Deep Learning in Scientific Computing

Introduction to Differentiable Physics - Part 1

Spring Semester 2023

Siddhartha Mishra
Ben Moseley

ETH zürich

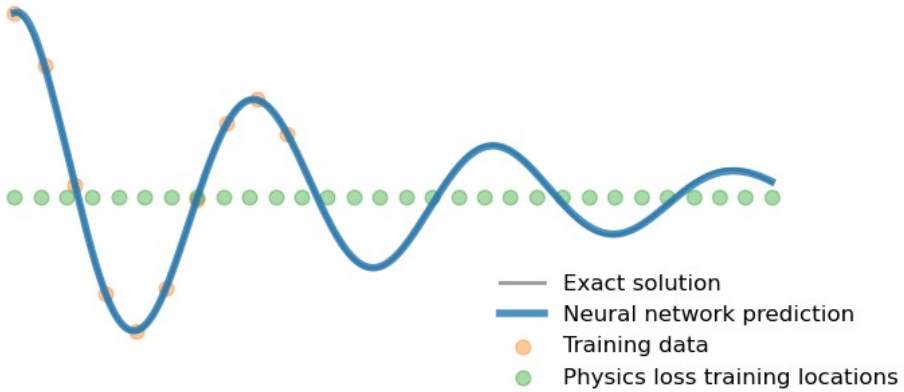
Course timeline

Tutorials		Lectures	
Tue 3:15-14:00, HG E5		Fri 12:15-14:00, HG D1.1	
21.02.		24.02.	~~Course introduction~~
28.02.	~~Intro to PyTorch~~	03.03.	~~Introduction to deep learning I~~
07.03.	~~Deep learning in PyTorch I~~	10.03.	~~Introduction to deep learning II~~
14.03.	~~Deep learning in PyTorch II~~	17.03.	~~Physics-informed neural networks – introduction and theory~~
21.03.	~~Implementing PINNs I~~	24.03.	~~Physics-informed neural networks – applications~~
28.03.	~~Implementing PINNs II~~	31.03.	~~Physics-informed neural networks – limitations and extensions~~
04.04.	~~Implementing PINNs III~~	07.04.	
11.04.		14.04.	
18.04.	~~Introduction to projects~~	21.04.	~~Introduction to operator learning~~
25.04.	~~Implementing neural operators I~~	28.04.	~~Operator networks and DeepONet~~
02.05.	~~Implementing neural operators II~~	05.05.	~~DeepONet continuation~~
09.05.	~~Project work~~	12.05.	~~Neural operators~~
16.05.	~~Implementing neural operators III~~	19.05.	~~Limitations of neural operators~~
23.05.	~~Project work~~	26.05.	Introduction to differentiable physics I
30.05.	Coding an autodiff engine	02.06.	Introduction to differentiable physics II

Lecture overview

- PINNs/ operator learning recap
- When should I use DNNs for scientific problems?
- Hybrid SciML approaches
  - Residual modelling
  - Opening the “black-box”
- Differentiable physics
  - How to train hybrid approaches
  - Autodifferentiation as a key enabler
  - Autodifferentiation recap

Course recap - PINNs



$$m\frac{d^2u}{dt^2} + \mu\frac{du}{dt} + ku = 0$$

$$NN(t;\theta) \approx u(t)$$

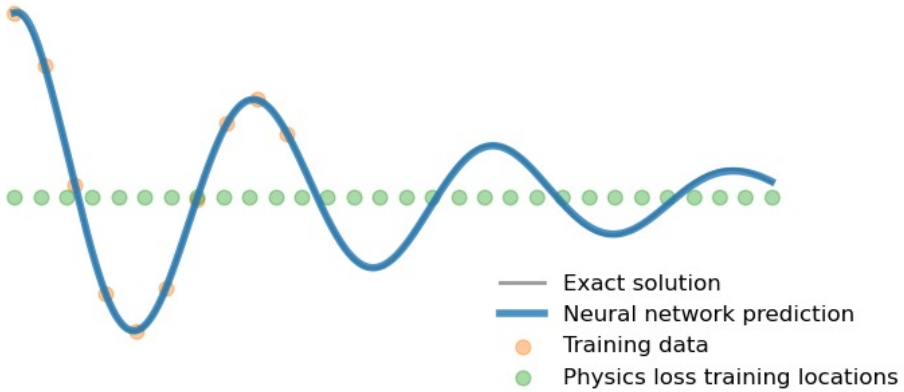
Boundary loss

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(t_i;\theta) - u(t_i))^2$$

$$+ \frac{\lambda}{M}\sum_{j}^{M}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right]NN(t_j;\theta)\right)^2$$

Physics loss

Course recap - PINNs



$$m\frac{d^2u}{dt^2} + \mu\frac{du}{dt} + ku = 0$$

$$NN(t;\theta) \approx u(t)$$

Boundary loss

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(t_i;\theta) - u(t_i))^2$$

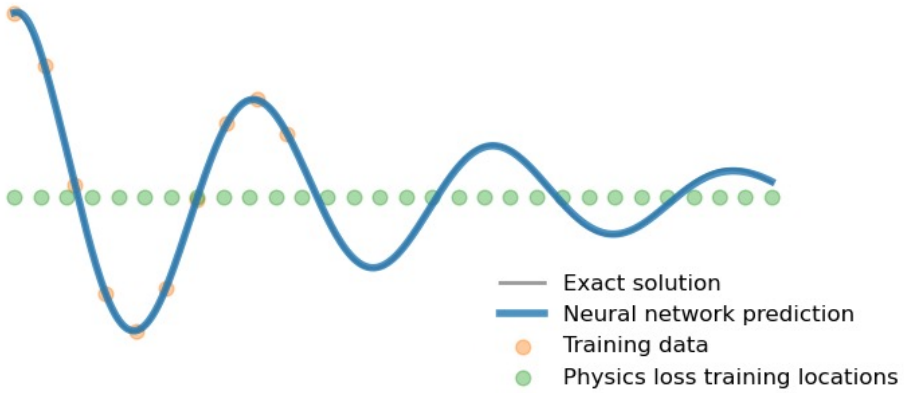
$$+ \frac{\lambda}{M}\sum_{j}^{M}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right]NN(t_j;\theta)\right)^2$$

Physics loss

Advantages of PINNs

Limitations of PINNs

Course recap - PINNs



$$m\frac{d^2u}{dt^2} + \mu\frac{du}{dt} + ku = 0$$

$$NN(t;\theta) \approx u(t)$$

Boundary loss

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(t_i;\theta) - u(t_i))^2 + \frac{\lambda}{M}\sum_{j}^{M}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right]NN(t_j;\theta)\right)^2$$

Physics loss

Advantages of PINNs

- **Mesh-free**
- Can jointly solve forward and inverse problems
- Often performs well on "**messy**" problems (where some observational data is available)
- Mostly **unsupervised**
- Can perform well for high-dimensional PDEs

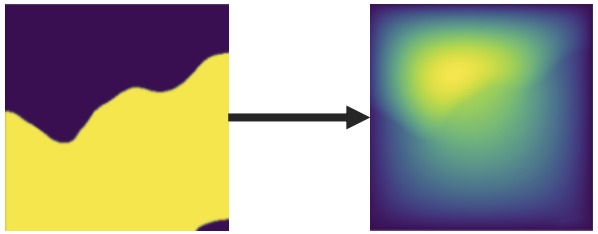
Limitations of PINNs

- **Computational cost** often high (especially for forward-only problems)
- Can be hard to **optimise**
- Challenging to **scale** to high-frequency, multi-scale problems

Course recap – Operator learning

Darcy PDE

$$\nabla \cdot \big(a(\boldsymbol{x})\nabla u(\boldsymbol{x})\big) = f(\boldsymbol{x})$$



Permeability, $a(x)$ Pressure, $u(x)$

$$\begin{tikzcd}
a(x) & \mathcal{X} \arrow[r, "\mathcal{G}"] \arrow[d, bend left, "\mathcal{P}"] \arrow[d, squiggly, "\mathcal{E}"'] & \mathcal{Y} \arrow[d, bend left, "\mathcal{R}"] \arrow[d, squiggly, "\mathcal{D}"'] & \hat{u}(x) \\
& \mathbb{R}^m \arrow[r, squiggly, "\mathcal{A}"'] & \mathbb{R}^p &
\end{tikzcd}$$

Fourier coeffs.

Fourier interpolation

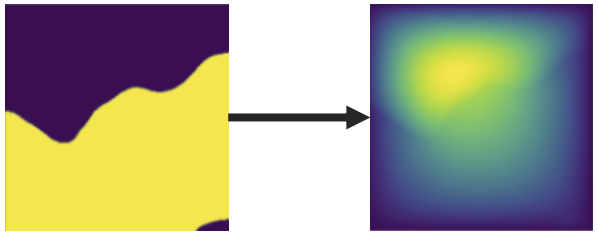
$$a(x) \to \overbrace{\{a_k\}_{k=1}^m \to NN(\{a_k\};\theta) \to \{u_k\}_{k=1}^p}^{\mathcal{G}_\theta^*[a]} \to \hat{u}(x)$$

$$L(\theta) = \frac{1}{NM}\sum_i^N \sum_j^M \left\| u_i(x_j) - \mathcal{G}_\theta^*[a_i](x_j) \right\|^2$$

Course recap – Operator learning

Darcy PDE

$$\nabla \cdot \big(a(\boldsymbol{x})\nabla u(\boldsymbol{x})\big) = f(\boldsymbol{x})$$



Permeability, $a(x)$ Pressure, $u(x)$

$$\begin{tikzcd}
a(x) & \mathcal{X} \arrow[r, "\mathcal{G}"] \arrow[d, squiggly, bend right, "\mathcal{E}"'] & \mathcal{Y} \arrow[d, bend left, "\mathcal{P}"] & \hat{u}(x) \\
& \mathbb{R}^m \arrow[u, bend left, "\mathcal{D}"'] \arrow[r, squiggly, "\mathcal{A}"'] & \mathbb{R}^p \arrow[u, squiggly, bend right, "\mathcal{R}"'] &
\end{tikzcd}$$

Fourier coeffs.

Fourier interpolation

Advantages of operator learning **Limitations of operator learning**

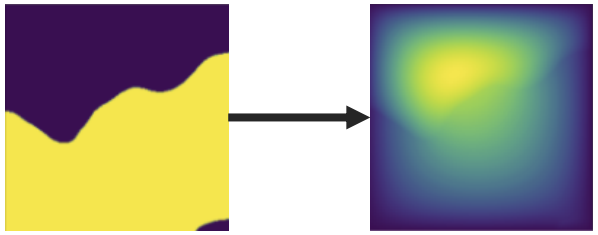
$$a(x) \to \overbrace{\{a_k\}_{k=1}^m \to NN(\{a_k\};\theta) \to \{u_k\}_{k=1}^p}^{\mathcal{G}_\theta^*[a]} \to \hat{u}(x)$$

$$L(\theta) = \frac{1}{NM}\sum_i^N \sum_j^M \left\| u_i(x_j) - \mathcal{G}_\theta^*[a_i](x_j) \right\|^2$$

Course recap – Operator learning

Darcy PDE

$$\nabla \cdot \big(a(\boldsymbol{x})\nabla u(\boldsymbol{x})\big) = f(\boldsymbol{x})$$



Permeability, $a(x)$ Pressure, $u(x)$

$$\begin{tikzcd}
a(x) & \mathrm{X} \arrow[r, "\mathcal{G}"] \arrow[d, bend right, "\mathcal{E}"'] & \mathrm{Y} & \hat{u}(x) \\
& \mathbb{R}^m \arrow[u, bend right, "\mathcal{D}"'] \arrow[r, squiggly, "\mathcal{A}"'] & \mathbb{R}^p \arrow[u, bend right, "\mathcal{R}"'] \arrow[d, bend right, "\mathcal{P}"'] & \\
& & &
\end{tikzcd}$$

Fourier coeffs.

Fourier interpolation

Advantages of operator learning

- Can be **orders of magnitude faster** than traditional simulation (once trained)

Limitations of operator learning

- Can require **lots** of training data, which can be expensive to obtain
- Can struggle to **generalise** to inputs outside of its training data
- Encoding / reconstruction steps require some **assumptions** about the regularity of $a(x)$ and $u(x)$

$$a(x) \to \overbrace{\{a_k\}_{k=1}^{m} \to NN(\{a_k\};\theta) \to \{u_k\}_{k=1}^{p}}^{\mathcal{G}_\theta^*[a]} \to \hat{u}(x)$$

$$L(\theta) = \frac{1}{NM}\sum_i^N \sum_j^M \left\| u_i(x_j) - \mathcal{G}_\theta^*[a_i](x_j) \right\|^2$$

When should I use deep neural networks for scientific problems?

Advantages of DNNs

- Usually very **fast** (once trained)
- Can represent highly **non-linear** functions

Limitations of DNNs

- Often lots of **training data required**
- Can be hard to **optimise**
- Can be hard to **interpret**
- Often struggle to **generalise**

When should I use deep neural networks for scientific problems?

Advantages of DNNs

- Usually very **fast** (once trained)
- Can represent highly **non-linear** functions

Limitations of DNNs

- Often lots of **training data required**
- Can be hard to **optimise**
- Can be hard to **interpret**
- Often struggle to **generalise**

General advice

Use DNNs to:

1) **Accelerate** your workflow, or
2) Learn the **parts** you are unsure of / have incomplete knowledge

Otherwise using DNNs may **not** be a good idea!

What are hybrid SciML approaches?

Note: all the DNNs so far have entirely **replaced** traditional algorithms

Key idea: incorporate DNNs directly into a traditional algorithm
= hybrid approach

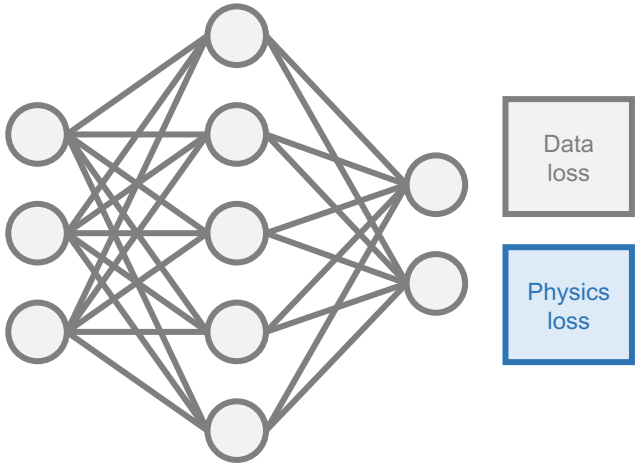
General advice

Use DNNs to:

1) **Accelerate** your workflow, or
2) Learn the **parts** you are unsure of / have incomplete knowledge

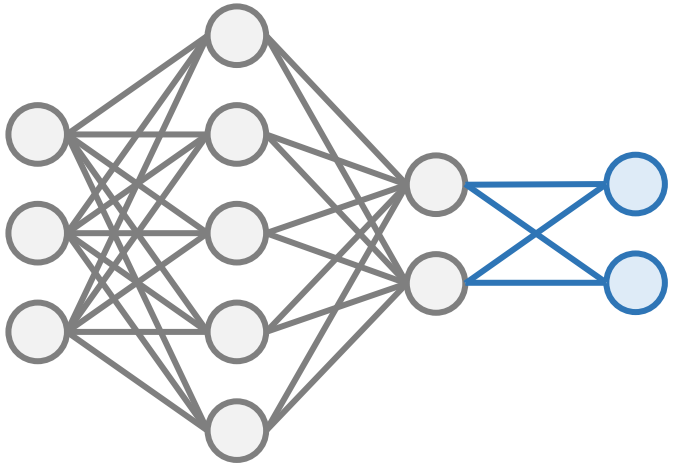
Ways to incorporate scientific principles into machine learning

Loss function



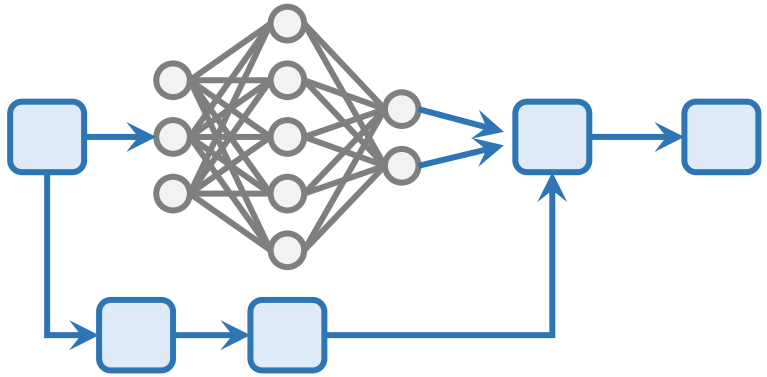
Example:
Physics-informed neural networks
(add governing equations to loss function)

Architecture



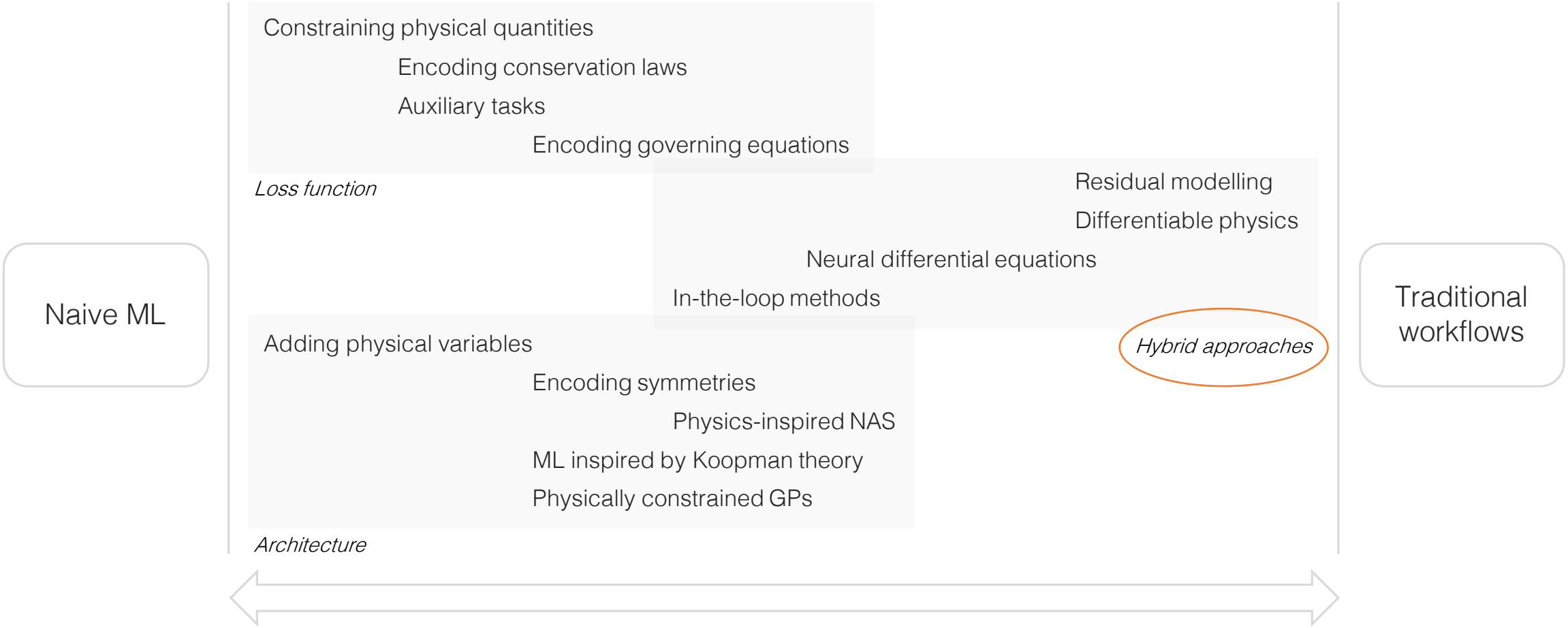
Example:
Encoding regularity / symmetries / conservation laws (e.g. energy conservation, rotational invariance), **operator learning**

Hybrid approaches



Example:
Neural differential equations
(incorporating neural networks into traditional PDE solvers)

A plethora of SciML techniques



B Moseley, Physics-informed machine learning: from concepts to real-world applications, PhD thesis, 2022

A plethora of SciML techniques

Naive ML

Fully learned

Fully data-driven

No physics constraints

Constraining physical quantities

Encoding conservation laws

Auxiliary tasks

Encoding governing equations

Loss function

Residual modelling

Differentiable physics

Neural differential equations

In-the-loop methods

Hybrid approaches

Adding physical variables

Encoding symmetries

Physics-inspired NAS

ML inspired by Koopman theory

Physically constrained GPs

Architecture

Traditional workflows

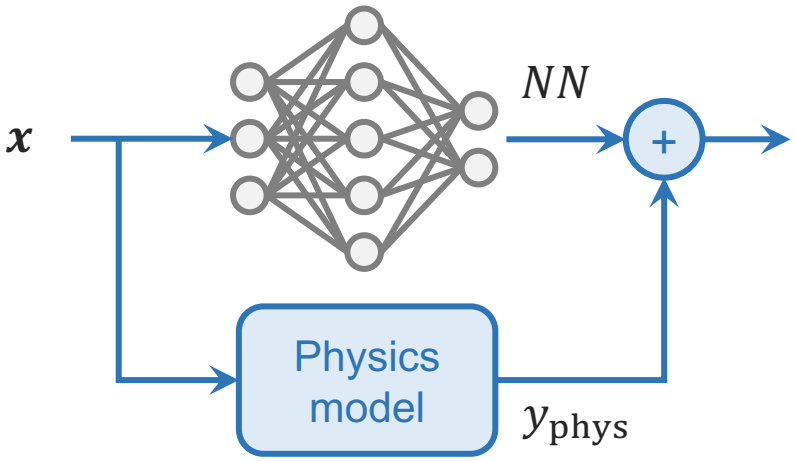
No learning

No data required

Hard physics constraints

B Moseley, Physics-informed machine learning: from concepts to real-world applications, PhD thesis, 2022

A simple hybrid approach – residual modelling



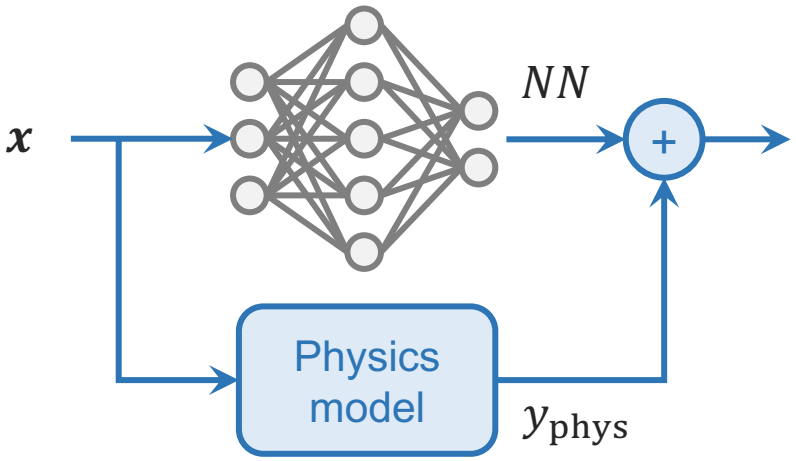
$$\hat{y} = y_{\text{phys}}(\boldsymbol{x}) + NN(\boldsymbol{x}; \theta)$$

I.e. neural network learns residual correction to physics model

Trained using many examples of inputs/outputs

When is this useful?

A simple hybrid approach – residual modelling

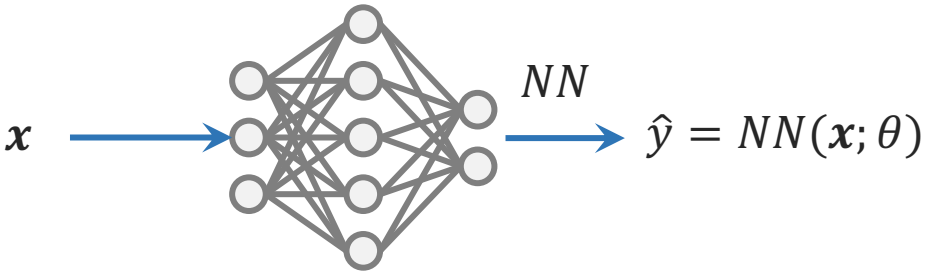


$$\hat{y} = y_{\text{phys}}(\boldsymbol{x}) + NN(\boldsymbol{x};\theta)$$

I.e. neural network learns residual correction to physics model

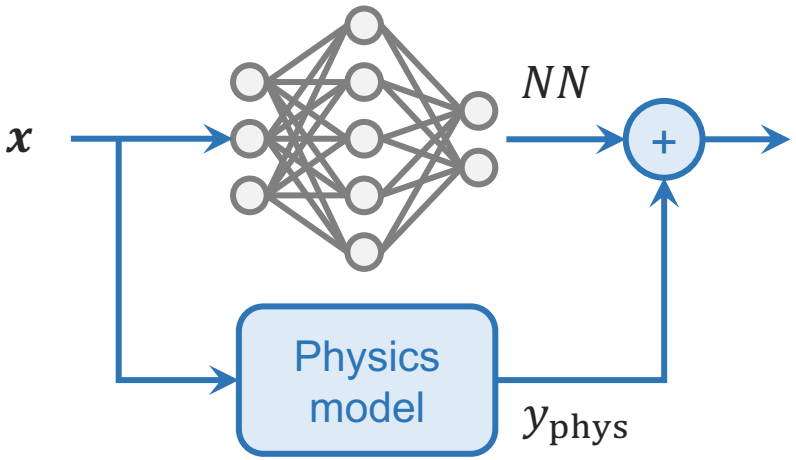
Trained using many examples of inputs/outputs

Useful when:
- We have incomplete understanding of physics
- More complex physical modeling is too expensive



Compared to naïve ML approach:
- **Easier** learning task: don't need to learn all the physics
- More **interpretable**

A simple hybrid approach – residual modelling



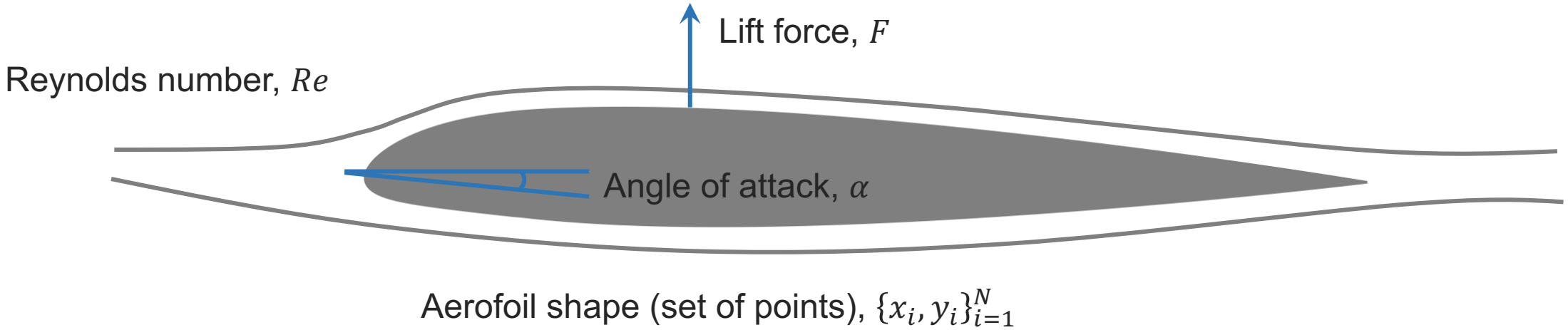
$$\hat{y} = y_{\text{phys}}(\boldsymbol{x}) + NN(\boldsymbol{x};\theta)$$

I.e. neural network learns residual correction to physics model

Trained using many examples of inputs/outputs

$$\begin{aligned} L(\theta) &= \sum_i^N \left(\hat{y}(\boldsymbol{x}_i;\theta) - y(\boldsymbol{x}_i)\right)^2 \\ &= \sum_i^N \left(NN(\boldsymbol{x}_i;\theta) - [y(\boldsymbol{x}_i) - y_{\text{phys}}(\boldsymbol{x}_i)]\right)^2 \\ &\equiv \sum_i^N (NN(\boldsymbol{x}_i;\theta) - r(\boldsymbol{x}_i))^2 \end{aligned}$$

Residual modelling – aerofoil example



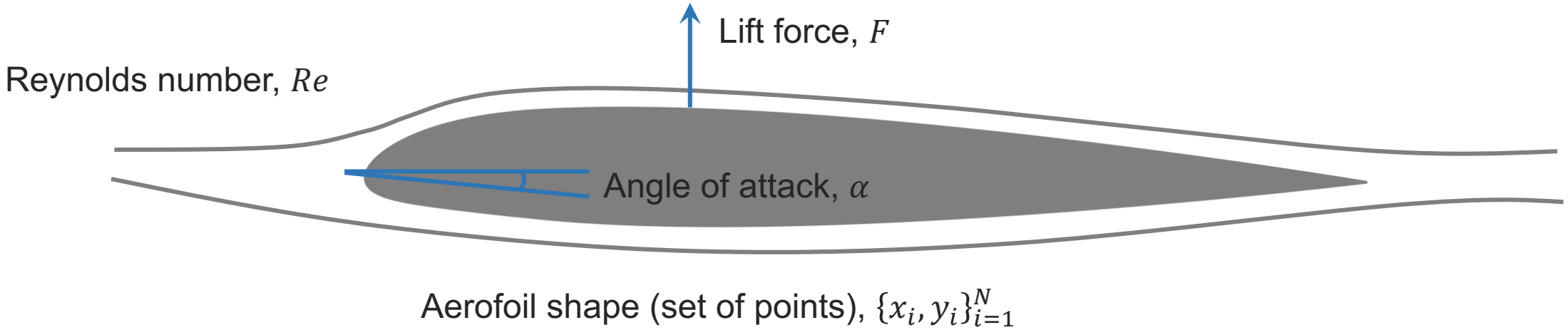
Simulation task:

Given $\{x_i, y_i\}_{i=1}^{N}$, Re and α

Predict F

Pawar et al, Physics guided machine learning using simplified theories, Physics of Fluids (2021)

Residual modelling – aerofoil example



- Full CFD simulations are typically accurate, but very expensive
- Faster approximate methods exist, but are usually less accurate

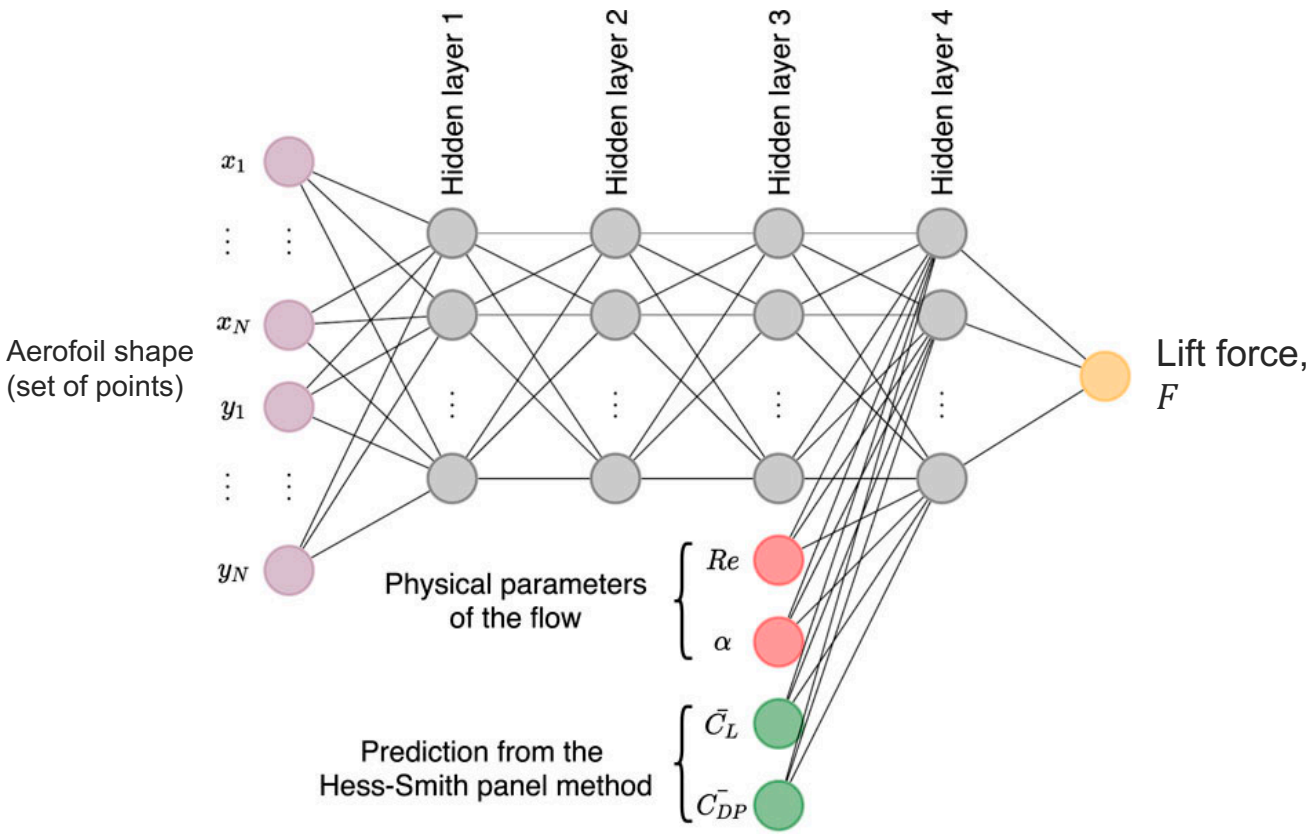
Simulation task:

Given $\{x_i, y_i\}_{i=1}^{N}$, Re and α

Predict F

Pawar et al, Physics guided machine learning using simplified theories, Physics of Fluids (2021)

Residual modelling – aerofoil example



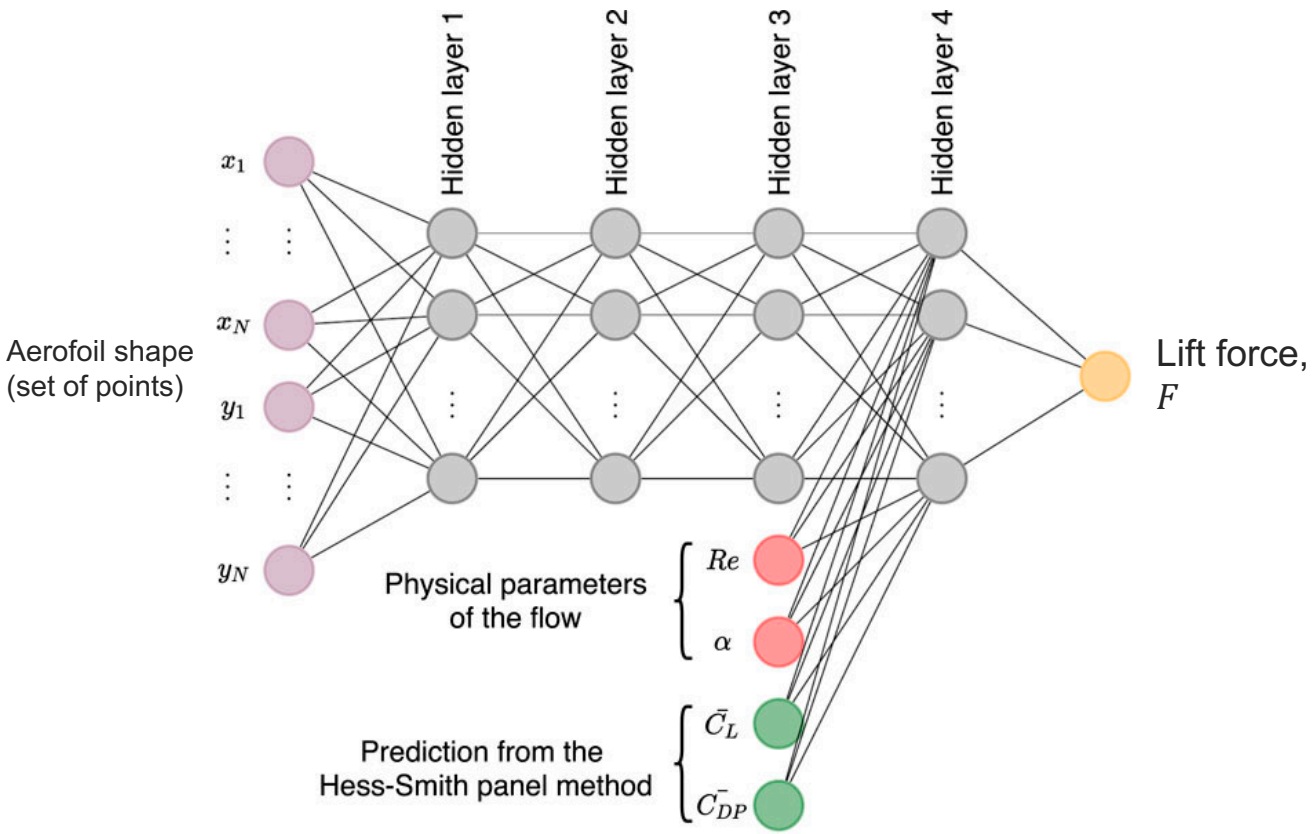
Hess-Smith panel method:
Fast **approximate** method for predicting lift force

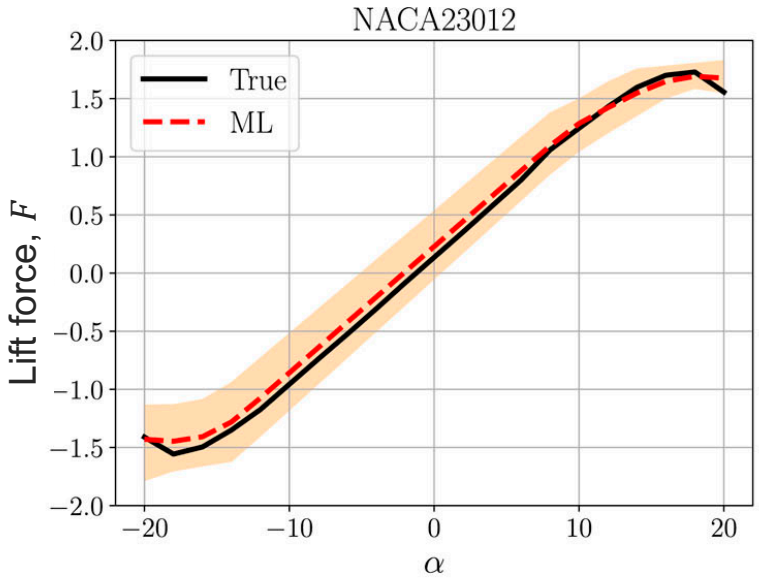
Training data:
Many example inputs/outputs generated from (expensive) **high-fidelity** CFD modelling

Goal:
A model which is **faster** than CFD and more **accurate** than approximate physics model

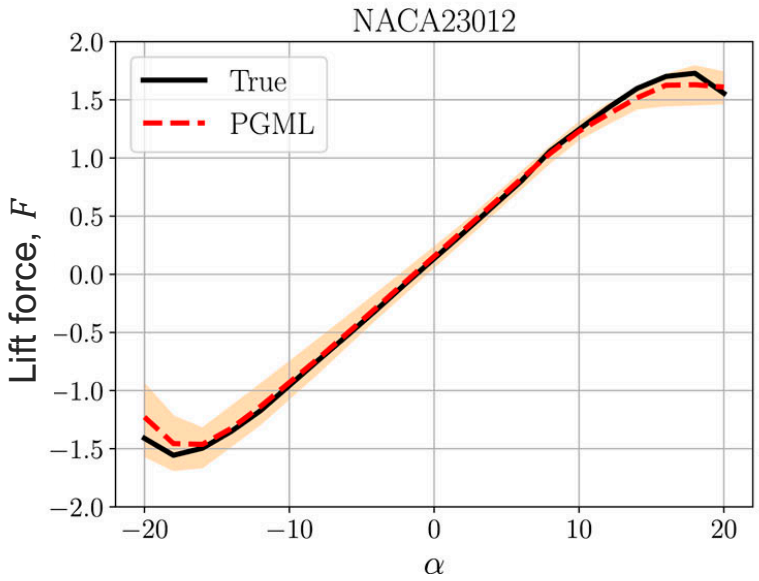
Pawar et al, Physics guided machine learning using simplified theories, Physics of Fluids (2021)

Residual modelling – aerofoil example





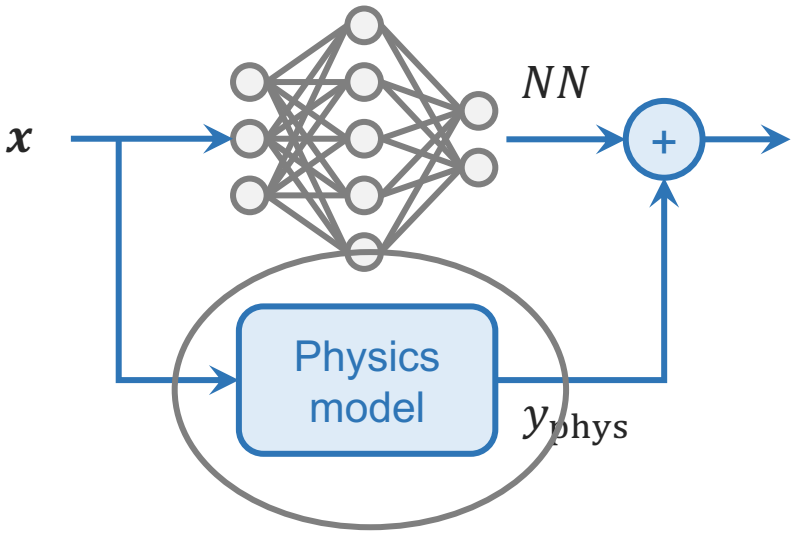
Naive NN (no physics inputs)



NN + physics model (hybrid approach)

Pawar et al, Physics guided machine learning using simplified theories, Physics of Fluids (2021)

Opening the black-box



$$\hat{y} = y_{\text{phys}}(\boldsymbol{x}) + NN(\boldsymbol{x}; \theta)$$

Residual methods treat the physics model as a "**black-box**"

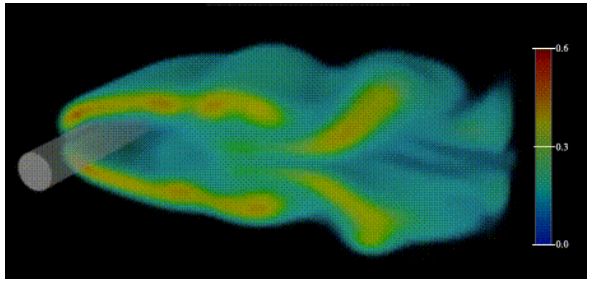
More complex hybrid methods open the box and insert ML **inside** the traditional algorithm

We insert ML where;

1) the algorithm is **slow**
2) we are **unsure** of our assumptions/ want to improve our modelling

Opening the black-box – finite difference solver

FD solver



Incompressible Navier-Stokes equation

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \nu \nabla^2 \boldsymbol{u} = -\frac{\mathbf{1}}{\rho} \nabla p$$

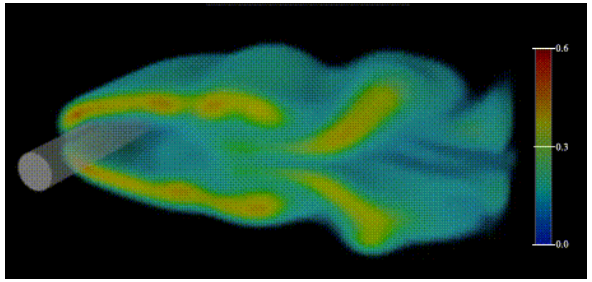
$$\nabla \cdot \boldsymbol{u} = 0$$

$\boldsymbol{u}(\boldsymbol{x}, t)$ is the flow velocity
$p(\boldsymbol{x}, t)$ is the pressure
$\rho(\boldsymbol{x})$ is the density
ν is the viscosity

Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Opening the black-box – finite difference solver

FD solver



Incompressible Navier-Stokes equation

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \nu \nabla^2 \boldsymbol{u} = -\frac{\mathbf{1}}{\rho}\nabla p$$

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

$\boldsymbol{u}(\boldsymbol{x}, t)$ is the flow velocity
$p(\boldsymbol{x}, t)$ is the pressure
$\rho(\boldsymbol{x})$ is the density
ν is the viscosity

"Operator splitting" numerical solver:

Discretise in time

$$\boldsymbol{u}_{t+1} = \boldsymbol{u}_t - \delta t(\boldsymbol{u}_t \cdot \nabla)\boldsymbol{u}_t + \delta t\, \nu \nabla^2 \boldsymbol{u}_t - \frac{\delta t}{\rho}\nabla p_{t+1} \quad (1)$$

Let

$$\boldsymbol{u}^* = \boldsymbol{u}_t - \delta t(\boldsymbol{u}_t \cdot \nabla)\boldsymbol{u}_t + \delta t\, \nu \nabla^2 \boldsymbol{u}_t - \frac{\delta t}{\rho}\nabla p_t \quad (2)$$

Then

$$\boldsymbol{u}_{t+1} = \boldsymbol{u}^* - \frac{\delta t}{\rho}\nabla(p_{t+1} - p_t)$$

Asserting $\nabla \cdot \boldsymbol{u}_{t+1} = 0 \Rightarrow$

$$0 = \nabla \cdot \boldsymbol{u}^* - \frac{\delta t}{\rho}\nabla^2(p_{t+1} - p_t)$$

$$\nabla^2(p_{t+1} - p_t) = \frac{\rho}{\delta t}\nabla \cdot \boldsymbol{u}^*$$

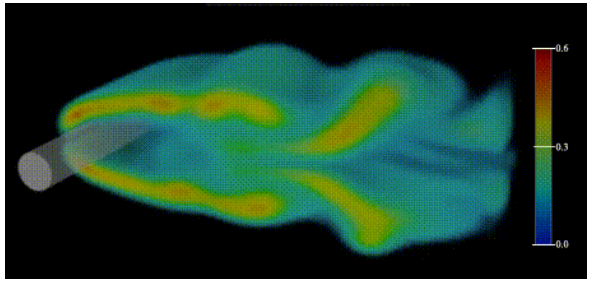
Discretise in space

$$L\left(p_{i,j,k,t+1} - p_{i,j,k,t}\right) = \frac{\rho_{i,j,k}}{\delta t} D\boldsymbol{u}^*_{i,j,k} \quad (3)$$

Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Opening the black-box – finite difference solver

FD solver



Incompressible Navier-Stokes equation

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \nu \nabla^2 \boldsymbol{u} = -\frac{\mathbf{1}}{\rho}\nabla p$$

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

$\boldsymbol{u}(\boldsymbol{x}, t)$ is the flow velocity
$p(\boldsymbol{x}, t)$ is the pressure
$\rho(\boldsymbol{x})$ is the density
ν is the viscosity

"Operator splitting" numerical solver:

Discretise in time

$$\boldsymbol{u}_{t+1} = \boldsymbol{u}_t - \delta t(\boldsymbol{u}_t \cdot \nabla)\boldsymbol{u}_t + \delta t\, \nu \nabla^2 \boldsymbol{u}_t - \frac{\delta t}{\rho}\nabla p_{t+1} \quad (1)$$

Let

$$\boldsymbol{u}^* = \boldsymbol{u}_t - \delta t(\boldsymbol{u}_t \cdot \nabla)\boldsymbol{u}_t + \delta t\, \nu \nabla^2 \boldsymbol{u}_t - \frac{\delta t}{\rho}\nabla p_t \quad (2)$$

Then

$$\boldsymbol{u}_{t+1} = \boldsymbol{u}^* - \frac{\delta t}{\rho}\nabla(p_{t+1} - p_t)$$

Asserting $\nabla \cdot \boldsymbol{u}_{t+1} = 0 \Rightarrow$

$$0 = \nabla \cdot \boldsymbol{u}^* - \frac{\delta t}{\rho}\nabla^2(p_{t+1} - p_t)$$

$$\nabla^2(p_{t+1} - p_t) = \frac{\rho}{\delta t}\nabla \cdot \boldsymbol{u}^*$$

Discretise in space

$$L\left(p_{i,j,k,t+1} - p_{i,j,k,t}\right) = \frac{\rho_{i,j,k}}{\delta t} D\boldsymbol{u}^*_{i,j,k} \quad (3)$$

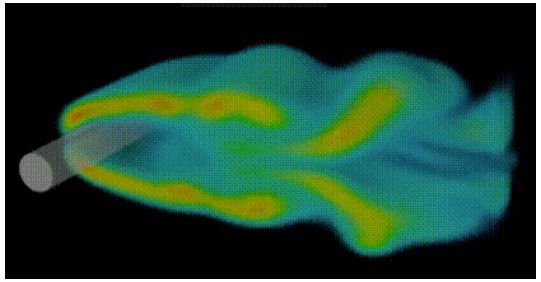
Basic algorithm:
Discretise $\boldsymbol{u}, p$ and ρ
Loop:
1. Compute $\boldsymbol{u}^*_{i,j,k}$ using (2)
2. Solve matrix equation (3) for $p_{i,j,k,t+1}$
3. Compute $\boldsymbol{u}_{i,j,k,t+1}$ using (1)

```
def NS_solver(u_0, p_0, rho, nu):
    "Pseudocode for solving NS equation"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

    return u_t, p_t
```

Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

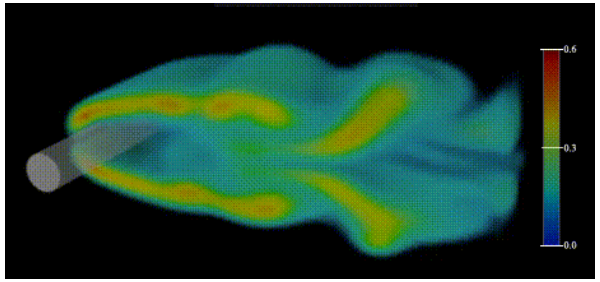
Computational cost / accuracy trade-off

Low fidelity FD solver



(32 x 32 x 64) cells
~10 seconds / 100 timesteps

High fidelity FD solver



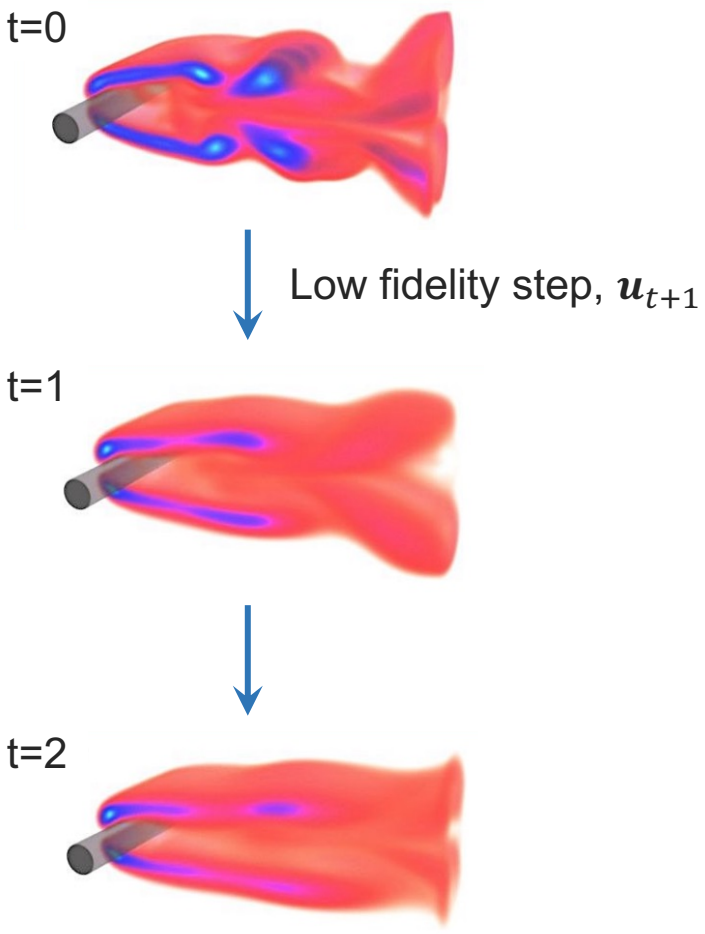
(128 x 128 x 256) cells
~1000 seconds / 100 timesteps

- Discretisation induces **errors** in the solver
- But finer grids are much more computationally expensive
- Can we use ML improve the accuracy of the **low fidelity** solver?

Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Traditional Navier-Stokes solver

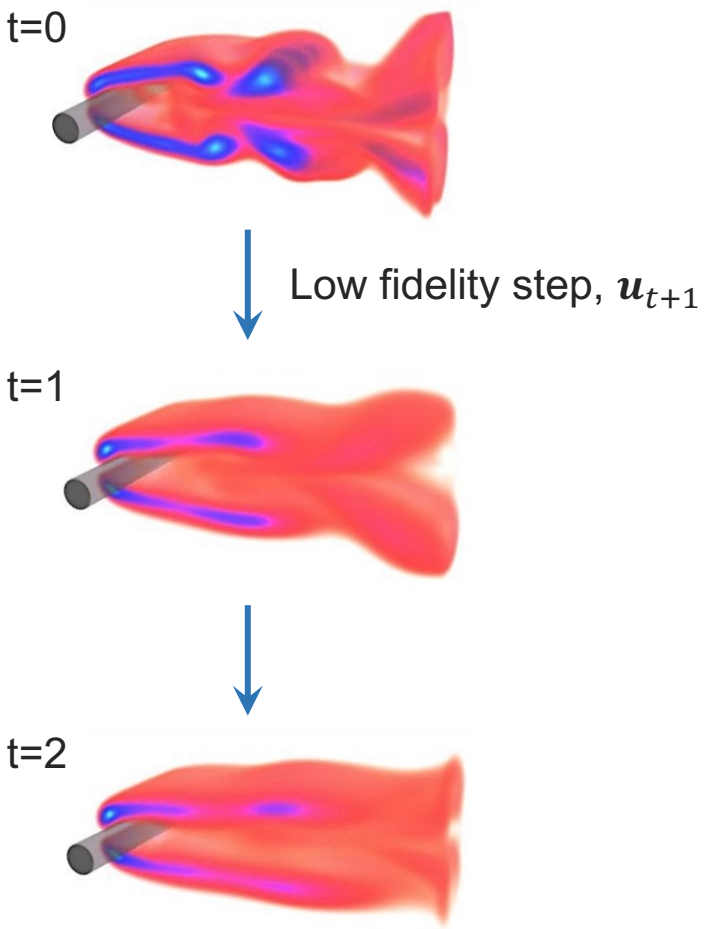
```python
def NS_solver(u_0, p_0, rho, nu):
    "Pseudocode for solving NS equation"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

    return u_t, p_t
```



Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Traditional Navier-Stokes solver

```python
def NS_solver(u_0, p_0, rho, nu):
    "Pseudocode for solving NS equation"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

    return u_t, p_t
```

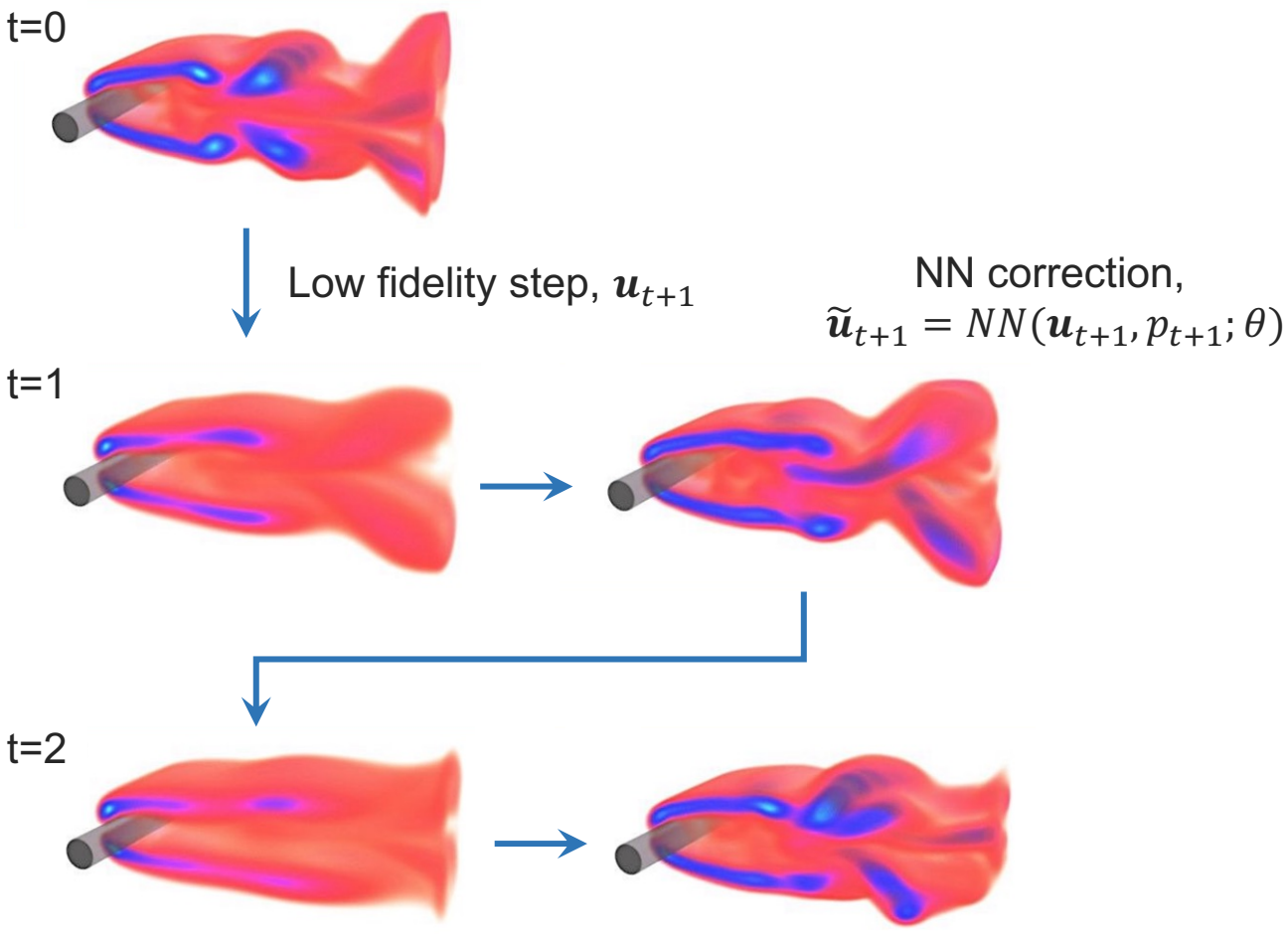


- Where could we insert ML **inside** this workflow to improve accuracy / efficiency?

Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Hybrid Navier-Stokes solver

```python
def NS_solver(u_0, p_0, rho, nu):
    "Pseudocode for solving NS equation"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

    return u_t, p_t


def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = NN(u_t, p_t, theta)

    return u_t, p_t
```



Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

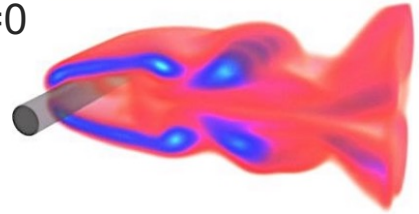
How do we train hybrid approaches?

```python
def NS_solver(u_0, p_0, rho, nu):
    "Pseudocode for solving NS equation"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

    return u_t, p_t


def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = NN(u_t, p_t, theta)

    return u_t, p_t
```

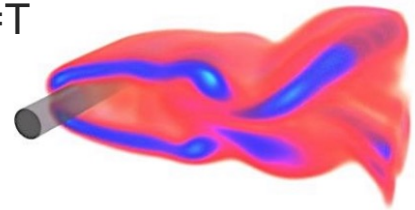
Initial velocity, $\boldsymbol{u}_0$

Final velocity from **high-fidelity** NS_solver, $\boldsymbol{u}_{\text{true } T}$

t=0



t=T



Assume our training data are $\{(\boldsymbol{u}_0^l, \boldsymbol{u}_{\text{true } T}^l)\}_l^N$ generated from high-fidelity simulations

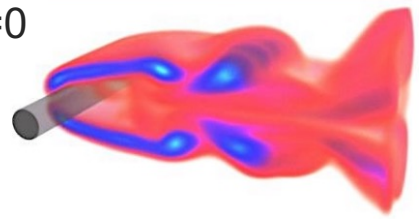
How do we train hybrid approaches?

```python
def NS_solver(u_0, p_0, rho, nu):
    "Pseudocode for solving NS equation"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

    return u_t, p_t


def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = NN(u_t, p_t, theta)

    return u_t, p_t
```

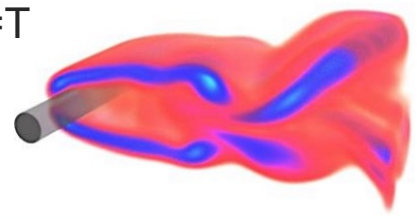
Initial velocity, $\boldsymbol{u}_0$

Final velocity from **high-fidelity** NS_solver, $\boldsymbol{u}_{\text{true } T}$

t=0



t=T



Assume our training data are $\{(\boldsymbol{u}_0^l, \boldsymbol{u}_{\text{true } T}^l)\}_l^N$ generated from high-fidelity simulations

How can we learn θ when the neural network is **inside** the traditional algorithm?

How do we train hybrid approaches?

Key idea: **autodifferentiation** allows us to differentiate **arbitrary** algorithms, not just neural networks!

We train neural networks using **autodifferentiation**

But autodifferentiation = exact gradients of **arbitrary** programs

So, we can use it to differentiate (and learn) traditional algorithms too!

How do we train hybrid approaches?

Key idea: **autodifferentiation** allows us to differentiate **arbitrary** algorithms, not just neural networks!

We train neural networks using **autodifferentiation**

But autodifferentiation = exact gradients of **arbitrary** programs

So, we can use it to differentiate (and learn) traditional algorithms too!

```
def NN(x, theta):
    "Defines a FCN"
    y = torch.tanh(theta[0]@x + theta[1])
    return y

theta.requires_grad_(True)
y = NN(x, theta)
loss = loss_fn(y, y_true)
dtheta = torch.autograd(loss, theta)
# for learning theta (training NN)
```

How do we train hybrid approaches?

Key idea: **autodifferentiation** allows us to differentiate **arbitrary** algorithms, not just neural networks!

```python
def NN(x, theta):
    "Defines a FCN"
    y = torch.tanh(theta[0]@x + theta[1])
    return y

theta.requires_grad_(True)
y = NN(x, theta)
loss = loss_fn(y, y_true)
dtheta = torch.autograd(loss, theta)
# for learning theta (training NN)
```

```python
def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = NN(u_t, p_t, theta)

    return u_t, p_t

theta.requires_grad_(True)
u_T,_ = Hybrid_NS_solver(u_0, p_0, rho, nu, theta)
loss = loss_fn(u_T, u_T_true)
dtheta = torch.autograd(loss, theta)
# for learning theta (training NN)
```

How do we train hybrid approaches?

Key idea: **autodifferentiation** allows us to differentiate **arbitrary** algorithms, not just neural networks!

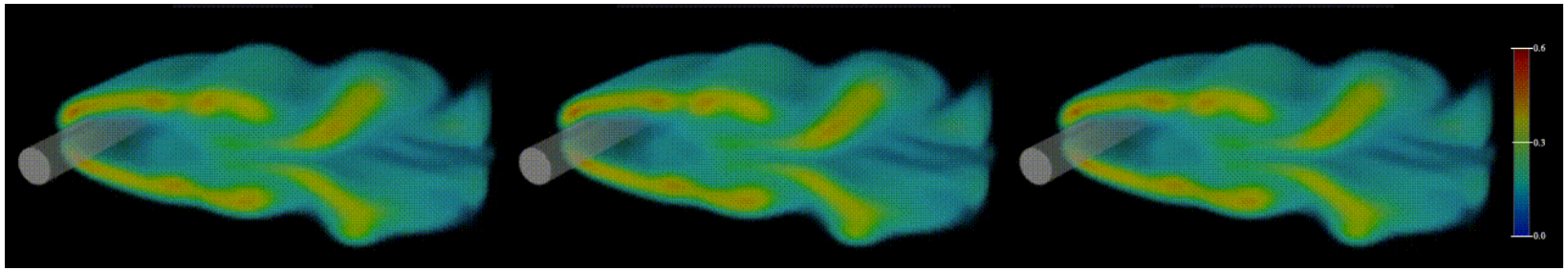
Differentiable physics = using autodifferentiation to differentiate physical algorithms

```
def NN(x, theta):
    "Defines a FCN"
    y = torch.tanh(theta[0]@x + theta[1])
    return y

theta.requires_grad_(True)
y = NN(x, theta)
loss = loss_fn(y, y_true)
dtheta = torch.autograd(loss, theta)
# for learning theta (training NN)
```

```
def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = NN(u_t, p_t, theta)

    return u_t, p_t

theta.requires_grad_(True)
u_T,_ = Hybrid_NS_solver(u_0, p_0, rho, nu, theta)
loss = loss_fn(u_T, u_T_true)
dtheta = torch.autograd(loss, theta)
# for learning theta (training NN)
```

Hybrid approach -

Low fidelity FD solver | Hybrid approach | High fidelity FD solver



32 x 32 x 64 grid cells
~10 seconds / 100 timesteps

128 x 128 x 256 cells
~1000 seconds / 100 timesteps

32 x 32 x 64 grid cells
~15 seconds / 100 timesteps

Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Autodifferentiation is a key enabler

Autodifferentiation is a **key enabler** of all of the SciML techniques studied so far

It allows us to **efficiently** differentiate through complicated loss functions and get gradients of learnable parameters

$$NN(t;\theta) \approx u(t)$$

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}\left(NN(t_i;\theta) - u(t_i)\right)^2 + \frac{\lambda}{M}\sum_{j}^{M}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right]NN(t_j;\theta)\right)^2$$

Physics-informed neural network

$$a(x) \to \overbrace{\{a_k\}_{k=1}^{m} \to NN(\{a_k\};\theta) \to \{u_k\}_{k=1}^{p}}^{\mathcal{G}_\theta^*[a]} \to \hat{u}(x)$$

$$L(\theta) = \frac{1}{NM}\sum_{i}^{N}\sum_{j}^{M}\left\|u_i(x_j) - \mathcal{G}_\theta^*[a_i](x_j)\right\|^2$$

Operator learning

Lecture overview

- PINNs/ operator learning recap
- When should I use DNNs for scientific problems?
- Hybrid SciML approaches
  - Residual modelling
  - Opening the “black-box”
- Differentiable physics
  - How to train hybrid approaches
  - Autodifferentiation as a key enabler
  - Autodifferentiation recap

5 min break

Recap - autodifferentiation

```python
def Hybrid_NS_solver(u_0, p_0, rho, nu, theta):
    "Pseudocode for solving NS equation, with NN correction"

    # u_0, p_0 have shape (NX, NY, NZ)
    u_t, p_t = u_0, p_0
    for t in range(0, T):
        u_star = f(u_t, p_t, rho, nu)
        p_t = matrix_solve(u_star, p_t, rho)
        u_t = g(u_t, p_t, rho, nu)

        u_t, p_t = NN(u_t, p_t, theta)

    return u_t, p_t

theta.requires_grad_(True)
u_T,_ = Hybrid_NS_solver(u_0, p_0, rho, nu, theta)
loss = loss_fn(u_T, u_T_true)
dtheta = torch.autograd(loss, theta)
# for learning theta (training NN)
```

Many programs can be decomposed in the following way:

Program:

Input: a vector $\boldsymbol{x} \in \mathbb{R}^n$

Function: A series of ***primitive operations*** *on the elements of* $\boldsymbol{x}$

add / multiply / trigonometric / …

Output: some transformed vector $\boldsymbol{y} \in \mathbb{R}^m$

Mathematically, the program defines a vector function $\boldsymbol{y}$: $\mathbb{R}^n \to \mathbb{R}^m$, composed of primitive operations:

$$\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{f}_N \circ, \dots, \circ\, \boldsymbol{f}_2 \circ \boldsymbol{f}_1(\boldsymbol{x})$$

Recap - autodifferentiation

Consider **any** vector function $\boldsymbol{y}$: $\mathbb{R}^n \to \mathbb{R}^m$, composed from many other vector functions

$$\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{f}_N \circ, \dots, \circ\, \boldsymbol{f}_2 \circ \boldsymbol{f}_1(\boldsymbol{x})$$

Then we can use the **multivariate chain rule** (= matrix multiplication of Jacobians) to evaluate its derivatives

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{f}_N}{\partial \boldsymbol{f}_{N-1}}, \dots, \frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{f}_1} \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}}$$

where

$$J_y \equiv \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Recap - autodifferentiation

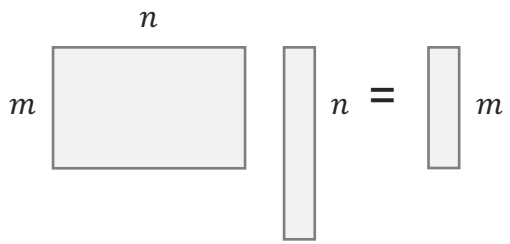
Modern autodifferentiation libraries allow us to efficiently compute:

The vector-Jacobian product (vjp):

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$$

or the Jacobian-vector product (jvp):

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \boldsymbol{v}$$



of **arbitrary** programs.

Recap - autodifferentiation

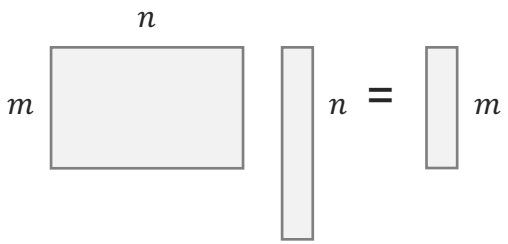
Modern autodifferentiation libraries allow us to efficiently compute:

The vector-Jacobian product (vjp):

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$$

or the Jacobian-vector product (jvp):

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \boldsymbol{v}$$



of **arbitrary** programs.

Why compute vjps / jvps, and not the full Jacobian?

Recap - autodifferentiation

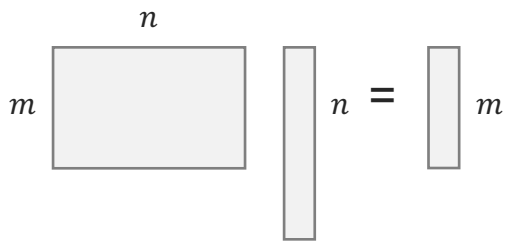
Modern autodifferentiation libraries allow us to efficiently compute:

The vector-Jacobian product (vjp):

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$$

or the Jacobian-vector product (jvp):

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \boldsymbol{v}$$



of **arbitrary** programs.

Why compute vjps / jvps, and not the full Jacobian?

1. In deep learning, vjps are usually the quantities needed when evaluating the gradient of **scalar** loss functions

$$L(\boldsymbol{\theta}): \mathbb{R}^n \to \mathbb{R}^1$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = \frac{\partial L}{\partial \boldsymbol{NN}} \frac{\partial \boldsymbol{NN}}{\partial \boldsymbol{\theta}}$$

$$\frac{\partial L}{\partial \boldsymbol{NN}} = \left(\frac{\partial L}{\partial NN_1}, \dots, \frac{\partial L}{\partial NN_k} \right)$$

Recap - autodifferentiation

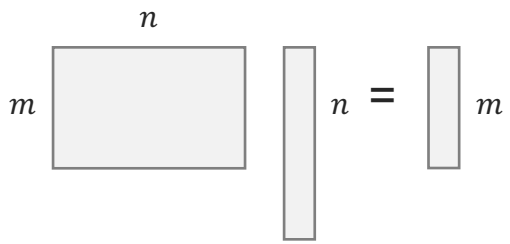
Modern autodifferentiation libraries allow us to efficiently compute:

The vector-Jacobian product (vjp):

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$$

or the Jacobian-vector product (jvp):

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \boldsymbol{v}$$



of **arbitrary** programs.

Why compute vjps / jvps, and not the full Jacobian?

1. In deep learning, vjps are usually the quantities needed when evaluating the gradient of **scalar** loss functions
2. Often evaluating vjp/jvps does not explicitly require the full Jacobian to be computed, making them **efficient** to compute

Consider

$$\boldsymbol{y} = \sin(\boldsymbol{x})$$

Then

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{pmatrix} \cos(x_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \cos(x_n) \end{pmatrix}$$

And

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = (v_1 \cos(x_1), \dots, v_n \cos(x_n))$$
$$= \boldsymbol{v} \cdot \cos(\boldsymbol{x})$$

Requires $\mathcal{O}(n)$ operations

Recap - autodifferentiation

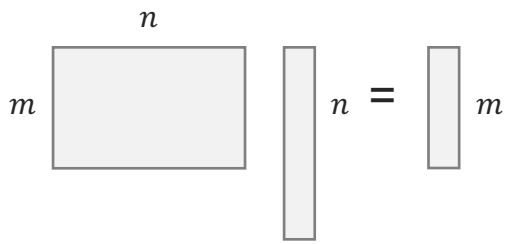
Modern autodifferentiation libraries allow us to efficiently compute:

The vector-Jacobian product (vjp):

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$$

or the Jacobian-vector product (jvp):

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \boldsymbol{v}$$



of **arbitrary** programs.

Why compute vjps / jvps, and not the full Jacobian?

1. In deep learning, vjps are usually the quantities needed when evaluating the gradient of **scalar** loss functions
2. Often evaluating vjp/jvps does not explicitly require the full Jacobian to be computed, making them **efficient** to compute
3. We can use vjps / jvps to compute the full Jacobian row by row / column by column if necessary

Let

$$\boldsymbol{v}^T = (1,0,\dots,0)$$

Then

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \left(\frac{\partial y_1}{\partial x_1}, \dots, \frac{\partial y_1}{\partial x_n}\right)$$

= First row of Jacobian

Recap - autodifferentiation

Note:

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_N}{\partial \boldsymbol{f}_{N-1}}, \dots, \frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{f}_1} \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}}$$

Then we can compute $\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by iteratively computing vector-Jacobian products, from left to right (reverse-mode):

Starting with $\boldsymbol{v}^T$,

$$\boldsymbol{v}^T \leftarrow \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_N}{\partial \boldsymbol{f}_{N-1}}$$

$$\boldsymbol{v}^T \leftarrow \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_{N-1}}{\partial \boldsymbol{f}_{N-2}}$$

$$\dots$$

$$\boldsymbol{v}^T \leftarrow \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}}$$

Recap - autodifferentiation

Note:

$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_N}{\partial \boldsymbol{f}_{N-1}}, \dots, \frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{f}_1} \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}}$$

Then we can compute $\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by iteratively computing vector-Jacobian products, from left to right (reverse-mode):

Starting with $\boldsymbol{v}^T$,

$$\boldsymbol{v}^T \leftarrow \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_N}{\partial \boldsymbol{f}_{N-1}}$$

$$\boldsymbol{v}^T \leftarrow \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_{N-1}}{\partial \boldsymbol{f}_{N-2}}$$

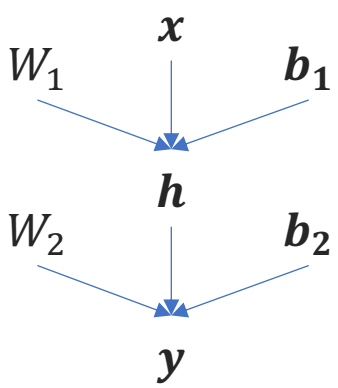
$$\dots$$

$$\boldsymbol{v}^T \leftarrow \boldsymbol{v}^T \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}}$$

- So, we only need to define the **vjp** for each **primitive operation** in order to compute $\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}$$



1) Decompose given function into its **primitive** operations

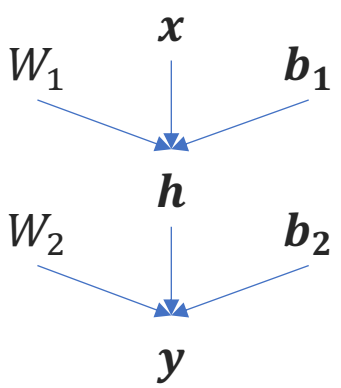
2) Build a **directed graph** of these operations

PyTorch

TensorFlow

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}$$



1) Decompose given function into its **primitive** operations

2) Build a **directed graph** of these operations

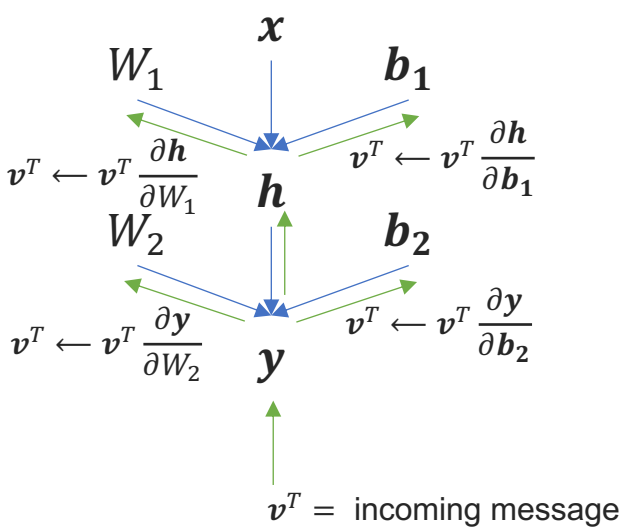
3) For each primitive operation, define
   1) Forward operation
   2) vector-Jacobian product
   3) Jacobian-vector product

PyTorch

TensorFlow

Autodiff in practice

$$\boldsymbol{y} = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}$$



1) Decompose given function into its **primitive** operations

2) Build a **directed graph** of these operations

3) For each primitive operation, define
   1) Forward operation
   2) vector-Jacobian product
   3) Jacobian-vector product

4) Evaluate the vjp or jvp of the function by applying the **chain rule (=message passing)** through the graph
   1) Forwards for jvp
   2) Backwards for vjp

PyTorch

TensorFlow

Differentiating through an Euler solver

Consider

$$\frac{d\boldsymbol{y}}{dx} = \boldsymbol{f}(x)$$

Solve using Euler method:

Given $\boldsymbol{y}_0, \boldsymbol{x}_0, \delta t$:

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + \delta x \boldsymbol{f}(x_i)$$
$$x_{i+1} = x_i + \delta x$$

Differentiating through an Euler solver

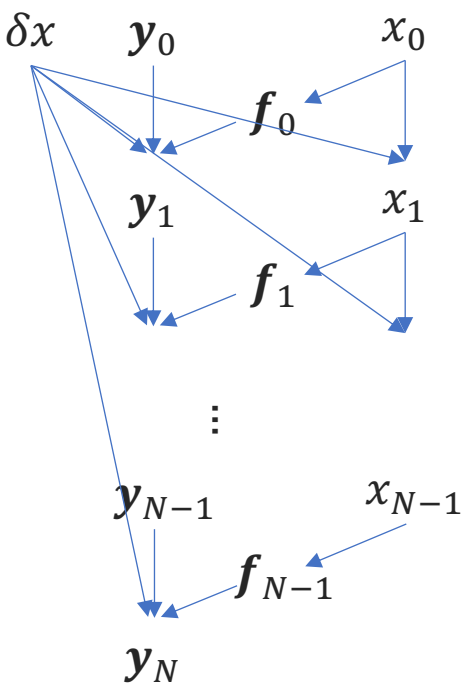
Consider

$$\frac{d\boldsymbol{y}}{dx} = \boldsymbol{f}(x)$$

Solve using Euler method:

Given $\boldsymbol{y}_0, \boldsymbol{x}_0, \delta t$:

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + \delta x \boldsymbol{f}(x_i)$$
$$x_{i+1} = x_i + \delta x$$



Differentiating through an Euler solver

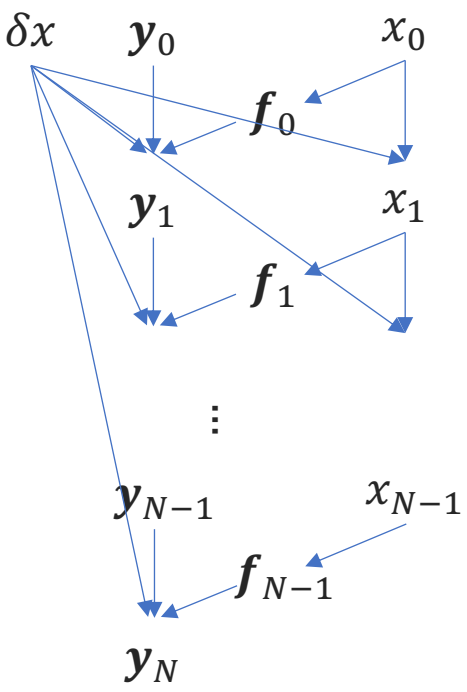
Consider

$$\frac{d\boldsymbol{y}}{dx} = \boldsymbol{f}(x)$$

Solve using Euler method:

Given $\boldsymbol{y}_0, \boldsymbol{x}_0, \delta t$:

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_{\boldsymbol{i}} + \delta x \boldsymbol{f}(x_i)$$
$$x_{i+1} = x_i + \delta x$$



We can differentiate through this program in the same way!

And, for example, learn $\boldsymbol{y}_0$ given observations of $\boldsymbol{y}_N$

Summary

- PINNs / operator learning entirely replace traditional algorithms with DNNs
- Hybrid approaches instead insert ML **inside** key **parts** of traditional algorithms
- Autodifferentiation is **the** key **enabler** for SciML
  - Allows hybrid approaches to be trained end-to-end
  - Can differentiate through **arbitrary** programs
  - This is an incredibly general and powerful idea