Introduction to Hybrid Workflows – Part 1

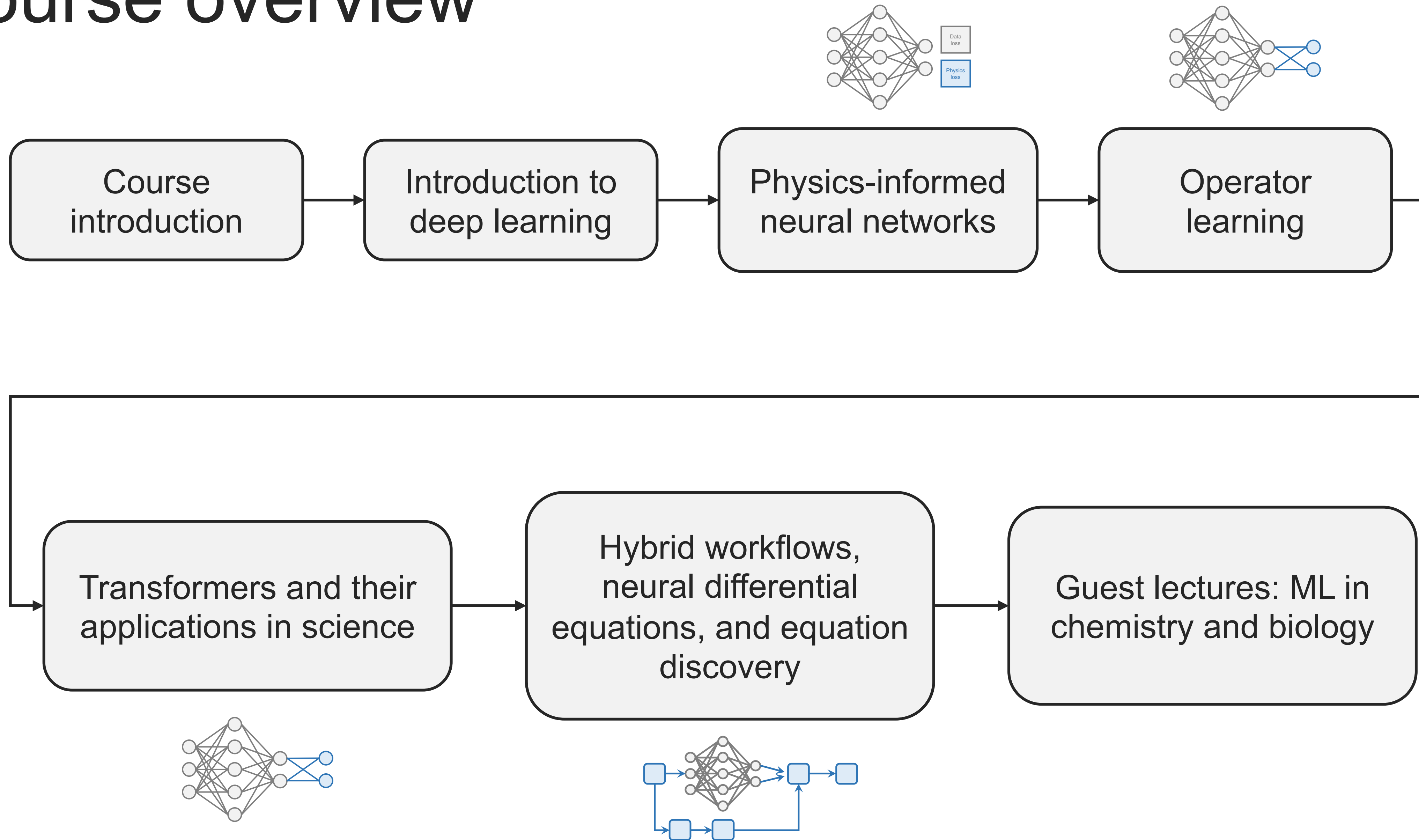
Spring Semester 2024

Siddhartha Mishra
Ben Moseley
ETH zürich

Course timeline

Tutorials		Lectures			
Mon 12:15-14:00 HG E 5		*Wed 08:15-10:00 ML H 44*		*Fri 12:15-13:00 ML H 44*	
19.02.		21.02.	Course introduction	23.02.	Introduction to deep learning I
26.02.	Introduction to PyTorch	28.02.	Introduction to deep learning II	01.03.	Introduction to PDEs
04.03.	Simple DNNs in PyTorch	06.03.	Physics-informed neural networks – introduction	08.03.	Physics-informed neural networks - limitations
11.03.	Implementing PINNs I	13.03.	Physics-informed neural networks – extensions	15.03.	Physics-informed neural networks – theory I
18.03.	Implementing PINNs II	20.03.	Physics-informed neural networks – theory II	22.03.	Supervised learning for PDEs I
25.03.	Operator learning I	27.03.	Supervised learning for PDEs II	29.03.	
01.04.		03.04.		05.04.	
08.04.	Operator learning II	10.04.	Introduction to operator learning I	12.04.	Introduction to operator learning II
15.04.		17.04.	Convolutional neural operators	19.04.	Time-dependent neural operators
22.04.	GNNs	24.04.	Large-scale neural operators	26.04.	Attention as a neural operator
29.04.	Transformers	01.05.		03.05.	Windowed attention and scaling laws
06.05.	Diffusion models	08.05.	**Introduction to hybrid workflows I**	10.05.	Introduction to hybrid workflows II
13.05.	Coding autodiff from scratch	15.05.	Neural differential equations	17.05.	Introduction to JAX
20.05.		22.05.	Symbolic regression and model discovery	24.05.	Course summary
27.05.	Intro to JAX / Neural ODEs	29.05.	Guest lecture: AlphaFold	31.05.	Guest lecture: AlphaFold

Course overview



Lecture overview

- Limitations of SciML approaches studied so far
- Hybrid SciML approaches
  - Residual modelling
  - Opening the “black-box”
  - How to train hybrid approaches
- Autodifferentiation
  - Autodifferentiation as a key enabler
  - What it is and how it works

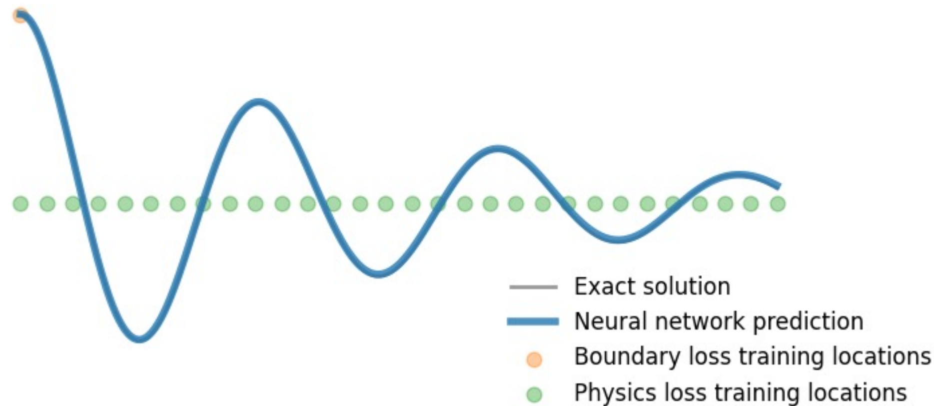
Lecture overview

- Limitations of SciML approaches studied so far
- Hybrid SciML approaches
  - Residual modelling
  - Opening the “black-box”
  - How to train hybrid approaches
- Autodifferentiation
  - Autodifferentiation as a key enabler
  - What it is and how it works

Learning objectives

- Be able to describe what a hybrid workflow is
- Understand how autodifferentiation is used to train hybrid workflows
- Understand how autodifferentiation works

Course recap - PINNs



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

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

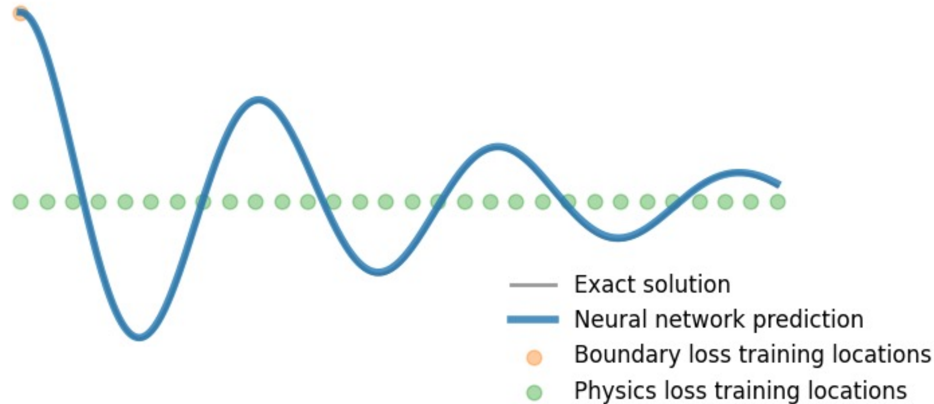
Boundary loss $L_b(\theta)$

$$L(\theta) = \lambda_1(NN(t=0;\theta) - 1)^2 + \lambda_2\left(\frac{dNN}{dt}(t=0;\theta) - 0\right)^2$$

Physics loss $L_p(\theta)$

$$+ \frac{1}{N_p}\sum_{i}^{N_p}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right]NN(t_i;\theta)\right)^2$$

Course recap - PINNs



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

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

Boundary loss $L_b(\theta)$

$$L(\theta) = \lambda_1(NN(t=0;\theta) - 1)^2 + \lambda_2\left(\frac{dNN}{dt}(t=0;\theta) - 0\right)^2$$

Physics loss $L_p(\theta)$

$$+ \frac{1}{N_p}\sum_{i}^{N_p}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right]NN(t_i;\theta)\right)^2$$

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

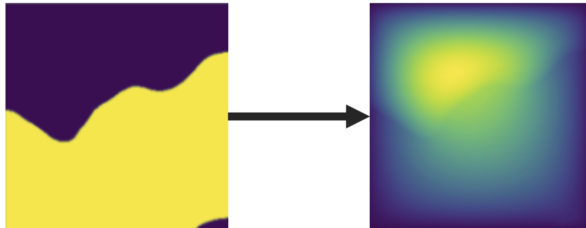
Limitations of PINNs

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

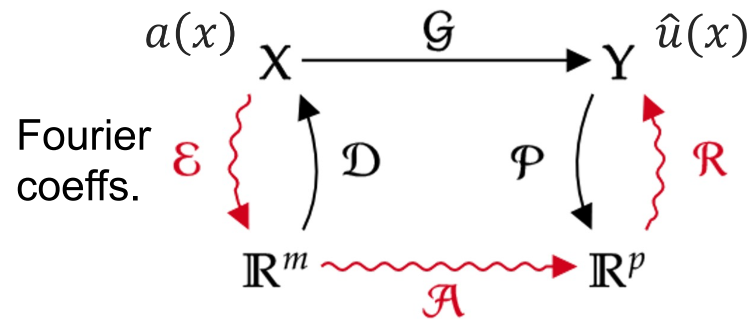
Course recap – Operator learning

Darcy PDE

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



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



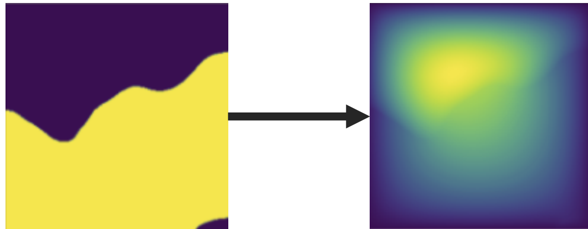
$$\overbrace{a(x) \to \{a_k\}_{k=1}^{m} \to NN(\{a_k\};\theta) \to \{u_k\}_{k=1}^{p}}^{\mathcal{G}_\theta^*[a]} \to \hat{u}(x)$$

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

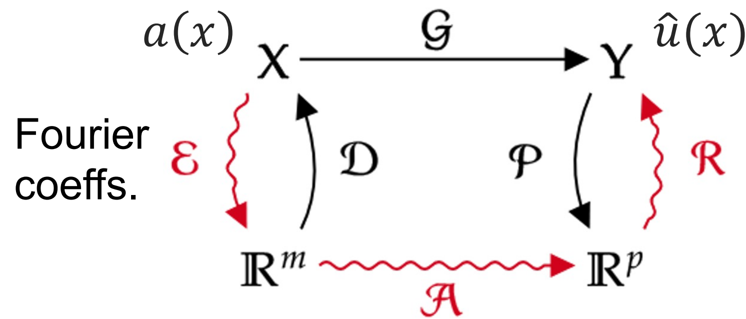
Course recap – Operator learning

Darcy PDE

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



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



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 $u(x)$

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

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

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

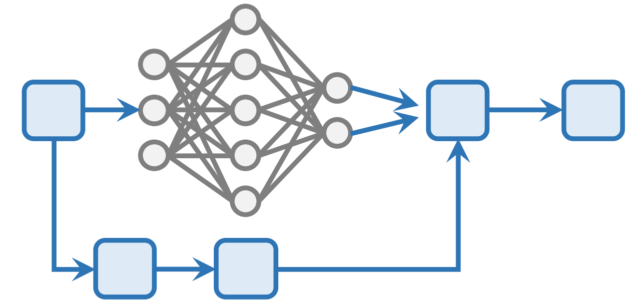
Use DNNs to:

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

Entirely replacing your existing workflow with a DNN may **not** be a good idea!

Hybrid SciML approaches

What if we **directly incorporate** DNNs into a traditional algorithm instead?
= hybrid approach



General advice

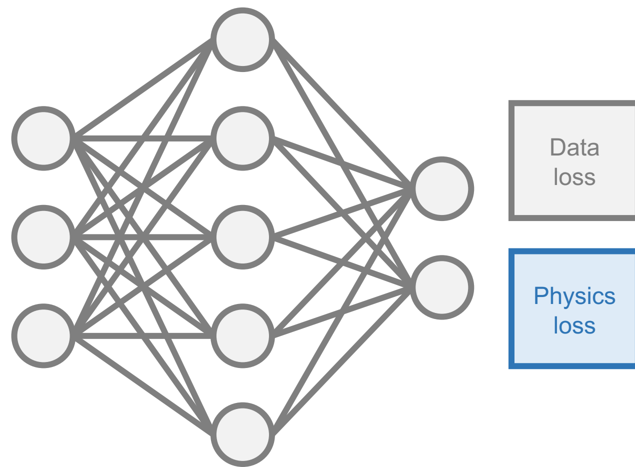
Use DNNs to:

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

Entirely replacing your existing workflow with a DNN may **not** be a good idea!

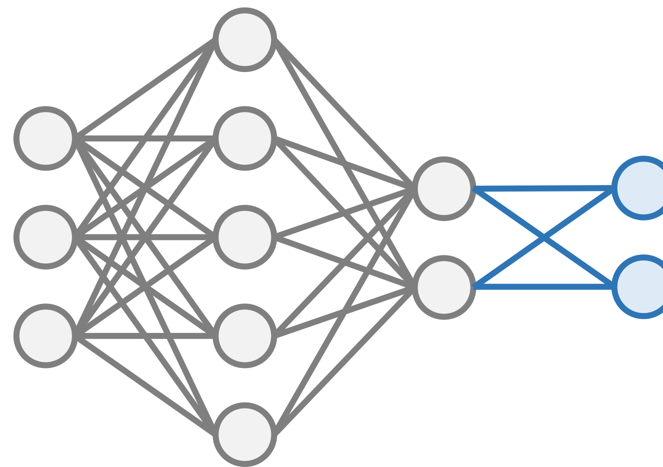
Ways to incorporate scientific principles into machine learning

Loss function



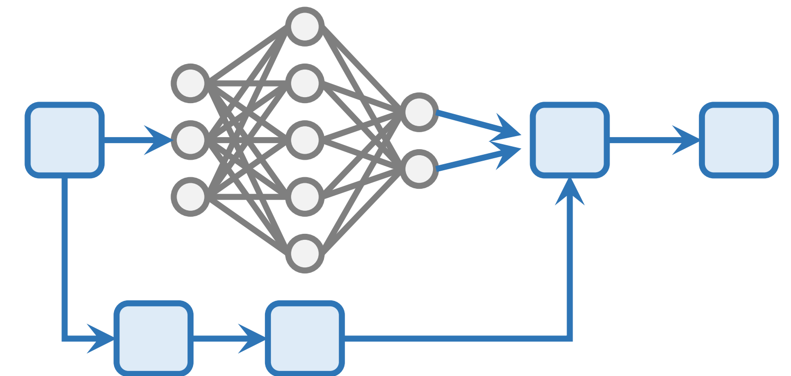
Example:
Physics-informed neural networks
(add governing equations to loss function)

Architecture



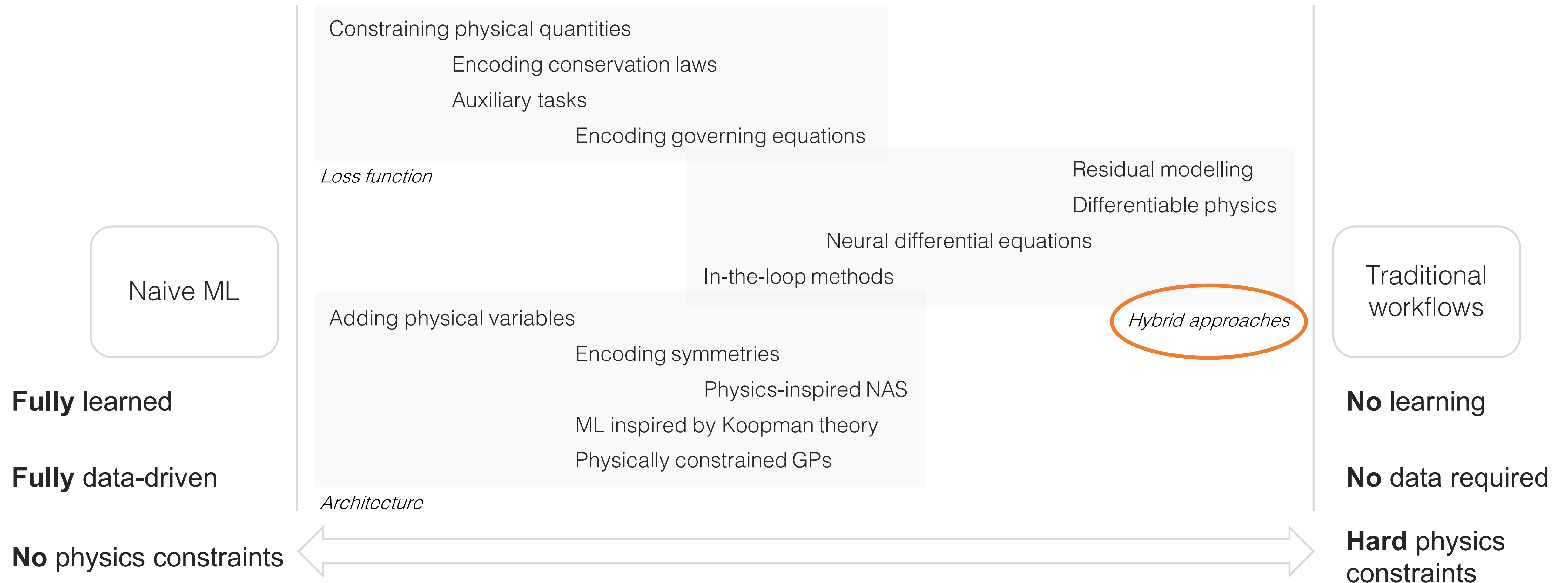
Example:
Encoding symmetries / conservation laws
(e.g. energy conservation, rotational invariance), operator learning

Hybrid approaches



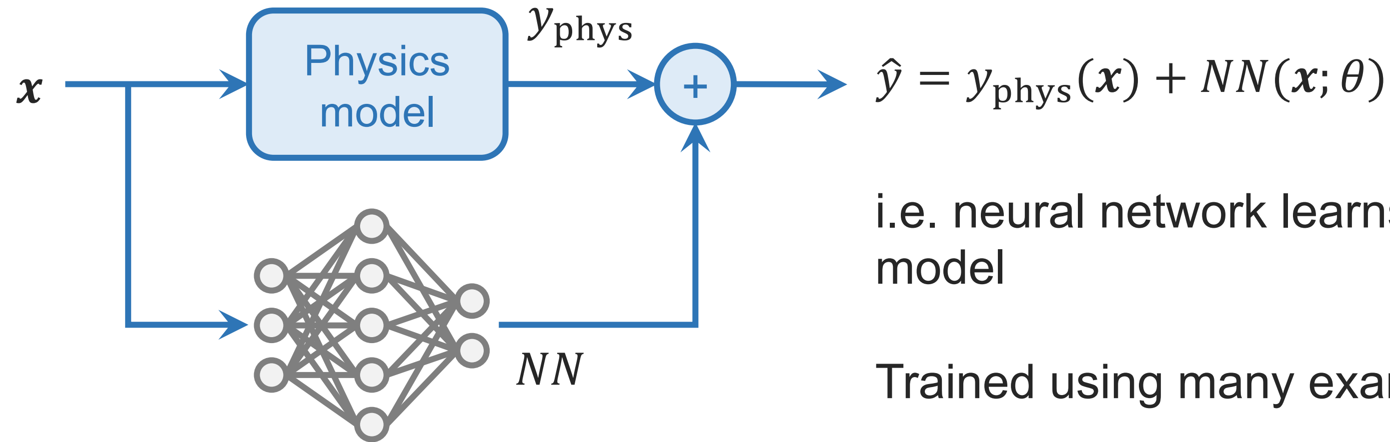
Example:
Neural differential equations
(incorporating neural networks into PDE models)

A plethora of SciML techniques



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

A simple hybrid approach – residual modelling



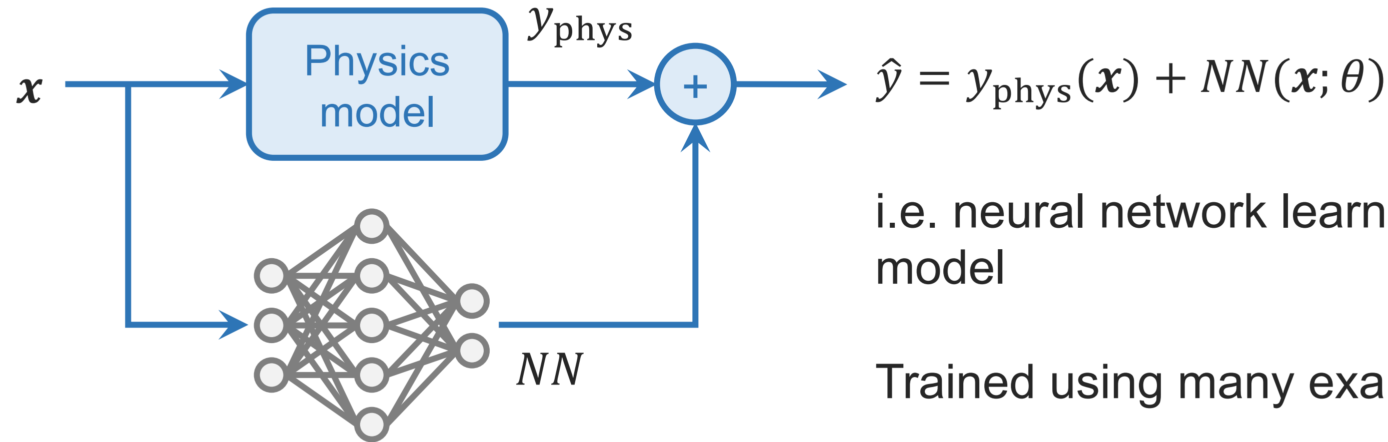
$\hat{y} = y_{\text{phys}}(\boldsymbol{x}) + NN(\boldsymbol{x}; \theta)$

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

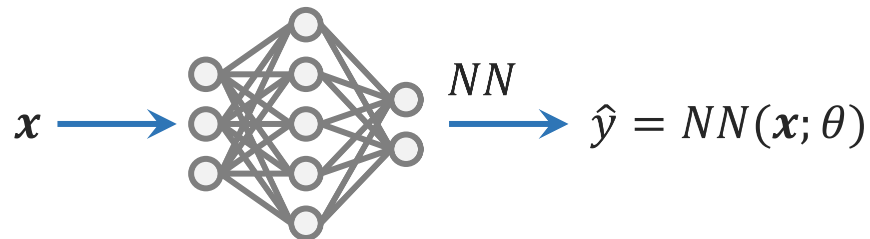


$\hat{y} = y_{\mathrm{phys}}(\boldsymbol{x}) + NN(\boldsymbol{x}; \theta)$

i.e. neural network learns **residual** correction to physics model

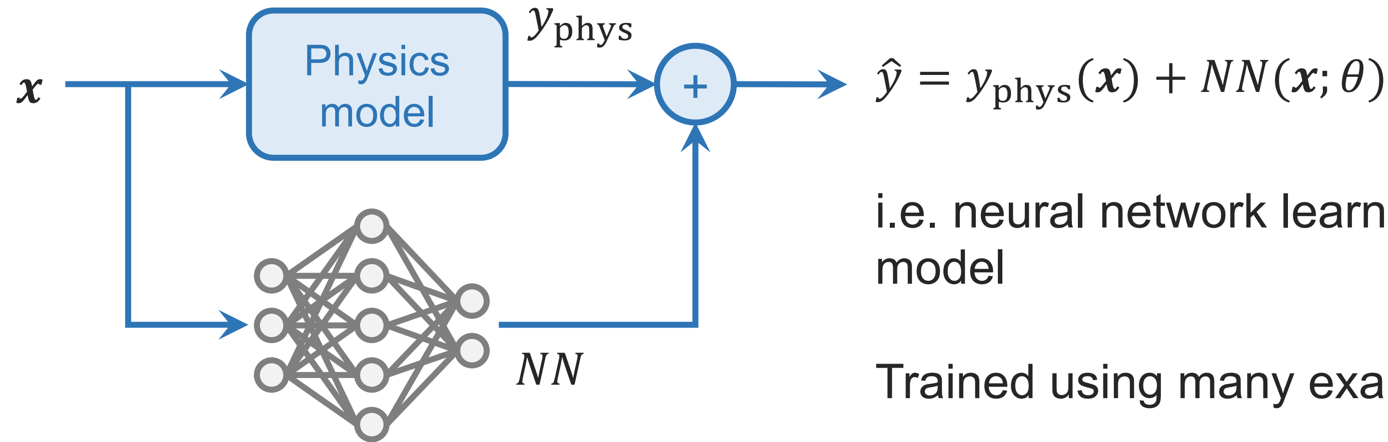
Trained using many examples of inputs/outputs

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

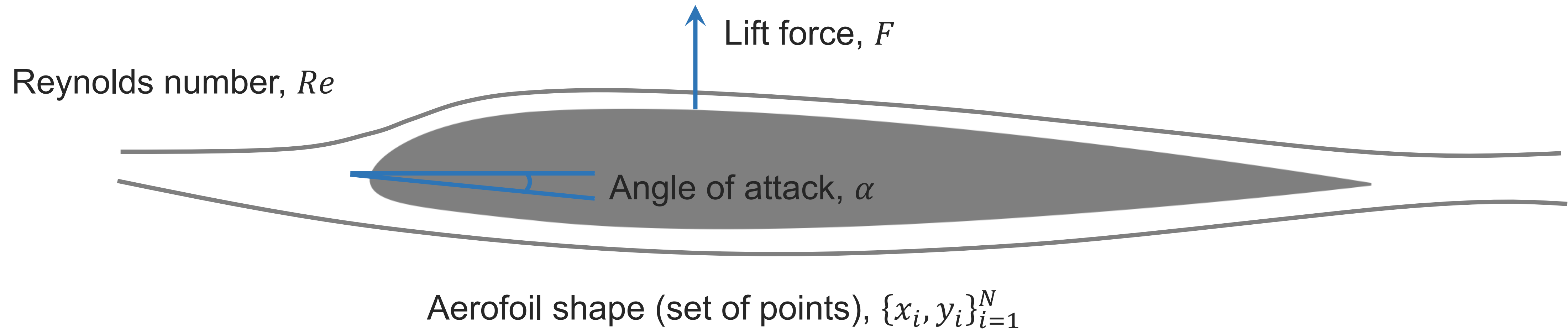
$$L(\theta) = \sum_i^N (\hat{y}(\boldsymbol{x}_i;\theta) - y_i)^2$$

$$= \sum_i^N \left(NN(\boldsymbol{x}_i;\theta) - [y_i - y_{\text{phys}}(\boldsymbol{x}_i)]\right)^2$$

$$\equiv \sum_i^N (NN(\boldsymbol{x}_i;\theta) - r(\boldsymbol{x}_i))^2$$

Note: can precompute $r(\boldsymbol{x}_i)$ in advance

Residual modelling – aerofoil example



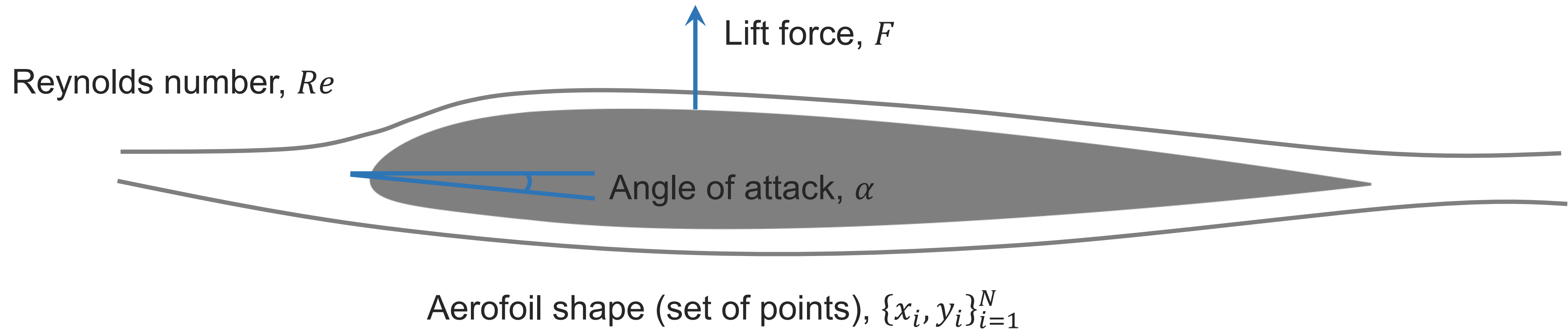
Simulation task:

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



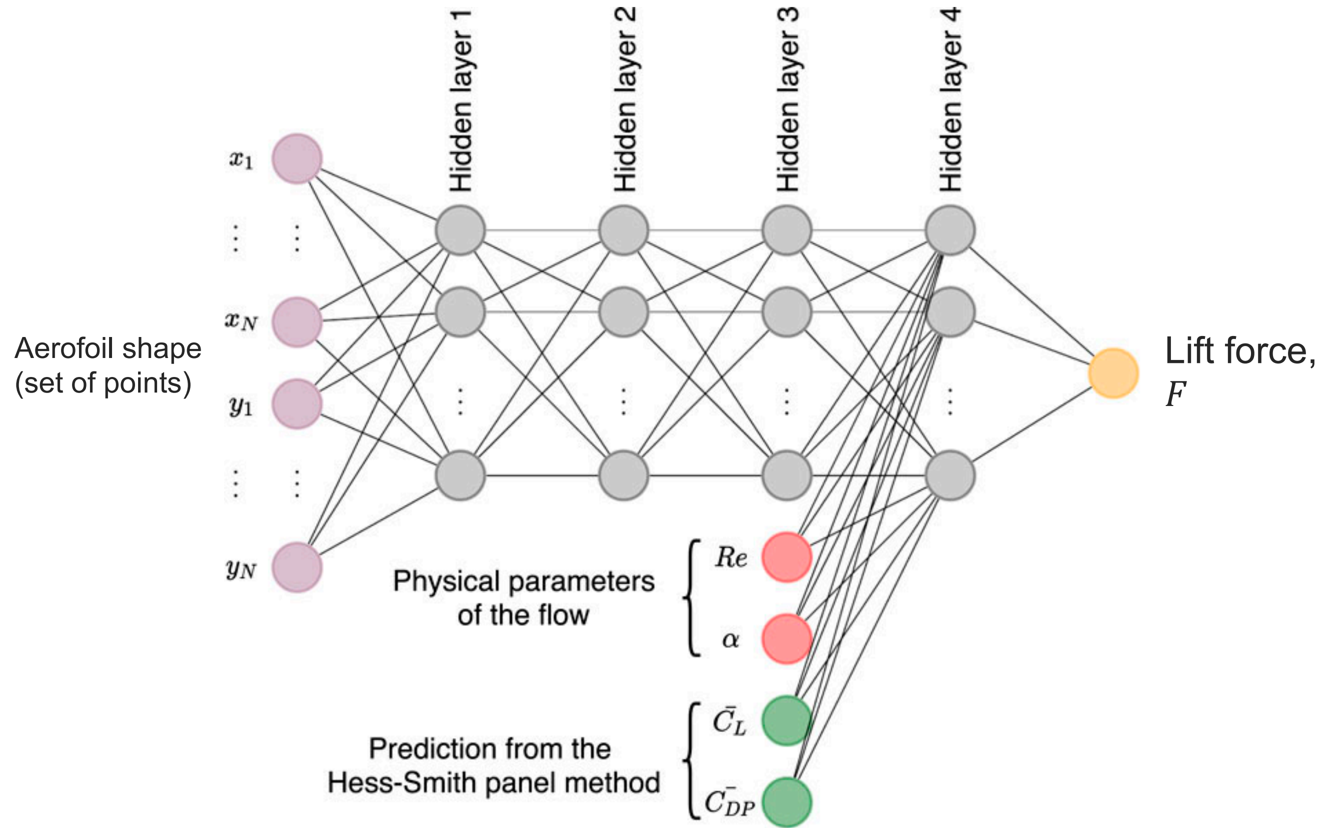
- Full CFD simulations are typically accurate, but very expensive
- Faster approximate methods exist, but are usually less accurate

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



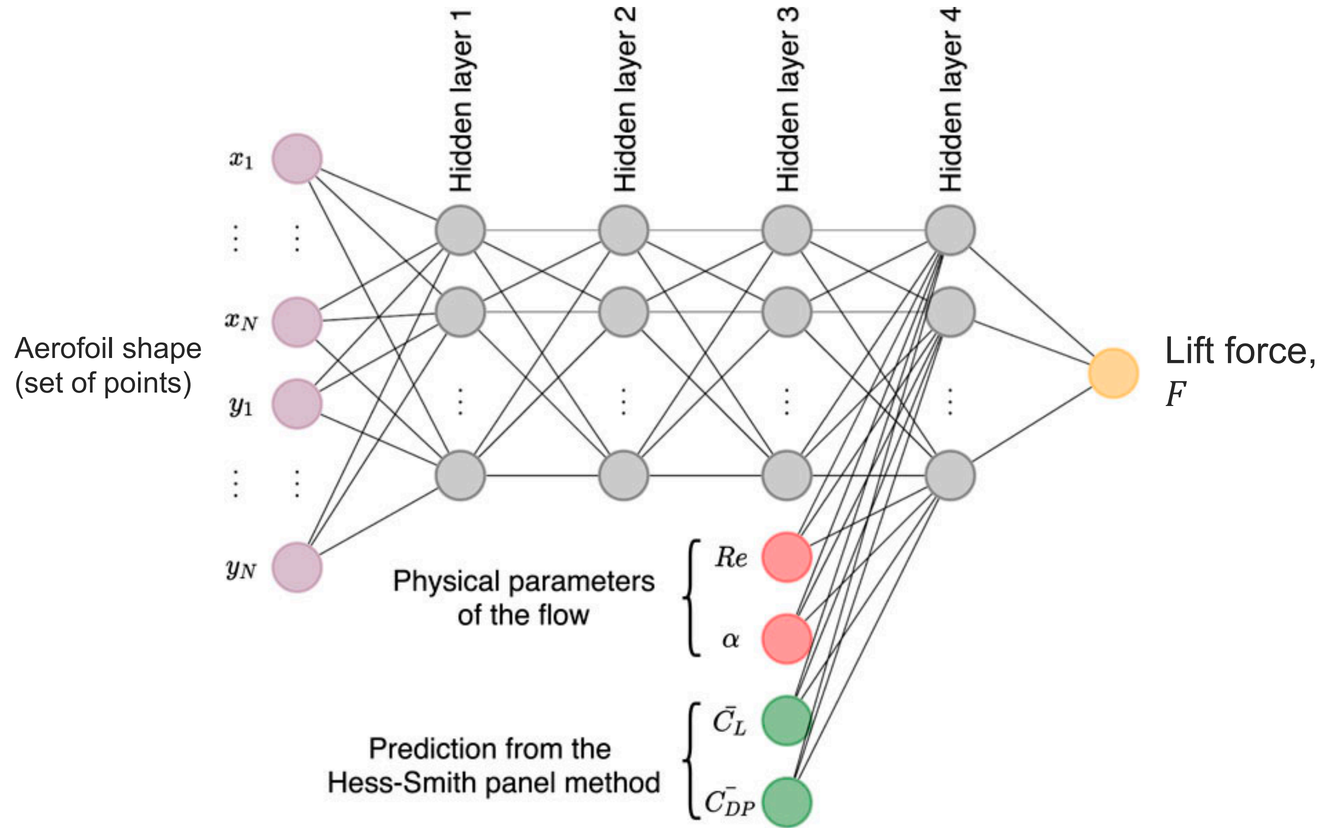
Hess-Smith panel method:
Fast **approximate** method for predicting lift force

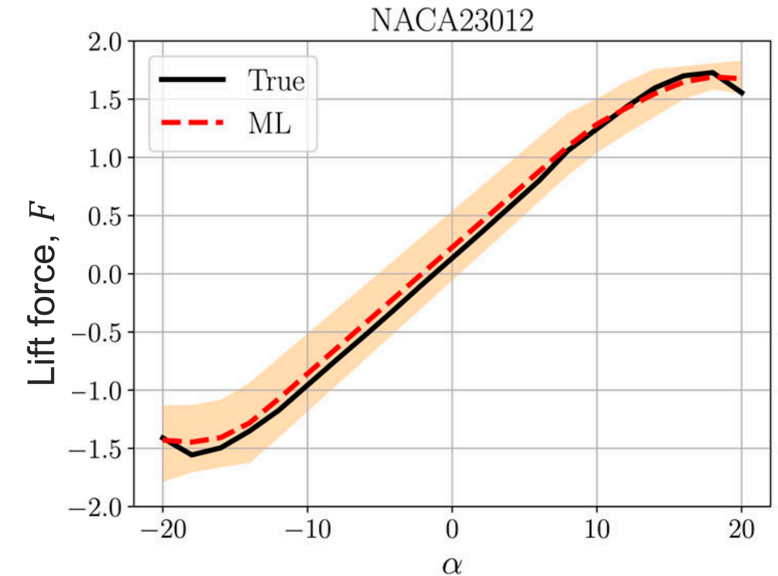
Training data:
Many example inputs/outputs generated from (expensive) **high-fidelity** CFD modelling

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

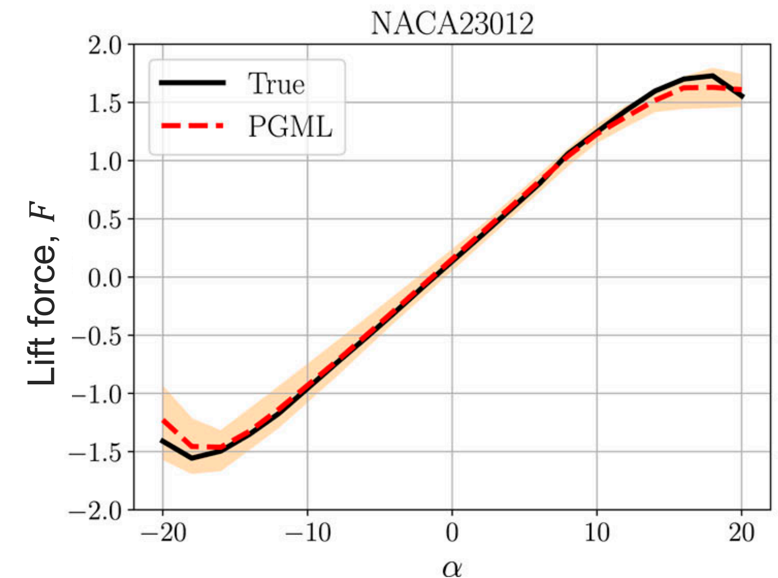
Pawar et al, Physics guided machine learning using simplified theories, Physics of Fluids (2021)

Residual modelling – aerofoil example





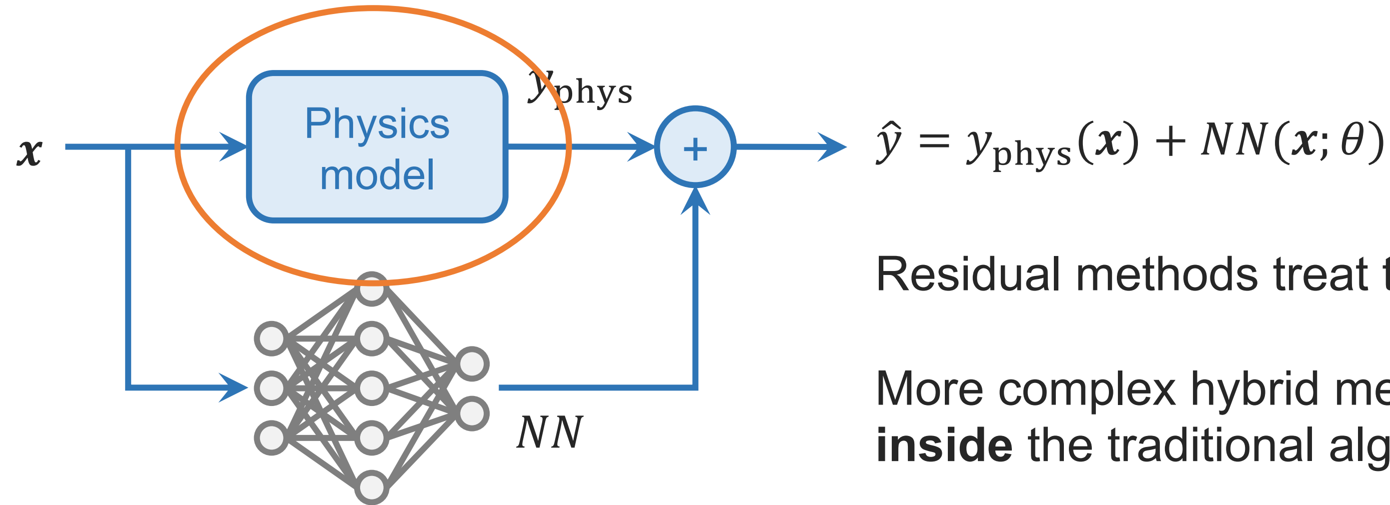
Naive NN (no physics inputs)



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**”

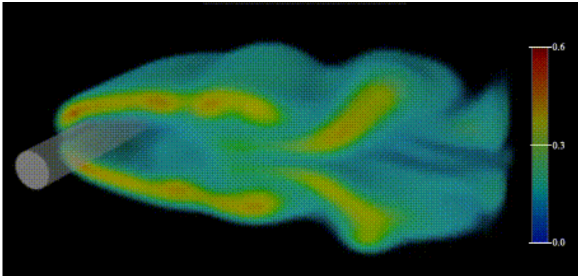
More complex hybrid methods open the box and insert ML **inside** the traditional algorithm

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

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

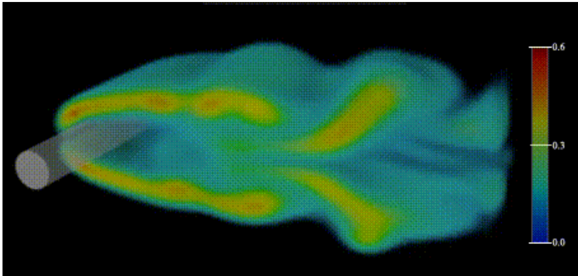
$$\nabla \cdot \boldsymbol{u} = 0$$

$\boldsymbol{u}(\boldsymbol{x}, t)$ 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 $\nabla \cdot \boldsymbol{u}_{t+1} = 0 \Rightarrow$

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

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

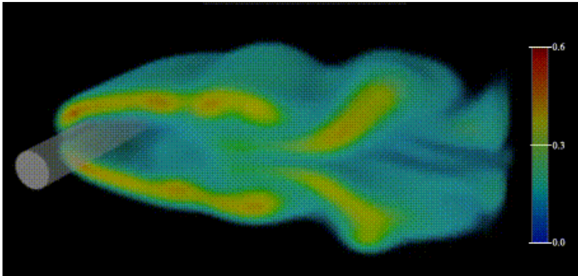
Discretise in space

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

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:

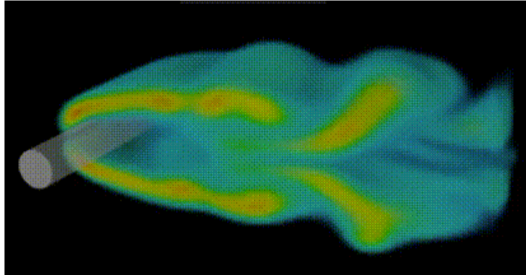
1. Compute $\boldsymbol{u}^*_{i,j,k}$ using (2)
2. Solve matrix equation (3) for $p_{i,j,k,t+1}$
3. Compute $\boldsymbol{u}_{i,j,k,t+1}$ using (1)

```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)

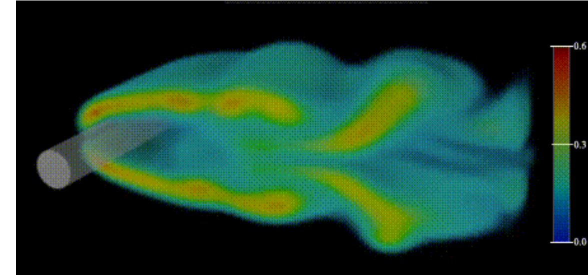
Computational cost / accuracy trade-off

Low fidelity FD solver



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

High fidelity FD solver



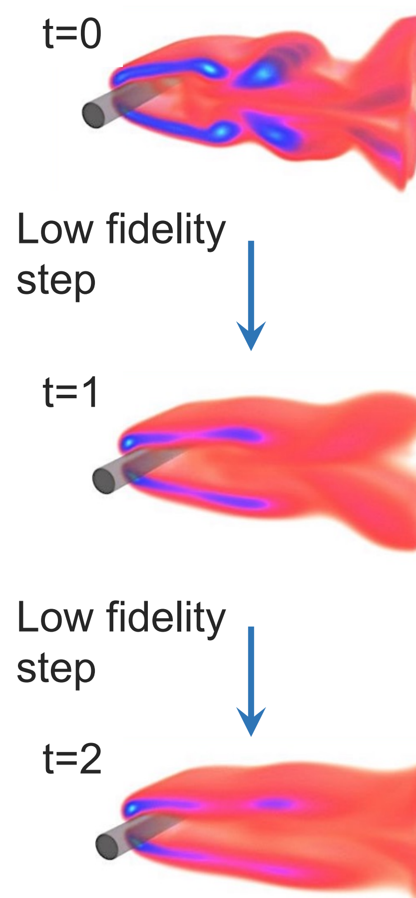
(128 x 128 x 256) cells
~1000 seconds / 100 timesteps

- 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

```
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
```

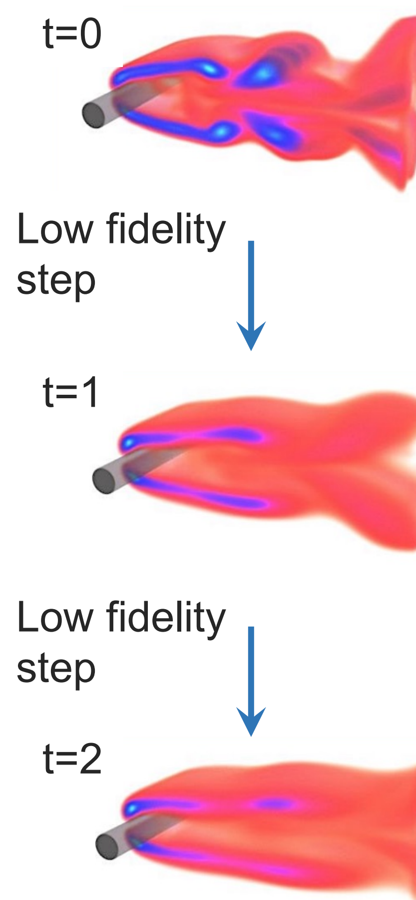


Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Traditional Navier-Stokes solver

```
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
```

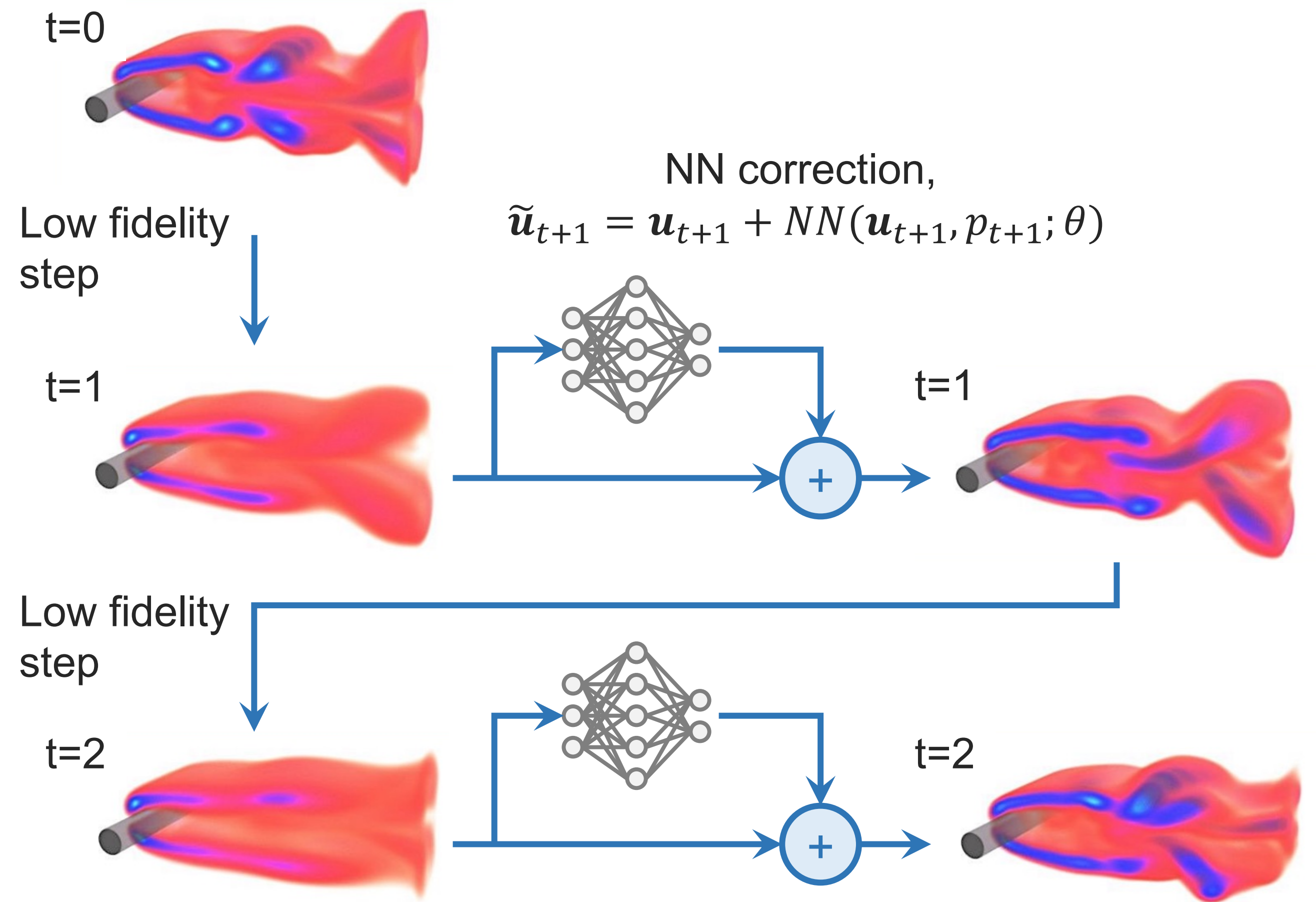
- Where could we insert ML **inside** this workflow to improve accuracy / efficiency?



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

Hybrid Navier-Stokes solver

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

    return u_t, p_t
```

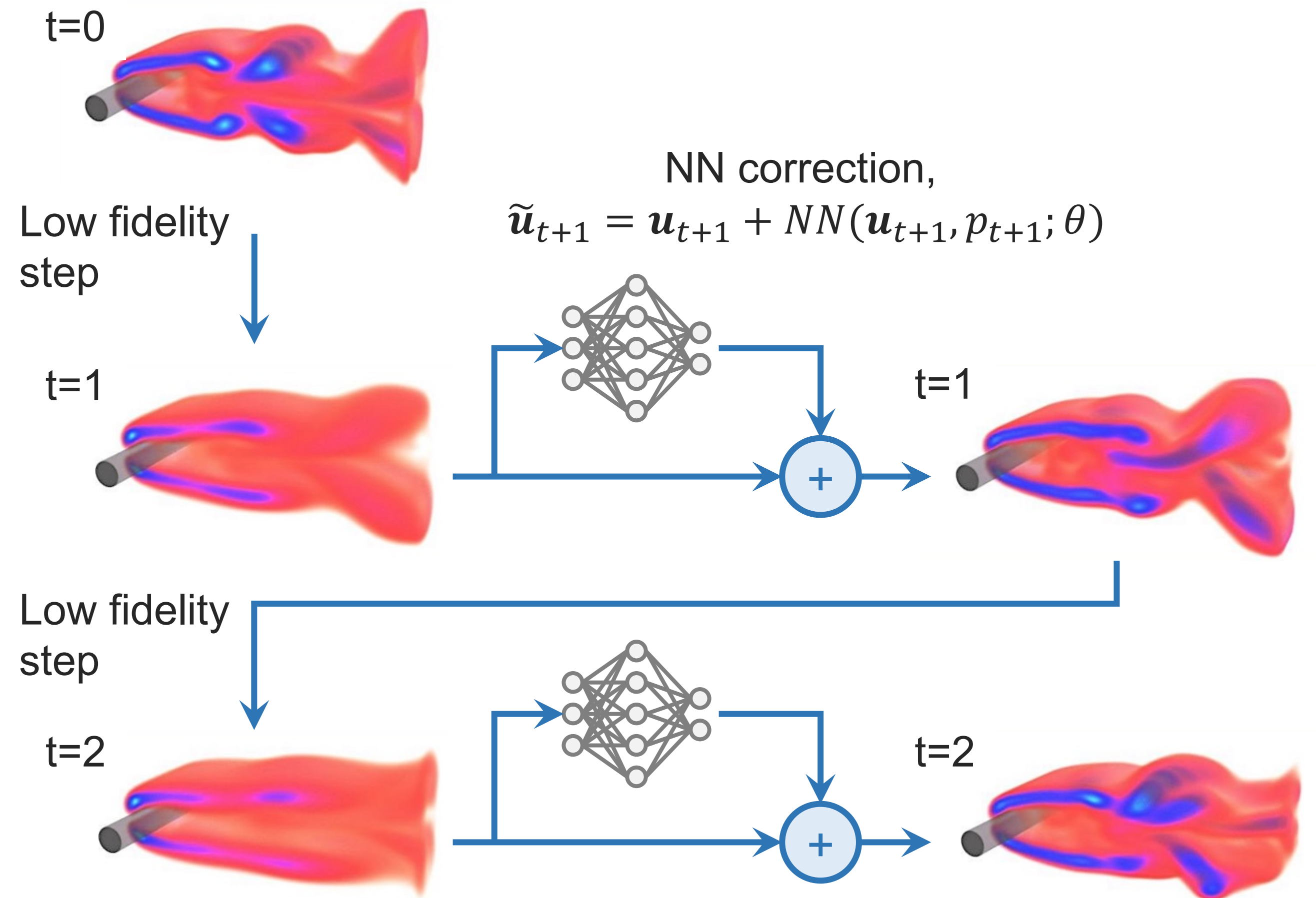


Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Hybrid Navier-Stokes solver

- How can we train $NN(\boldsymbol{u}_{t+1}, p_{t+1}; \theta)$?
- What training data do we need? (Hint: what inputs/labels do we need to train the network?)
- What loss function should we use?

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

    return u_t, p_t
```

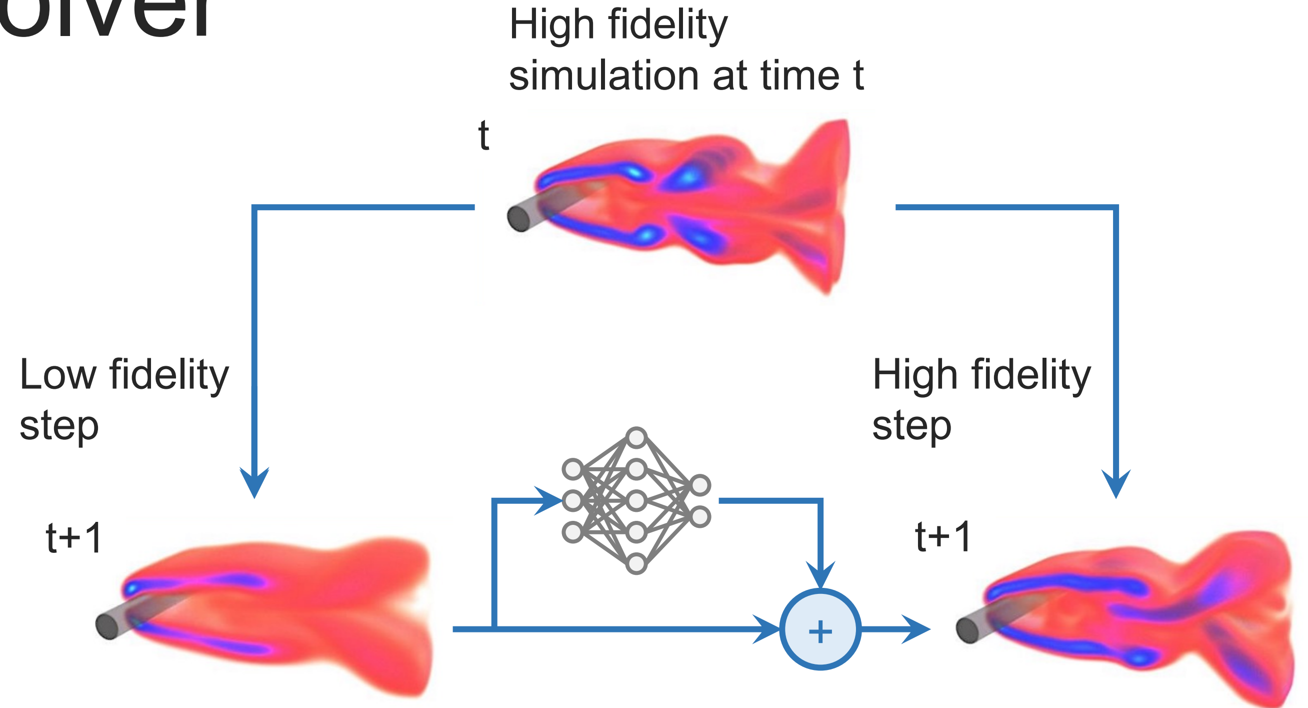


Um et al, Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers, NeurIPS (2020)

Hybrid Navier-Stokes solver

- How can we train $NN(\boldsymbol{u}_{t+1}, p_{t+1}; \theta)$?

Option 1: use pairs of low fidelity / high fidelity timesteps as training data



$$
\begin{aligned}
L(\theta) &= \sum_{t}^{N} \left\| \boldsymbol{u}_{t+1}^{L} + NN(\boldsymbol{u}_{t+1}^{L}, p_{t+1}^{L}; \theta) - \boldsymbol{u}_{t+1}^{H} \right\|^{2} \\
&= \sum_{t}^{N} \left\| NN(\boldsymbol{u}_{t+1}^{L}, p_{t+1}^{L}; \theta) - (\boldsymbol{u}_{t+1}^{H} - \boldsymbol{u}_{t+1}^{L}) \right\|^{2}
\end{aligned}
$$

Note: can precompute residual in advance

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

Hybrid Navier-Stokes solver

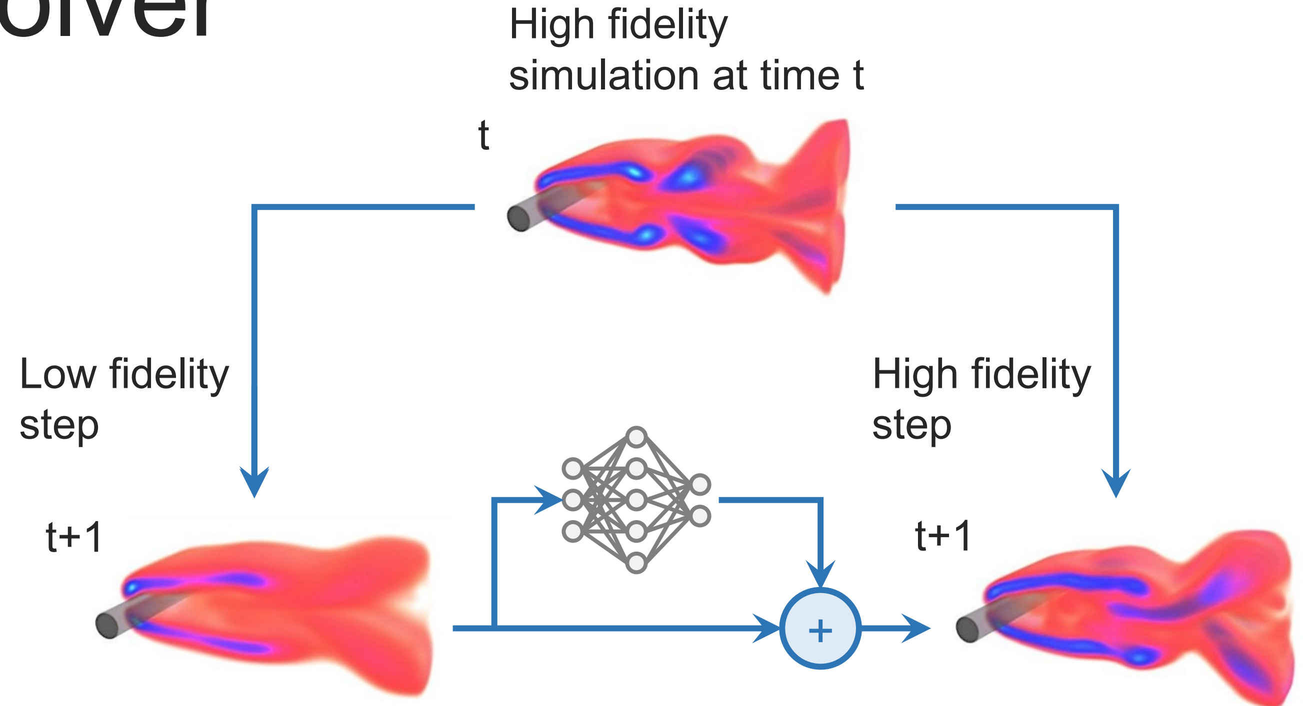
- How can we train $NN(\boldsymbol{u}_{t+1}, p_{t+1}; \theta)$?

Option 1: use pairs of low fidelity / high fidelity timesteps as training data

During training, neural network only sees **exact** low fidelity timesteps as input

Problem:

- But during inference, neural network sees **different** inputs (low fidelity timesteps + previous NN corrections) ⚠
- Leads to a train/test distribution shift, and error accumulation over time



$$L(\theta) = \sum_{t}^{N} \left\| \boldsymbol{u}_{t+1}^{L} + NN(\boldsymbol{u}_{t+1}^{L}, p_{t+1}^{L}; \theta) - \boldsymbol{u}_{t+1}^{H} \right\|^2$$

$$= \sum_{t}^{N} \left\| NN(\boldsymbol{u}_{t+1}^{L}, p_{t+1}^{L}; \theta) - (\boldsymbol{u}_{t+1}^{H} - \boldsymbol{u}_{t+1}^{L}) \right\|^2$$

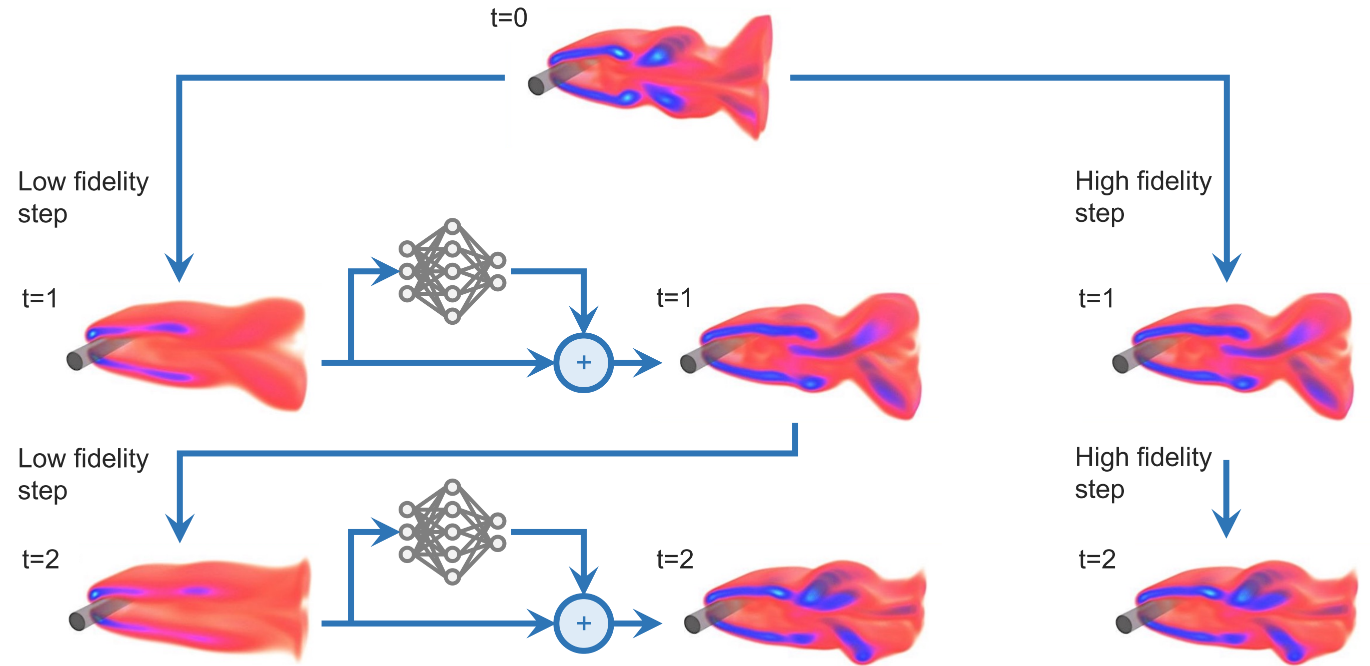
Note: can precompute residual in advance

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

Hybrid Navier-Stokes solver

- How can we train $NN(\boldsymbol{u}_{t+1}, p_{t+1}; \theta)$?

Option 2: match outputs of hybrid solver to high-fidelity simulation directly



$$L(\theta) = \sum_{i}^{N} \sum_{t}^{T} \left\| HybridSolver_t(\boldsymbol{u}_{0_i}; \theta) - \boldsymbol{u}_t^H(\boldsymbol{u}_{0_i}) \right\|^2$$

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

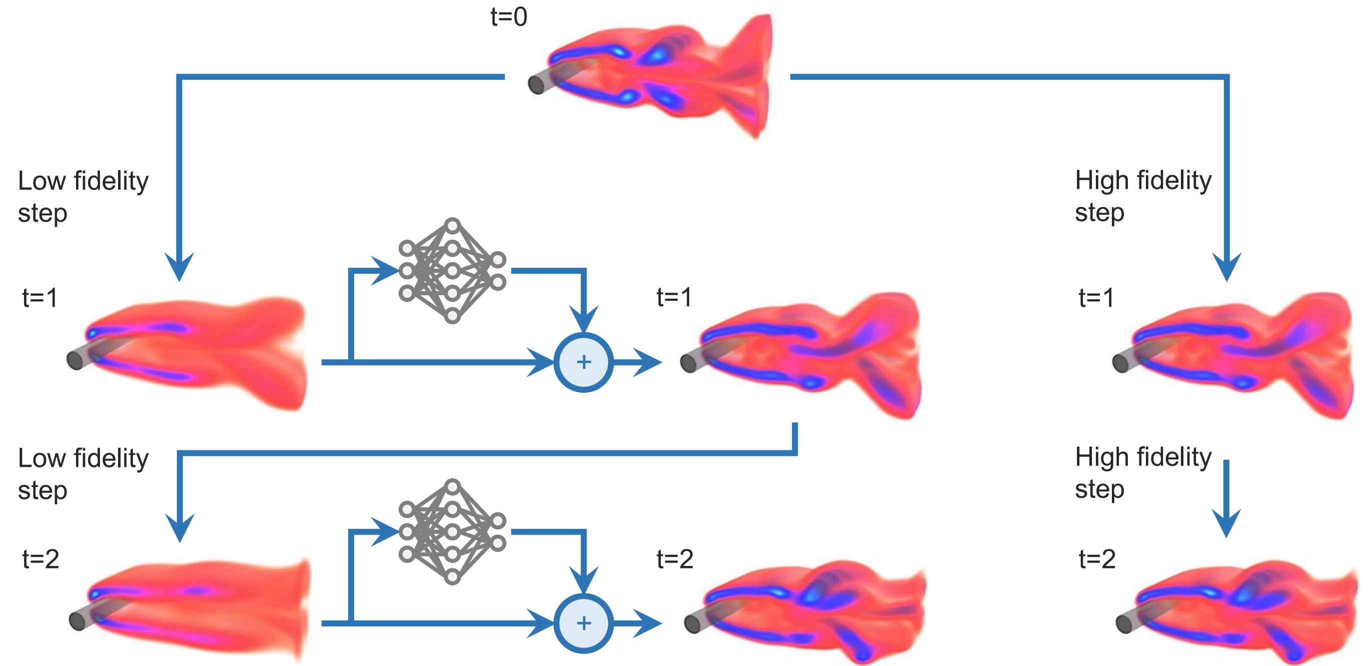
Hybrid Navier-Stokes solver

- How can we train $NN(\boldsymbol{u}_{t+1}, p_{t+1}; \theta)$?

Option 2: match outputs of hybrid solver to high-fidelity simulation directly

NN learns to correct its previous errors ✓
Reduces distributional shift ✓

Requires $HybridSolver$ to be **differentiable**!



$$L(\theta) = \sum_{i}^{N} \sum_{t}^{T} \left\| HybridSolver_t\left(\boldsymbol{u}_{0_i}; \theta\right) - \boldsymbol{u}_t^H\left(\boldsymbol{u}_{0_i}\right) \right\|^2$$

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

Hybrid Navier-Stokes solver

- How can we train $NN(\boldsymbol{u}_{t+1}, p_{t+1}; \theta)$?

Option 2: match outputs of hybrid solver to high-fidelity simulation directly

NN learns to correct its previous errors ✓
Reduces distributional shift ✓

Requires $HybridSolver$ to be **differentiable**!

.. we can just use autodifferentiation!

```
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 = (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.grad(loss, theta)
# for learning theta (training NN)
```

$$L(\theta) = \sum_{i}^{N} \sum_{t}^{T} \left\| HybridSolver_t(\boldsymbol{u}_{0_i}; \theta) - \boldsymbol{u}_t^H(\boldsymbol{u}_{0_i}) \right\|^2$$

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 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.grad(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 = (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.grad(loss, theta)
# for learning theta (training NN)
```

$$L(\theta) = \sum_{i}^{N} \sum_{t}^{T} \left\| HybridSolver_t\left(\boldsymbol{u}_{0_i}; \theta\right) - \boldsymbol{u}_t^H(\boldsymbol{u}_{0_i}) \right\|^2$$

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?

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

We train neural networks using **autodifferentiation**

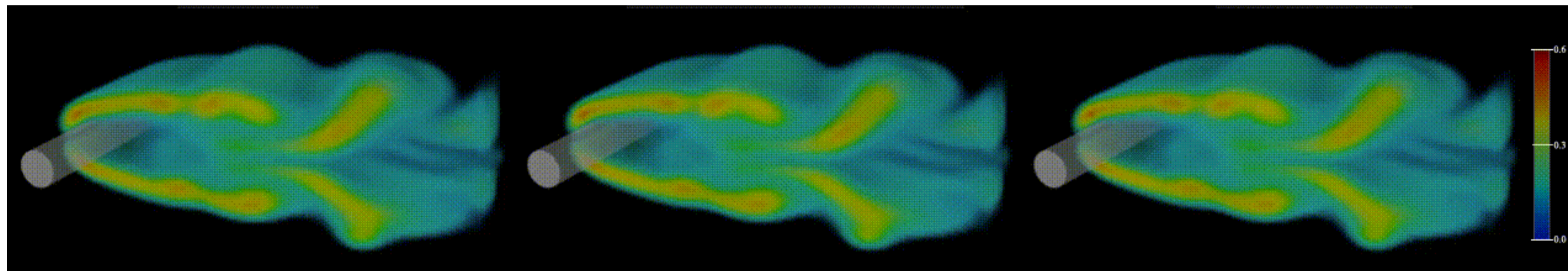
But autodifferentiation = exact gradients of **arbitrary** programs

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

Differentiable physics = using autodifferentiation to differentiate physical algorithms

NS solver results

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



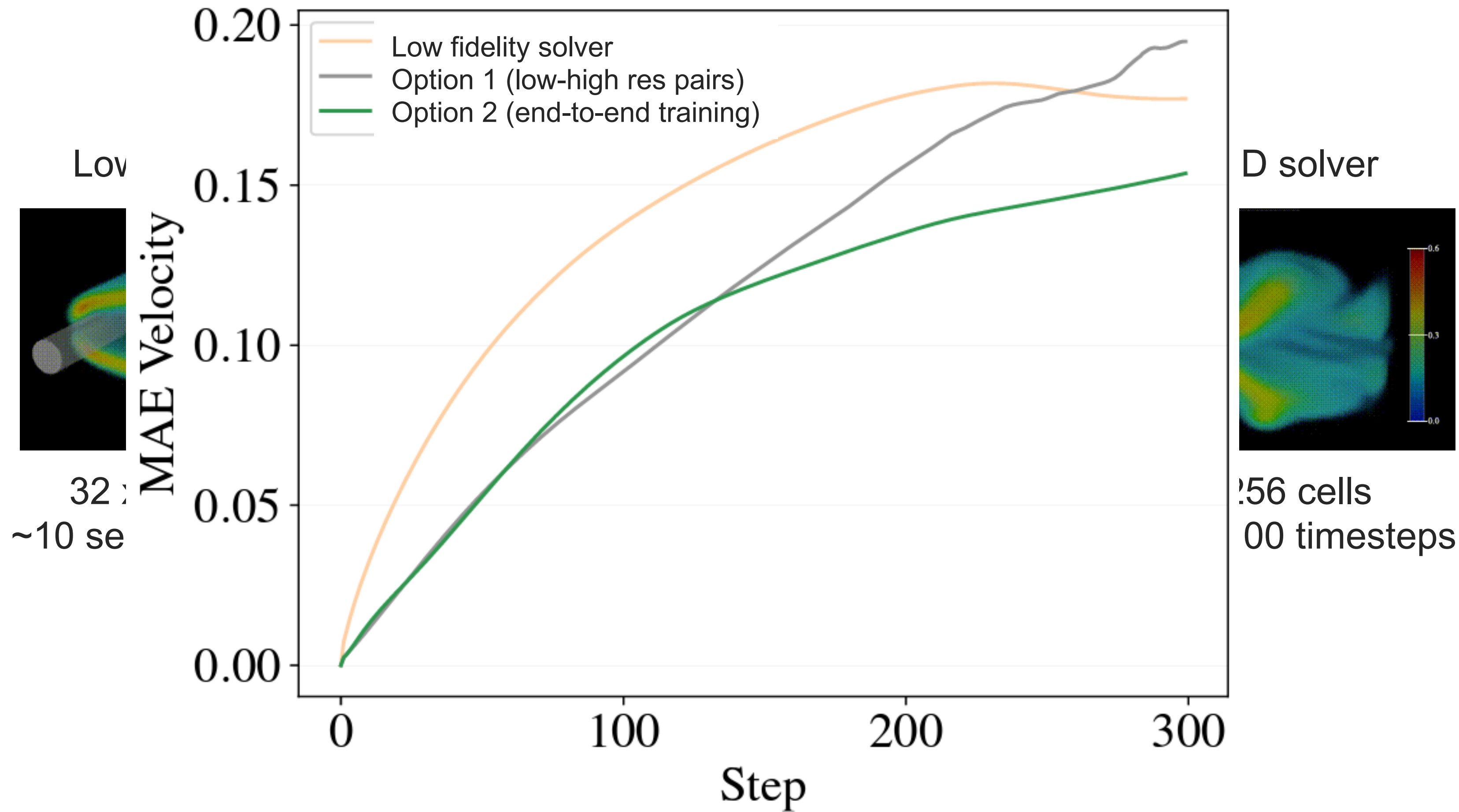
32 x 32 x 64 grid cells
~10 seconds / 100 timesteps

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)

NS solver results



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

Lecture overview

- Limitations of SciML approaches studied so far
- Hybrid SciML approaches
  - Residual modelling
  - Opening the “black-box”
  - How to train hybrid approaches
- Autodifferentiation
  - Autodifferentiation as a key enabler
  - What it is and how it works

Learning objectives

- Be able to describe what a hybrid workflow is
- Understand how autodifferentiation is used to train hybrid workflows
- Understand how autodifferentiation works

5 min break

Lecture overview

- Limitations of SciML approaches studied so far
- Hybrid SciML approaches
  - Residual modelling
  - Opening the "black-box"
  - How to train hybrid approaches
- Autodifferentiation
  - Autodifferentiation as a key enabler
  - What it is and how it works

Learning objectives

- Be able to describe what a hybrid workflow is
- Understand how autodifferentiation is used to train hybrid workflows
- Understand how autodifferentiation works

Autodifferentiation is a key enabler

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

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

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

$$L(\theta) = \lambda_1 (NN(t=0;\theta) - 1)^2 + \lambda_2 \left(\frac{dNN}{dt}(t=0;\theta) - 0\right)^2 + \frac{1}{N_p}\sum_i^{N_p}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right] NN(t_i;\theta)\right)^2$$

Physics-informed neural network

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

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

Operator learning

$$L(\theta) = \sum_i^N \sum_t^T \left\| HybridSolver_t(u_{0_i};\theta) - u_t^H(u_{0_i}) \right\|^2$$

Hybrid algorithms

Programs as vector functions

```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 = (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.grad(loss, theta)
# for learning theta (training NN)
```

Many (scientific) 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 $\boldsymbol{y}\colon \mathbb{R}^n \rightarrow \mathbb{R}^m$, composed of primitive operations:

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

Chain rule for vector functions

Consider **any** vector function $\boldsymbol{y}\colon \mathbb{R}^n \to \mathbb{R}^m$, 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

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

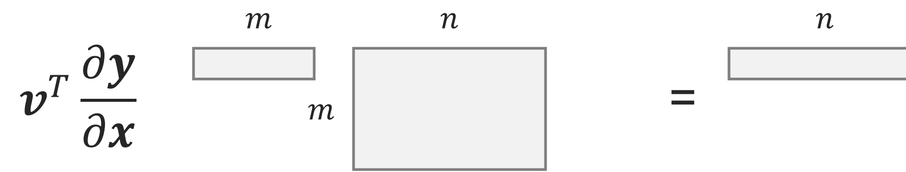
where

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

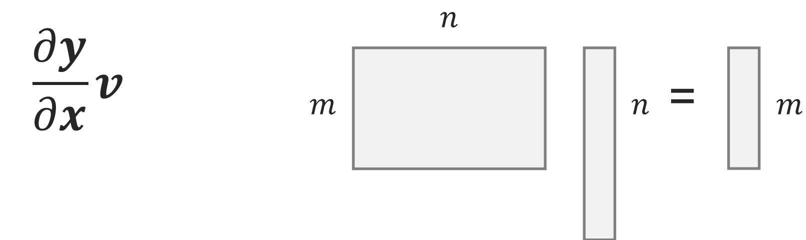
Autodifferentiation

Modern autodifferentiation libraries allow us to efficiently compute:

The vector-Jacobian product (vjp):



or the Jacobian-vector product (jvp):

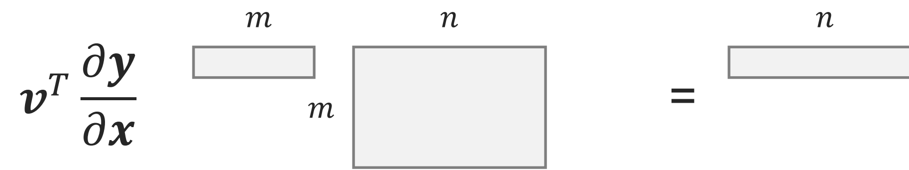


of **arbitrary** programs.

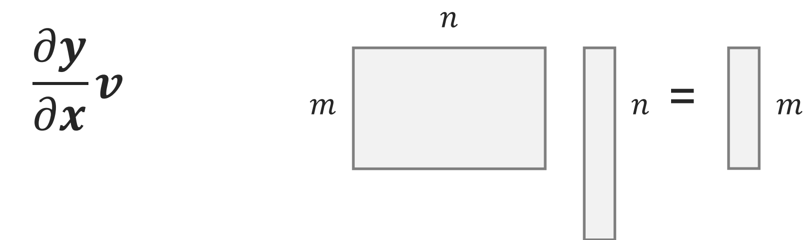
Autodifferentiation

Modern autodifferentiation libraries allow us to efficiently compute:

The vector-Jacobian product (vjp):



or the Jacobian-vector product (jvp):



of **arbitrary** programs.

Why is it useful to evaluate the vjp / jvp?

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):

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

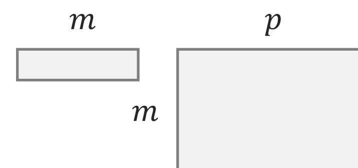
of **arbitrary** programs.

Why is it useful to evaluate the vjp / jvp?

Consider training a neural network:

$$L(\boldsymbol{\theta}): \mathbb{R}^p \rightarrow \mathbb{R}^1$$

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



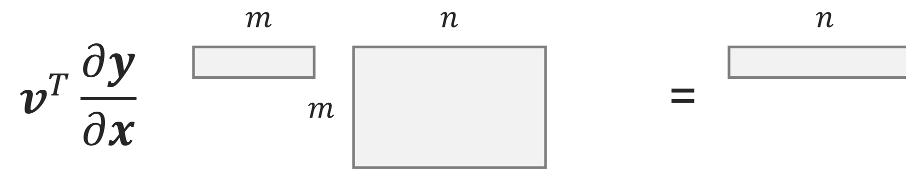
m = total number of network outputs
p = total number of parameters

We need to compute a vjp!

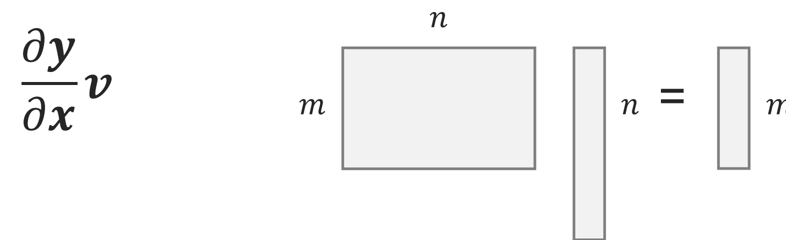
Chain rule

Modern autodifferentiation libraries allow us to efficiently compute:

The vector-Jacobian product (vjp):



or the Jacobian-vector product (jvp):



of **arbitrary** programs.

We can evaluate the vjp / jvp using the **chain rule**, for example:

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

Dimensionality

Let's think about dimensionality. Consider a simple MLP with $m = 100$ outputs, $h = 100$ hidden units, and 10,000 inputs.
Then W_1 has 100 x 10,000 = 1M elements.

$$NN(\boldsymbol{x};\theta) = \overbrace{W_2\sigma(\underbrace{W_1\boldsymbol{x} + \boldsymbol{b}_1}_{\boldsymbol{g}}) + \boldsymbol{b}_2}^{\boldsymbol{f}} = \boldsymbol{f} \circ \boldsymbol{g}(\boldsymbol{x};\theta)$$

Then:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}}\frac{\partial \boldsymbol{g}}{\partial W_1}$$

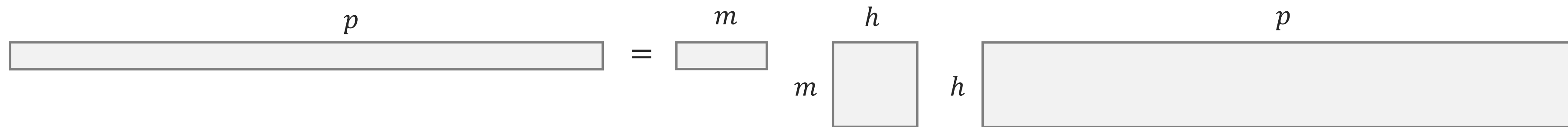
Dimensionality

Let's think about dimensionality. Consider a simple MLP with $m = 100$ outputs, $h = 100$ hidden units, and 10,000 inputs. Then W_1 has 100 x 10,000 = 1M elements.

$$NN(\boldsymbol{x};\theta) = \overbrace{W_2\sigma(\underbrace{W_1\boldsymbol{x} + \boldsymbol{b}_1}_{\boldsymbol{g}}) + \boldsymbol{b}_2}^{\boldsymbol{f}} = \boldsymbol{f} \circ \boldsymbol{g}(\boldsymbol{x};\theta)$$

Then:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}} \frac{\partial \boldsymbol{f}}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}} \frac{\partial \boldsymbol{g}}{\partial W_1}$$



(1 x 1M) = (1 x 100) (100 x 100) (100 x 1M)

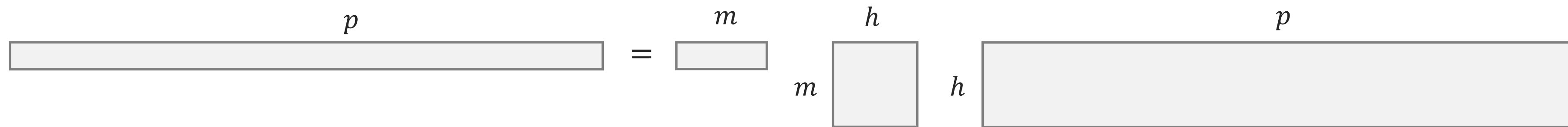
Dimensionality

Let's think about dimensionality. Consider a simple MLP with $m = 100$ outputs, $h = 100$ hidden units, and 10,000 inputs. Then W_1 has 100 x 10,000 = 1M elements.

$$NN(\boldsymbol{x};\theta) = \overbrace{W_2\sigma(\underbrace{W_1\boldsymbol{x} + \boldsymbol{b}_1}_{\boldsymbol{g}}) + \boldsymbol{b}_2}^{\boldsymbol{f}} = \boldsymbol{f} \circ \boldsymbol{g}(\boldsymbol{x};\theta)$$

Then:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}}\frac{\partial \boldsymbol{g}}{\partial W_1}$$



(1 x 1M) = (1 x 100) (100 x 100) (100 x 1M)

Consider evaluating the chain rule (RHS):

1) From right to left (forward)

(100 x 100) (100 x 1M) → (100 x 1M)
(1 x 100) (100 x 1M) → (1 x 1M)

= lots of computation (large matrix-matrix multiply)

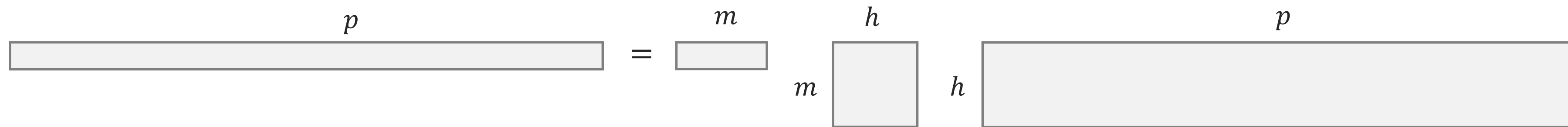
Dimensionality

Let's think about dimensionality. Consider a simple MLP with m = 100 outputs, h = 100 hidden units, and 10,000 inputs. Then W_1 has 100 x 10,000 = 1M elements.

$$NN(\boldsymbol{x};\theta) = \overbrace{W_2\sigma(\underbrace{W_1\boldsymbol{x}+\boldsymbol{b}_1}_{\boldsymbol{g}})+\boldsymbol{b}_2}^{\boldsymbol{f}} = \boldsymbol{f}\circ\boldsymbol{g}(\boldsymbol{x};\theta)$$

Then:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}}\frac{\partial \boldsymbol{g}}{\partial W_1}$$



Consider evaluating the chain rule (RHS):

(1 x 1M) = (1 x 100) (100 x 100) (100 x 1M)

1) From right to left (forward)

(100 x 100) (100 x 1M) → (100 x 1M)
(1 x 100) (100 x 1M) → (1 x 1M)

= lots of computation (large matrix-matrix multiply)

2) From left to right (reverse)

(1 x 100) (100 x 100) → (1 x 100)
(1 x 100) (100 x 1M) → (1 x 1M)

= much less computation (vector-matrix multiplies)

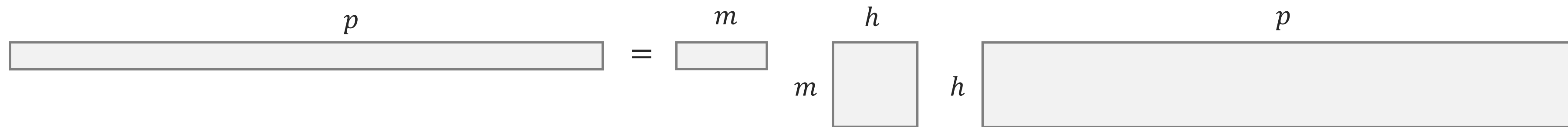
Dimensionality

Let's think about dimensionality. Consider a simple MLP with $m = 100$ outputs, $h = 100$ hidden units, and 10,000 inputs.
Then W_1 has 100 x 10,000 = 1M elements.

$$NN(\boldsymbol{x};\theta) = \overbrace{W_2\sigma(\underbrace{W_1\boldsymbol{x} + \boldsymbol{b}_1}_{\boldsymbol{g}}) + \boldsymbol{b}_2}^{\boldsymbol{f}} = \boldsymbol{f} \circ \boldsymbol{g}(\boldsymbol{x};\theta)$$

Then:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}}\frac{\partial \boldsymbol{g}}{\partial W_1}$$



Consider evaluating the chain rule (RHS):

(1 x 1M) = (1 x 100) (100 x 100) (100 x 1M)

1) From right to left (forward)

(100 x 100) (100 x 1M) → (100 x 1M)
(1 x 100) (100 x 1M) → (1 x 1M)

= lots of computation (large matrix-matrix multiply)

2) From left to right (reverse)

(1 x 100) (100 x 100) → (1 x 100)
(1 x 100) (100 x 1M) → (1 x 1M)

= much less computation (vector-matrix multiplies)

=> Order matters! Evaluating vjps in reverse mode is usually most efficient

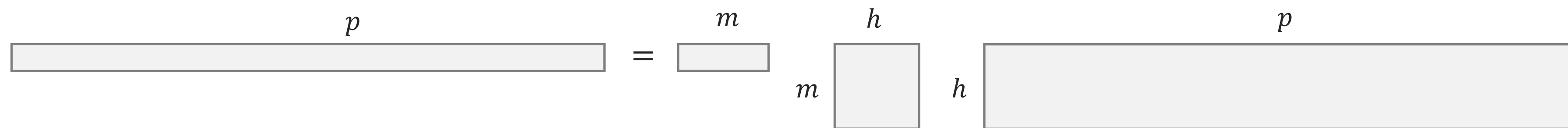
Dimensionality

Let's think about dimensionality. Consider a simple MLP with $m = 100$ outputs, $h = 100$ hidden units, and 10,000 inputs.
Then W_1 has 100 x 10,000 = 1M elements.

$$NN(\boldsymbol{x};\theta) = \overbrace{W_2\sigma(\underbrace{W_1\boldsymbol{x} + \boldsymbol{b}_1}_{\boldsymbol{g}}) + \boldsymbol{b}_2}^{\boldsymbol{f}} = \boldsymbol{f} \circ \boldsymbol{g}(\boldsymbol{x};\theta)$$

Then:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}}\frac{\partial \boldsymbol{g}}{\partial W_1}$$



Consider evaluating the chain rule (RHS): (1 x 1M) = (1 x 100) (100 x 100) (100 x 1M)

2) From left to right (reverse)

But what about the last computation ($\frac{\partial L}{\partial \boldsymbol{g}}\frac{\partial \boldsymbol{g}}{\partial W_1}$)?

This is still expensive! ($\frac{\partial \boldsymbol{g}}{\partial W_1}$ has 100M elements, or ~0.5 GB) →

(1 x 100) (100 x 100) → (1 x 100)
(1 x 100) (100 x 1M) → (1 x 1M)

= much less computation (vector-matrix multiplies)

=> Order matters! Evaluating vjps in reverse mode is usually most efficient

Dimensionality

Note:

$$\frac{\partial \boldsymbol{g}}{\partial W_1} = \frac{\partial (W_1 \boldsymbol{x} + \boldsymbol{b}_1)}{\partial W_1} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n & 0 & 0 & \cdots & 0 \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & \cdots & 0 & x_1 & x_2 & \cdots & x_n \end{pmatrix}$$

Then

$$\frac{\partial L}{\partial W_1} = \frac{dL}{\partial \boldsymbol{g}} \frac{\partial \boldsymbol{g}}{\partial W_1} = \left(\frac{dL}{\partial g_1}(x_1, \cdots x_{10,000}), \cdots, \frac{dL}{\partial g_{100}}(x_1, \cdots, x_{10,000}) \right)$$

$$= \frac{\partial L}{\partial \boldsymbol{g}}^{\boldsymbol{T}} \otimes \boldsymbol{x}$$

This is just the (flattened) **outer product** of two vectors (100 x 1) ⊗ (10,000 x 1)

⇒ We don't have to fully populate the last Jacobian ($\frac{\partial \boldsymbol{g}}{\partial W_1}$) when computing its vector-Jacobian product

⇒ Often, vjps (and jvps) can be computed efficiently **without** needing to populate the full Jacobian

Dimensionality

Another example:

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

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

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

$\Rightarrow$ We don't have to fully populate the last Jacobian ($\frac{\partial \boldsymbol{g}}{\partial W_1}$) when computing its vector-Jacobian product

$\Rightarrow$ Often, vjps (and jvps) can be computed efficiently **without** needing to populate the full Jacobian

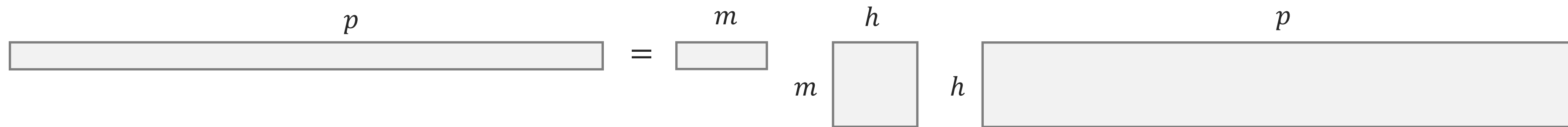
Dimensionality

Let's think about dimensionality. Consider a simple MLP with $m = 100$ outputs, $h = 100$ hidden units, and 10,000 inputs.
Then W_1 has 100 x 10,000 = 1M elements.

$$NN(\boldsymbol{x};\theta) = \overbrace{W_2\sigma(\underbrace{W_1\boldsymbol{x}+\boldsymbol{b}_1}_{\boldsymbol{g}})+\boldsymbol{b}_2}^{\boldsymbol{f}} = \boldsymbol{f}\circ\boldsymbol{g}(\boldsymbol{x};\theta)$$

Then:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial W_1} = \frac{\partial L}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}}\frac{\partial \boldsymbol{g}}{\partial W_1}$$



Consider evaluating the chain rule (RHS):

(1 x 1M) = (1 x 100) (100 x 100) (100 x 1M)

= Efficient training code

Allows us to train neural networks with **billions** of parameters

2) From left to right (reverse)

(1 x 100) (100 x 100) → (1 x 100)
~~(1 x 100) (100 x 1M) → (1 x 1M)~~
(100 x 1) ⊗ (10,000 x 1) → (1 x 1M)

= much less computation (vector-matrix multiplies)

Vector-Jacobian product

vjp:

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

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 $\boldsymbol{v}^T$,

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

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

...

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

- We only need to define the **vjp** for each **primitive operation** to compute $\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$
- Usually, we do not need to explicitly compute the full intermediate Jacobians $\frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{f}_{i-1}}$

Jacobian-vector product

jvp:

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

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

Starting with $\boldsymbol{v}$,

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

$$\boldsymbol{v} \leftarrow \frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{f}_1} \boldsymbol{v}$$

$$\dots$$

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

- We only need to define the **jvp** for each **primitive operation** to compute $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \boldsymbol{v}$
- Usually, we do not need to explicitly compute the full intermediate Jacobians $\frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{f}_{i-1}}$

Full Jacobian

- What if we want the full Jacobian? $J_y = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$

Full Jacobian

- What if we want the full Jacobian? $J_y = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$

- We can combine vjps / jvps to compute the full Jacobian row by row / column by column if necessary

Let

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

Then

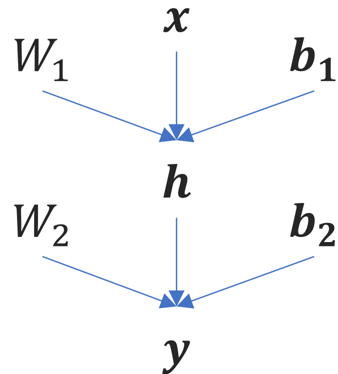
$$\boldsymbol{v}^T \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \left(\frac{\partial y_1}{\partial x_1}, \dots, \frac{\partial y_1}{\partial x_n}\right)$$

= First row of Jacobian

- Note jvps are usually more efficient for “tall” Jacobians, whilst vjps are more efficient for “wide” Jacobians

Autodiff in practice

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

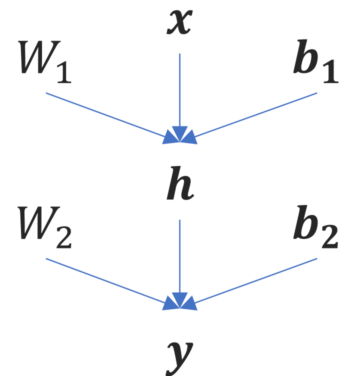


1) Decompose given function into its **primitive** operations

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

Autodiff in practice

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



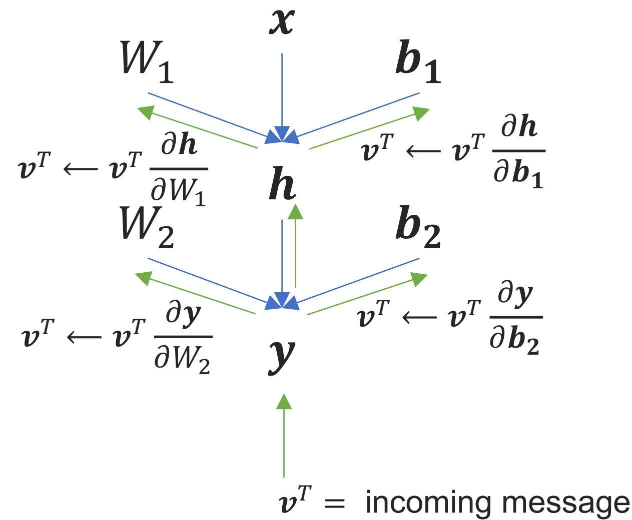
1) Decompose given function into its **primitive** operations

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

Autodiff in practice

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



TensorFlow

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

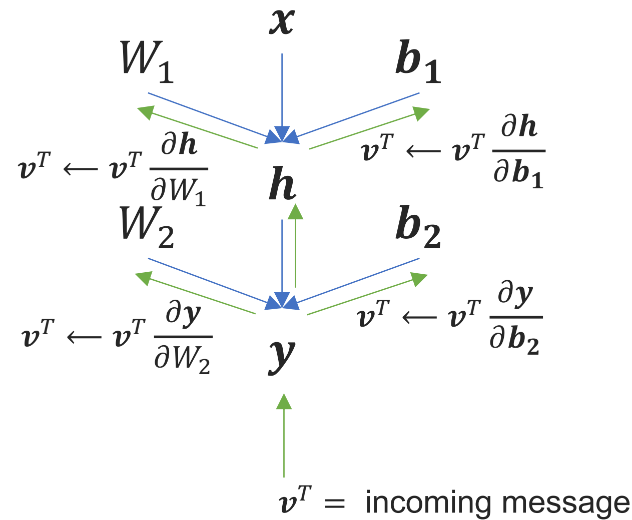
2) Build a **directed graph** of these operations

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

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

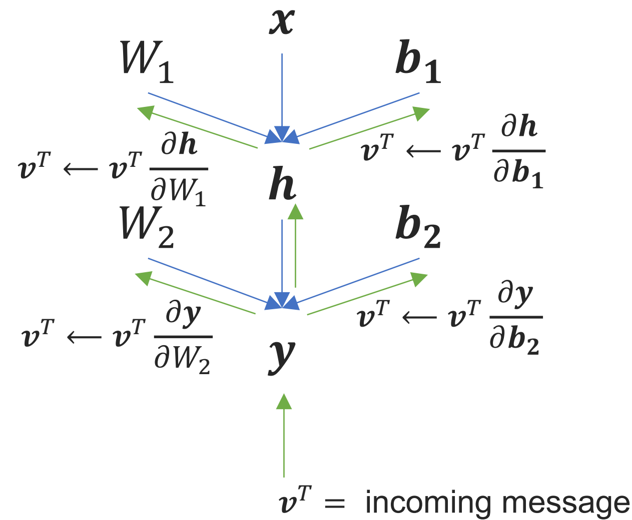
PyTorch

TensorFlow

- How does required memory scale with depth of computation for vjp vs jvp?

Autodiff in practice

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



PyTorch

TensorFlow

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

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

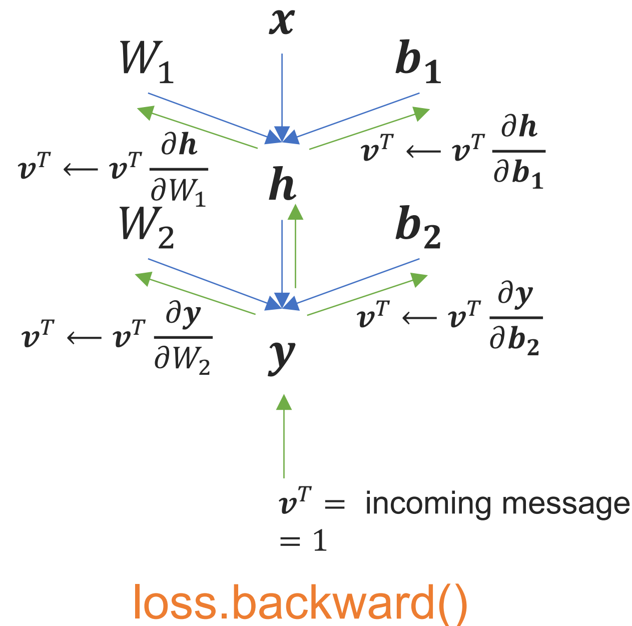
3) For each primitive operation, define
   1) Forward operation
   2) vector-Jacobian product
   3) Jacobian-vector product

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

- How does required memory scale with depth of computation for vjp vs jvp?
- vjp: memory scales linearly with depth (need to store forward computations)
- jvp: memory independent of depth (can compute jvp alongside forward pass)

Autodiff in practice

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



torch.autograd.grad(*outputs*, *inputs*, *grad_outputs=None*, *retain_graph=None*, *create_graph=False*, *only_inputs=True*, *allow_unused=None*, *is_grads_batched=False*, *materialize_grads=False*) [SOURCE]

Computes and returns the sum of gradients of outputs with respect to the inputs.

`grad_outputs` should be a sequence of length matching `output` containing the “vector” in vector-Jacobian product, usually the pre-computed gradients w.r.t. each of the outputs. If an output doesn’t require_grad, then the gradient can be `None`).

Note autodiff is **not**

- Symbolic differentiation
- Finite differences

It is a way of efficiently computing exact gradients!

Hybrid workflows in practice

Step 1: **rewrite** your traditional scientific algorithm in an autodifferentiation framework (e.g. PyTorch/JAX)

Hybrid workflows in practice

Step 1: **rewrite** your traditional scientific algorithm in an autodifferentiation framework (e.g. PyTorch/JAX)

Step 2: make parts of this algorithm **learnable** (either to accelerate it, or to improve accuracy)

Hybrid workflows in practice

Step 1: **rewrite** your traditional scientific algorithm in an autodifferentiation framework (e.g. PyTorch/JAX)

Step 2: make parts of this algorithm **learnable** (either to accelerate it, or to improve accuracy)

Step 3: get some **training examples** of what you want the input/output of the algorithm to be

Hybrid workflows in practice

Step 1: **rewrite** your traditional scientific algorithm in an autodifferentiation framework (e.g. PyTorch/JAX)

Step 2: make parts of this algorithm **learnable** (either to accelerate it, or to improve accuracy)

Step 3: get some **training examples** of what you want the input/output of the algorithm to be

Step 4: train your algorithm by **(auto)differentiating** through it and using gradient descent

Hybrid workflows in practice

Step 1: **rewrite** your traditional scientific algorithm in an autodifferentiation framework (e.g. PyTorch/JAX)

Step 2: make parts of this algorithm **learnable** (either to accelerate it, or to improve accuracy)

Step 3: get some **training examples** of what you want the input/output of the algorithm to be

Step 4: train your algorithm by **(auto)differentiating** through it and using gradient descent

Bonus: your code now runs on the GPU!

Summary

- Hybrid approaches insert learnable components **inside** traditional algorithms
- Autodifferentiation is **the** key **enabler** for SciML
    - Allows hybrid approaches to be trained end-to-end
    - Is an incredibly general and powerful tool