Deep Learning in Scientific Computing

Introduction to Differentiable Physics - Part 2

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

- Differentiable physics recap
- Coding a simple hybrid approach in PyTorch
- Hybrid approaches for inverse problems
- Neural differential equations (NDEs)
- Course summary

Hybrid approaches - recap

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:

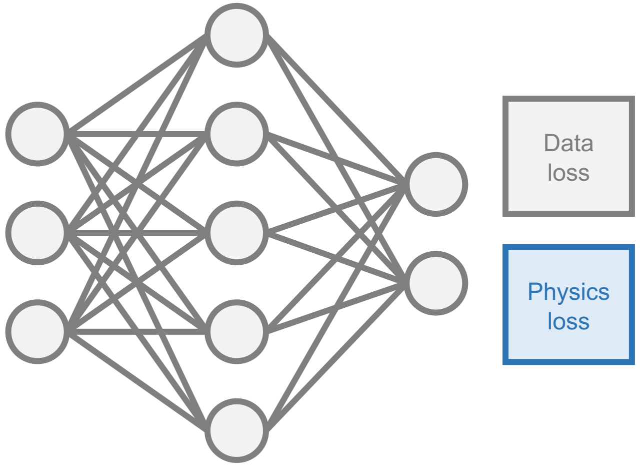
1) **Accelerate** your workflow, or
2) Learn the **parts** you are unsure of / have incomplete knowledge

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

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

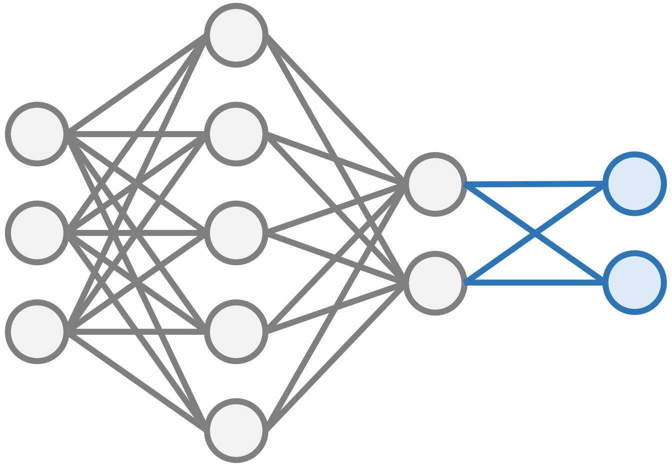
Ways to incorporate scientific principles into machine learning

Loss function



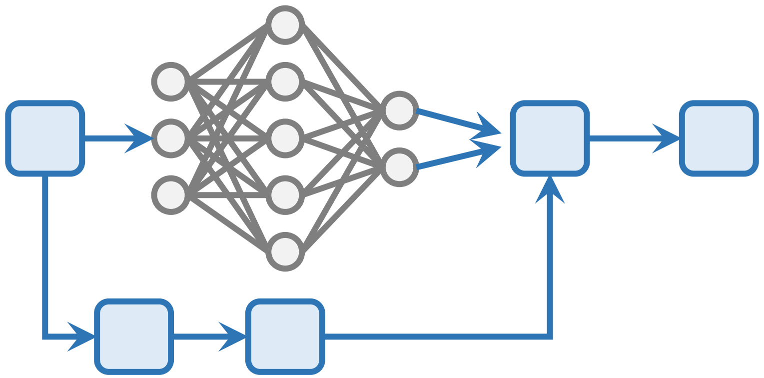
Example:
Physics-informed neural networks
(add governing equations to loss function)

Architecture



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

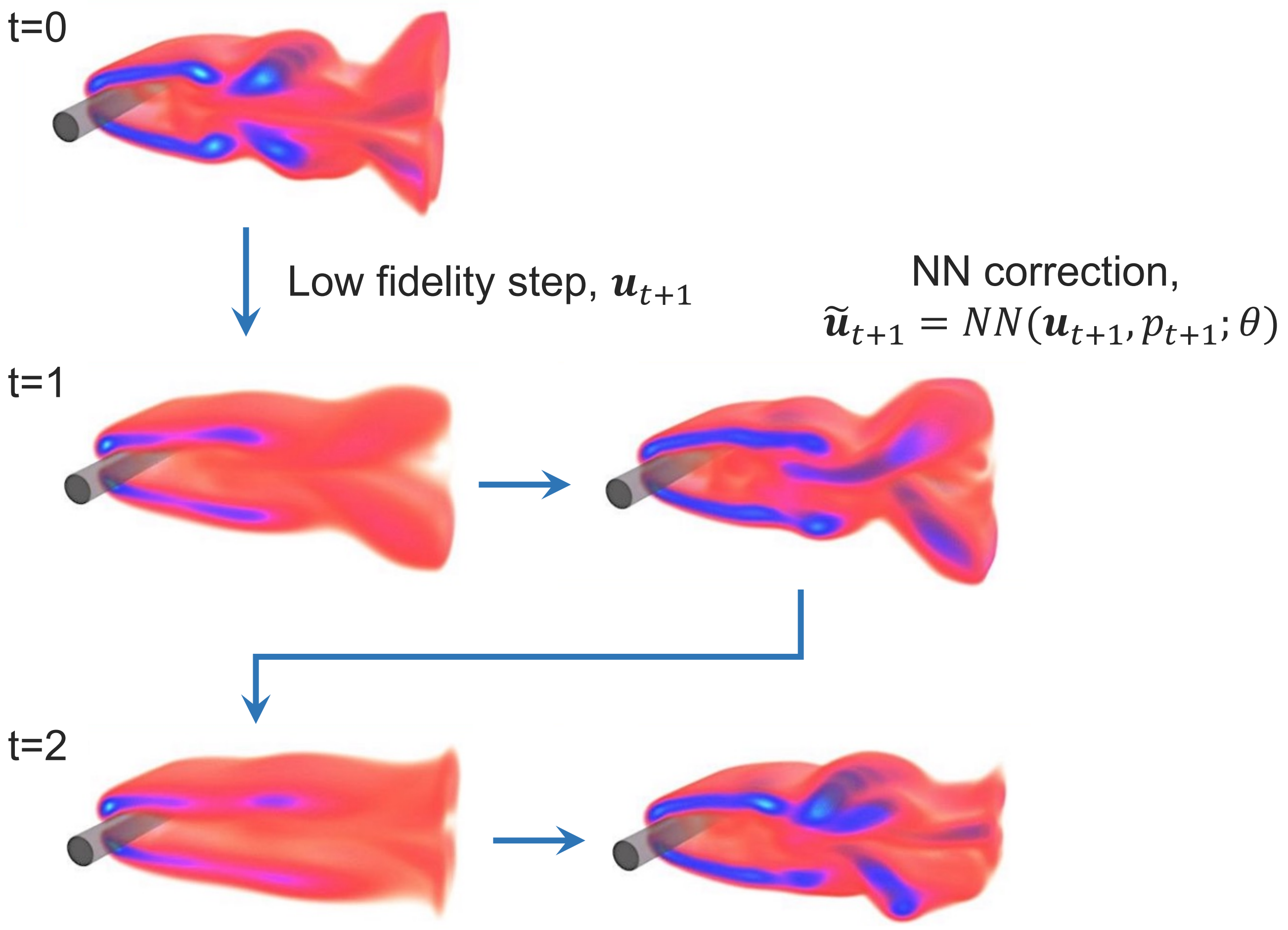
Hybrid approaches



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

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 = 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?

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

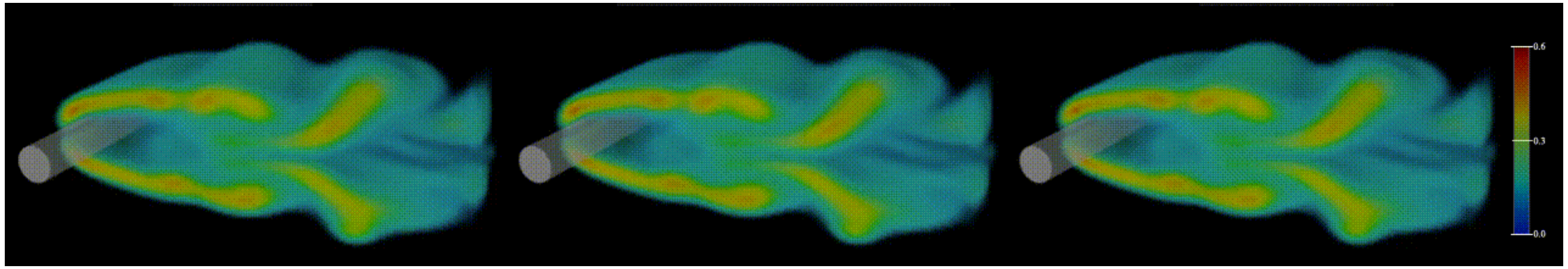
Differentiable physics = using autodifferentiation to learn physical algorithms

```
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 - 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)

Lecture overview

- Differentiable physics recap
- Coding a simple hybrid approach in PyTorch
- Hybrid approaches for inverse problems
- Neural differential equations (NDEs)
- Course summary

Coding a simple hybrid approach in PyTorch

More complex hybrid approaches

```
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
```

- There are many **other** ways we can insert neural networks into our existing algorithms

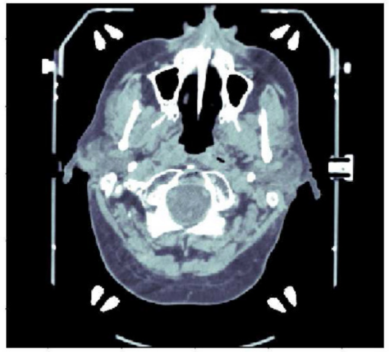
More complex hybrid approaches

```
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
```

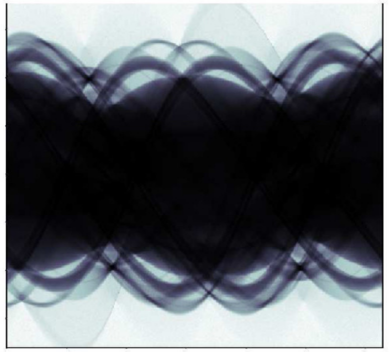
- There are many **other** ways we can insert neural networks into our existing algorithms
- Many traditional algorithms (simulation, inversion, control, data assimilation, equation discovery, …) are **iterative**
- **“In-the-loop”** methods are a class of hybrid approaches which insert neural networks into the **inner loops** of traditional algorithms

A more complex "in-the-loop" example

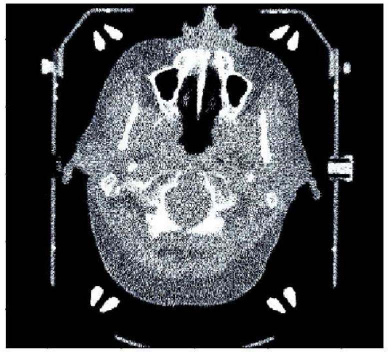
Consider the following **inverse** problem:



Ground truth computed tomography image



Resulting tomographic data (sinogram)



Result of inverse algorithm

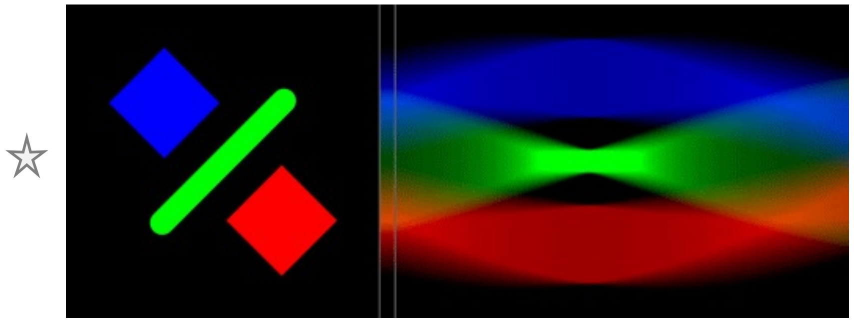
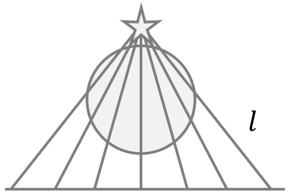


Image source: Wikipedia

$a(x)$

$$b = F(a) = I_0 \exp(-\int_l a(x)\,dx))$$

$\hat{a}(x)$



Adler et al, Solving ill-posed inverse problems using iterative deep neural networks, Inverse Problems (2017)

$$b = F(a)$$

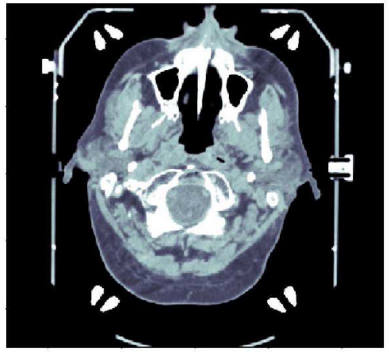
a = set of input conditions

F = physical model of the system

b = resulting properties given F and a

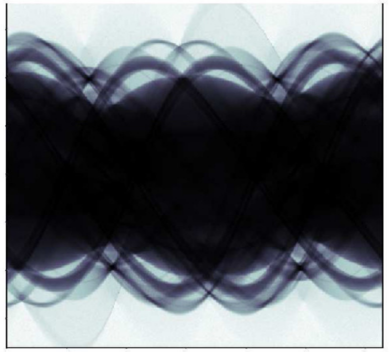
A more complex "in-the-loop" example

Consider the following **inverse** problem:



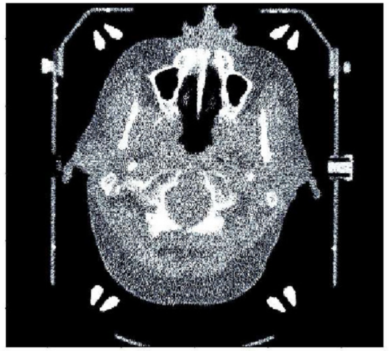
Ground truth computed tomography image

$a(x)$



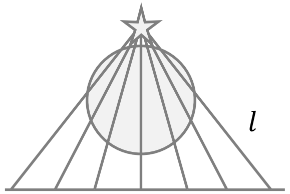
Resulting tomographic data (sinogram)

$$b = F(a) = I_0 \exp(-\int_l a(x)\, dx))$$



Result of inverse algorithm

$\hat{a}(x)$



Adler et al, Solving ill-posed inverse problems using iterative deep neural networks, Inverse Problems (2017)

This problem can be framed as an **optimisation** problem:

$$\min_{\hat{a}} \|b - F(\hat{a})\|^2$$

Assuming F is a differentiable, we can use **gradient descent** to learn $\hat{a}$:

Loss function:

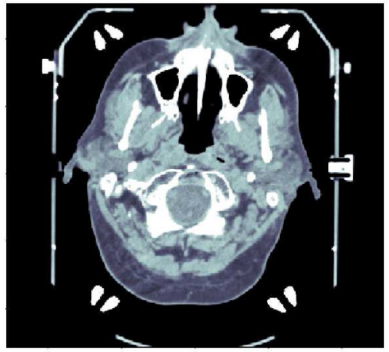
$$L(\hat{a}) = \|b - F(\hat{a})\|^2$$

Gradient descent:

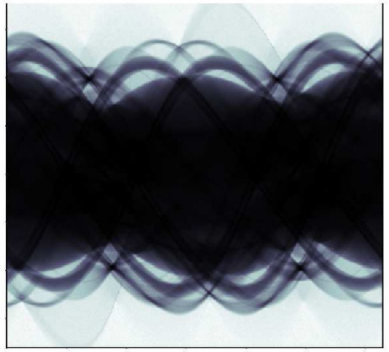
$$\hat{a} \leftarrow \hat{a} - \gamma \frac{\partial L(\hat{a})}{\partial \hat{a}}$$

A more complex “in-the-loop” example

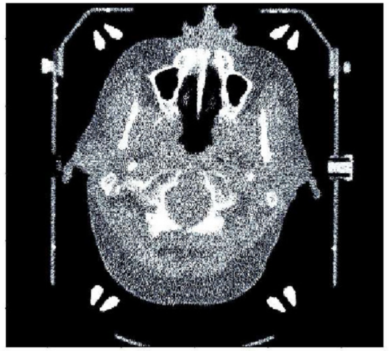
Consider the following **inverse** problem:



Ground truth computed tomography image

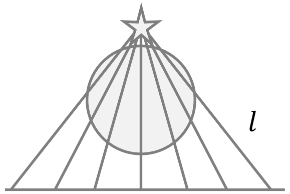


Resulting tomographic data (sinogram)



Result of inverse algorithm

$a(x)$ $\qquad b = F(a) = I_0 \exp(-\int_l a(x)\, dx)$ $\qquad \hat{a}(x)$



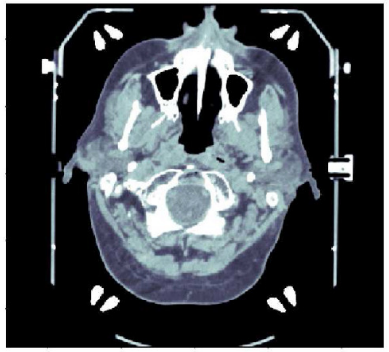
Adler et al, Solving ill-posed inverse problems using iterative deep neural networks, Inverse Problems (2017)

However, this problem can be very **ill-posed** if;

- The measurements are noisy
- There are not enough observations

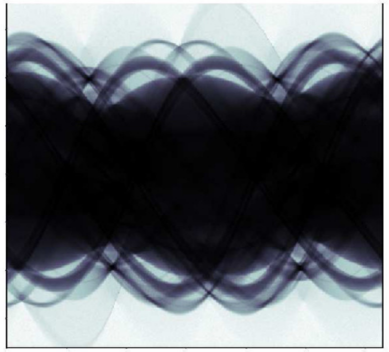
A more complex "in-the-loop" example

Consider the following **inverse** problem:



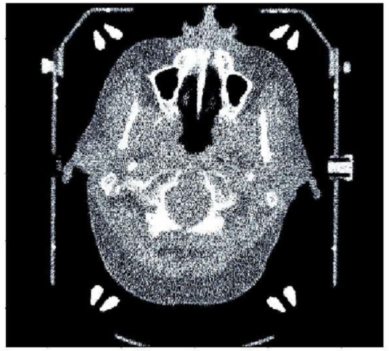
Ground truth computed tomography image

$a(x)$



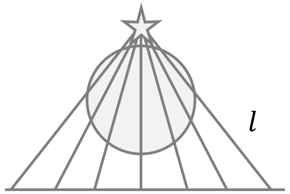
Resulting tomographic data (sinogram)

$$b = F(a) = I_0 \exp(-\int_l a(x)\, dx))$$



Result of inverse algorithm

$\hat{a}(x)$



Adler et al, Solving ill-posed inverse problems using iterative deep neural networks, Inverse Problems (2017)

However, this problem can be very **ill-posed** if;

- The measurements are noisy
- There are not enough observations

To improve, add **regularization**:

$$L(\hat{a}) = \|b - F(\hat{a})\|^2 + \lambda\, R(\hat{a})$$

Where, for example

$$R(\hat{a}) = \|\nabla \hat{a}\|$$

Which asserts a **prior** that the output image should be "smooth" (= total variation regularization)

A more complex "in-the-loop" example

```
def X_ray_tomography(a_hat_0, b):
    "Pseudocode for carrying out X ray tomography"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        a_hat -= gamma*da

    return a_hat
```

1. Start with initial guess $\hat{a}$
2. Loop:
    1. Compute gradient, $\frac{\partial L(\hat{a})}{\partial \hat{a}}$
    2. Take gradient descent step,

$$\hat{a} \leftarrow \hat{a} - \gamma \frac{\partial L(\hat{a})}{\partial \hat{a}}$$

However, this problem can be very **ill-posed** if;

- The measurements are noisy
- There are not enough observations

To improve, add **regularization**:

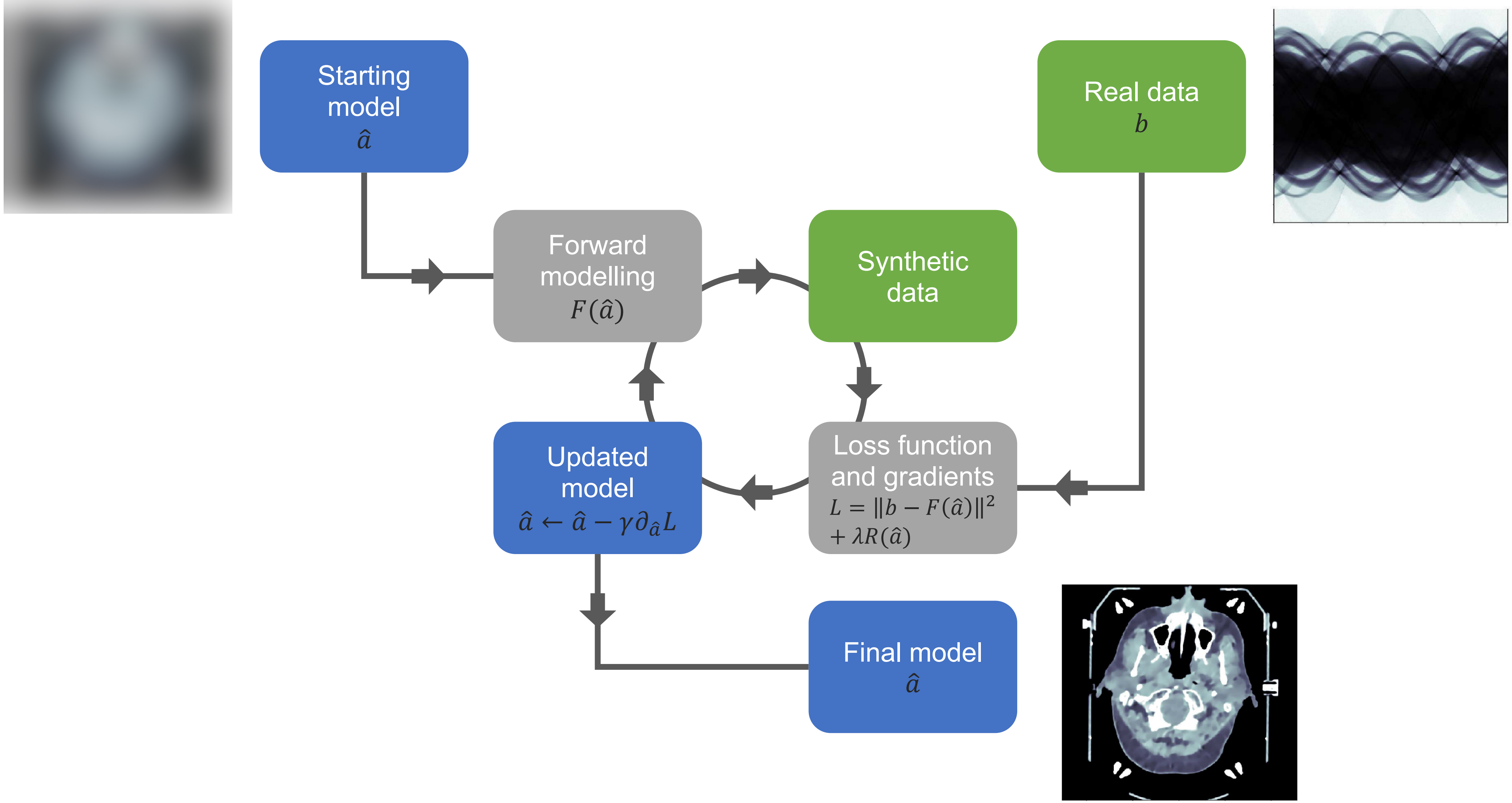
$$L(\hat{a}) = \|b - F(\hat{a})\|^2 + \lambda R(\hat{a})$$

Where, for example

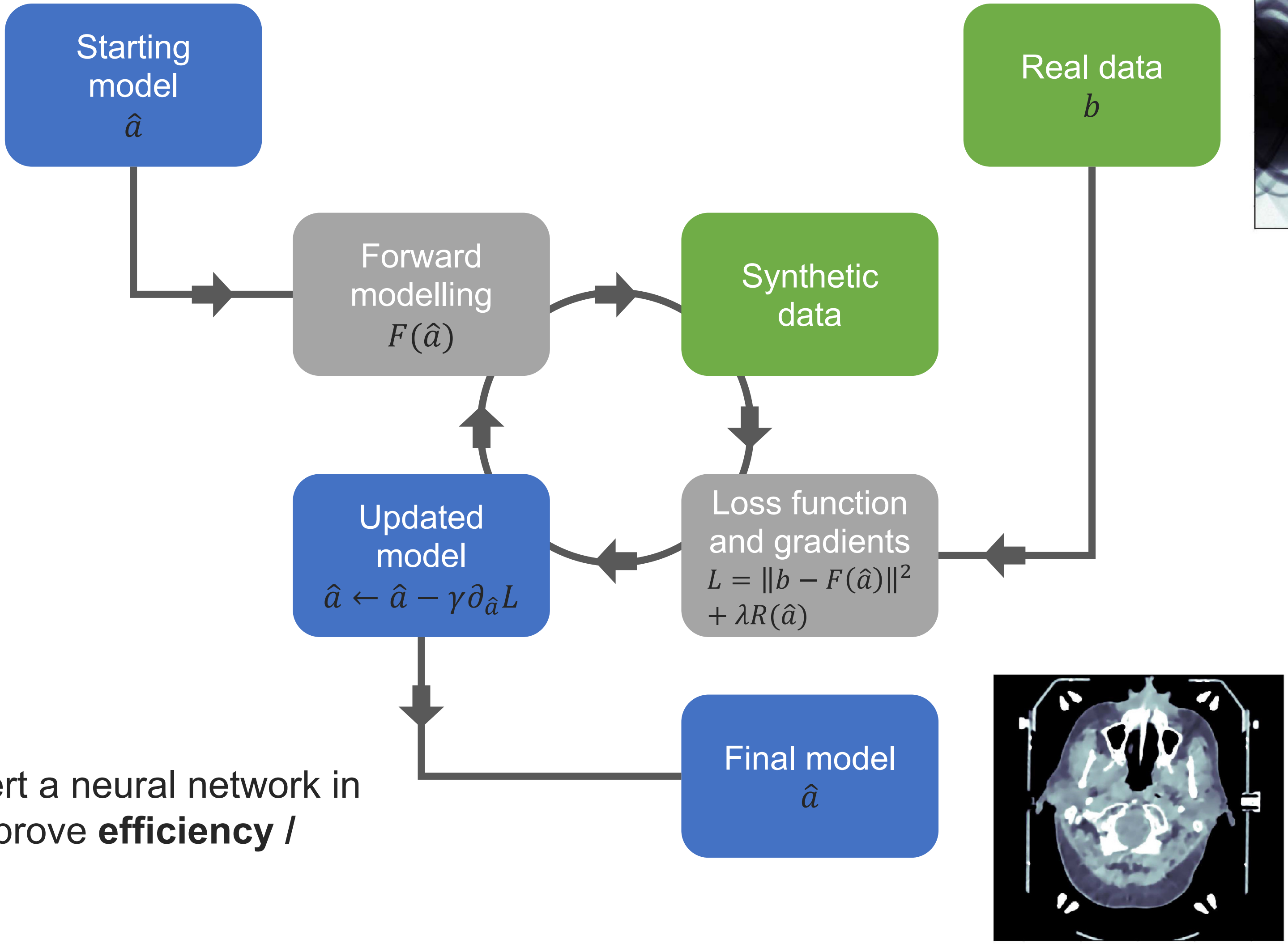
$$R(\hat{a}) = \|\nabla \hat{a}\|$$

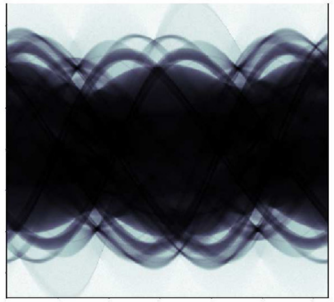
Which asserts a **prior** that the output image should be "smooth" (= total variation regularization)

X-ray tomography



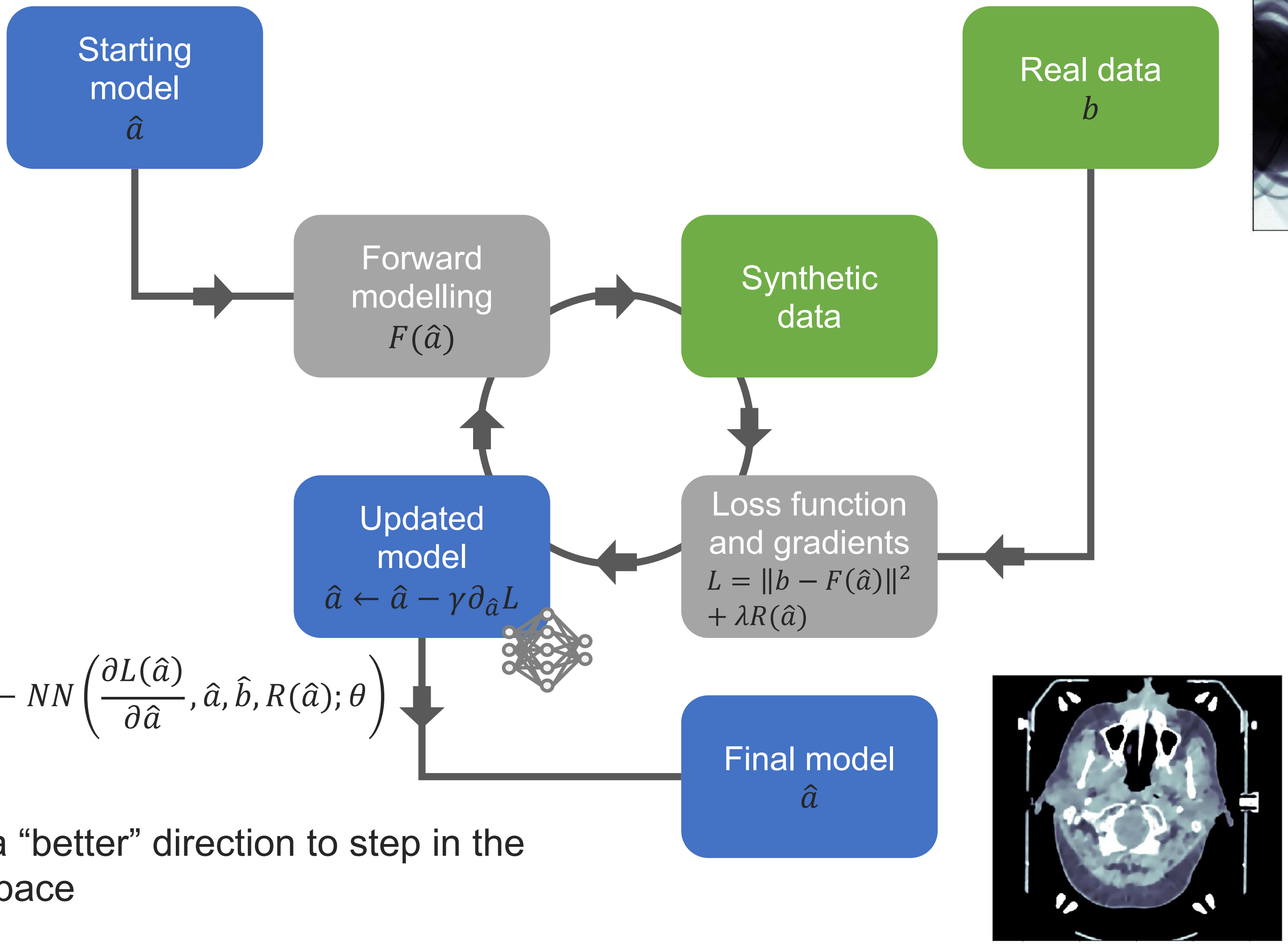
X-ray tomography

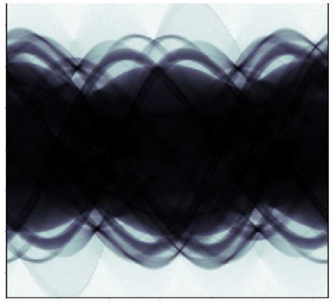




- Where can we insert a neural network in this workflow to improve **efficiency / accuracy**?

Hybrid X-ray tomography





$$\frac{\partial L(\hat{a})}{\partial \hat{a}} \leftarrow NN\left(\frac{\partial L(\hat{a})}{\partial \hat{a}}, \hat{a}, \hat{b}, R(\hat{a}); \theta\right)$$

Idea: **learn** a “better” direction to step in the parameter space

Hybrid X-ray tomography

```
def X_ray_tomography(a_hat_0, b):
    "Pseudocode for carrying out X ray tomography"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        a_hat -= gamma*da

    return a_hat
```

1. Start with initial guess $\hat{a}$
2. Loop:
    1. Compute gradient, $\frac{\partial L(\hat{a})}{\partial \hat{a}}$
    2. Take gradient descent step,

$$\hat{a} \leftarrow \hat{a} - \gamma\, NN\left(\frac{\partial L(\hat{a})}{\partial \hat{a}}, \hat{a}, \hat{b}, R(\hat{a}); \theta\right)$$

```
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat
```

Idea: **learn** a “better” direction to step in the parameter space

Hybrid X-ray tomography

```python
def X_ray_tomography(a_hat_0, b):
    "Pseudocode for carrying out X ray tomography"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        a_hat -= gamma*da

    return a_hat
```

1. Start with initial guess $\hat{a}$
2. Loop:
    1. Compute gradient, $\frac{\partial L(\hat{a})}{\partial \hat{a}}$
    2. Take gradient descent step,

$$\hat{a} \leftarrow \hat{a} - \gamma \, NN\left(\frac{\partial L(\hat{a})}{\partial \hat{a}}, \hat{a}, \hat{b}, R(\hat{a}); \theta\right)$$

```python
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat
```

Idea: **learn** a “better” direction to step in the parameter space

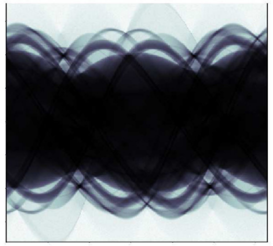
- How do we train this hybrid approach (learn θ)?

Hybrid X-ray tomography

Input to function:

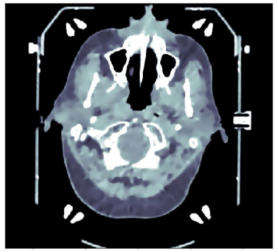
$\hat{a}_0$

b



Output:

$\hat{a}$



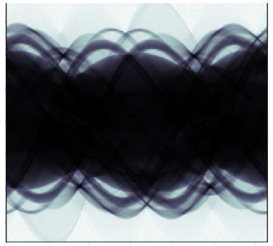
```
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat
```

Hybrid X-ray tomography

Input to function:

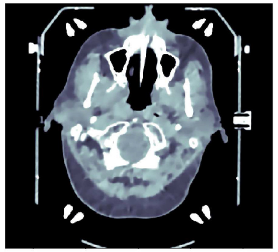
$\hat{a}_0$

b



Output:

$\hat{a}$



```python
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat
```

We train this hybrid approach using lots of examples of inputs/outputs $(\hat{a}_0, b, a)$ and the loss function

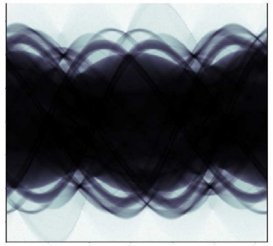
$$L(\theta) = \sum_{i}^{N} \|H(\hat{a}_{0\,i}, b_i; \theta) - a_i\|^2$$

Hybrid X-ray tomography

Input to function:

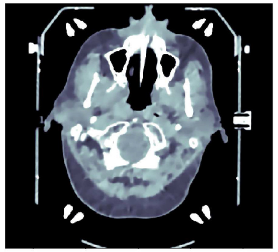
$\hat{a}_0$

b



Output:

$\hat{a}$



We train this hybrid approach using lots of examples of inputs/outputs $(\hat{a}_0, b, a)$ and the loss function

$$L(\theta) = \sum_i^N \|H(\hat{a}_{0\,i}, b_i; \theta) - a_i\|^2$$

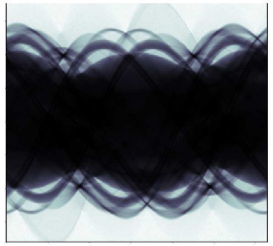
```python
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

Hybrid X-ray tomography

Input to function:

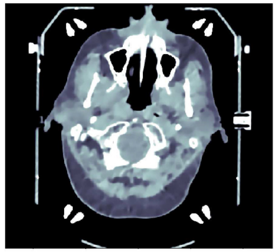
$\hat{a}_0$

b



Output:

$\hat{a}$



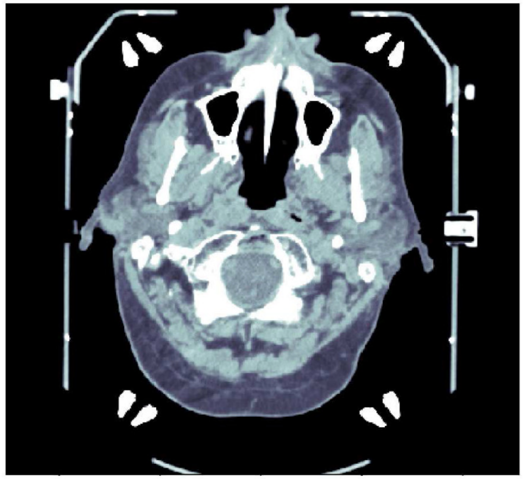
```
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

We train this hybrid approach using lots of examples of inputs/outputs $(\hat{a}_0, b, a)$ and the loss function

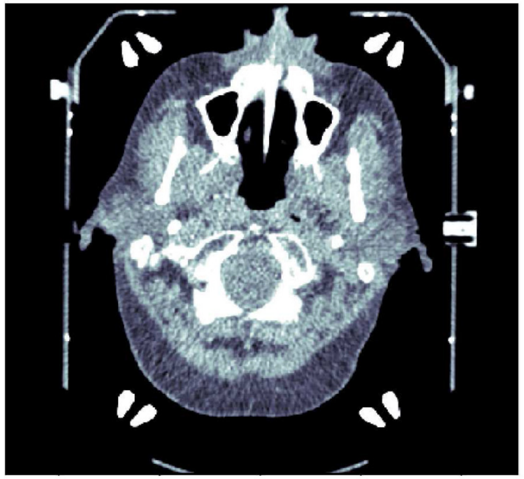
$$L(\theta) = \sum_i^N \| H(\hat{a}_{0\,i}, b_i; \theta) - a_i \|^2$$

"Gradient descent on gradient descent"
"Learned gradient descent"
"Learning to learn"

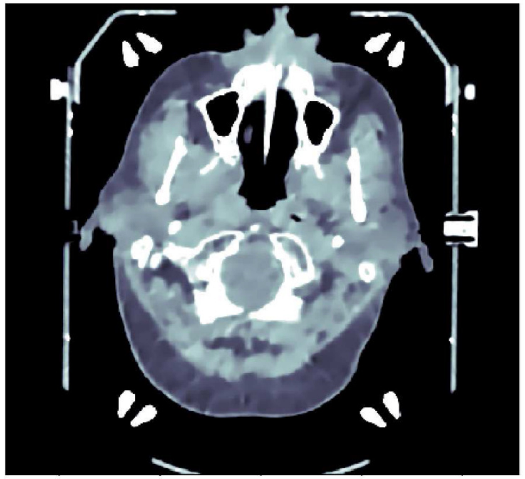
Hybrid X-ray tomography



Ground truth



Traditional inversion



Learned gradient descent

Adler et al, Solving ill-posed inverse problems using
iterative deep neural networks, Inverse Problems (2017)

Adding even more flexibility

- We can use **more** than one learnable component if we want!
- Where else would it be useful to add another?

```
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

Idea: **learn** a “better” direction to step in the parameter space

Adding even more flexibility

```python
def Hybrid2_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0

    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + theta[0]*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta[1])
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid2_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

Idea 2: learn regularisation **hyperparameter** too

```python
def Hybrid_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0
    lam = 1
    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + lam*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta)
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

Idea: **learn** a “better” direction to step in the parameter space

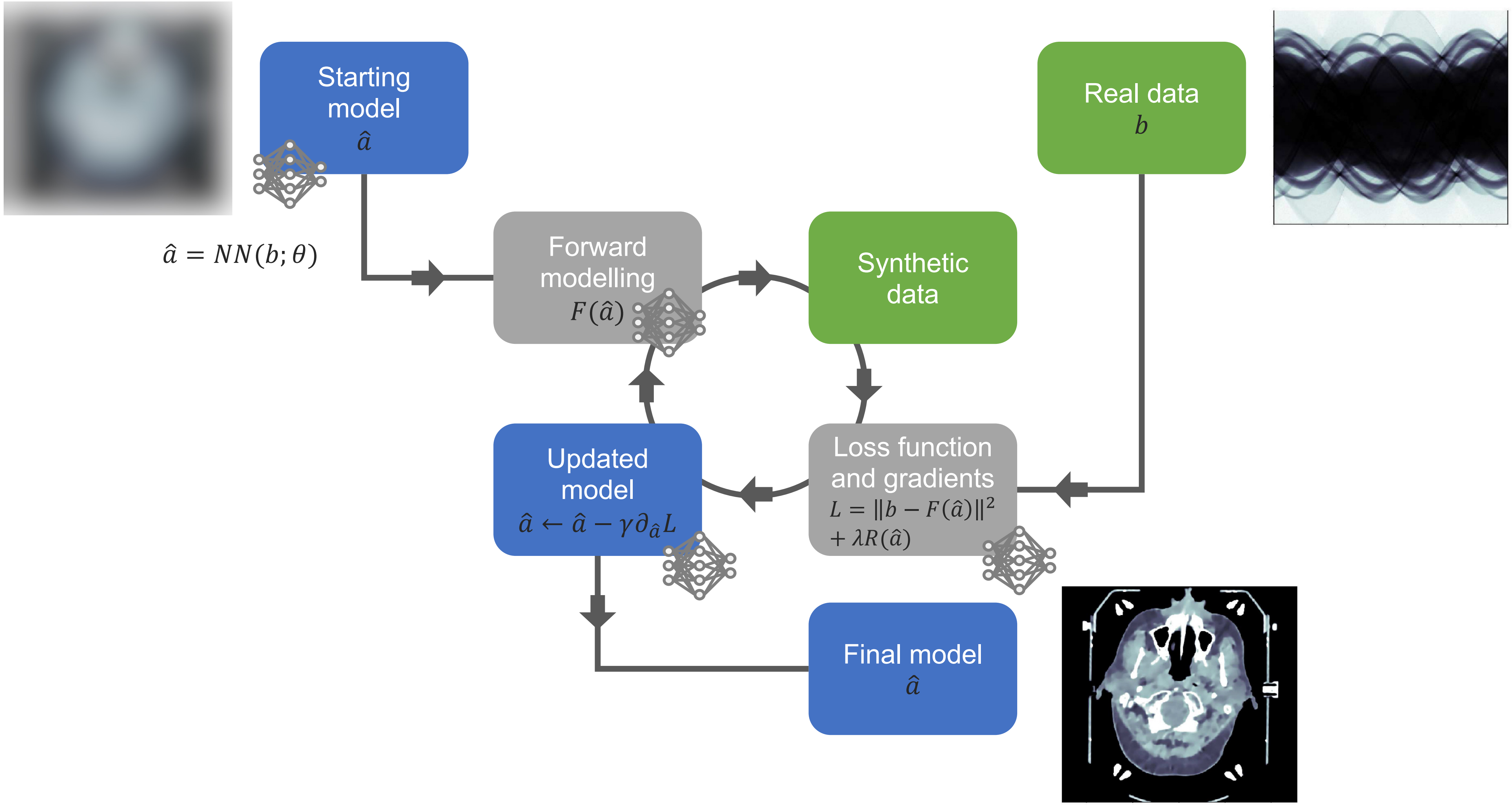
Adding even more flexibility

```python
def Hybrid2_X_ray_tomography(a_hat_0, b, theta):
    "Pseudocode for carrying out X ray tomography, with NN correction"

    # a_hat_0 is the initial image guess, of shape (NX, NY)
    # b are the observed measurements, of shape (MX, MY)

    a_hat = a_hat_0

    for i in range(0, n_steps):
        a_hat = a_hat.requires_grad_(True)
        b_hat = numerical_integrate(a_hat)
        R = total_variation(b_hat)
        loss = torch.mean((b-b_hat)**2) + theta[0]*R
        da = torch.autograd.grad(loss, a_hat)
        da = NN(da, a_hat, b_hat, R, theta[1])
        a_hat -= gamma*da

    return a_hat

# learn NN parameters
theta.requires_grad_(True)
for i in range(0, n_steps2):
    a, b = # train NN using many example inverse problems
    a_hat = Hybrid2_X_ray_tomography(a_hat_0, b, theta)
    loss = loss_fn(a, a_hat)
    dtheta = torch.autograd.grad(loss, theta)
    theta -= gamma*dtheta
```

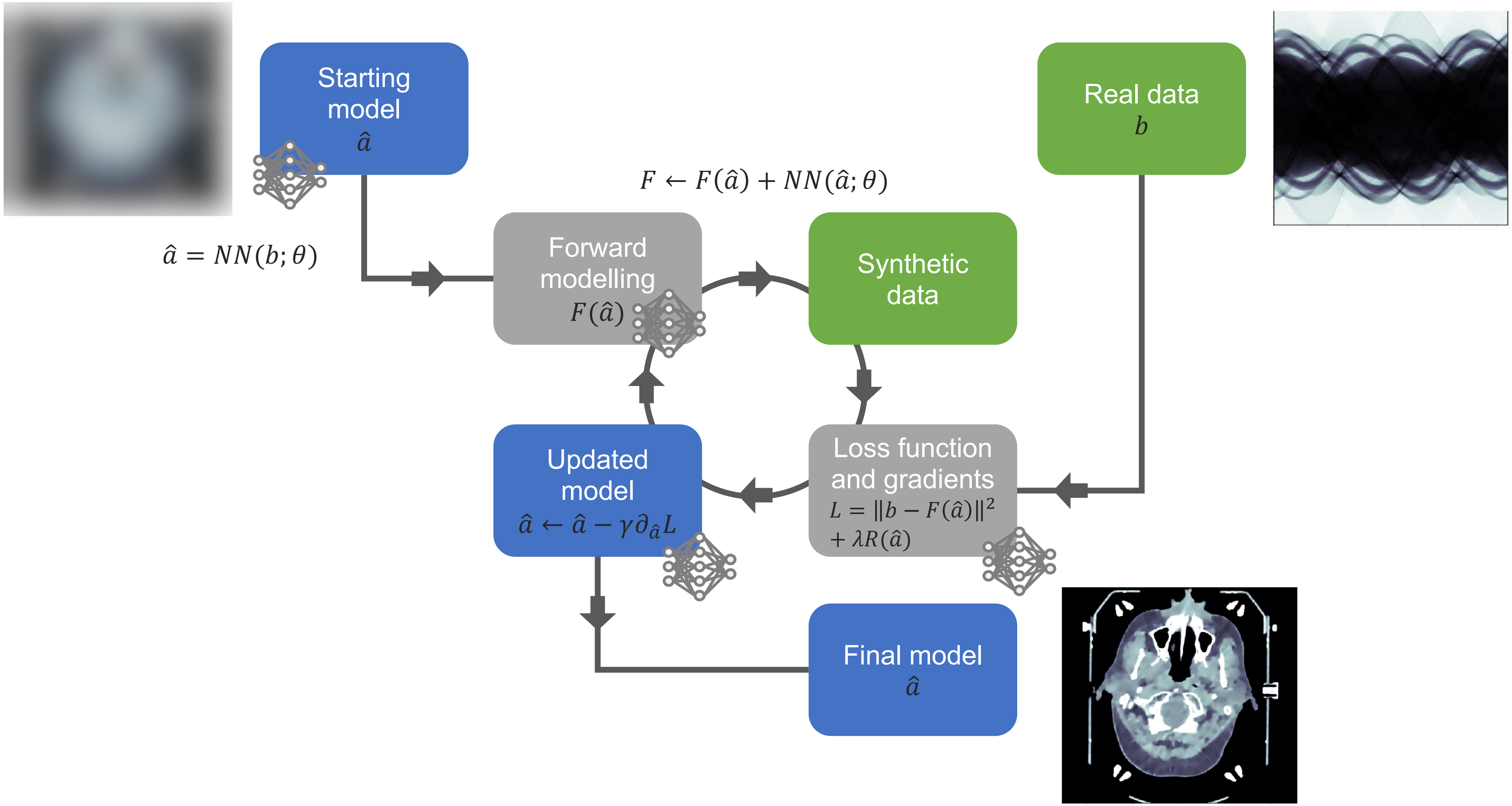
Key idea:
Traditional algorithms can be made as **learnable** (flexible) or as **unlearnable** (rigid) as you like

This allows you to balance the pros/cons of using NNs!

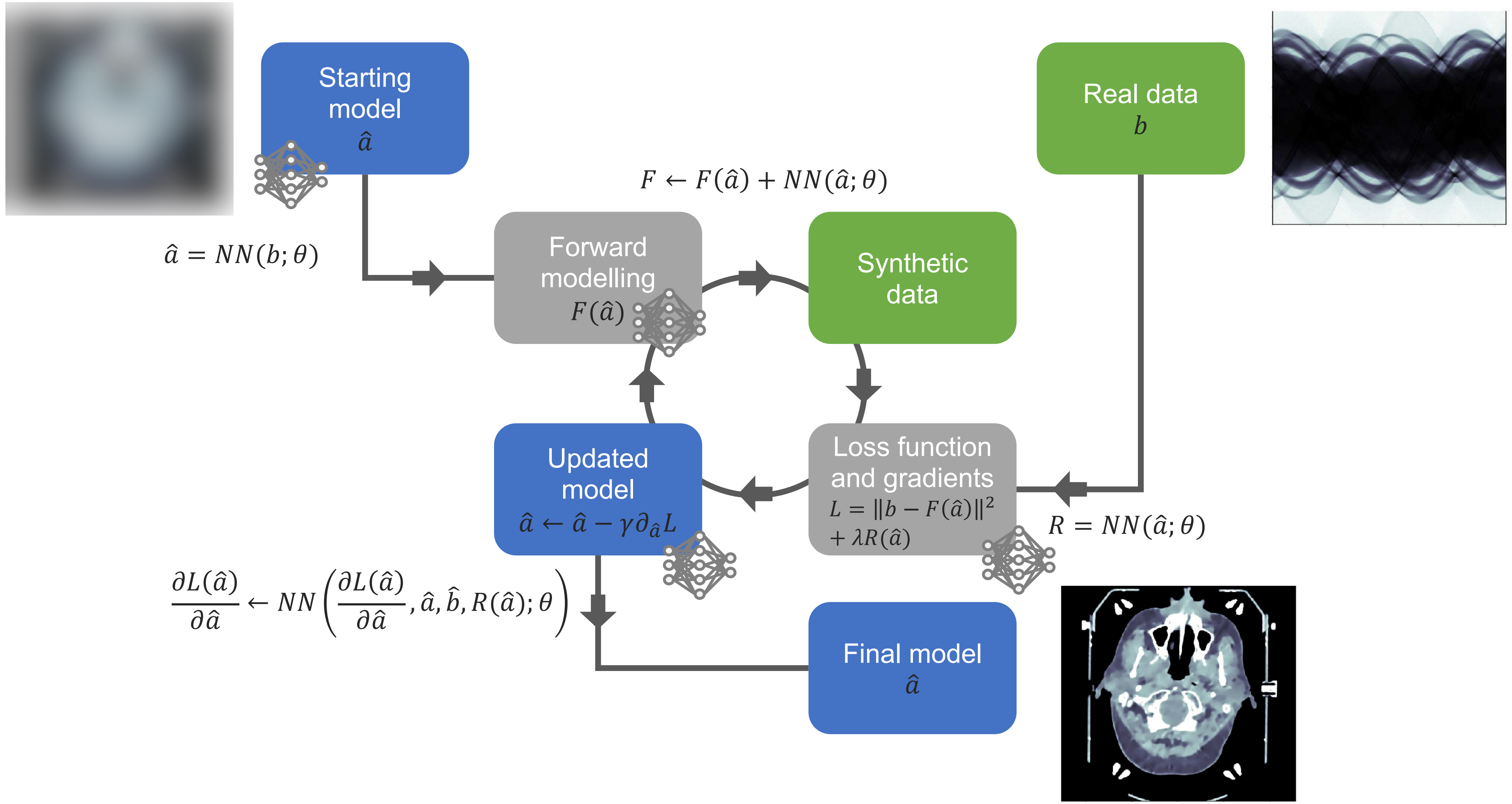
We can add learnable components everywhere!



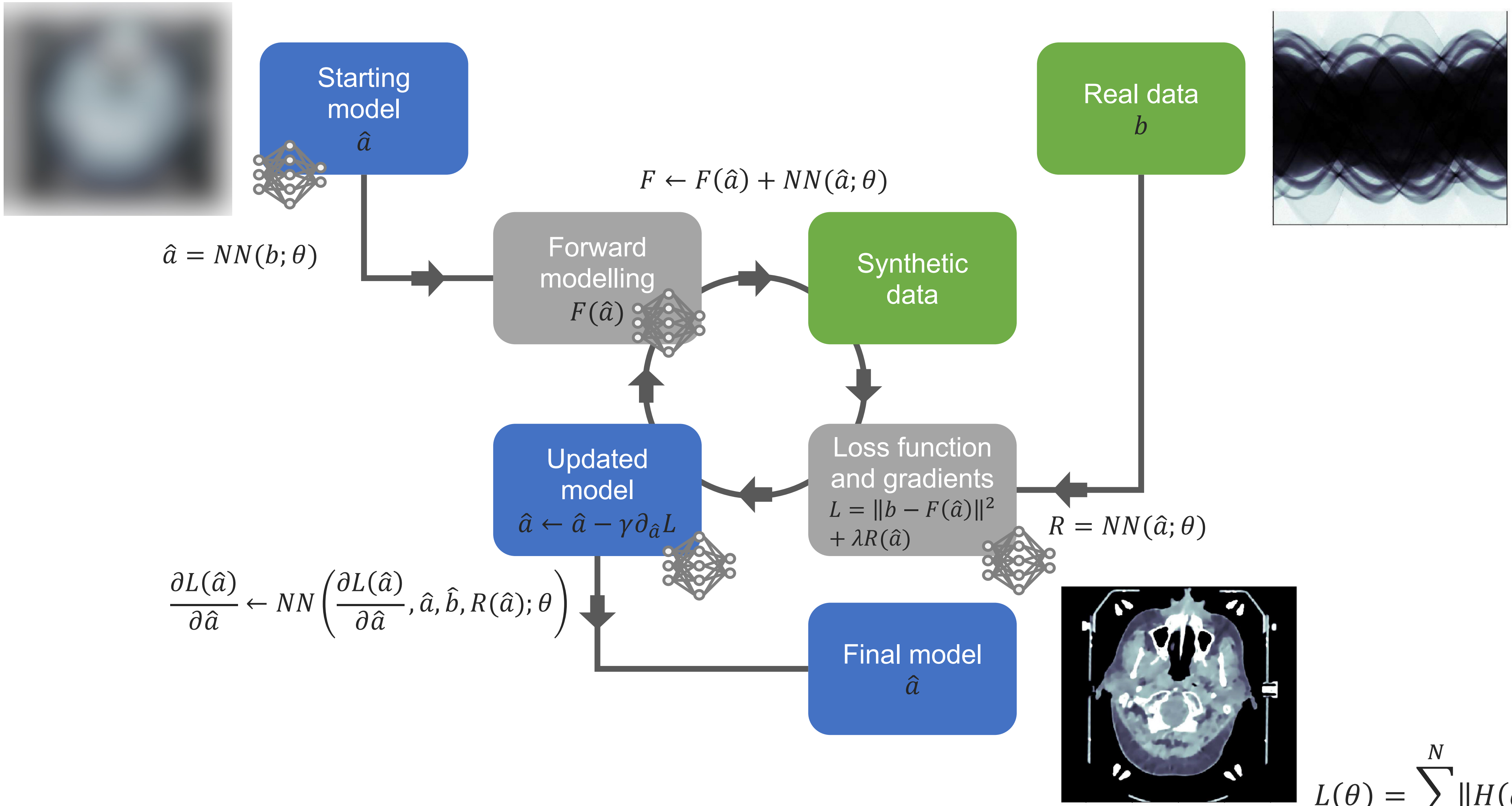
We can add learnable components everywhere!



We can add learnable components everywhere!



We can add learnable components everywhere!



$$L(\theta) = \sum_{i}^{N} \|H(\hat{a}_{0\,i}, b_i;\theta) - a_i\|^2$$

5 min break

Lecture overview

- Differentiable physics recap
- Coding a simple hybrid approach in PyTorch
- Hybrid approaches for inverse problems
- Neural differential equations (NDEs)
- Course summary

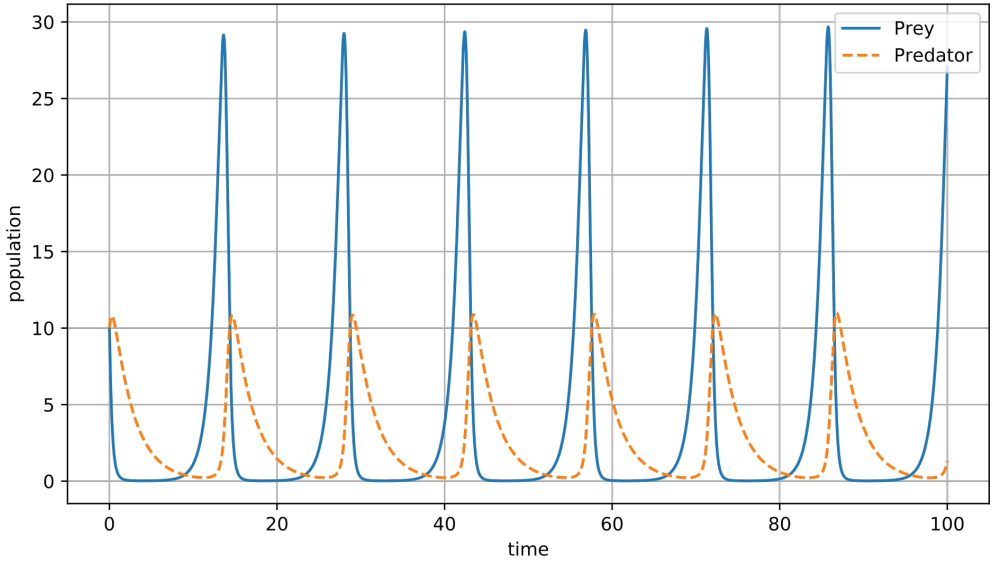
Ordinary differential equations

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}, t)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

For example, the Lotka-Volterra system:

$$\frac{dx}{dt} = \alpha x - \beta xy$$
$$\frac{dy}{dt} = \gamma xy - \delta y$$



Source: wikipedia

x = population density of prey
y = population density of predator
α, β = max prey birth rate, effect of predators on prey growth rate
δ, γ = max predator death rate, effect of prey on predator growth rate

$$\alpha, \beta = 1.1, 0.4$$
$$\delta, \gamma = 0.4, 0.1$$
$$x_0 = y_0 = 10$$

Ordinary differential equations

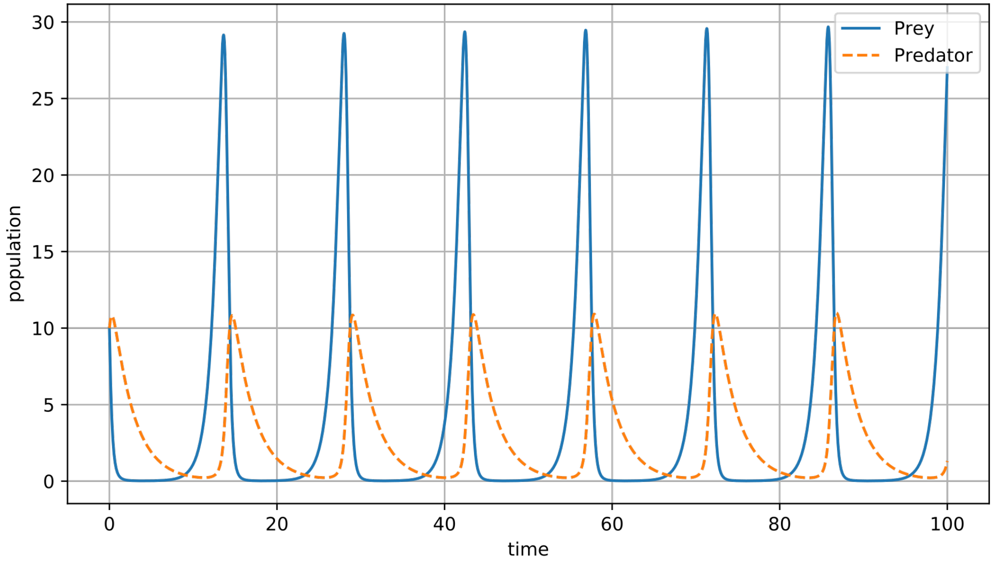
Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}, t)$$
$$\boldsymbol{x}(t = 0) = \boldsymbol{x}_0$$

For example, the Lotka-Volterra system:

$$\frac{dx}{dt} = \alpha x - \beta xy$$
$$\frac{dy}{dt} = \gamma xy - \delta y$$

- What if we are unsure of $\boldsymbol{f}(\boldsymbol{x}, t)$?



Source: wikipedia

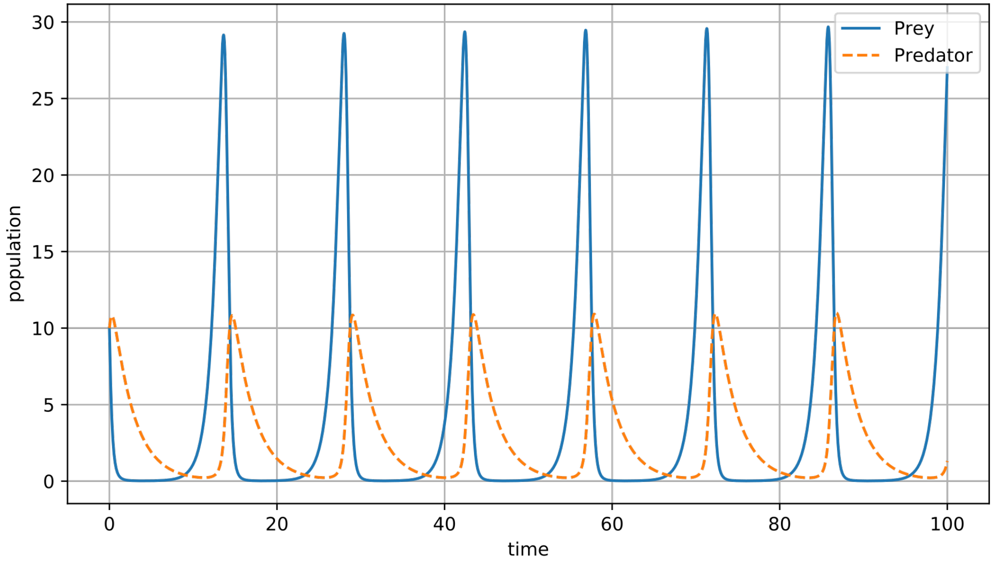
Neural differential equations

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}, t; \theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

For example, the Lotka-Volterra system:

$$\frac{dx}{dt} = \alpha x + NN(x, y; \theta_1)$$
$$\frac{dy}{dt} = NN(x, y; \theta_2) - \theta_3 y$$



Source: wikipedia

- What if we are unsure of $\boldsymbol{f}(\boldsymbol{x}, t)$?
- Use neural networks to represent uncertain parts

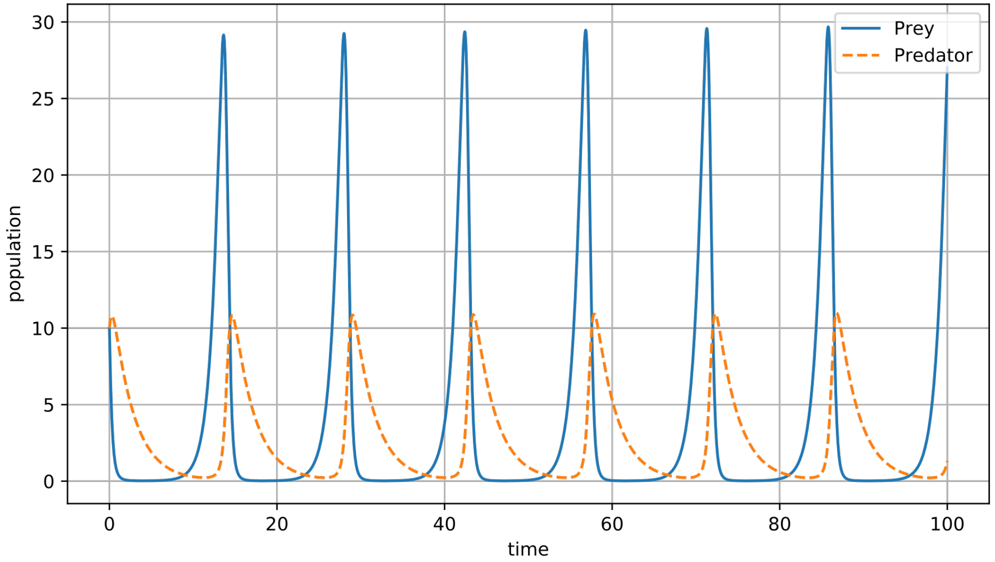
Neural differential equations

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}, t; \theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

For example, the Lotka-Volterra system:

$$\frac{dx}{dt} = \alpha x + NN(x, y; \theta_1)$$
$$\frac{dy}{dt} = NN(x, y; \theta_2) - \theta_3 y$$



Source: wikipedia

- What if we are unsure of $\boldsymbol{f}(\boldsymbol{x}, t)$?
- Use neural networks to represent uncertain parts
- This is known as a **neural differential equation**

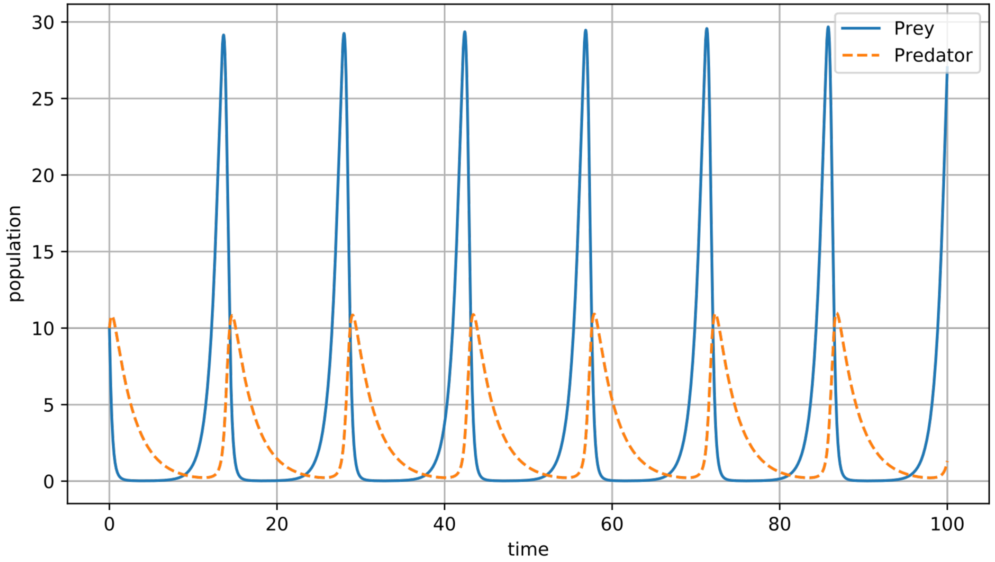
Neural differential equations

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}, t; \theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

For example, the Lotka-Volterra system:

$$\frac{dx}{dt} = \alpha x + NN(x, y; \theta_1)$$
$$\frac{dy}{dt} = NN(x, y; \theta_2) - \theta_3 y$$



Source: wikipedia

- What if we are unsure of $\boldsymbol{f}(\boldsymbol{x}, t)$?
- Use neural networks to represent uncertain parts
- This is known as a **neural differential equation**
- We can use a hybrid approach to **learn** the dynamics

Neural differential equations

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}, t; \theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

For example, the Lotka-Volterra system:

$$\frac{dx}{dt} = \alpha x + NN(x, y; \theta_1)$$
$$\frac{dy}{dt} = NN(x, y; \theta_2) - \theta_3 y$$

- What if we are unsure of $\boldsymbol{f}(\boldsymbol{x}, t)$?
- Use neural networks to represent uncertain parts
- This is known as a **neural differential equation**
- We can use a hybrid approach to **learn** the dynamics

A simple way of solving an ODE is to **discretise** in time and use the **Euler method**:
Given $\boldsymbol{x}_0, t_0, \delta t$:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \delta t \boldsymbol{f}(\boldsymbol{x}_i, t_i)$$
$$t_{i+1} = t_i + \delta t$$

For the Lotka-Volterra system:

$$x_{i+1} = x_i + \delta t(\alpha x_i - \beta x_i y_i)$$
$$y_{i+1} = y_i + \delta t(\gamma x_i y_i - \delta y_i)$$
$$t_{i+1} = t_i + \delta t$$

For the learnable Lotka-Volterra system:

$$x_{i+1} = x_i + \delta t(\alpha x_i + NN(x_i, y_i; \theta_1))$$
$$y_{i+1} = y_i + \delta t(NN(x_i, y_i; \theta_2) - \theta_3 y_i)$$
$$t_{i+1} = t_i + \delta t$$

Hybrid Lotka-Volterra solver

A simple way of solving an ODE is to **discretise** in time and use the **Euler method**:
Given $\boldsymbol{x}_0, t_0, \delta t$:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \delta t \boldsymbol{f}(\boldsymbol{x}_i, t_i)$$
$$t_{i+1} = t_i + \delta t$$

For the Lotka-Volterra system:

$$x_{i+1} = x_i + \delta t(\alpha x_i - \beta x_i y_i)$$
$$y_{i+1} = y_i + \delta t(\gamma x_i y_i - \delta y_i)$$
$$t_{i+1} = t_i + \delta t$$

For the learnable Lotka-Volterra system:

$$x_{i+1} = x_i + \delta t(\alpha x_i + NN(x_i, y_i; \theta_1))$$
$$y_{i+1} = y_i + \delta t(NN(x_i, y_i; \theta_2) - \theta_3 y_i)$$
$$t_{i+1} = t_i + \delta t$$

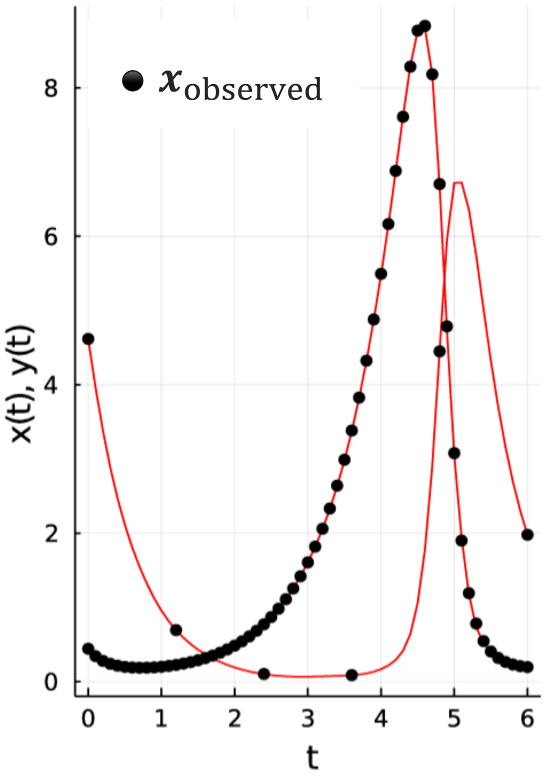
```python
def Hybrid_LV_Euler_solver(x0, y0, dt, theta):
    """Pseudocode for solving Lotka-Volterra system,
    with learnable dynamics"""

    x, y = x0, y0
    for t in range(0, T):
        x = x + dt*(alpha*x + NN(x, y, theta[0]))
        y = y + dt*(NN(x, y, theta[1]) - theta[2]*y)
    return x, y
```

Hybrid Lotka-Volterra solver

Train the hybrid solver using loss function:

$$L(\theta) = \sum_{i}^{T} \| \boldsymbol{x}_i(\boldsymbol{x}_0, t_0, \delta t, \theta) - \boldsymbol{x}_{\text{observed } i} \|^2$$

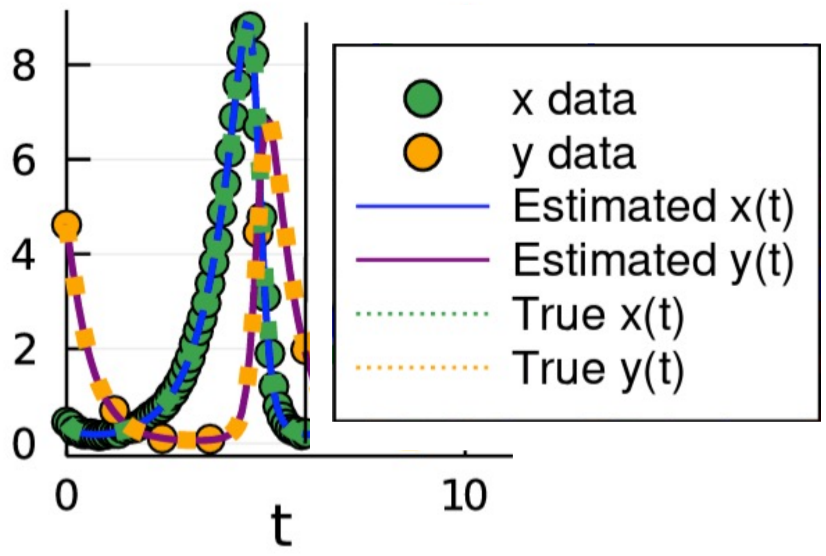
```python
def Hybrid_LV_Euler_solver(x0, y0, dt, theta):
    """Pseudocode for solving Lotka-Volterra system,
    with learnable dynamics"""

    x, y = x0, y0
    for t in range(0, T):
        x = x + dt*(alpha*x + NN(x, y, theta[0]))
        y = y + dt*(NN(x, y, theta[1]) - theta[2]*y)
    return x, y
```



Rackauckas et al, Universal differential equations for scientific machine learning, ArXiv (2021)

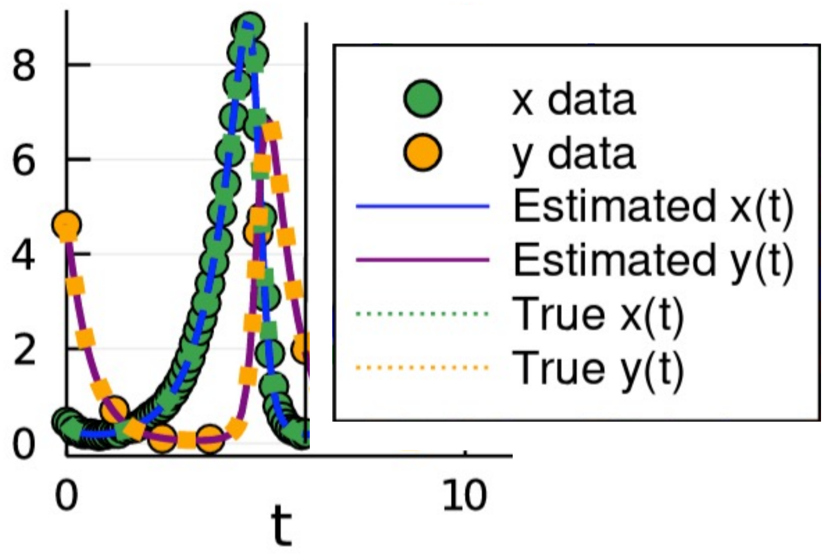
Hybrid Lotka-Volterra solver

Rackauckas et al, Universal differential equations for scientific machine learning, ArXiv (2021)



Hybrid Lotka-Volterra solver

Rackauckas et al, Universal differential equations for scientific machine learning, ArXiv (2021)

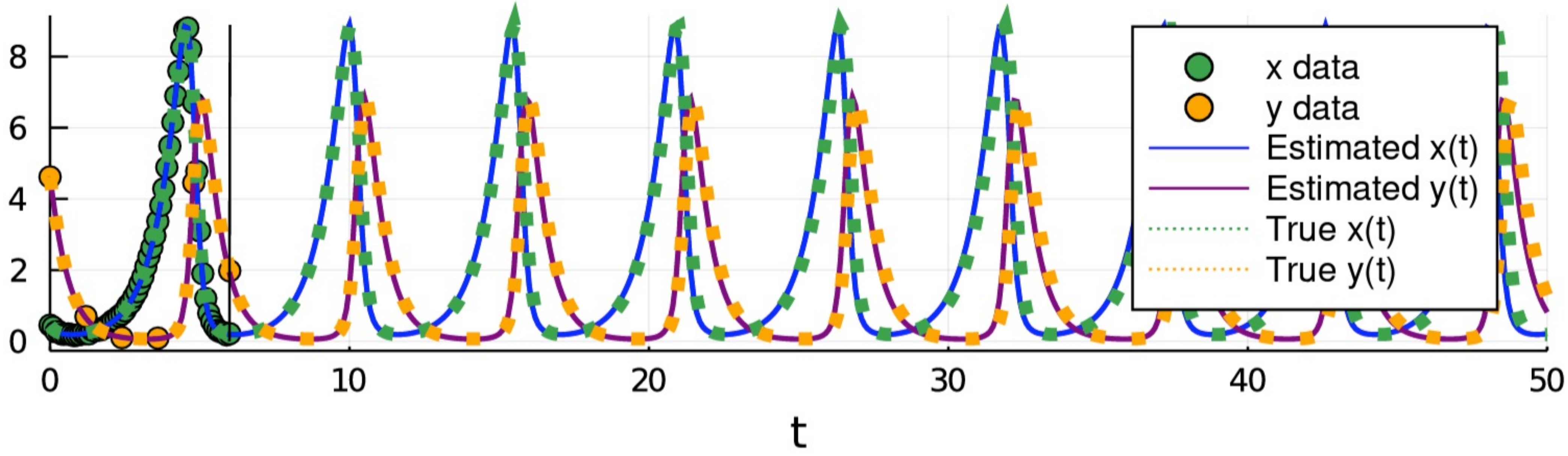


$$\frac{dx}{dt} = \alpha x + NN(x, y; \theta_1)$$
$$\frac{dy}{dt} = NN(x, y; \theta_2) - \theta_3 y$$

- Note, after training, we can do **symbolic regression** on $NN(x, y; \theta_1)$ and $NN(x, y; \theta_2)$ to "discover" their functional form, i.e. that $NN(x, y; \theta_1) \approx -\beta xy$

Hybrid Lotka-Volterra solver

Rackauckas et al, Universal differential equations for scientific machine learning, ArXiv (2021)

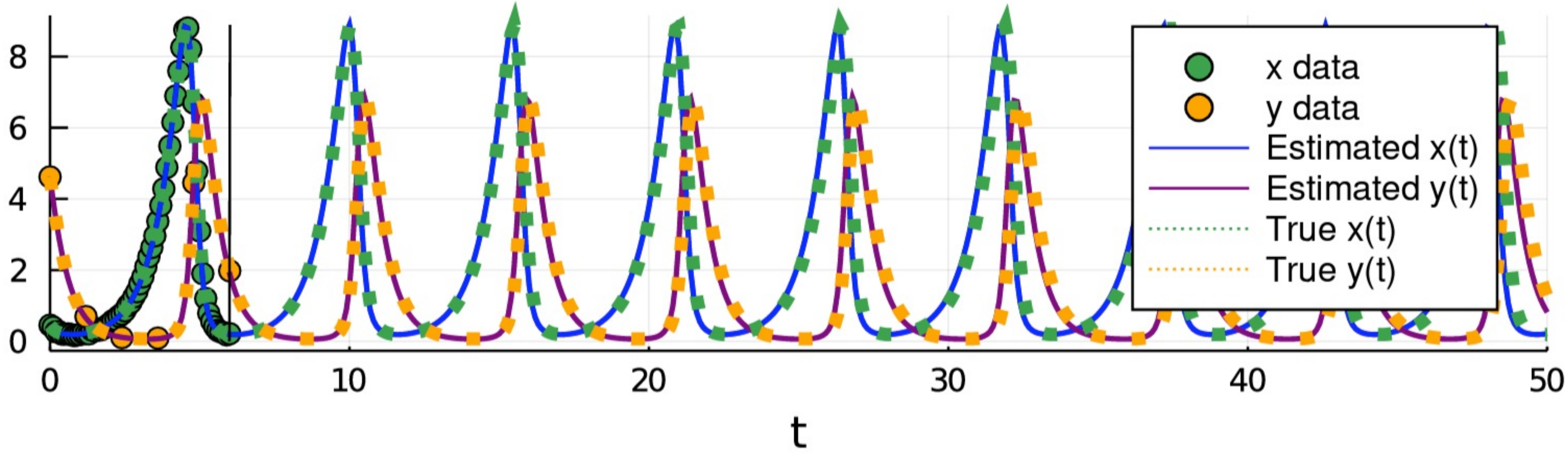


$$\frac{dx}{dt} = \alpha x + NN(x, y; \theta_1)$$
$$\frac{dy}{dt} = NN(x, y; \theta_2) - \theta_3 y$$

- This model generalizes well!

Hybrid Lotka-Volterra solver

Rackauckas et al, Universal differential equations for scientific machine learning, ArXiv (2021)



$$\frac{dx}{dt} = \alpha x + NN(x, y; \theta_1)$$
$$\frac{dy}{dt} = NN(x, y; \theta_2) - \theta_3 y$$

- This model generalizes well!

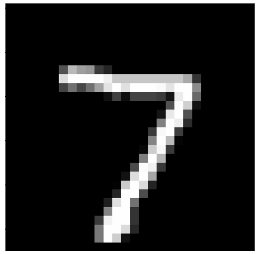
Idea: what if we use differential equations to model **any** dataset (not just physical systems)?

Neural differential equations

Input
$\boldsymbol{x}$

PDE solver
With learnable
parameters θ

Output
$\boldsymbol{y}$



$P(\boldsymbol{x} = 7)$

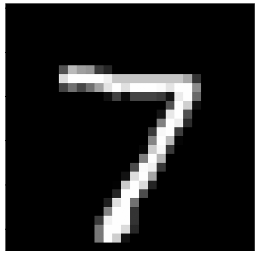
Idea: what if we use differential equations to model **any** dataset (not just physical systems)?

Neural differential equations

Input
$\boldsymbol{x}$

PDE solver
With learnable
parameters θ

Output
$\boldsymbol{y}$



$P(\boldsymbol{x} = 7)$

Idea: what if we use differential equations to model **any** dataset (not just physical systems)?

- We can think of the PDE solver as a “custom” NN architecture

Neural ordinary differential equations

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x};\theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

Solver using Euler method:

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

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \delta t \boldsymbol{f}(\boldsymbol{x}_i;\theta)$$

Neural ordinary differential equations

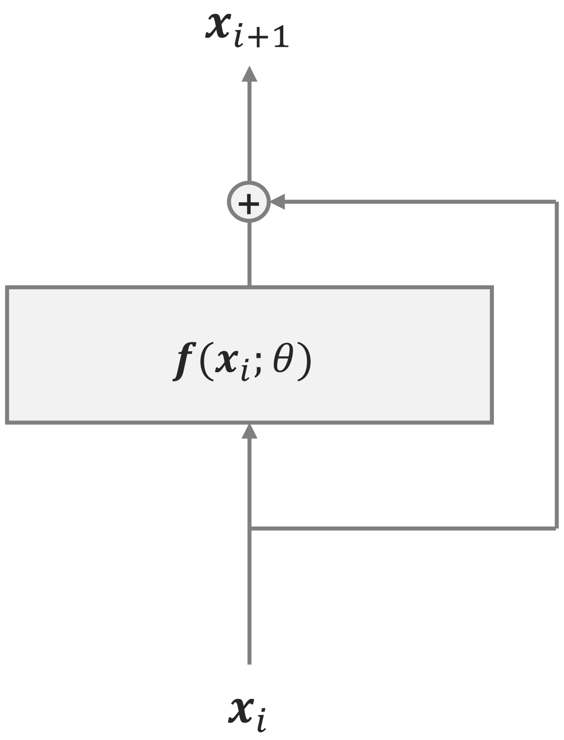
Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}; \theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

Solver using Euler method:

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

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \delta t \boldsymbol{f}(\boldsymbol{x}_i; \theta)$$



Neural ordinary differential equations

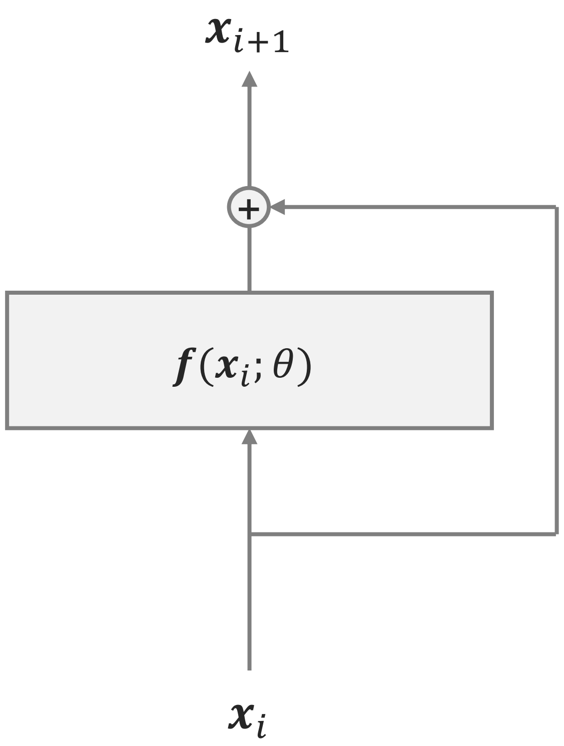
Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x};\theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

Solver using Euler method:

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

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \delta t \boldsymbol{f}(\boldsymbol{x}_i;\theta)$$



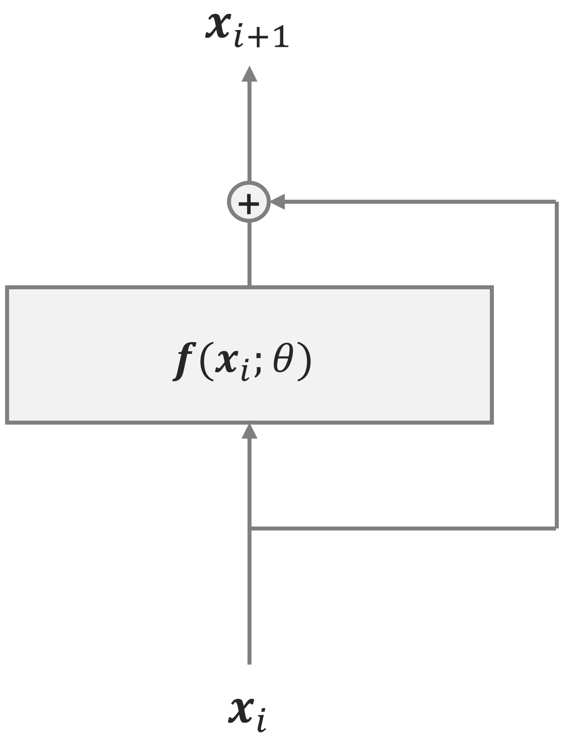
This is **identical** to the **residual layer** used in standard residual networks (ResNets)!

ResNets are Euler solvers

I.e., ResNets are Euler solvers!

$\Rightarrow$ In the **limit** of an infinite numbers of layers (i.e. as $\delta t \to 0$), a ResNet computes the solution to

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}, \theta)$$



This is **identical** to the **residual layer** used in standard residual networks (ResNets)!

Higher-order solvers

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}; \theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

We are not limited to Euler solvers! Many **other** solvers could be used, for example higher-order Runge-Kutta methods, e.g. RK4:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \frac{\delta t}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4)$$
$$\boldsymbol{k}_1 = \boldsymbol{f}(\boldsymbol{x}_i; \theta)$$
$$\boldsymbol{k}_2 = \boldsymbol{f}\left(\boldsymbol{x}_i + \frac{\delta t}{2}\boldsymbol{k}_1; \theta\right)$$
$$\boldsymbol{k}_3 = \boldsymbol{f}\left(\boldsymbol{x}_i + \frac{\delta t}{2}\boldsymbol{k}_2; \theta\right)$$
$$\boldsymbol{k}_4 = \boldsymbol{f}(\boldsymbol{x}_i + \delta t\boldsymbol{k}_3; \theta)$$

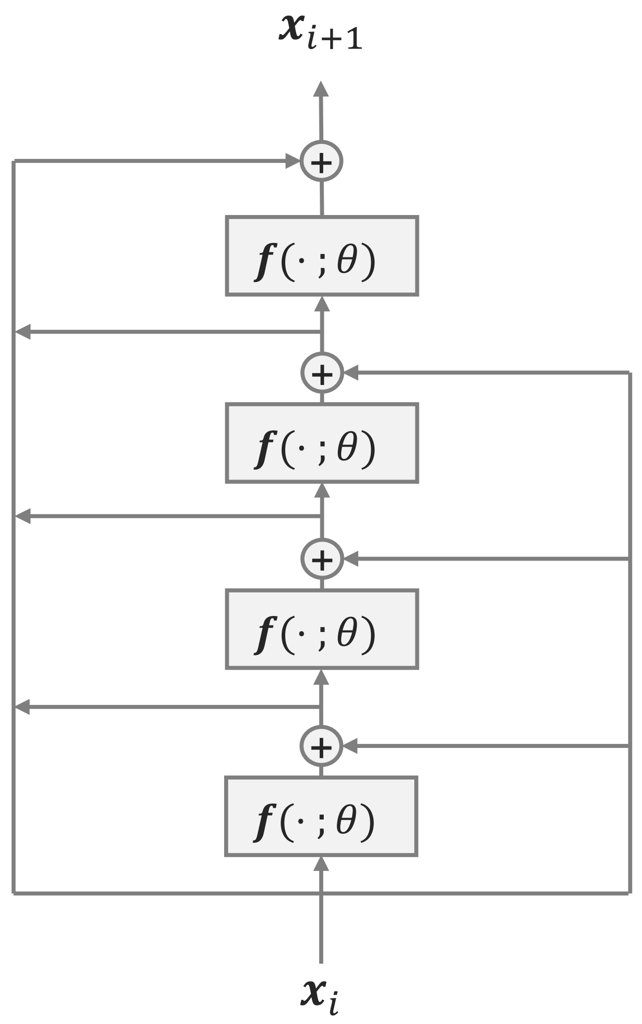
Higher-order solvers

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}; \theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

We are not limited to Euler solvers! Many **other** solvers could be used, for example higher-order Runge-Kutta methods, e.g. RK4:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \frac{\delta t}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4)$$
$$\boldsymbol{k}_1 = \boldsymbol{f}(\boldsymbol{x}_i; \theta)$$
$$\boldsymbol{k}_2 = \boldsymbol{f}\left(\boldsymbol{x}_i + \frac{\delta t}{2}\boldsymbol{k}_1; \theta\right)$$
$$\boldsymbol{k}_3 = \boldsymbol{f}\left(\boldsymbol{x}_i + \frac{\delta t}{2}\boldsymbol{k}_2; \theta\right)$$
$$\boldsymbol{k}_4 = \boldsymbol{f}(\boldsymbol{x}_i + \delta t\boldsymbol{k}_3; \theta)$$



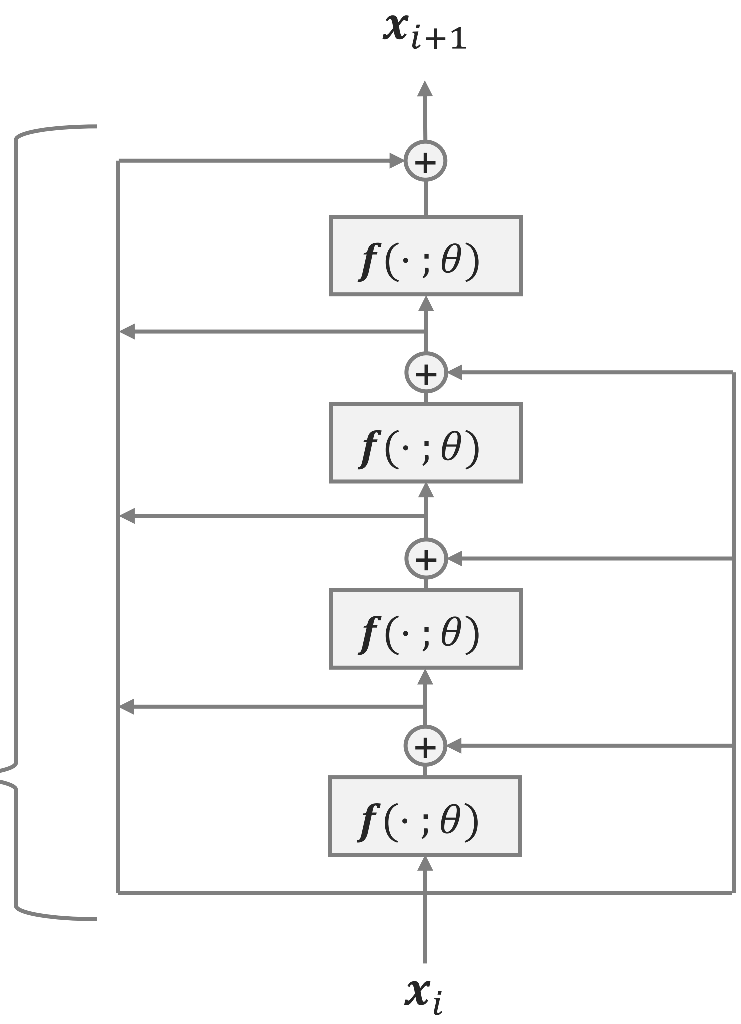
Higher-order solvers

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}; \theta)$$
$$\boldsymbol{x}(t = 0) = \boldsymbol{x}_0$$

We are not limited to Euler solvers! Many **other** solvers could be used, for example higher-order Runge-Kutta methods, e.g. RK4:

“Custom” residual block



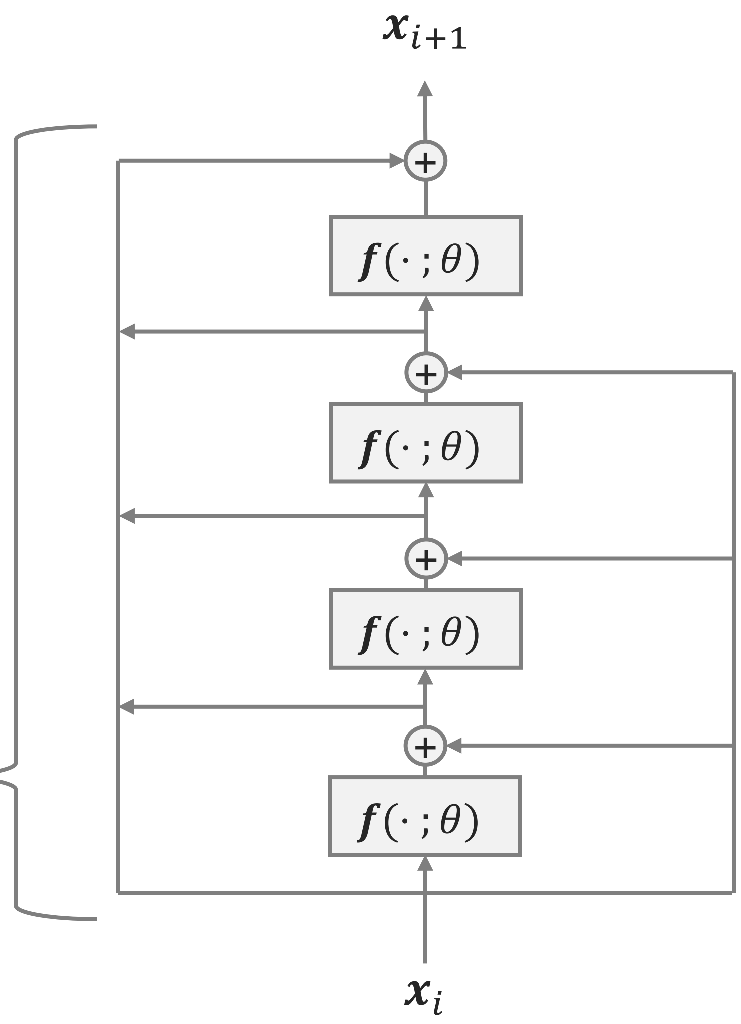
Higher-order solvers

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}; \theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

We are not limited to Euler solvers! Many **other** solvers could be used, for example higher-order Runge-Kutta methods, e.g. RK4:

“Custom” residual block



=> Other solvers define other NN architectures

Higher-order solvers

Consider solving an **ordinary differential equation** (ODE):

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}; \theta)$$
$$\boldsymbol{x}(t=0) = \boldsymbol{x}_0$$

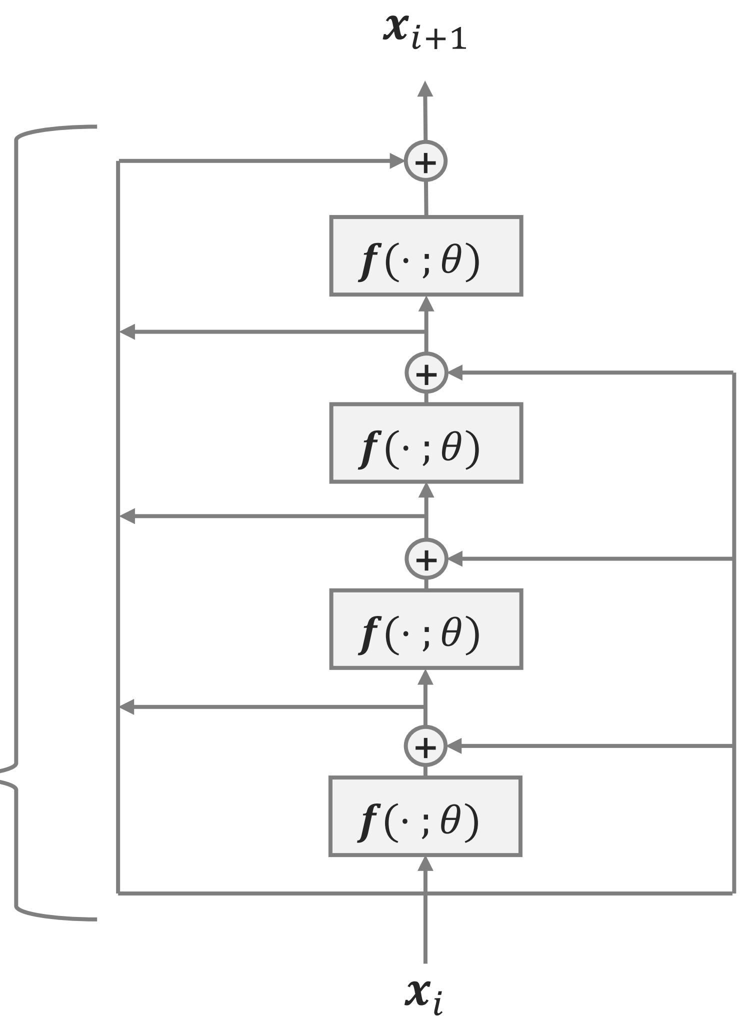
We are not limited to Euler solvers! Many **other** solvers could be used, for example higher-order Runge-Kutta methods, e.g. RK4:

	Test Error
1-Layer MLP†	1.60%
ResNet	0.41%
RK-Net	0.47%

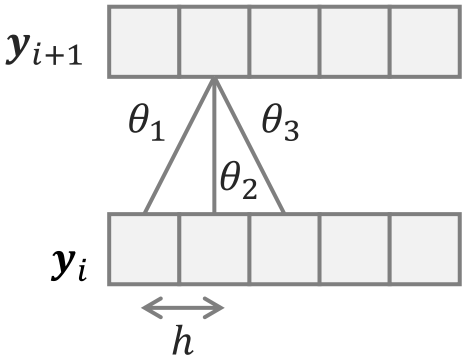
Performance on MNIST (digit classification)

Chen et al, Neural ordinary differential equations, NeurIPS (2018)

“Custom” residual block



Neural differential equations

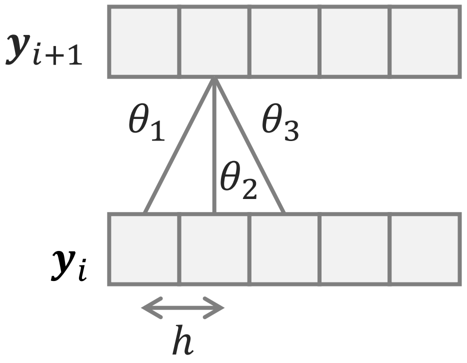


Consider a 1D convolutional layer:

$$\begin{aligned}\boldsymbol{y}_{i+1} &= \boldsymbol{\theta} \star \boldsymbol{y}_i \\ &= (\theta_1 \quad \theta_2 \quad \theta_3) \star \boldsymbol{y}_i\end{aligned}$$

Ruthotto and Haber, Deep Neural Networks Motivated by Partial Differential Equations, Journal of Mathematical Imaging and Vision (2019)

Neural differential equations



Consider a 1D convolutional layer:

$$
\begin{aligned}
&\boldsymbol{y}_{i+1} = \boldsymbol{\theta} \star \boldsymbol{y}_i \\
&= (\theta_1 \quad \theta_2 \quad \theta_3) \star \boldsymbol{y}_i
\end{aligned}
$$

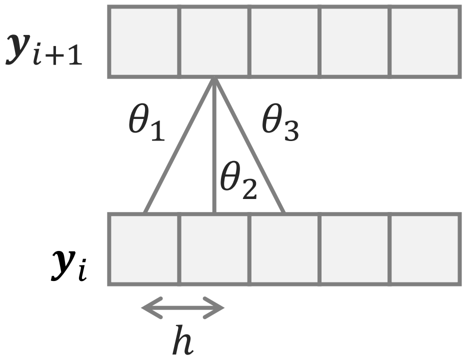
Let us **transform** $\boldsymbol{\theta}$ to a new vector $\boldsymbol{\beta}(\boldsymbol{\theta})$ which is (uniquely) given by

$$
\begin{pmatrix}
\frac{1}{4} & -\frac{1}{2h} & -\frac{1}{h^2} \\
\frac{1}{2} & 0 & \frac{2}{h^2} \\
\frac{1}{4} & \frac{1}{2h} & -\frac{1}{h^2}
\end{pmatrix}
\begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix}
=
\begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix}
$$

For some $h > 0$.

Ruthotto and Haber, Deep Neural Networks Motivated by Partial Differential Equations, Journal of Mathematical Imaging and Vision (2019)

Neural differential equations



Consider a 1D convolutional layer:

$$\begin{aligned}\boldsymbol{y}_{i+1} &= \boldsymbol{\theta} \star \boldsymbol{y}_i \\ &= (\theta_1 \quad \theta_2 \quad \theta_3) \star \boldsymbol{y}_i\end{aligned}$$

Let us **transform** $\boldsymbol{\theta}$ to a new vector $\boldsymbol{\beta}(\boldsymbol{\theta})$ which is (uniquely) given by

$$\begin{pmatrix} \frac{1}{4} & -\frac{1}{2h} & -\frac{1}{h^2} \\ \frac{1}{2} & 0 & \frac{2}{h^2} \\ \frac{1}{4} & \frac{1}{2h} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix}$$

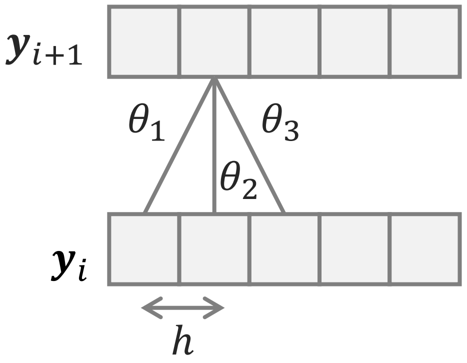
For some $h > 0$.

Then we can **re-write** the convolutional layer as

$$\boldsymbol{y}_{i+1} = \left(\frac{\beta_1(\boldsymbol{\theta})}{4} (1 \quad 2 \quad 1) + \frac{\beta_2(\boldsymbol{\theta})}{2h} (-1 \quad 0 \quad 1) + \frac{\beta_3(\boldsymbol{\theta})}{h^2} (-1 \quad 2 \quad -1) \right) \star \boldsymbol{y}_i$$

Ruthotto and Haber, Deep Neural Networks Motivated by Partial Differential Equations, Journal of Mathematical Imaging and Vision (2019)

Neural differential equations



Consider a 1D convolutional layer:

$$\begin{aligned}\boldsymbol{y}_{i+1} &= \boldsymbol{\theta} \star \boldsymbol{y}_i \\ &= (\theta_1 \quad \theta_2 \quad \theta_3) \star \boldsymbol{y}_i\end{aligned}$$

Let us **transform** $\boldsymbol{\theta}$ to a new vector $\boldsymbol{\beta}(\boldsymbol{\theta})$ which is (uniquely) given by

$$\begin{pmatrix} \frac{1}{4} & -\frac{1}{2h} & -\frac{1}{h^2} \\ \frac{1}{2} & 0 & \frac{2}{h^2} \\ \frac{1}{4} & \frac{1}{2h} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix}$$

For some $h > 0$.

Then we can **re-write** the convolutional layer as

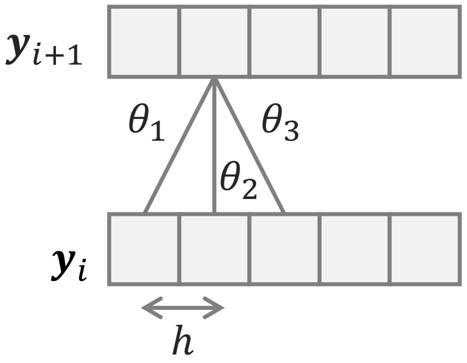
$$\boldsymbol{y}_{i+1} = \left(\frac{\beta_1(\boldsymbol{\theta})}{4} (1 \quad 2 \quad 1) + \frac{\beta_2(\boldsymbol{\theta})}{2h} (-1 \quad 0 \quad 1) + \frac{\beta_3(\boldsymbol{\theta})}{h^2} (-1 \quad 2 \quad -1) \right) \star \boldsymbol{y}_i$$

In the **limit** $h \to 0$,

$$y_{i+1} = \beta_1(\boldsymbol{\theta}) y_i + \beta_2(\boldsymbol{\theta}) \frac{\partial y_i}{\partial x} + \beta_3(\boldsymbol{\theta}) \frac{\partial^2 y_i}{\partial x^2}$$

Ruthotto and Haber, Deep Neural Networks Motivated by Partial Differential Equations, Journal of Mathematical Imaging and Vision (2019)

Neural differential equations



Consider a 1D convolutional layer:

$$\begin{aligned}\boldsymbol{y}_{i+1} &= \boldsymbol{\theta} \star \boldsymbol{y}_i \\ &= (\theta_1 \quad \theta_2 \quad \theta_3) \star \boldsymbol{y}_i\end{aligned}$$

Let us **transform** $\boldsymbol{\theta}$ to a new vector $\boldsymbol{\beta}(\boldsymbol{\theta})$ which is (uniquely) given by

$$\begin{pmatrix} \frac{1}{4} & -\frac{1}{2h} & -\frac{1}{h^2} \\ \frac{1}{2} & 0 & \frac{2}{h^2} \\ \frac{1}{4} & \frac{1}{2h} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix}$$

For some $h > 0$.

Then we can **re-write** the convolutional layer as

$$\boldsymbol{y}_{i+1} = \left(\frac{\beta_1(\boldsymbol{\theta})}{4} (1 \quad 2 \quad 1) + \frac{\beta_2(\boldsymbol{\theta})}{2h} (-1 \quad 0 \quad 1) + \frac{\beta_3(\boldsymbol{\theta})}{h^2} (-1 \quad 2 \quad -1) \right) \star \boldsymbol{y}_i$$

In the **limit** $h \to 0$,

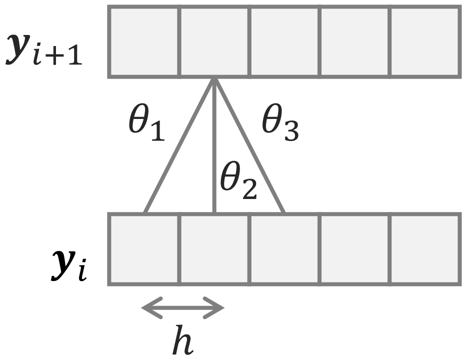
$$y_{i+1} = \beta_1(\boldsymbol{\theta}) y_i + \beta_2(\boldsymbol{\theta}) \frac{\partial y_i}{\partial x} + \beta_3(\boldsymbol{\theta}) \frac{\partial^2 y_i}{\partial x^2}$$

Consider a residual CNN, then

$$y_{i+1} = y_i + \beta_1(\boldsymbol{\theta}) y_i + \beta_2(\boldsymbol{\theta}) \frac{\partial y_i}{\partial x} + \beta_3(\boldsymbol{\theta}) \frac{\partial^2 y_i}{\partial x^2}$$

Ruthotto and Haber, Deep Neural Networks Motivated by Partial Differential Equations, Journal of Mathematical Imaging and Vision (2019)

Neural differential equations



Consider a 1D convolutional layer:

$$\begin{aligned}\boldsymbol{y}_{i+1} &= \boldsymbol{\theta} \star \boldsymbol{y}_i \\ &= (\theta_1 \quad \theta_2 \quad \theta_3) \star \boldsymbol{y}_i\end{aligned}$$

Let us **transform** $\boldsymbol{\theta}$ to a new vector $\boldsymbol{\beta}(\boldsymbol{\theta})$ which is (uniquely) given by

$$\begin{pmatrix} \frac{1}{4} & -\frac{1}{2h} & -\frac{1}{h^2} \\ \frac{1}{2} & 0 & \frac{2}{h^2} \\ \frac{1}{4} & \frac{1}{2h} & -\frac{1}{h^2} \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix}$$

For some $h > 0$.

Then we can **re-write** the convolutional layer as

$$\boldsymbol{y}_{i+1} = \left(\frac{\beta_1(\boldsymbol{\theta})}{4}(1 \quad 2 \quad 1) + \frac{\beta_2(\boldsymbol{\theta})}{2h}(-1 \quad 0 \quad 1) + \frac{\beta_3(\boldsymbol{\theta})}{h^2}(-1 \quad 2 \quad -1) \right) \star \boldsymbol{y}_i$$

In the **limit** $h \to 0$,

$$y_{i+1} = \beta_1(\boldsymbol{\theta}) y_i + \beta_2(\boldsymbol{\theta}) \frac{\partial y_i}{\partial x} + \beta_3(\boldsymbol{\theta}) \frac{\partial^2 y_i}{\partial x^2}$$

Consider a residual CNN, then

$$y_{i+1} = y_i + \beta_1(\boldsymbol{\theta}) y_i + \beta_2(\boldsymbol{\theta}) \frac{\partial y_i}{\partial x} + \beta_3(\boldsymbol{\theta}) \frac{\partial^2 y_i}{\partial x^2}$$

In the **limit** of infinite layers, the residual CNN solves

$$\frac{\partial y}{\partial t} = \beta_1(\boldsymbol{\theta}) y + \beta_2(\boldsymbol{\theta}) \frac{\partial y}{\partial x} + \beta_3(\boldsymbol{\theta}) \frac{\partial^2 y}{\partial x^2}$$

Ruthotto and Haber, Deep Neural Networks Motivated by Partial Differential Equations, Journal of Mathematical Imaging and Vision (2019)

Neural differential equations

Neural network architectures $\Longleftrightarrow$ Differential equation solvers

Understanding of architectures / training algorithms $\Longleftrightarrow$ Understanding of PDEs / their solutions

Lecture overview

- Differentiable physics recap
- Coding a simple hybrid approach in PyTorch
- Hybrid approaches for inverse problems
- Neural differential equations (NDEs)
- Course summary

Scientific machine learning (SciML)

Major problem

Despite big breakthroughs in science + AI

Naively using deep learning for scientific tasks usually leads to:

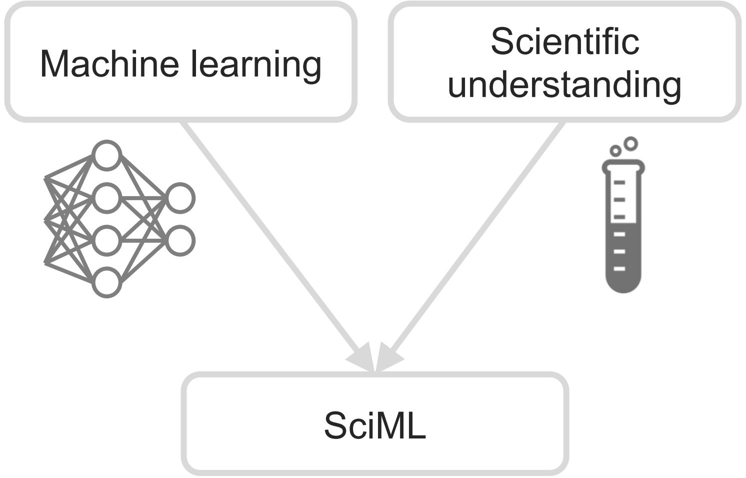
- Lack of interpretability
- Poor generalisation
- Lots of training data required

Do neural networks really "**understand**" the scientific tasks they are being applied to?

Traditional scientific method:

- Revolves around theory and experiment
- a good theory should be explainable and make **novel** predictions

Solution



more **powerful, robust, interpretable** models

General trends in SciML

- Incorporating scientific understanding nearly always **improves** the performance of ML algorithms
- SciML approaches can be as flexible (learnable) or as inflexible (unlearnable) as necessary
- There are a plethora of SciML approaches; chose the one which **suits** your problem
- SciML approaches **still** suffer from the limitations of deep neural networks (generalisation, lack of interpretability, scalability, …)
- SciML approaches can be applied to:
  - **many** different problems (simulation, inversion, data assimilation, control, equation discovery, …)
  - **many** different fields
- SciML requires truly **interdisciplinary** research

Course learning objectives

- Aware of advanced **applications** of deep learning in scientific computing
- Familiar with the **design**, **implementation** and **theory** of these algorithms
- Understand the **pros/cons** of using deep learning
- Understand key scientific machine learning **concepts** and themes

Thank you!