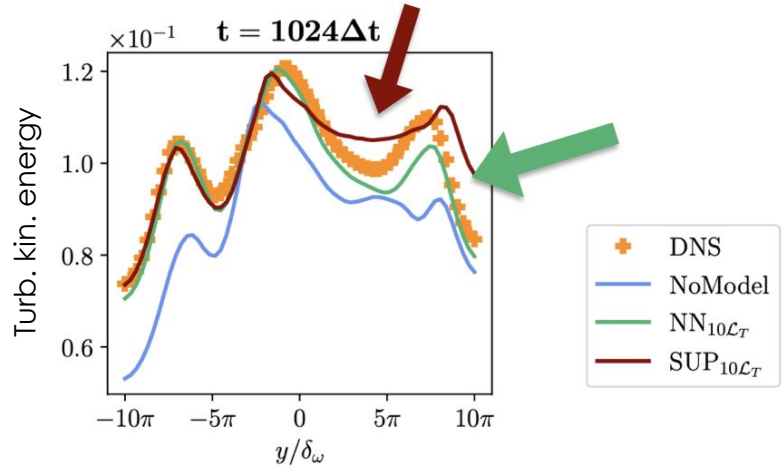
Coarse-grained simulations

- f_{CNN} correction step can learn to implicitly up-sample the solution
  ⇒ the main simulation can run spatially or temporarily under-resolved ⇒ **speedup**

- **how:** perform high-resolution simulation (DNS) + train hybrid solver on a coarse-grained grid / approximate equations
  - use supervised L2 loss against the high-resolution ground truth
    + additional losses capturing differences in important quantities (energy spectrum, strain, mean flow, …)



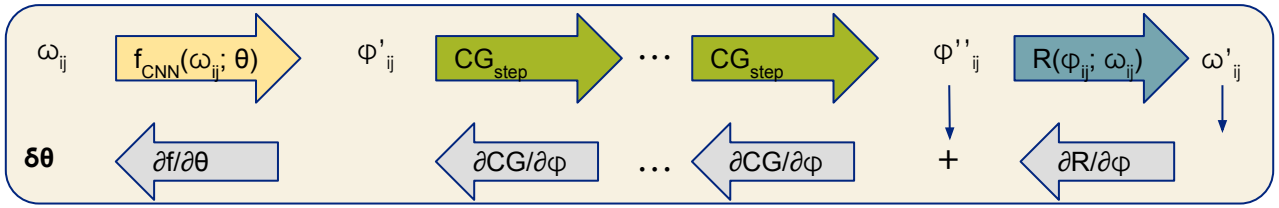




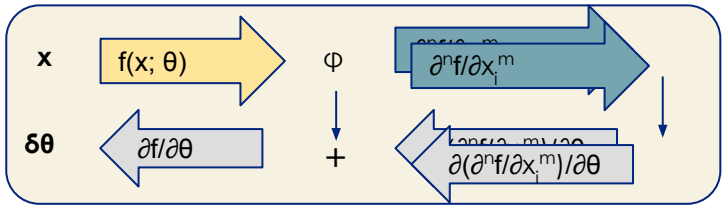
[Bjoern (2022)]

Summary

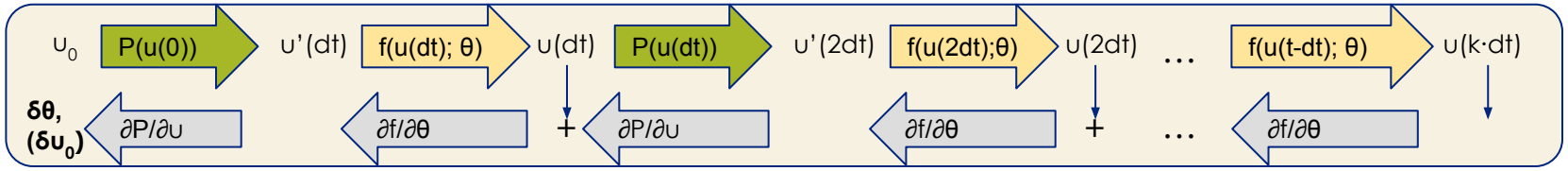
- Differentiable physics can improve standard methods of solving PDEs in terms of accuracy, speed and flexibility
  - slowly penetrating into high-temperature fusion plasma simulations
  - many schemes and applications how to combine NN and classical methods are possible, we touched just a few:

- 

NN model of D^{-1} + iterative optimizer for speed-up

- 

PINN for simplicity, flexibility and getting mesh-free, but typically slow

- 

hybrid solver for precision and speed-up (coarse-graining)

Hybrid operators

Pros:

- **leverages and improves existing efficient numerical solvers and discretizations**
  - efficient
  - good control of solution precision

Cons:

- more complicated implementation
- needs discretization
- needs deeper understanding of the solved problem

Physics Informed Neural Networks

Pros:

- **simple flexible formulation, ease of use**
  - simple to combine multiple PDEs (e.g. grad-shafranov equilibrium + braginskii transport in SOL)
- exact analytical derivatives via AD
- mesh-free

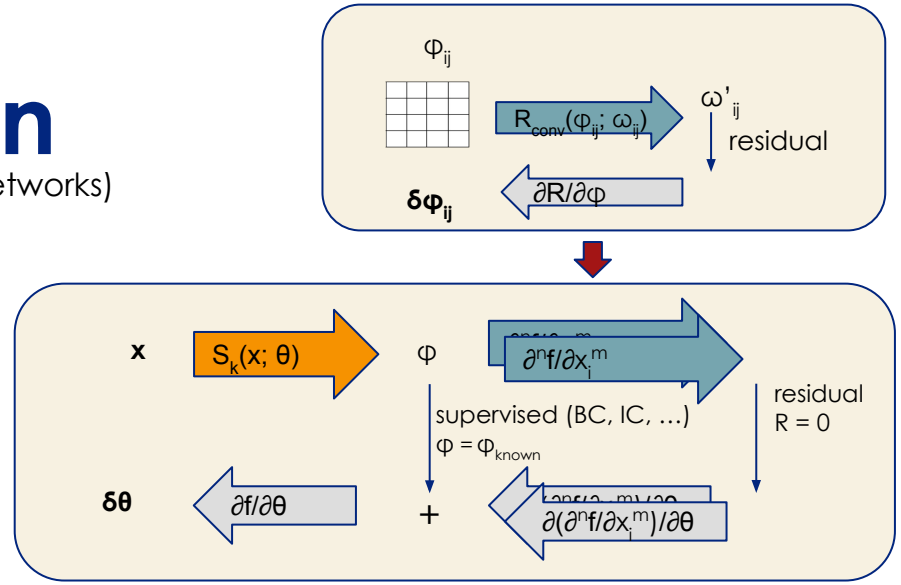
Cons:

- expensive evaluation
- incompatible with existing numerical methods
- poor control of solution precision
  - depends on the capacity of NN to represent it and proper convergence

Backup slides

Poisson equation - spline solution

(typically not used but may help with understanding the principle of Physics Informed Neural Networks)



Poisson equation: $\Delta\varphi = \omega$

Find a spline representation of φ that solves the Poisson equation:

spline $S_k(x; \theta)$: piecewise n-D polynomial with parametric continuity C^{k-1}

AD can compute exact spline derivatives

1. randomly initialize spline parameters **θ**

$$\varphi = S_k(x, y; \boldsymbol{\theta}_{random})$$

2. AD: 2nd derivatives at random points

exact derivatives: $\partial^2 S_k(x, y; \boldsymbol{\theta})/\partial x^2$; $\partial^2 S_k(x, y; \boldsymbol{\theta})/\partial y^2$

3. Exact residual loss

$$L(x, y, \boldsymbol{\theta}) = \sum[\partial^2 S_k(x, y; \boldsymbol{\theta})/\partial x^2 + \partial^2 S_k(x, y; \boldsymbol{\theta})/\partial y^2 - \omega(x, y)] + \alpha \sum_{BC}(\varphi'_{BC} - \varphi_{BC})$$

4. optimize, e.g by gradient descent, to find **θ** that will minimize L, i.e. fit the spline to the PDE

finds single C^{k-1} **continuous solution with exact derivatives** to single ω at a time