AI in the Sciences and Engineering

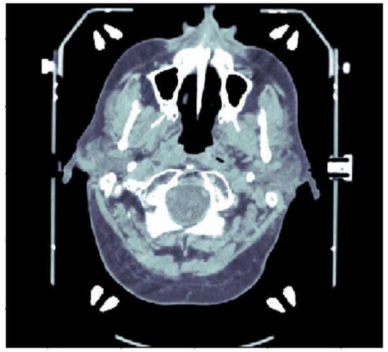
Neural Differential Equations

Spring Semester 2024

Siddhartha Mishra
Ben Moseley

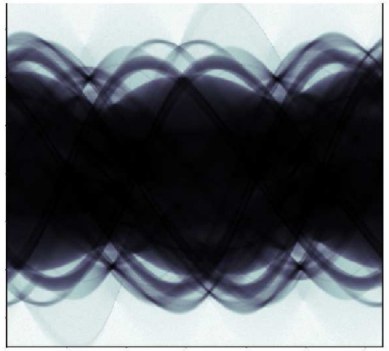
ETH zürich

Recap - computed tomography



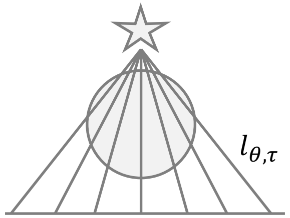
Ground truth computed tomography image

$a(x, y)$



Resulting tomographic data (sinogram)

$$b(\theta, \tau) = F(a) = I_0 e^{-\int_{l_{\theta,\tau}} a(x,y)\, ds}$$



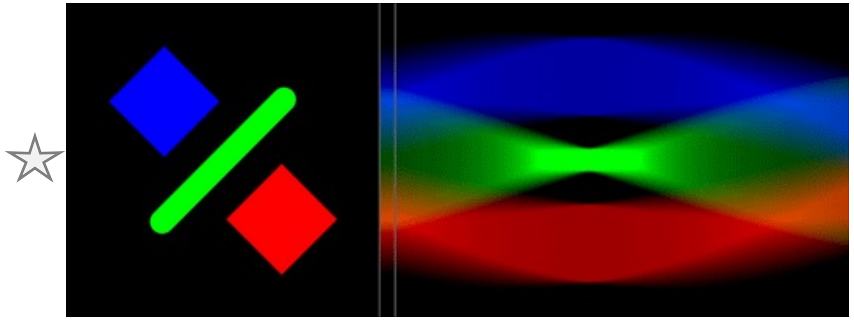


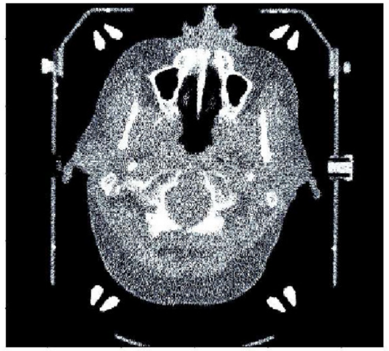
Image source: Wikipedia

$$b = F(a)$$

a = set of input conditions

F = physical model of the system

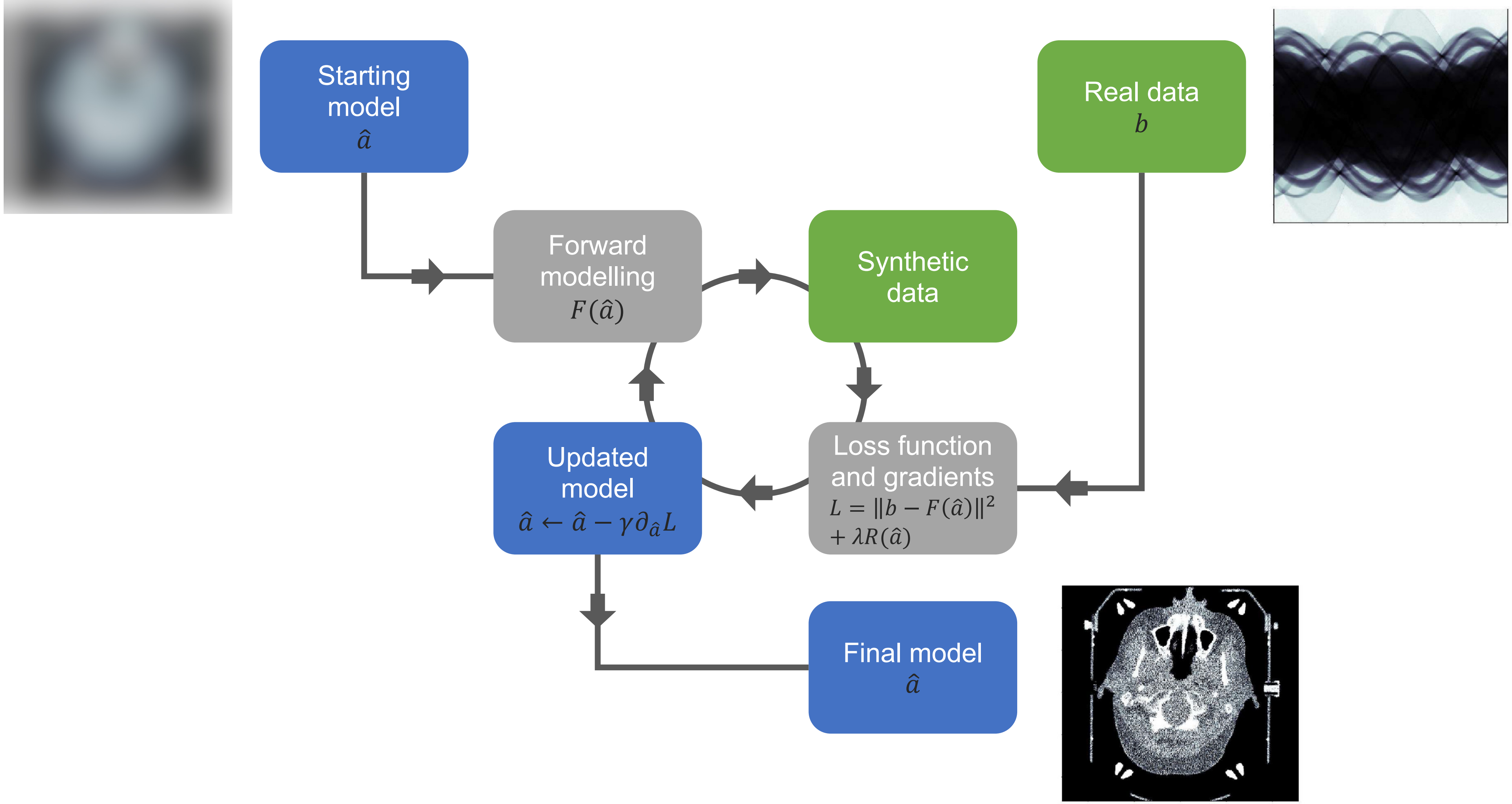
b = resulting properties given F and a



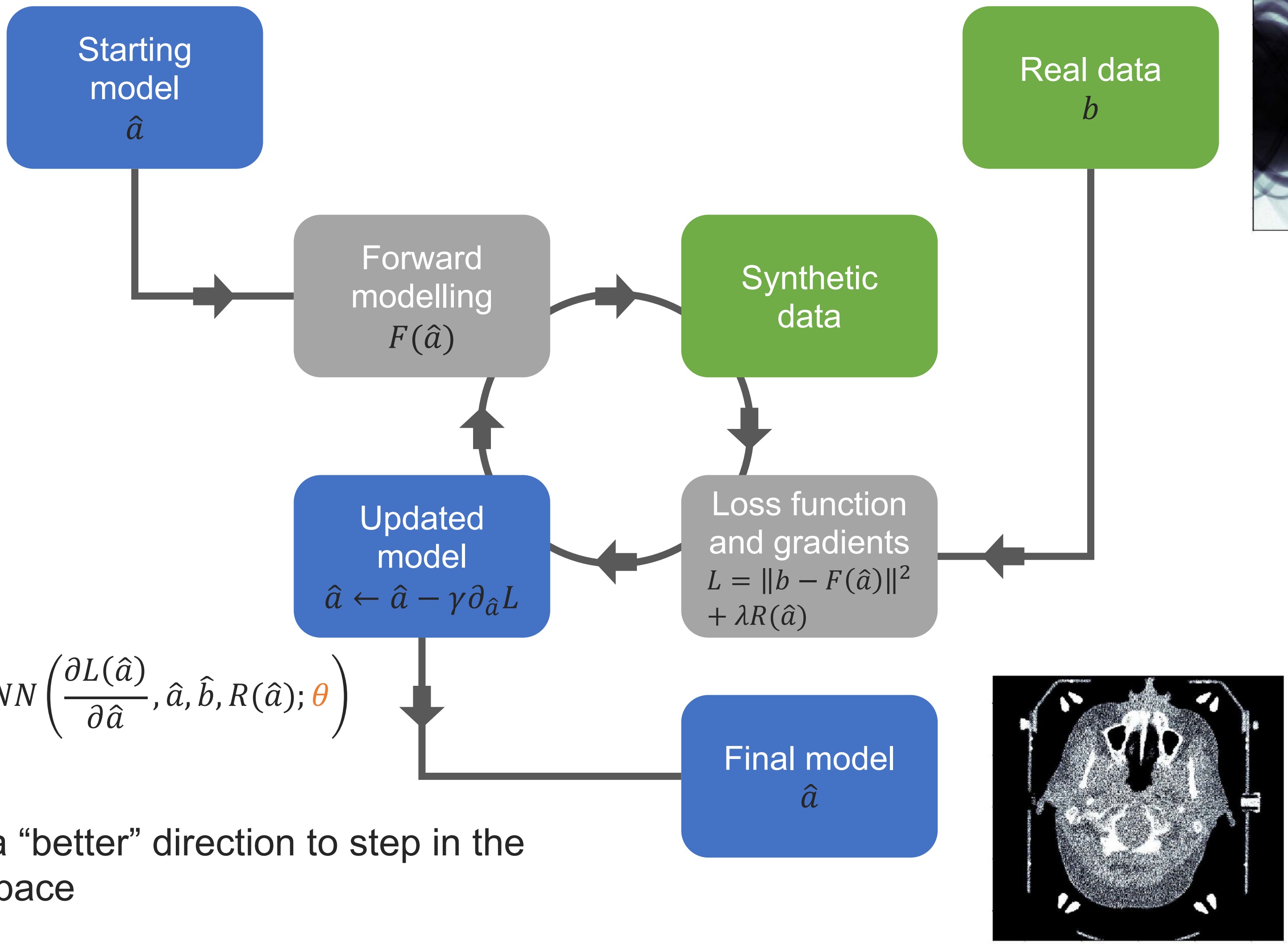
Result of inverse algorithm

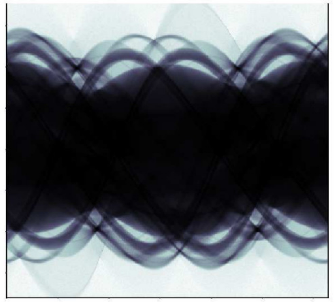
$\hat{a}$

Recap - solving the inverse problem



Recap - hybrid computed tomography



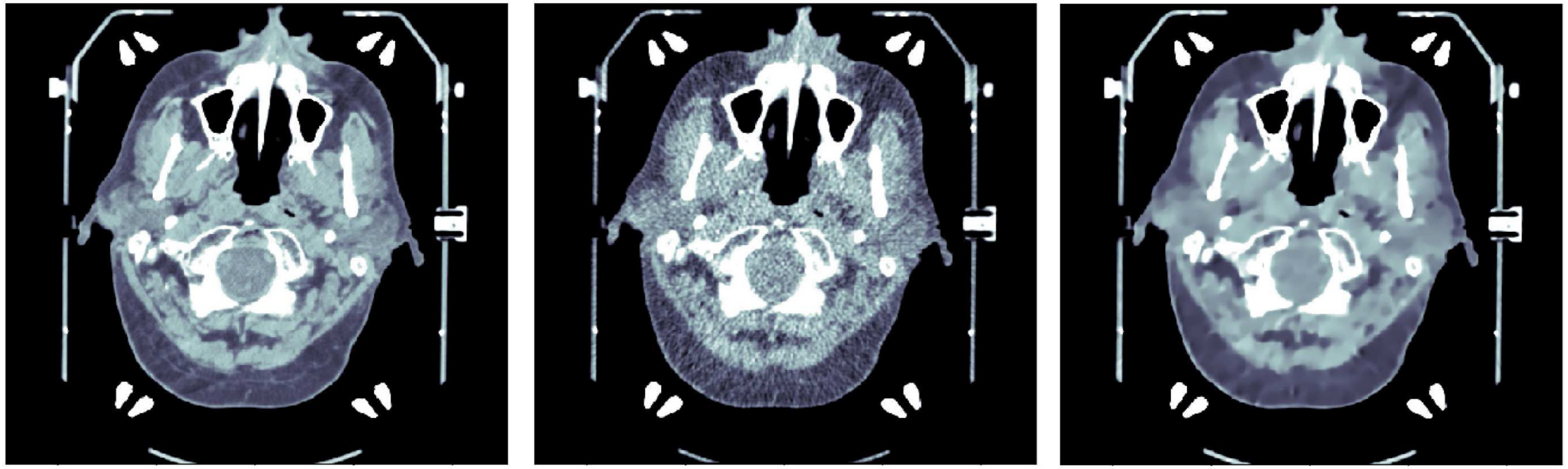


$$\frac{\partial L(\hat{a})}{\partial \hat{a}} \leftarrow NN\left(\frac{\partial L(\hat{a})}{\partial \hat{a}}, \hat{a}, \hat{b}, R(\hat{a}); \theta\right)$$

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

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

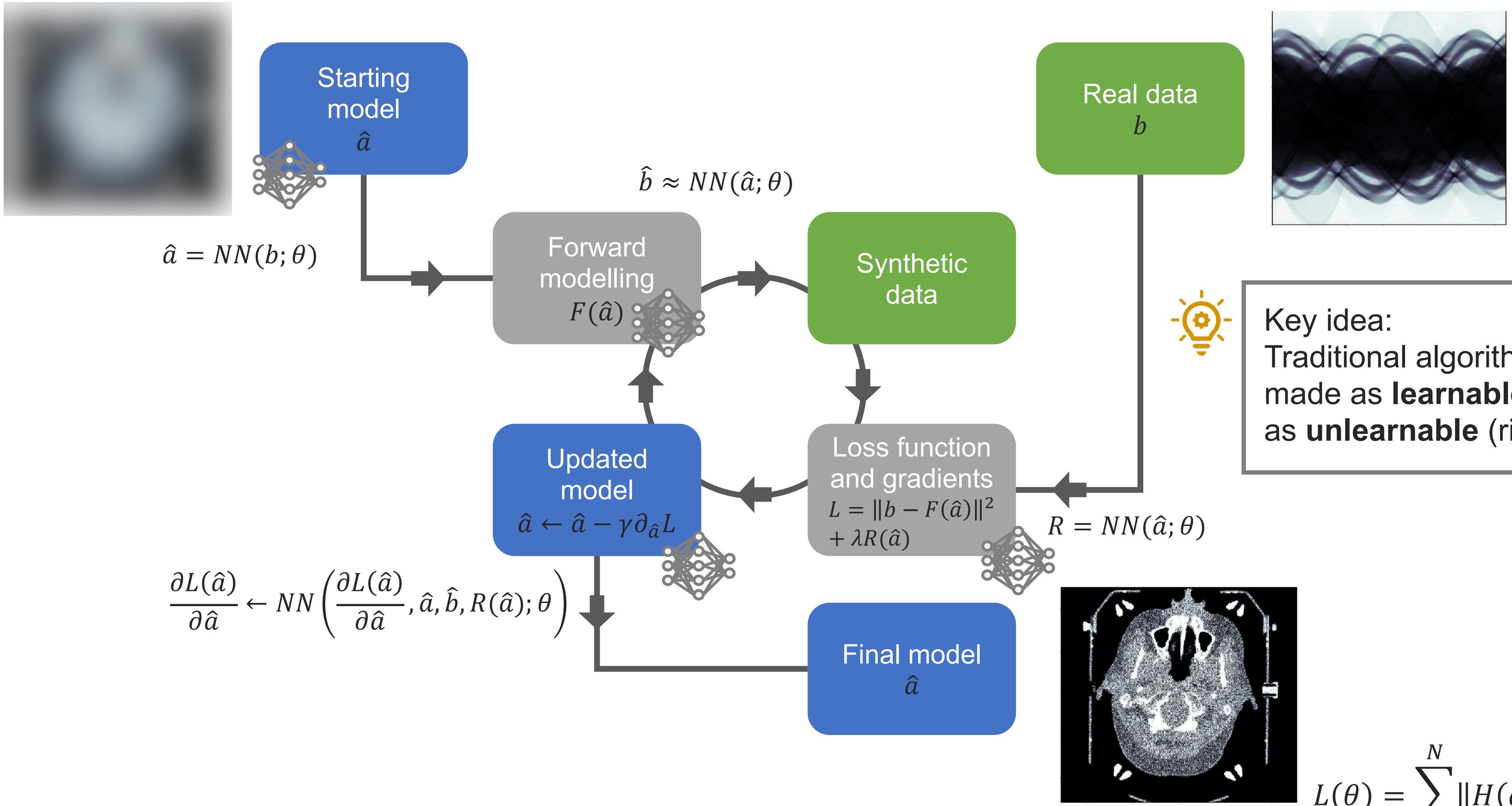
Recap - hybrid computed tomography



Ground truth | Traditional inversion | **Learned gradient descent**

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

Recap - hybrid computed tomography



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

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

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

Lecture overview

- What is a neural differential equation (NDE)?
- The link between NDEs and neural network architectures
- State of the art NDEs
  - Coupled oscillatory RNNs
  - Diffusion models

Lecture overview

- What is a neural differential equation (NDE)?
- The link between NDEs and neural network architectures
- State of the art NDEs
  - Coupled oscillatory RNNs
  - Diffusion models

Learning objectives

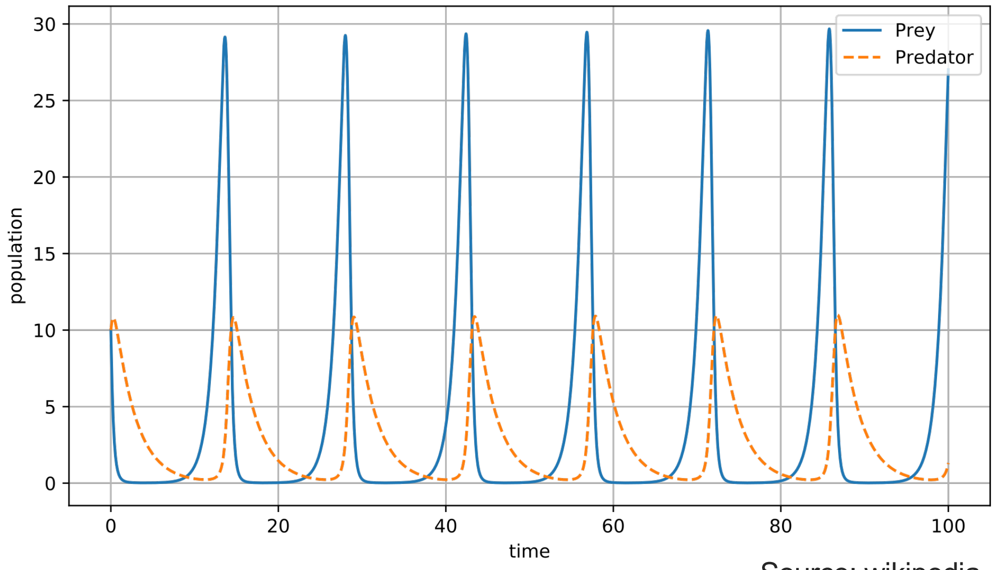
- Be able to define an NDE
- Explain the connection between numerical PDE solvers and neural network architectures
- Be aware of state-of-the-art applications of NDEs

Lotka-Volterra system

The Lotka-Volterra system models **predator-prey** dynamics:

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

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



Source: wikipedia

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

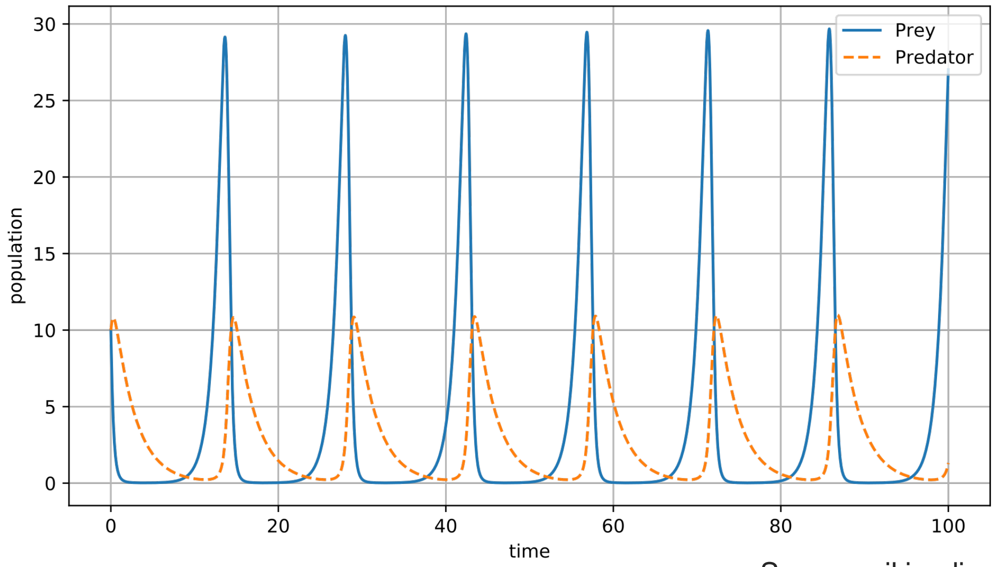
Lotka-Volterra system

The Lotka-Volterra system models **predator-prey** dynamics:

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

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

- How can we solve this system of ODEs (numerically)?



Source: wikipedia

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

Solving Lotka-Volterra system

The Lotka-Volterra system models **predator-prey** dynamics:

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

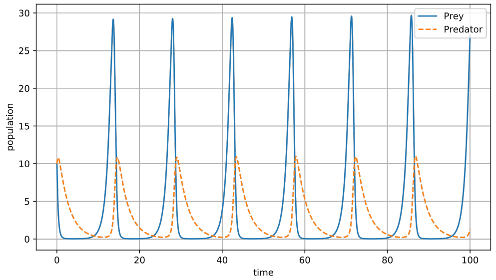
x = population density of prey
y = population density of predator
α, β = max prey birth rate, effect of predators on prey growth rate
δ, γ = max predator death rate, effect of prey on predator growth rate

We can solve numerically using the **Euler method**:

$$\frac{x_{i+1} - x_i}{t_{i+1} - t_i} \approx \alpha x_i - \beta x_i y_i$$
$$\frac{y_{i+1} - y_i}{t_{i+1} - t_i} \approx \gamma x_i y_i - \delta y_i$$

Rearrange:

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



Assumptions of Lotka-Volterra system

The Lotka-Volterra system models **predator-prey** dynamics:

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

$x =$ population density of prey
$y =$ population density of predator
$\alpha, \beta =$ max prey birth rate, effect of predators on prey growth rate
$\delta, \gamma =$ max predator death rate, effect of prey on predator growth rate

Assumptions:

- The prey population always finds ample food.
- The food supply of the predator population depends entirely on the size of the prey population.
- The rate of change of population is proportional to its size.
- Predators have limitless appetite.
- …

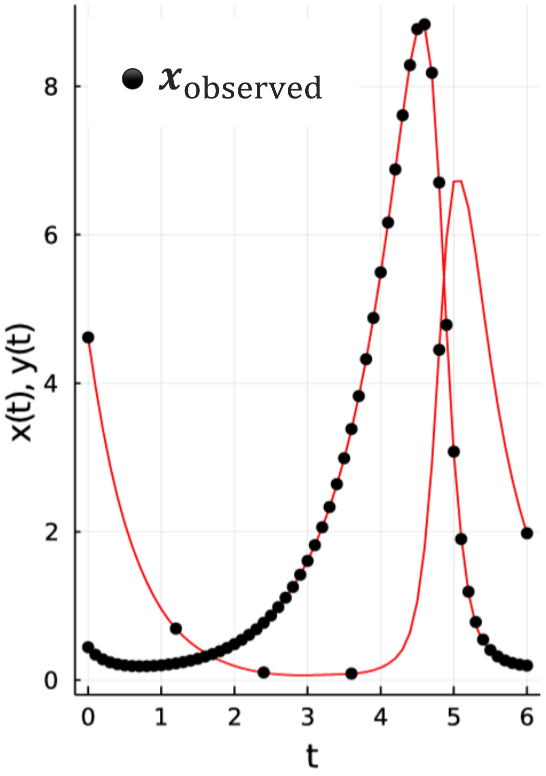
Learning Lotka-Volterra system

The Lotka-Volterra system models **predator-prey** dynamics:

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

x = population density of prey
y = population density of predator

- What if we are unsure of the RHS of the equation? How could we "learn" the ODEs based on population measurements?



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

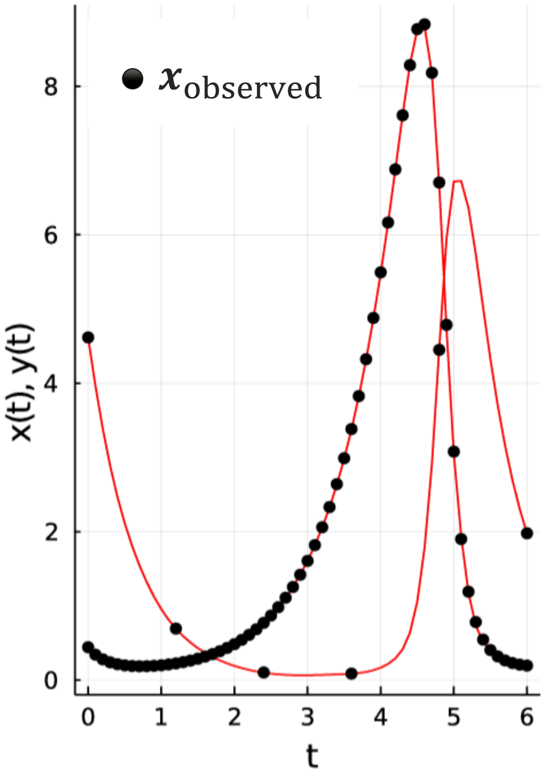
Learning Lotka-Volterra system

The Lotka-Volterra system models **predator-prey** dynamics:

$$\frac{dx}{dt} = \alpha x + NN_1(x, y; \boldsymbol{\theta}_1)$$
$$\frac{dy}{dt} = NN_2(x, y; \boldsymbol{\theta}_2) - \theta_3 y$$

x = population density of prey
y = population density of predator

> Key idea: use NNs to represent parts of differential equations we don't know
>
> = **neural differential equation** (NDE)



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

Learning Lotka-Volterra system

The Lotka-Volterra system models **predator-prey** dynamics:

$$\frac{dx}{dt} = \alpha x + NN_1(x, y; \boldsymbol{\theta}_1)$$
$$\frac{dy}{dt} = NN_2(x, y; \boldsymbol{\theta}_2) - \theta_3 y$$

x = population density of prey
y = population density of predator

Key idea: use NNs to represent parts of differential equations we don't know

= **neural differential equation** (NDE)

How can we solve this system of ODEs (numerically)?

Learning Lotka-Volterra system

The Lotka-Volterra system models **predator-prey** dynamics:

$$\frac{dx}{dt} = \alpha x + NN_1(x, y; \boldsymbol{\theta}_1)$$
$$\frac{dy}{dt} = NN_2(x, y; \boldsymbol{\theta}_2) - \theta_3 y$$

x = population density of prey
y = population density of predator

Key idea: use NNs to represent parts of differential equations we don't know

= **neural differential equation** (NDE)

We can solve numerically using the same **Euler method**:

$$\frac{x_{i+1} - x_i}{t_{i+1} - t_i} \approx \alpha x_i - NN_1(x_i, y_i; \boldsymbol{\theta}_1)$$
$$\frac{y_{i+1} - y_i}{t_{i+1} - t_i} \approx NN_2(x_i, y_i; \boldsymbol{\theta}_2) - \theta_3 y_i$$

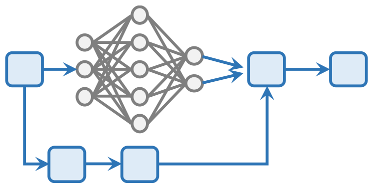
Rearrange:

$$x_{i+1} = x_i + \Delta t(\alpha x_i - NN_1(x_i, y_i; \boldsymbol{\theta}_1))$$
$$y_{i+1} = y_i + \Delta t(NN_2(x_i, y_i; \boldsymbol{\theta}_2) - \theta_3 y_i)$$
$$t_{i+1} = t_i + \Delta t$$

Learning Lotka-Volterra system

Note this is an example of a **hybrid** simulation workflow:

```
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
```



We can solve numerically using the same **Euler method**:

$$\frac{x_{i+1} - x_i}{t_{i+1} - t_i} \approx \alpha x_i - NN_1(x_i, y_i; \boldsymbol{\theta}_1)$$

$$\frac{y_{i+1} - y_i}{t_{i+1} - t_i} \approx NN_2(x_i, y_i; \boldsymbol{\theta}_2) - \theta_3 y_i$$

Rearrange:

$$x_{i+1} = x_i + \Delta t(\alpha x_i - NN_1(x_i, y_i; \boldsymbol{\theta}_1))$$

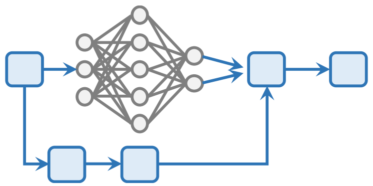
$$y_{i+1} = y_i + \Delta t(NN_2(x_i, y_i; \boldsymbol{\theta}_2) - \theta_3 y_i)$$

$$t_{i+1} = t_i + \Delta t$$

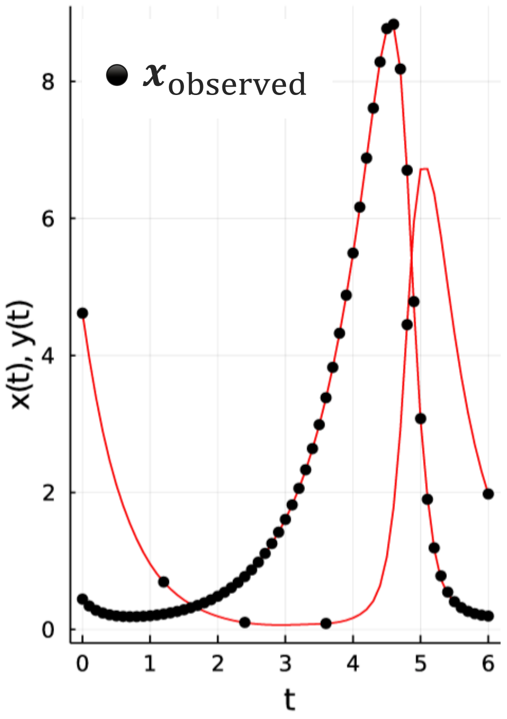
Learning Lotka-Volterra system

Note this is an example of a **hybrid** simulation workflow:

```
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
```



Suppose we are given these population measurements:



- How can we train the neural networks using this data?

Learning Lotka-Volterra system

Note this is an example of a **hybrid** simulation workflow:

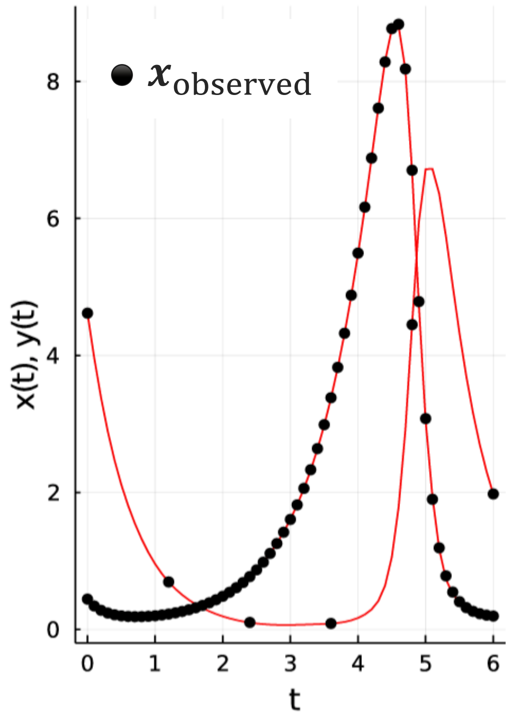
```python
def Hybrid_LV_Euler_solver(x0, y0, dt, theta):
    """Pseudocode for solving Lotka-Volterra system,
    with learnable dynamics"""

    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
```

Train the hybrid solver using loss function:

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

(Using autodifferentiation + gradient descent)

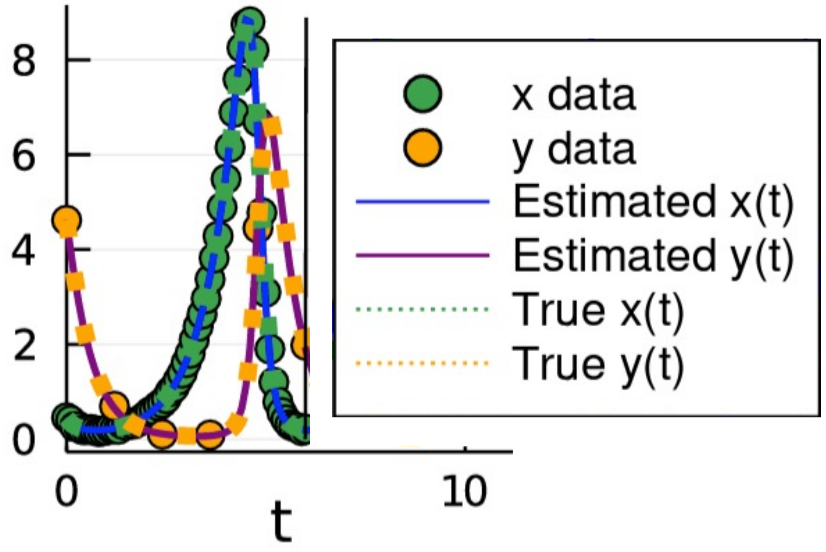
Suppose we are given these population measurements:



- How can we train the neural networks using this data?

Hybrid Lotka-Volterra solver

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



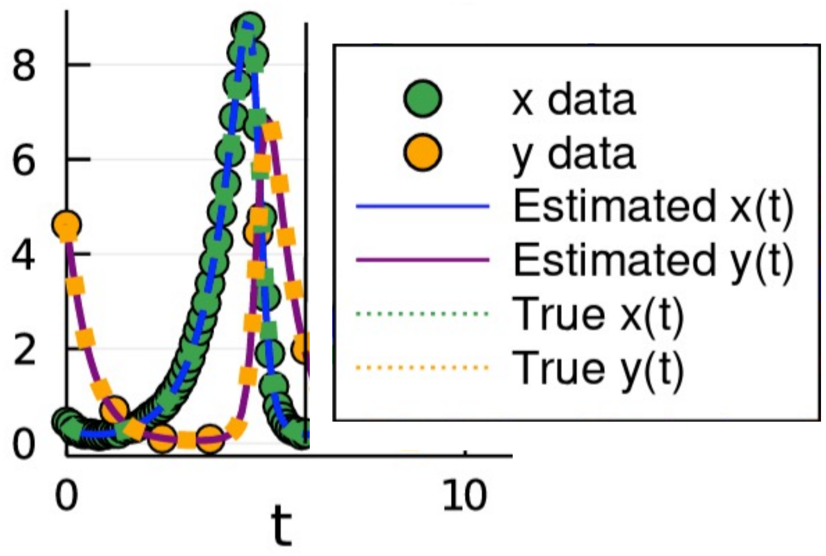
$$\frac{dx}{dt} = \alpha x + NN_1(x, y; \boldsymbol{\theta}_1)$$

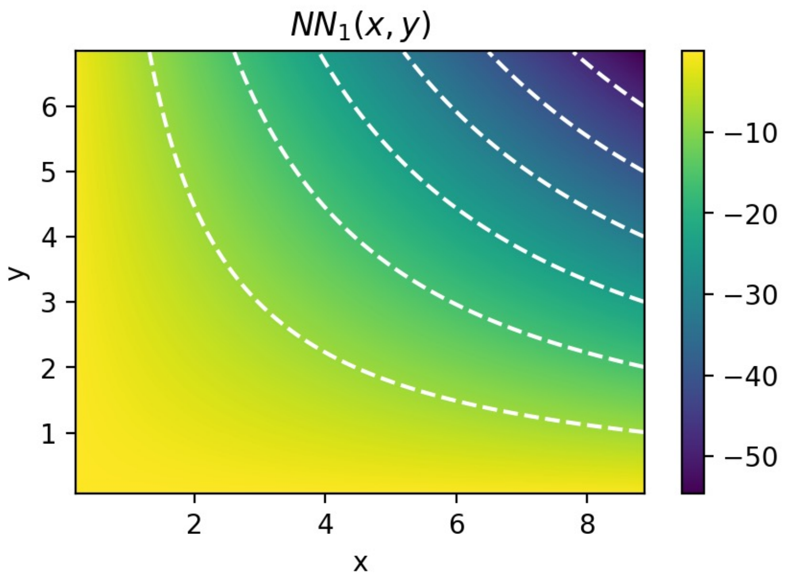
$$\frac{dy}{dt} = NN_2(x, y; \boldsymbol{\theta}_2) - \theta_3 y$$

$$L(\boldsymbol{\theta}) = \sum_i^T \|\boldsymbol{x}_{\text{Euler } i}(\boldsymbol{x}_0, \Delta t, \boldsymbol{\theta}) - \boldsymbol{x}_{\text{observed } i}\|^2$$

Hybrid Lotka-Volterra solver

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





$$\frac{dx}{dt} = \alpha x + NN_1(x,y;\boldsymbol{\theta}_1)$$

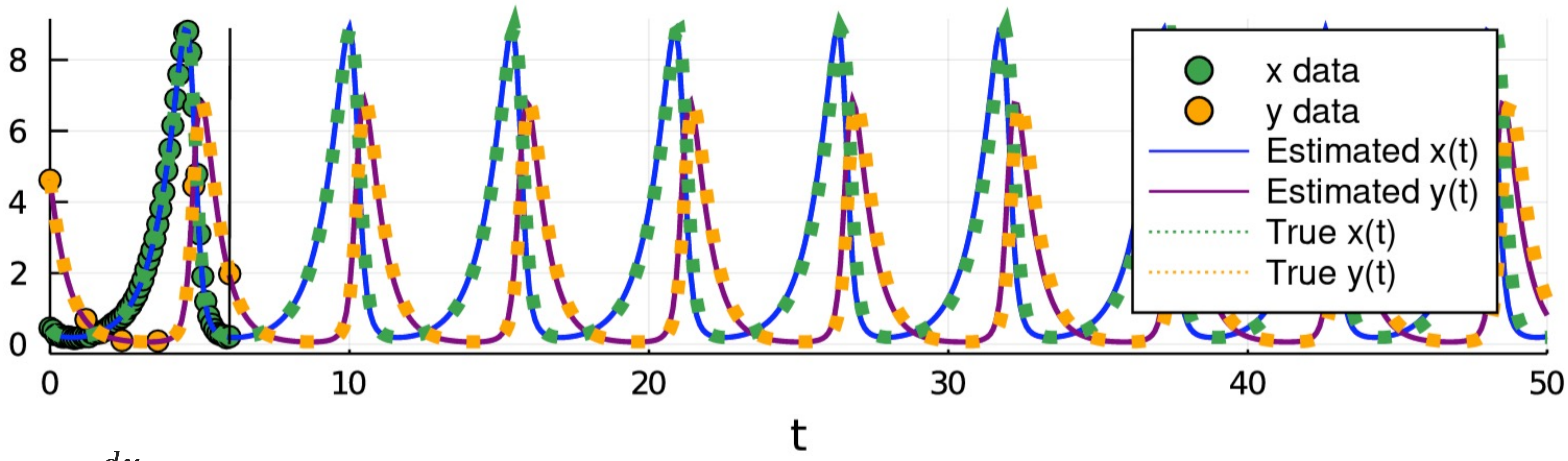
$$\frac{dy}{dt} = NN_2(x,y;\boldsymbol{\theta}_2) - \theta_3 y$$

$$L(\boldsymbol{\theta}) = \sum_i^T \|\boldsymbol{x}_{\mathrm{Euler}\,i}(\boldsymbol{x}_0, \Delta t, \boldsymbol{\theta}) - \boldsymbol{x}_{\mathrm{observed}\,i}\|^2$$

- Note, after training, we can do **symbolic regression** on $NN_1(x,y;\boldsymbol{\theta}_1)$ and $NN_2(x,y;\boldsymbol{\theta}_2)$ to “discover” their functional form, e.g. that $NN_1(x,y;\boldsymbol{\theta}_1) \approx -\beta xy$

Hybrid Lotka-Volterra solver

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



$$\frac{dx}{dt} = \alpha x + NN_1(x, y; \boldsymbol{\theta}_1)$$

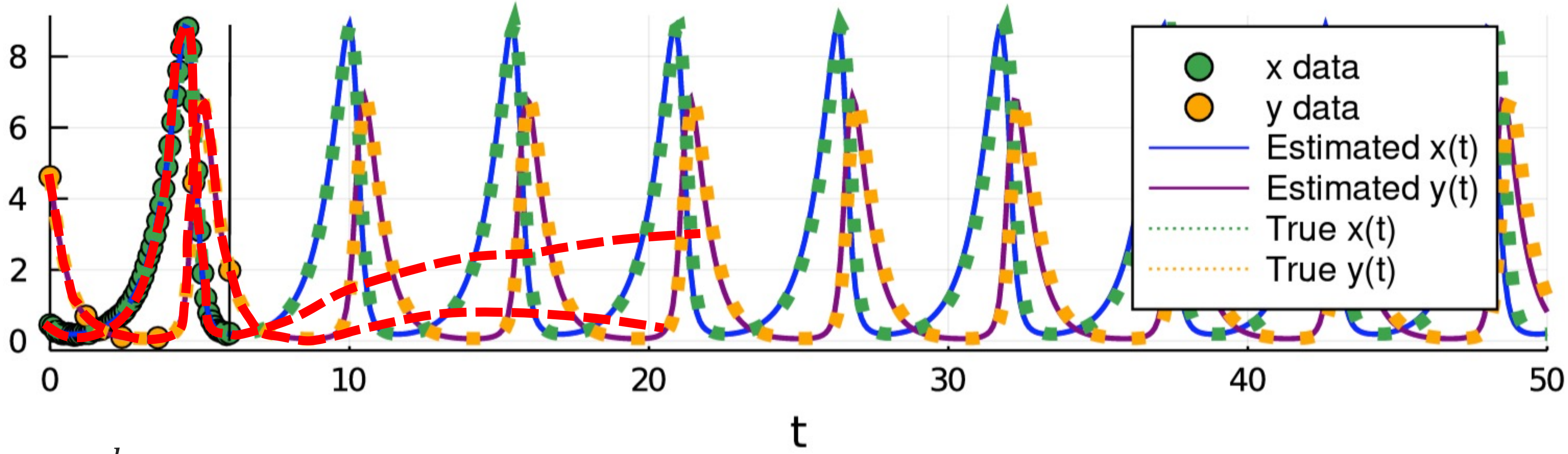
$$\frac{dy}{dt} = NN_2(x, y; \boldsymbol{\theta}_2) - \theta_3 y$$

- This model generalizes well!

$$L(\boldsymbol{\theta}) = \sum_{i}^{T} \|\boldsymbol{x}_{\mathrm{Euler}\ i}(\boldsymbol{x}_0, \Delta t, \boldsymbol{\theta}) - \boldsymbol{x}_{\mathrm{observed}\ i}\|^2$$

Hybrid Lotka-Volterra solver

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



$$\frac{dx}{dt} = \alpha x + NN_1(x, y; \boldsymbol{\theta}_1)$$

$$\frac{dy}{dt} = NN_2(x, y; \boldsymbol{\theta}_2) - \theta_3 y$$

Red curve = comparison to training

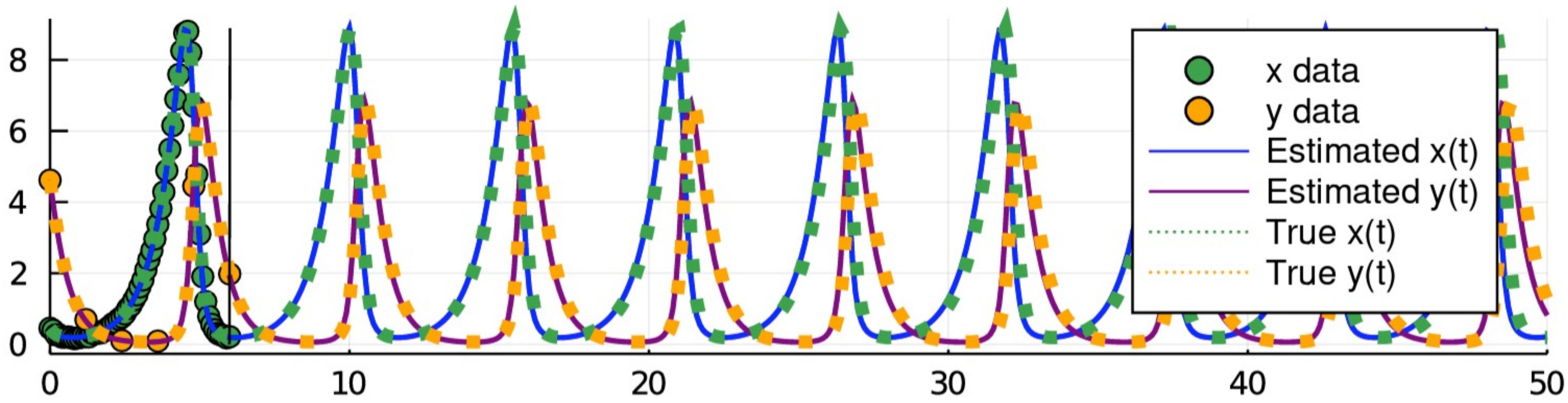
$$\boldsymbol{x} \approx NN(t; \boldsymbol{\theta})$$

$$L(\boldsymbol{\theta}) = \sum_i^T \|\boldsymbol{x}_{\text{Euler } i}(\boldsymbol{x}_0, \Delta t, \boldsymbol{\theta}) - \boldsymbol{x}_{\text{observed } i}\|^2$$

$$L(\boldsymbol{\theta}) = \sum_i^T \|NN(t_i; \boldsymbol{\theta}) - \boldsymbol{x}_{\text{observed } i}\|^2$$

Hybrid Lotka-Volterra solver

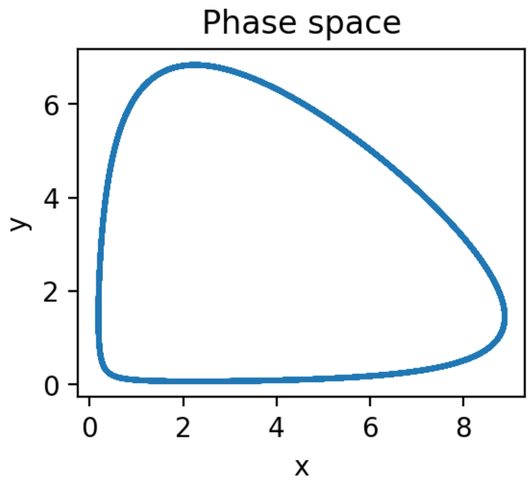
Rackauckas et al, Universal differential equations for scientific machine learning, ArXiv (2021)



$$\frac{dx}{dt} = \alpha x + NN_1(x, y; \boldsymbol{\theta}_1)$$

$$\frac{dy}{dt} = NN_2(x, y; \boldsymbol{\theta}_2) - \theta_3 y$$

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



- Model generalizes well because neural networks see **entire phase space** in their inputs during training

Summary - neural differential equations

Key idea: use NNs to represent parts of differential equations we don't know

= **neural differential equation** (NDE)

- We can solve NDEs using numerical methods
- We can train NDEs using autodifferentiation
- They can be used to "discover" underlying dynamics
- They can be thought of as a **hybrid** technique

Lecture overview

- What is a neural differential equation (NDE)?
- The link between NDEs and neural network architectures
- State of the art NDEs
  - Coupled oscillatory RNNs
  - Diffusion models

Learning objectives

- Be able to define an NDE
- Explain the connection between numerical PDE solvers and neural network architectures
- Be aware of state-of-the-art applications of NDEs

Neural ordinary differential equations

More generally, we define a **neural ordinary differential equation** as:

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

Where $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$ is a learnable function (a neural network)

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

Neural ordinary differential equations

More generally, we define a **neural ordinary differential equation** as:

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

Where $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$ is a learnable function (a neural network)

Solver using Euler method:

Given $\boldsymbol{x}_0 = \boldsymbol{x}(t = 0), \Delta t$:

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

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

Neural ordinary differential equations

More generally, we define a **neural ordinary differential equation** as:

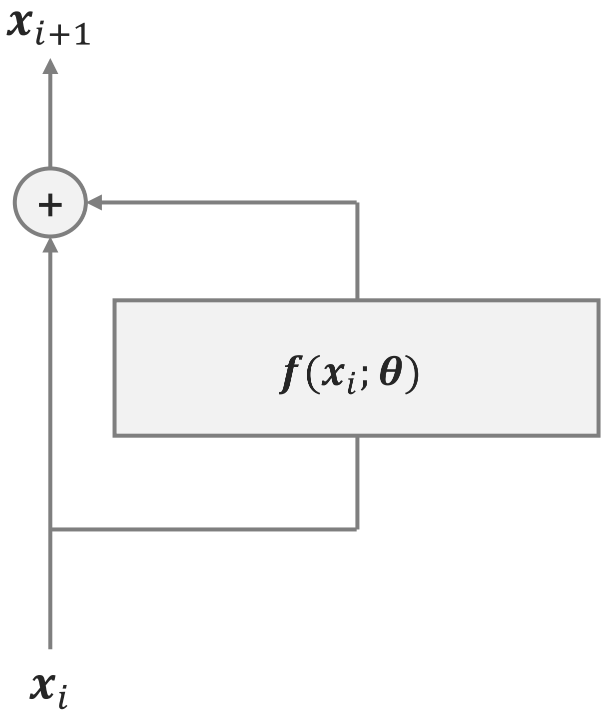
$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$$

Where $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$ is a learnable function (a neural network)

Solver using Euler method:

Given $\boldsymbol{x}_0 = \boldsymbol{x}(t = 0), \Delta t$:

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



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

Neural ordinary differential equations

More generally, we define a **neural ordinary differential equation** as:

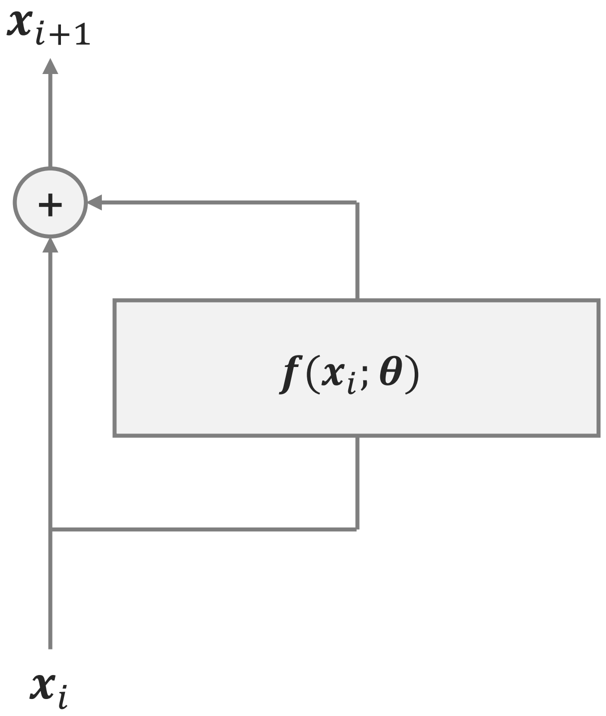
$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta})$$

Where $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta})$ is a learnable function (a neural network)

Solver using Euler method:

Given $\boldsymbol{x}_0 = \boldsymbol{x}(t=0), \Delta t$:

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



The Euler step is **identical** to a **residual layer** used in standard residual networks (ResNets)!

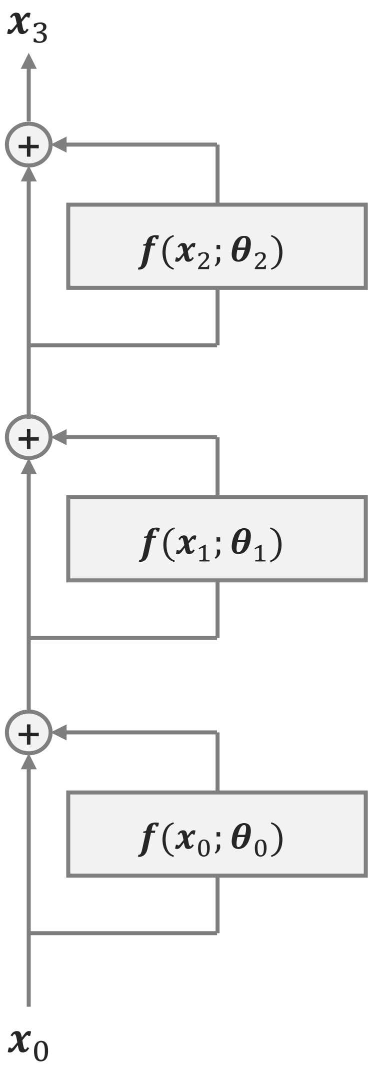
ResNets are Euler solvers

ResNets ⟺ Euler ODE solvers

In the **limit** of infinite numbers of layers (i.e. as $\Delta t \to 0$), ResNets solve the ODE

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

Training a ResNet ⟺ learning the RHS of the ODE



Neural ordinary differential equations

We define a **neural ordinary differential equation** as:

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

Where $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$ is a learnable function (a neural network)

We are not limited to Euler solvers! What else could we use to solve this ODE?

Higher-order solvers

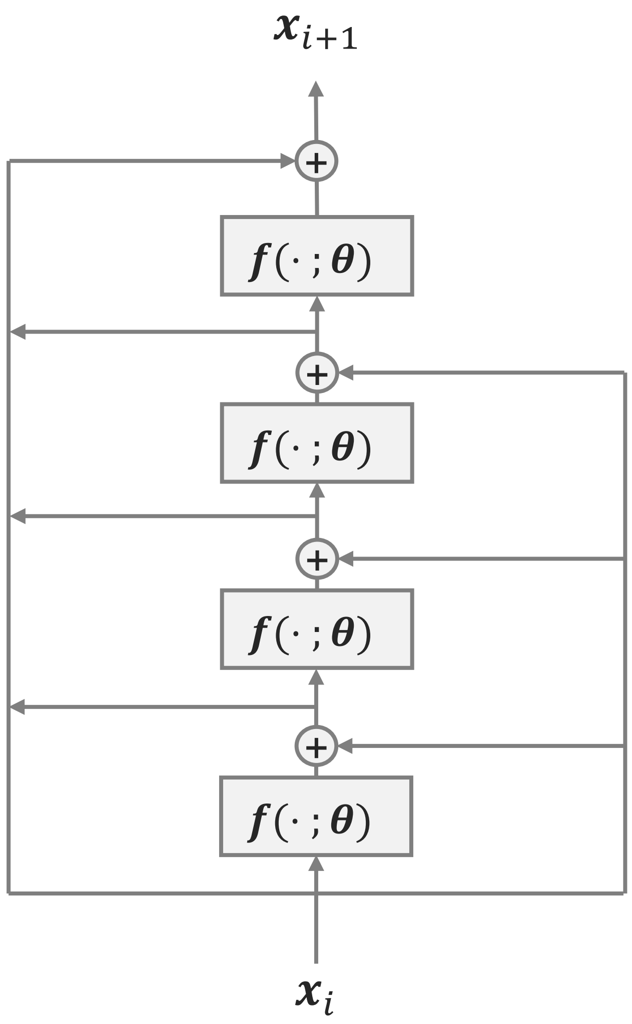
We define a **neural ordinary differential equation** as:

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

Where $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$ is a learnable function (a neural network)

Many **other** solvers could be used, for example higher-order Runge-Kutta methods, e.g. RK4:

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



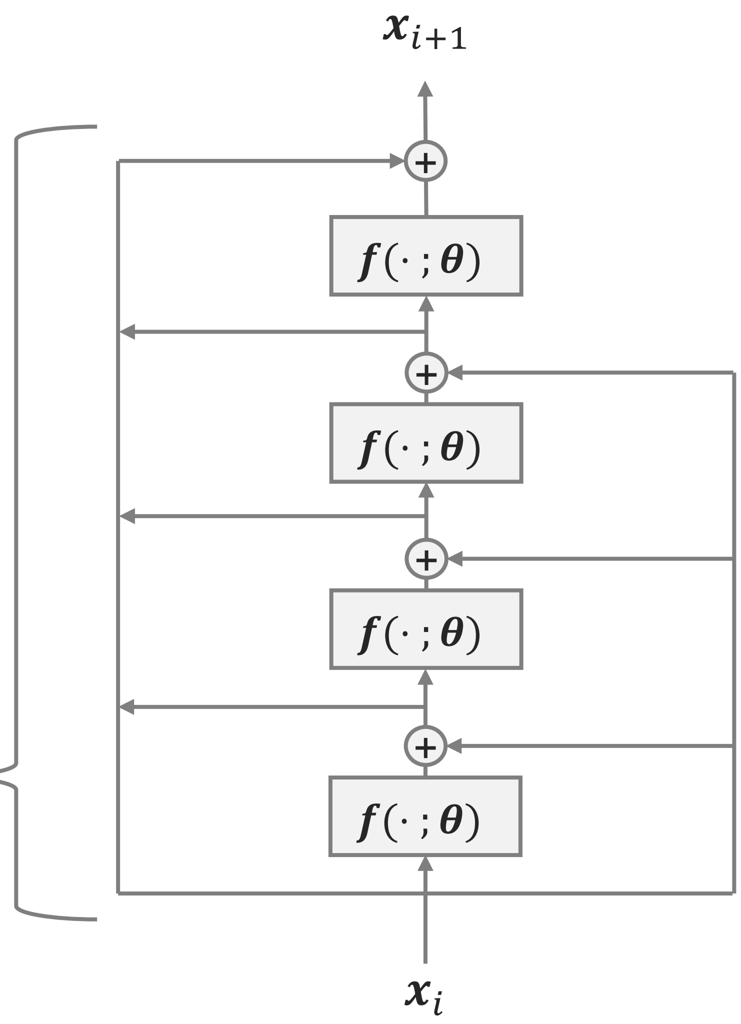
Higher-order solvers

We define a **neural ordinary differential equation** as:

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

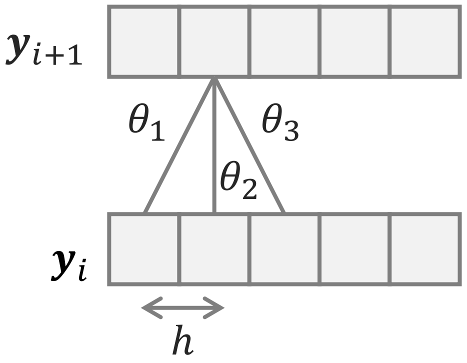
Where $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$ is a learnable function (a neural network)

“Custom” residual block



=> Other solvers define other NN architectures

CNNs as PDE solvers



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)

Summary – NDEs and NN architectures

Discretised NDE solvers ⟺ Neural network architectures

Understanding of PDEs / their solutions ⟺ Understanding of architectures / training algorithms

NDEs can help us **interpret** the dynamics of neural network architectures

Lecture overview

- What is a neural differential equation (NDE)?
- The link between NDEs and neural network architectures
- State of the art NDEs
  - Coupled oscillatory RNNs
  - Diffusion models

Learning objectives

- Be able to define an NDE
- Explain the connection between numerical PDE solvers and neural network architectures
- Be aware of state-of-the-art applications of NDEs

5 min break

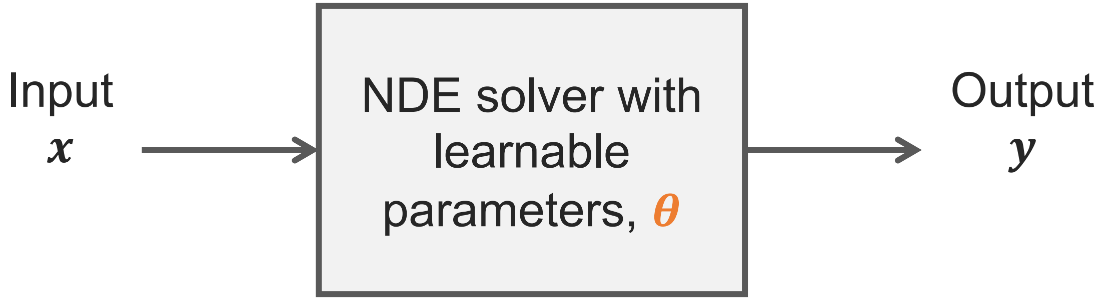
Lecture overview

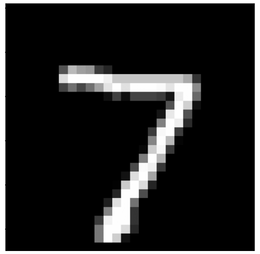
- What is a neural differential equation (NDE)?
- The link between NDEs and neural network architectures
- State of the art NDEs
  - Coupled oscillatory RNNs
  - Diffusion models

Learning objectives

- Be able to define an NDE
- Explain the connection between numerical PDE solvers and neural network architectures
- Be aware of state-of-the-art applications of NDEs

Using NDEs for ML tasks



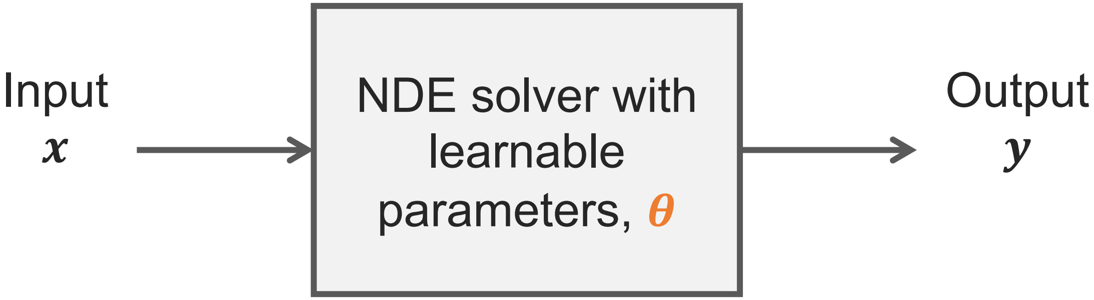


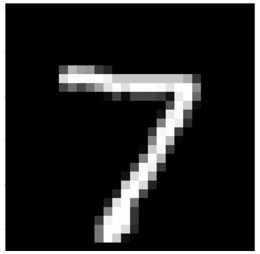
$P(\boldsymbol{x} = 7)$

Key idea: discretised NDE solvers can be thought of as “custom” NN architectures

- what if we use NDEs to model **any** dataset (not just physical systems)?

Using NDEs for ML tasks





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

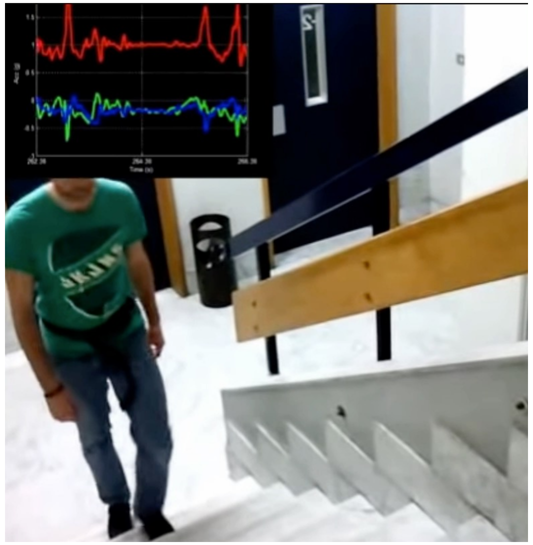
	Test Error
1-Layer MLP†	1.60%
ResNet	0.41%
RK-Net	0.47%

Performance on MNIST (digit classification)

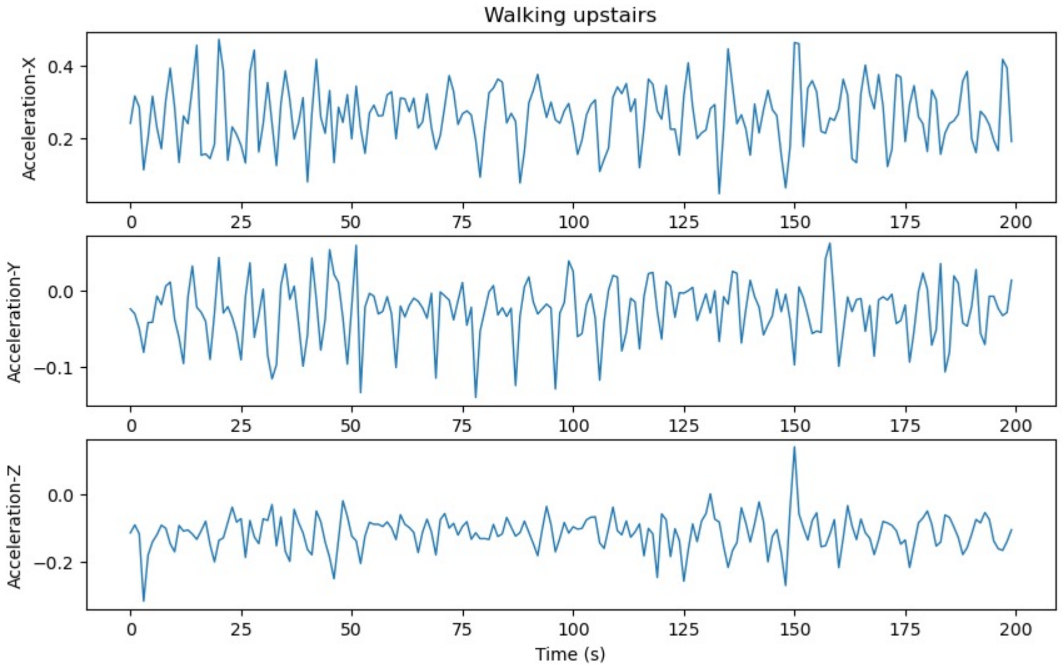
Chen et al, Neural ordinary differential equations, NeurIPS (2018)

Human activity recognition

- Consider the task of **human activity recognition**

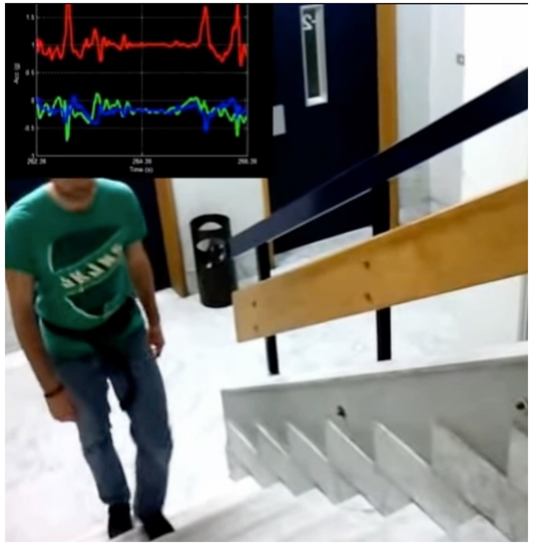


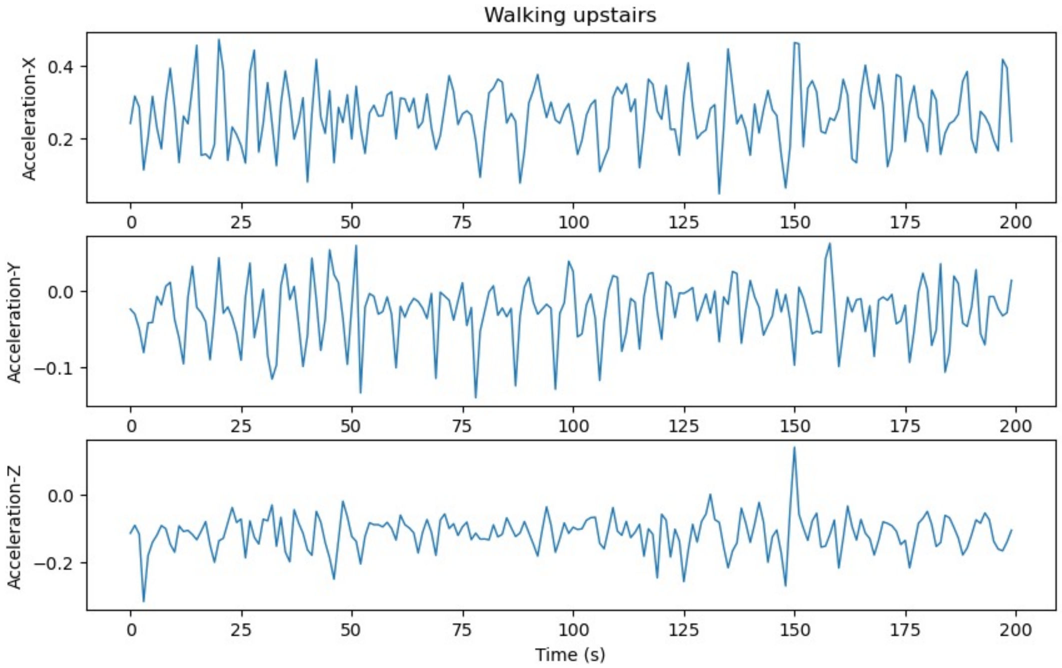
Anguita et al. Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine. 4th International Workshop of Ambient Assisted Living (2012)



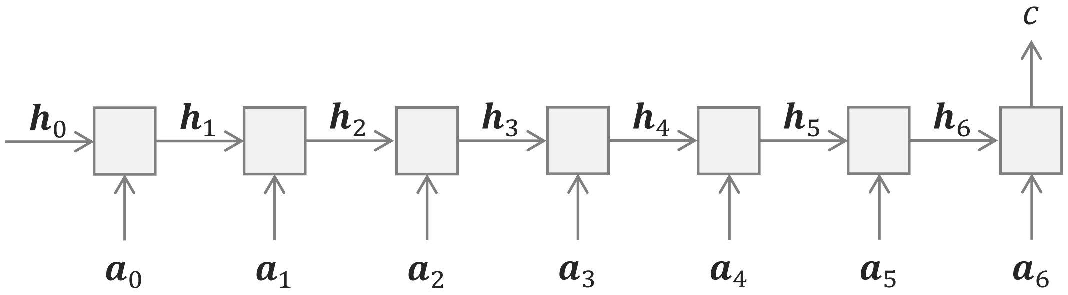
Human activity recognition

- Consider the task of **human activity recognition**





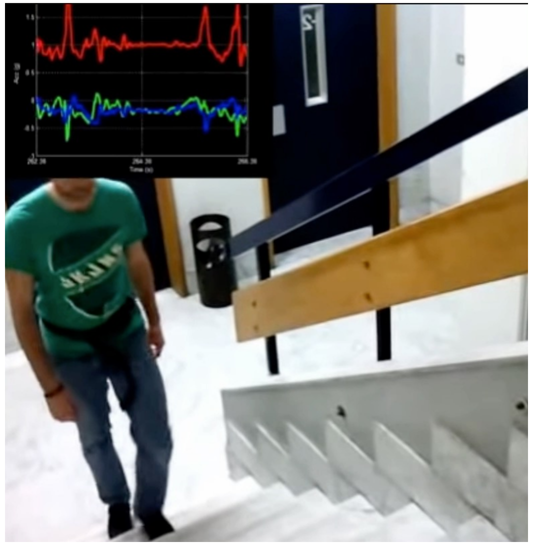
- One way to predict the class is to use a **recursive neural network** (RNN)

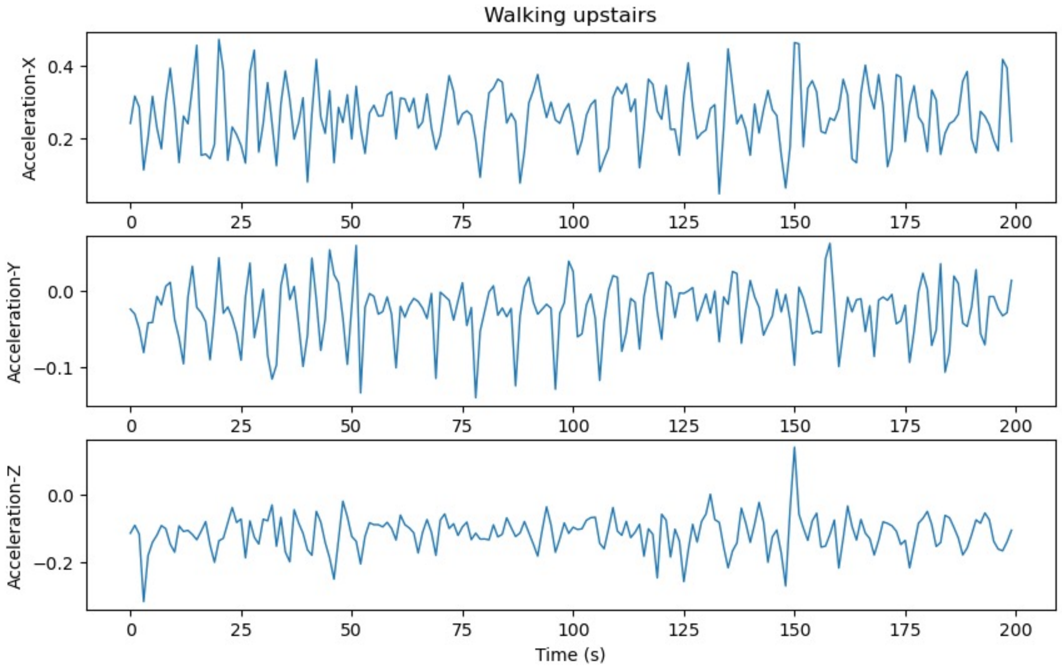


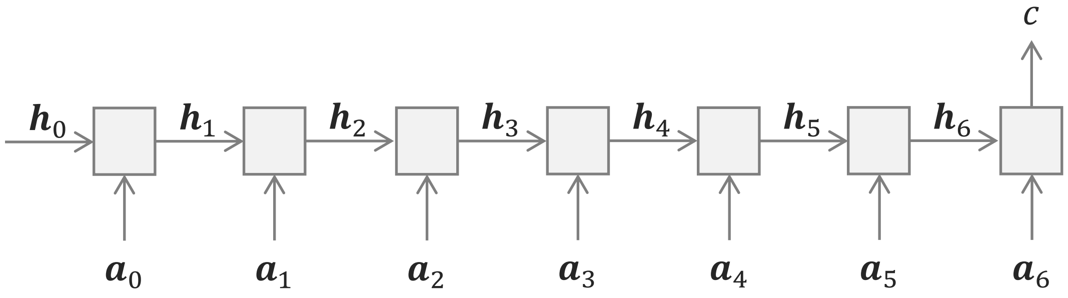
- It is often hard to know what architecture to use in the RNN cell: MLP? CNN? LSTM cell?

Human activity recognition

- Consider the task of **human activity recognition**



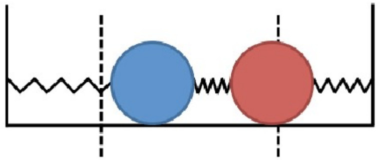




- One way to predict the class is to use a **recursive neural network** (RNN)
- The data looks "oscillatory" – can we incorporate this into the RNN design?

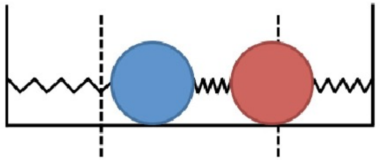
Coupled harmonic oscillators

- From above: Discretised NDE solvers ⟺ Neural network architectures
- Idea: use **coupled harmonic oscillators** to design a neural network architecture

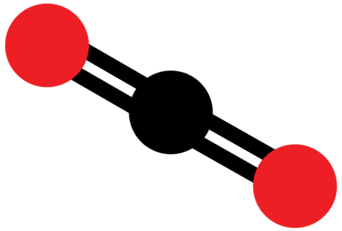


Coupled harmonic oscillators

- From above: Discretised NDE solvers ⟺ Neural network architectures
- Idea: use **coupled harmonic oscillators** to design a neural network architecture



- Coupled harmonic oscillators are found across physics, engineering and biology



CO_2

EEG readings (source: Wikipedia)

Tacoma Narrows suspension bridge, 1940

Rusch and Mishra, Coupled Oscillatory Recurrent Neural Network (coRNN): An accurate and (gradient) stable architecture for learning long time dependencies. ICLR (2021)

Coupled harmonic oscillators

- 1D damped harmonic oscillator

$$m\frac{d^2x}{dt^2} = -\mu\frac{dx}{dt} - kx + f$$

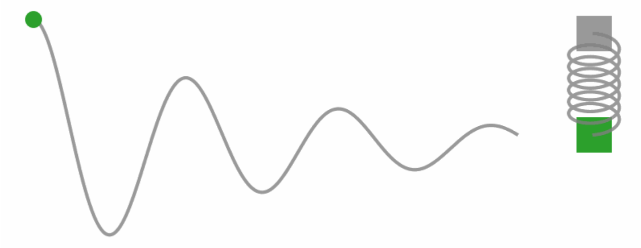
$x =$ displacement of oscillator

$m =$ mass of oscillator

$\mu =$ coefficient of friction

$k =$ spring constant

$f =$ external driving force



Coupled harmonic oscillators

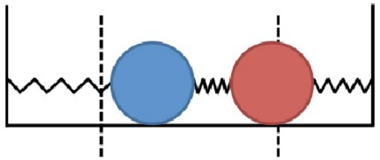
- **ND coupled**, **nonlinear**, damped harmonic oscillator

$$M \frac{d^2\boldsymbol{x}}{dt^2} = \tanh\left(-W \frac{d\boldsymbol{x}}{dt} - V\boldsymbol{x} + \boldsymbol{f}\right)$$

where

$$M = \begin{pmatrix} m_1 & 0 & 0 \\ 0 & \dots & 0 \\ 0 & 0 & m_n \end{pmatrix}$$

and W, V are coefficient of friction and spring constant matrices, where their off-diagonal elements represent **interactions** between oscillators



- 1D damped harmonic oscillator

$$m \frac{d^2x}{dt^2} = -\mu \frac{dx}{dt} - kx + f$$

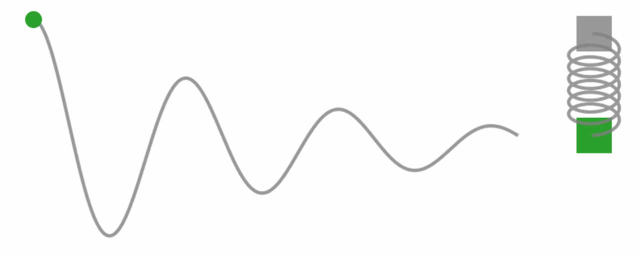
x = displacement of oscillator

m = mass of oscillator

μ = coefficient of friction

k = spring constant

f = external driving force



Solving coupled harmonic oscillators

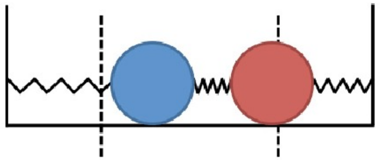
- **ND coupled**, **nonlinear**, damped harmonic oscillator

$$M\frac{d^2\boldsymbol{x}}{dt^2} = \tanh\left(-W\frac{d\boldsymbol{x}}{dt} - V\boldsymbol{x} + \boldsymbol{f}\right)$$

where

$$M = \begin{pmatrix} m_1 & 0 & 0 \\ 0 & \dots & 0 \\ 0 & 0 & m_n \end{pmatrix}$$

and W, V are coefficient of friction and spring constant matrices, where their off-diagonal elements represent **interactions** between oscillators



- How can we solve this system of ODEs? (assuming $M = 1$)

Solving coupled harmonic oscillators

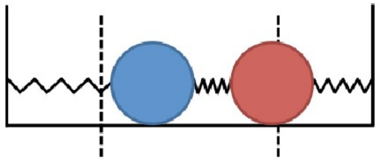
- **ND coupled**, **nonlinear**, damped harmonic oscillator

$$M \frac{d^2\boldsymbol{x}}{dt^2} = \tanh\left(-W \frac{d\boldsymbol{x}}{dt} - V\boldsymbol{x} + \boldsymbol{f}\right)$$

where

$$M = \begin{pmatrix} m_1 & 0 & 0 \\ 0 & \dots & 0 \\ 0 & 0 & m_n \end{pmatrix}$$

and W, V are coefficient of friction and spring constant matrices, where their off-diagonal elements represent **interactions** between oscillators



Introduce velocity variable:

$$\boldsymbol{v} = \frac{d\boldsymbol{x}}{dt}$$

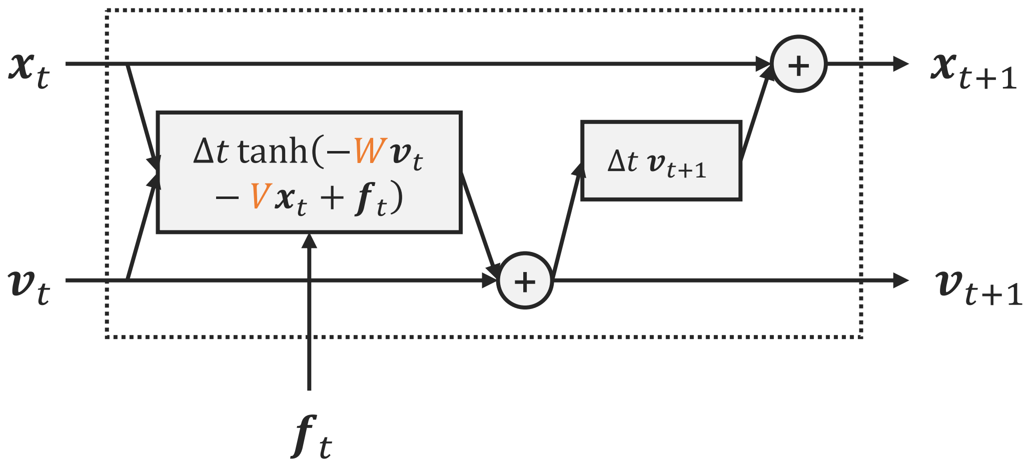
Then

$$M \frac{d\boldsymbol{v}}{dt} = \tanh(-W\boldsymbol{v} - V\boldsymbol{x} + \boldsymbol{f})$$

Assume $M = 1$, and discretise in time:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \Delta t \boldsymbol{v}_{t+1}$$
$$\boldsymbol{v}_{t+1} = \boldsymbol{v}_t + \Delta t \tanh(-W\boldsymbol{v}_t - V\boldsymbol{x}_t + \boldsymbol{f}_t)$$

Solving coupled harmonic oscillators



Introduce velocity variable:

$$\boldsymbol{v} = \frac{d\boldsymbol{x}}{dt}$$

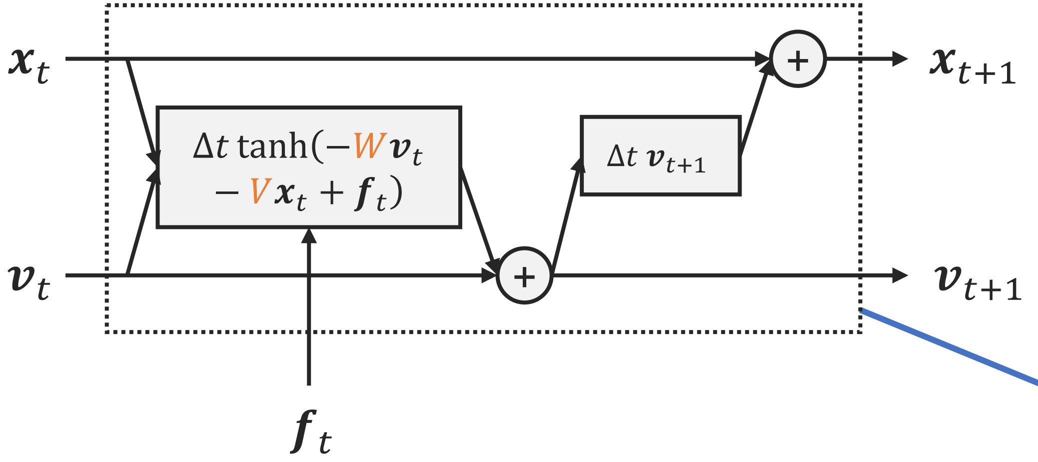
Then

$$M\frac{d\boldsymbol{v}}{dt} = \tanh(-W\boldsymbol{v} - V\boldsymbol{x} + \boldsymbol{f})$$

Assume $M = 1$, and discretise in time:

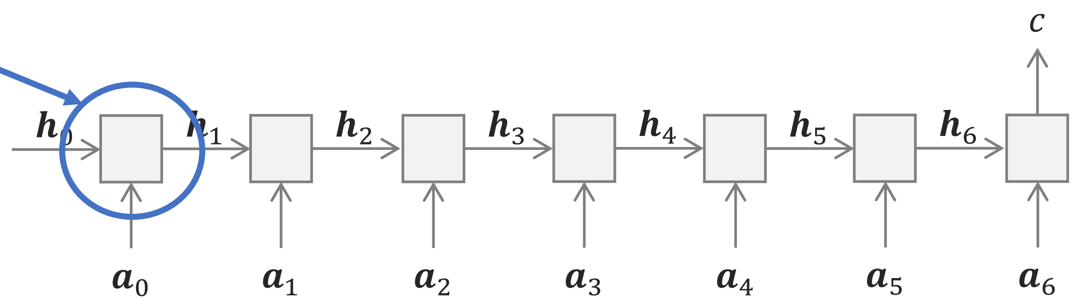
$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \Delta t \boldsymbol{v}_{t+1}$$
$$\boldsymbol{v}_{t+1} = \boldsymbol{v}_t + \Delta t \tanh(-W\boldsymbol{v}_t - V\boldsymbol{x}_t + \boldsymbol{f}_t)$$

Coupled oscillatory RNNs (CoRNNs)



We can interpret the ODE solver as an RNN, and treat W and V as **learnable**, shared weight matrices

= Physics-inspired RNN design!



Rusch and Mishra, Coupled Oscillatory Recurrent Neural Network (coRNN): An accurate and (gradient) stable architecture for learning long time dependencies. ICLR (2021)

Coupled oscillatory RNNs (CoRNNs)

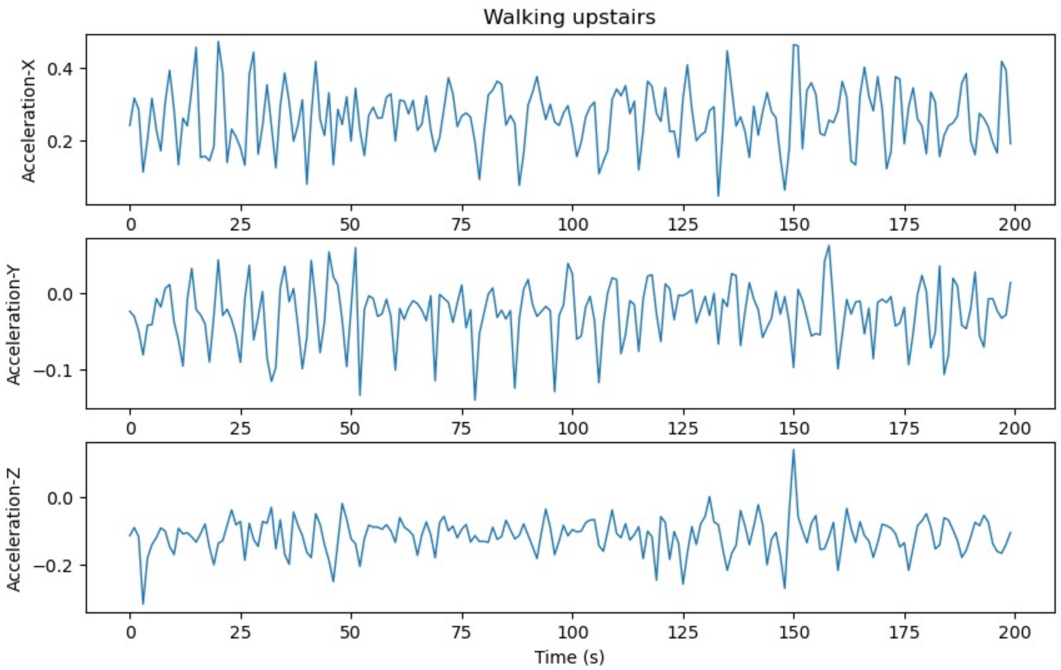


Table 3: Test accuracies on HAR-2.

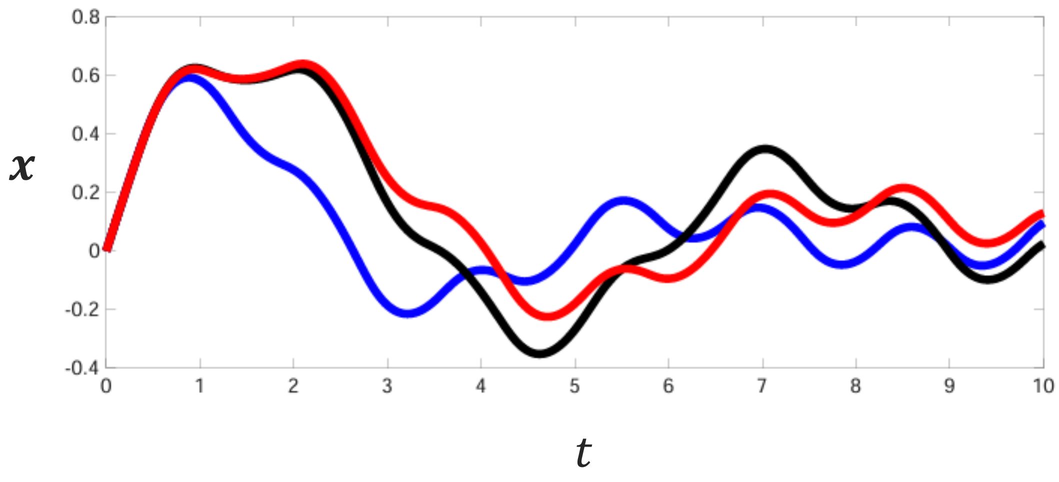
Model	test accuracy	# units	# params
GRU (Kusupati et al., 2018)	93.6%	75	19k
LSTM (Kag et al., 2020)	93.7%	64	16k
FastRNN (Kusupati et al., 2018)	94.5%	80	7k
FastGRNN (Kusupati et al., 2018)	95.6%	80	7k
anti.sym. RNN (Kag et al., 2020)	93.2%	120	8k
incremental RNN (Kag et al., 2020)	96.3%	64	4k
coRNN	**97.2%**	64	9k

Table 4: Test accuracies on IMDB.

Model	test accuracy	# units	# params
LSTM (Campos et al., 2018)	86.8%	128	220k
Skip LSTM(Campos et al., 2018)	86.6%	128	220k
GRU (Campos et al., 2018)	86.2%	128	164k
Skip GRU (Campos et al., 2018)	86.6%	128	164k
ReLU GRU (Dey & Salemt, 2017)	84.8%	128	99k
coRNN	**87.4%**	128	46k

Rusch and Mishra, Coupled Oscillatory Recurrent Neural Network (coRNN): An accurate and (gradient) stable architecture for learning long time dependencies. ICLR (2021)

Interpreting network dynamics



- We can plot the evolution of the **hidden state** of the CoRNN (= displacement of the oscillators)
- Using the underlying ODE, it can be shown that the energy of the system (and therefore magnitude of the oscillations) is **bounded**
- This leads to the result that CoRNNs do not suffer from **exploding gradients***

(*see paper for proof)

Rusch and Mishra, Coupled Oscillatory Recurrent Neural Network (coRNN): An accurate and (gradient) stable architecture for learning long time dependencies. ICLR (2021)

Lecture summary

- A neural differential equation uses neural networks to represent **learnable parts** of the equation

- A discretised NDE solver can be thought of as neural network architecture with **interpretable dynamics**

- State of the art ML models, e.g. diffusion models, solve NDEs