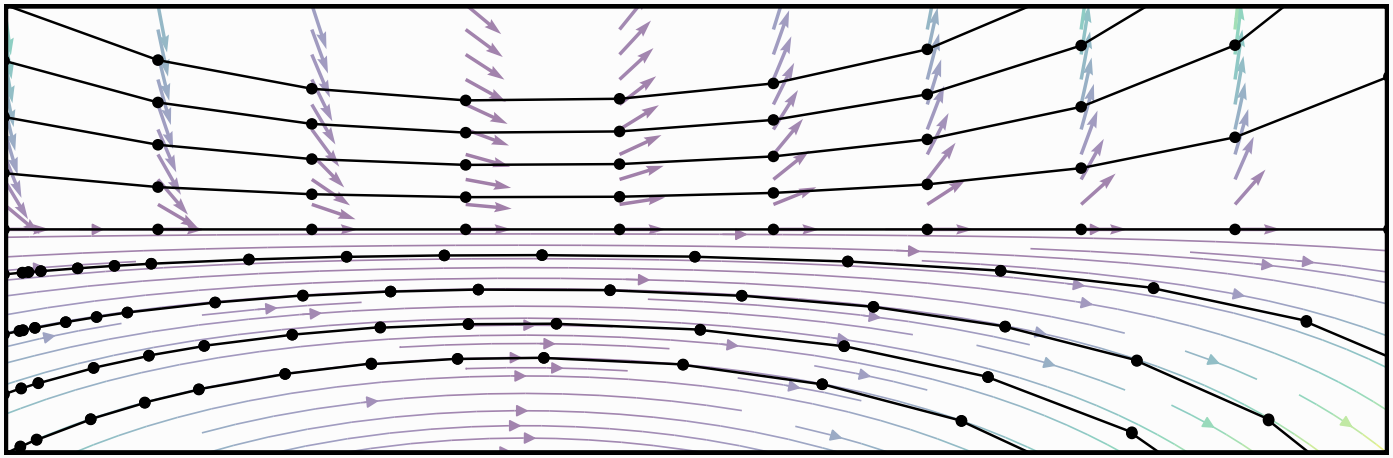
Neural ODEs - Seminar Talk

Rada Beronova & Georg Trede



University of Heidelberg

June 13, 2023

Overview

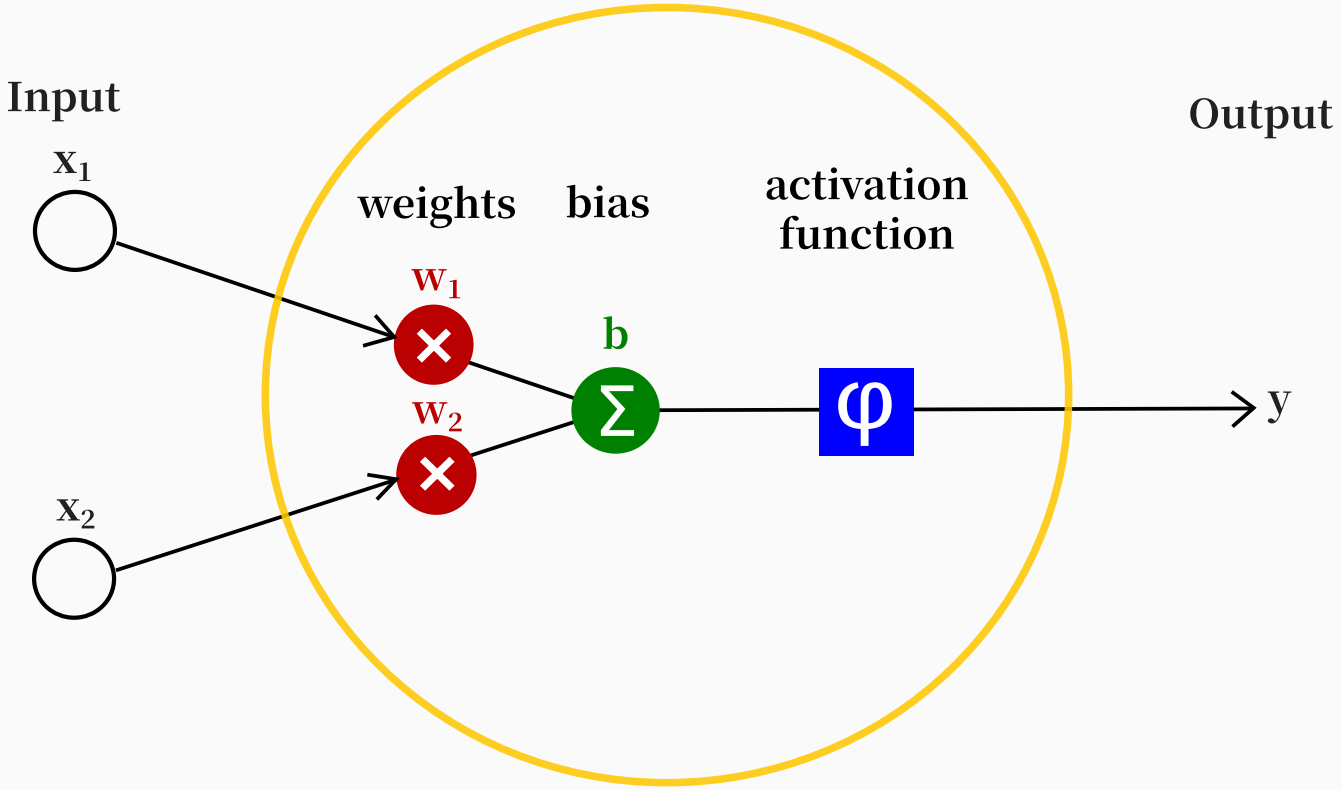
Introduction

Neural Networks

Basic idea

Input data is processed through different layers of artificial neurons stacked together to produce the desired output.

Neural Networks - Neuron



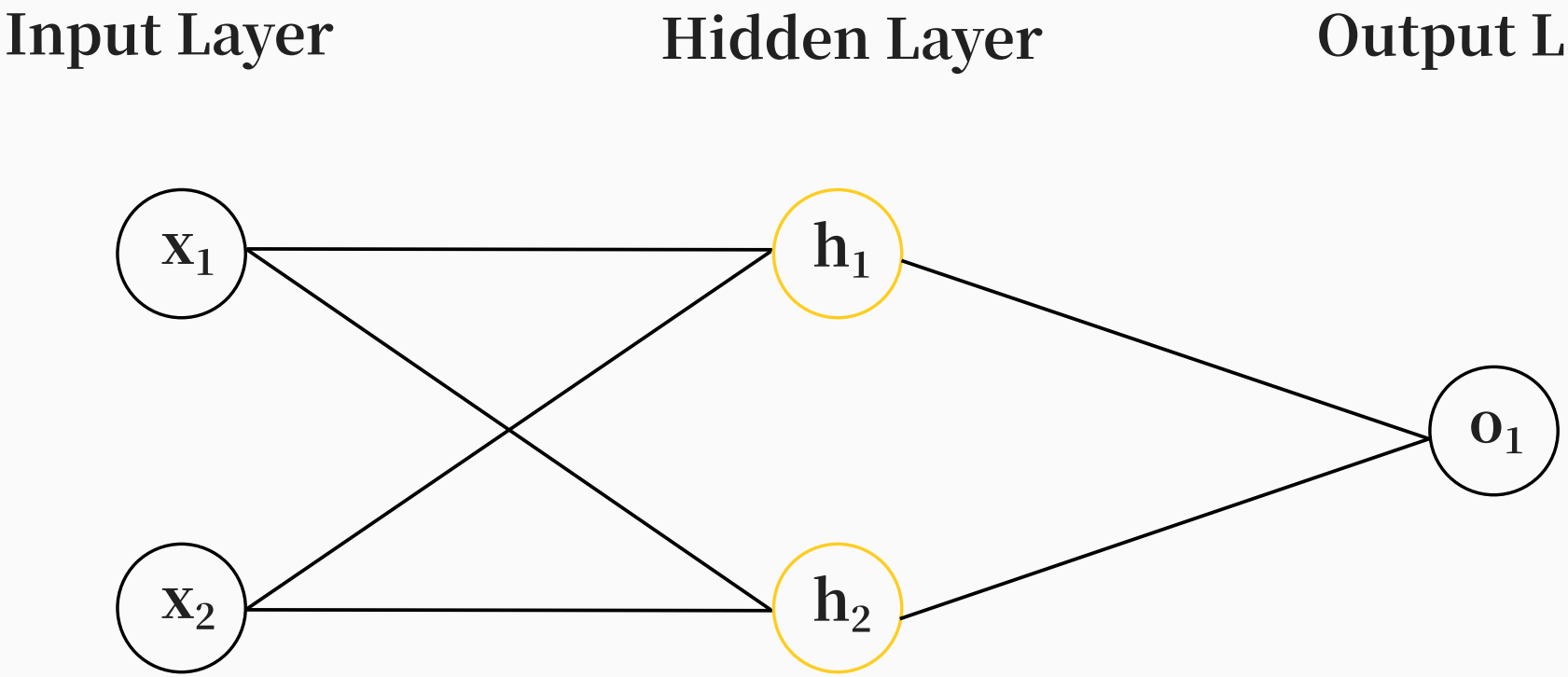
$$\phi(x_1 \cdot w_1 + x_2 \cdot w_2 + b) = y$$

Neural Networks - Neuron

What does feedforward mean?

The process of passing inputs through the model to get an output is known as **feedforward**.

Neural Networks - Example: Feedforward



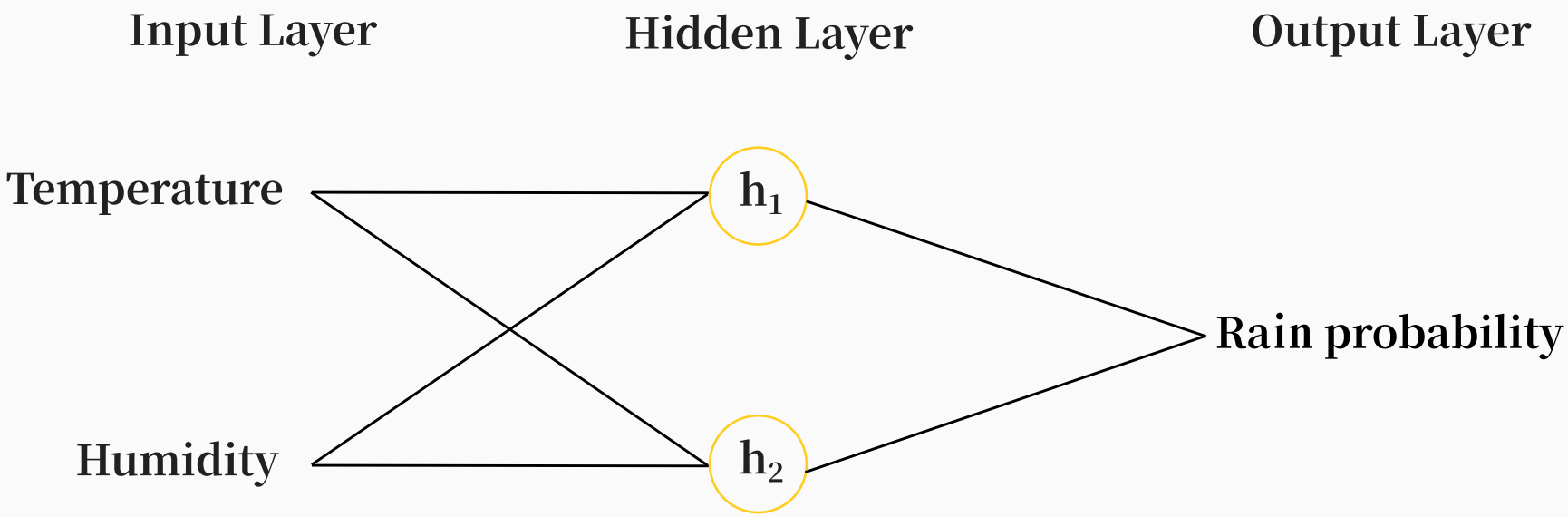
Neural Networks - Example: Feedforward

What happens when we pass in the input?

$$h_1 = \phi(x_1 \cdot w_{1,1} + x_2 \cdot w_{2,1} + b_1)$$
$$h_2 = \phi(x_1 \cdot w_{1,2} + x_2 \cdot w_{2,2} + b_2)$$

$$\Rightarrow o_1 = \phi(h_1 \cdot w_{h1} + h_2 \cdot w_{h2} + b_3)$$

Neural Networks - Training



Neural Networks - Training

What is a **loss function?**

A function that compares the target and predicted output values.

Example: **Mean-squared-error** loss

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2 = L(y_{pred})$$

$\Rightarrow$ **Training a neural network = Minimising its Loss**

Neural Networks - Training

How can we minimise the loss?

Backpropagation

How would loss change if we change $w_{1,1}$?

$$\frac{\partial L}{\partial w_{1,1}} = \frac{\partial L}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_{1,1}}$$

$\Rightarrow$ This process of calculating partial derivatives by working backwards is known as **backpropagation**.

Neural Networks - Training

How can we minimise the loss?

Problem

How does the neural network know how to change the weights and biases to minimise loss?

Solution

We can use an optimisation algorithm!
For example, **Gradient descent**:

$$w_{1,1} \leftarrow w_{1,1} - \eta \frac{\partial L}{\partial w_{1,1}} \quad \text{with } \eta\text{: learning rate}$$

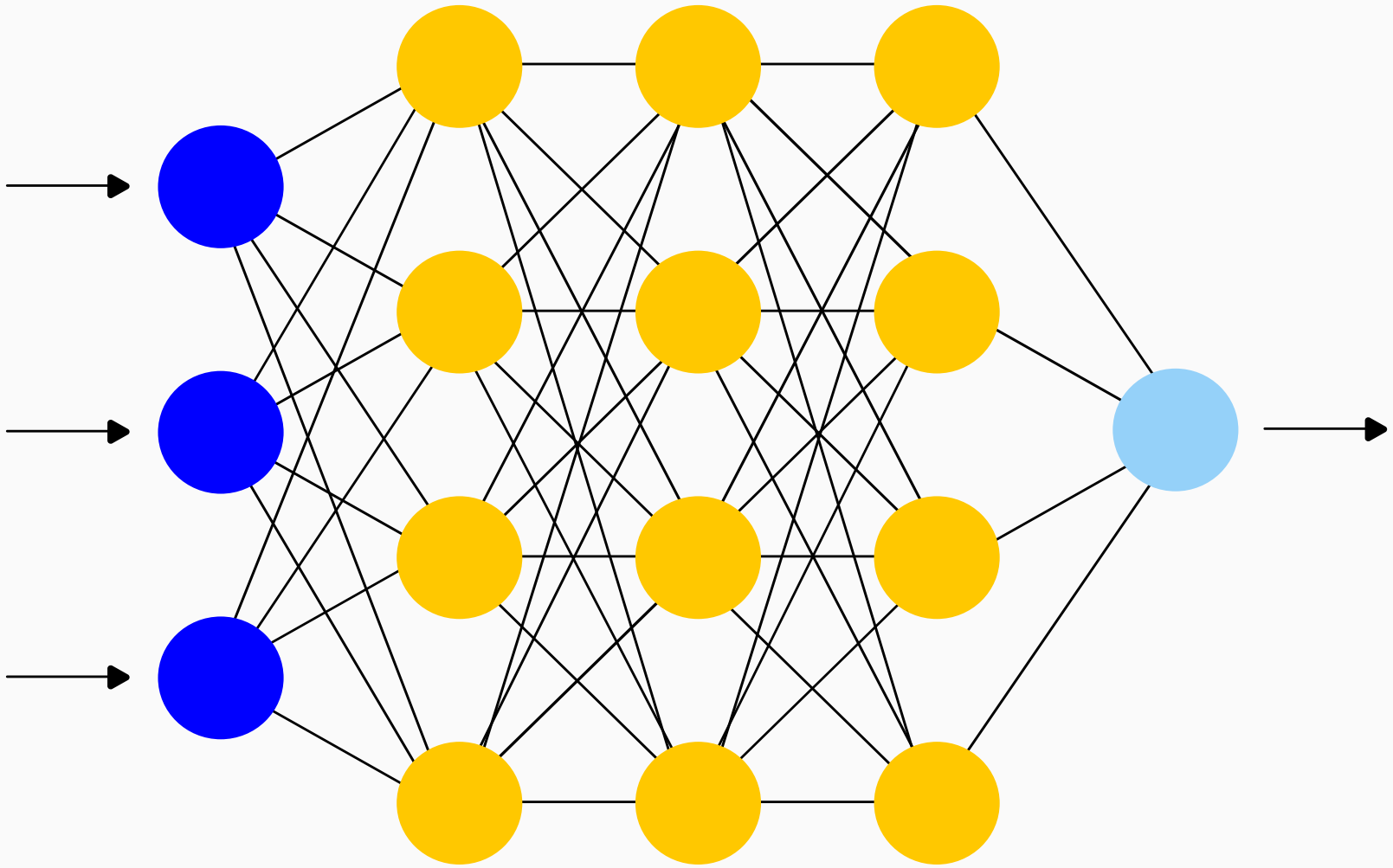
$\Rightarrow$ If we do this for **every weight and bias** in the network, the loss will slowly decrease, and our **network will improve**.

Neural Networks - Training

The whole training process:

1. Choose one sample from given dataset
2. Calculate the forward propagation
3. Compare the calculated output to the expected output (loss)
4. Calculate all the partial derivatives of loss with respect to weights and biases (backpropagation)
5. Use the update equation to adjust each weight and bias
6. Go back to step 1

Deep Neural Networks

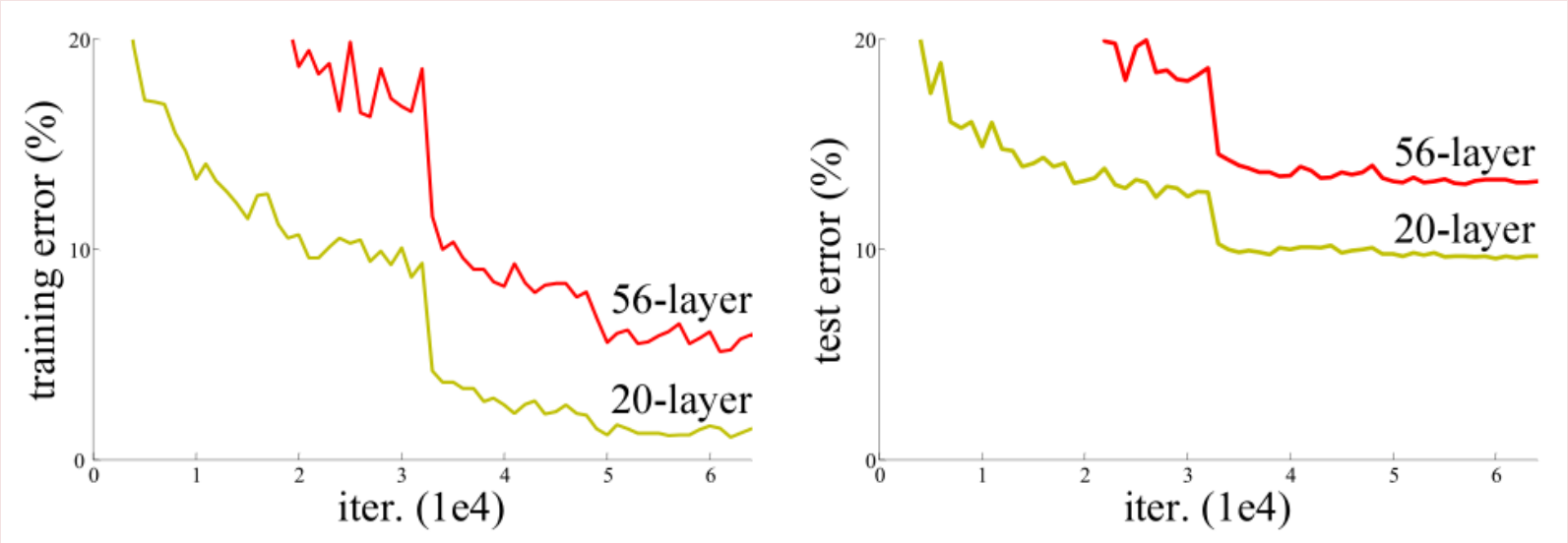


Problem: **vanishing gradient**

With **increasing** network **depth**:

→ **Decreasing gradient** during backpropagation

→ **Accuracy** gets saturated and then **degrades** rapidly

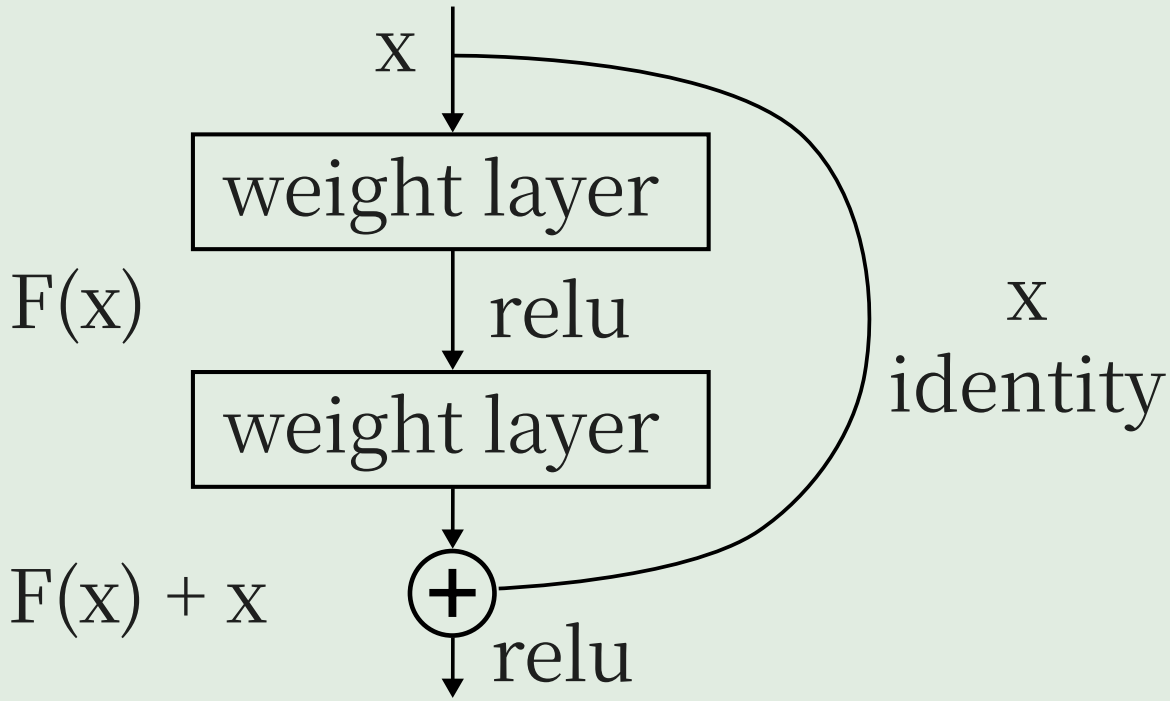


Residual Networks (ResNets)

Solution: **ResNets**

Idea: Breaking a deep NN down into small chunks of network that are connected through skip links.

$\Rightarrow$ **Residual blocks**



$\Rightarrow$ **Skip connections**

Residual Networks (ResNets)

Residual blocks

Idea: Instead of letting the layers learn the underlying mapping, let the network fit the residual mapping.

$$\mathcal{F}(x) := \mathcal{H}(x) - x$$

$\mathcal{H}(x)$: underlying mapping, $\mathcal{F}(x)$: fitted mapping

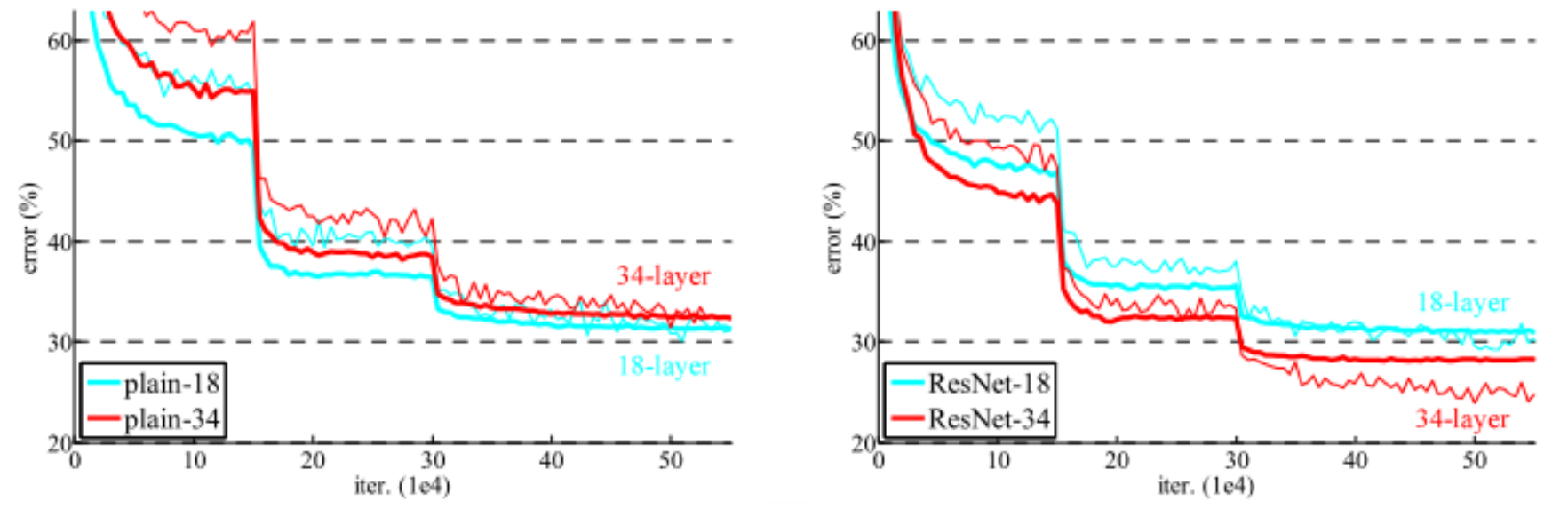
$$\Rightarrow y = \mathcal{H}(x) = \mathcal{F}(x) + x$$

$\Rightarrow$ possibility to just learn identity function
$\rightarrow$ input = output
$\rightarrow$ "skipping" layers without degrading network performance

Residual Networks (ResNets)

Main takeaways:

- By **stacking** many residual blocks, we can form a very deep NN with a lower error rate
- The **skip-connections** allow the network to skip any unneeded layers (regularisation)
  $\Rightarrow$ Help with vanishing gradient
  $\Rightarrow$ **Improved accuracy** and **faster convergence**



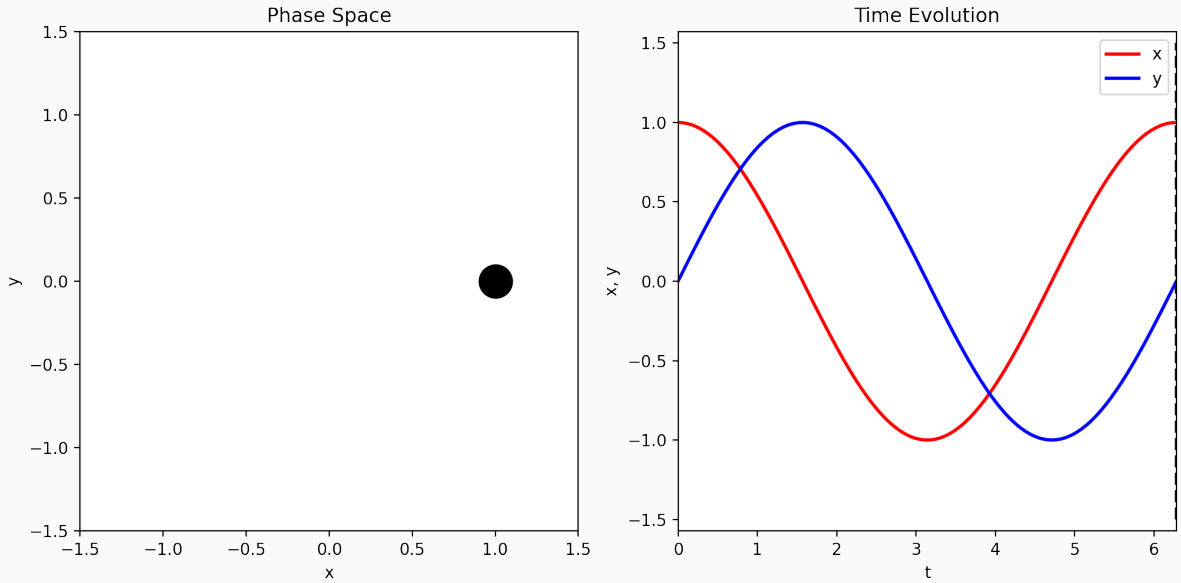
Limitations of classical ODEs

You're asked!

Can you guess the underlying ODEs?

Limitations of classical ODEs

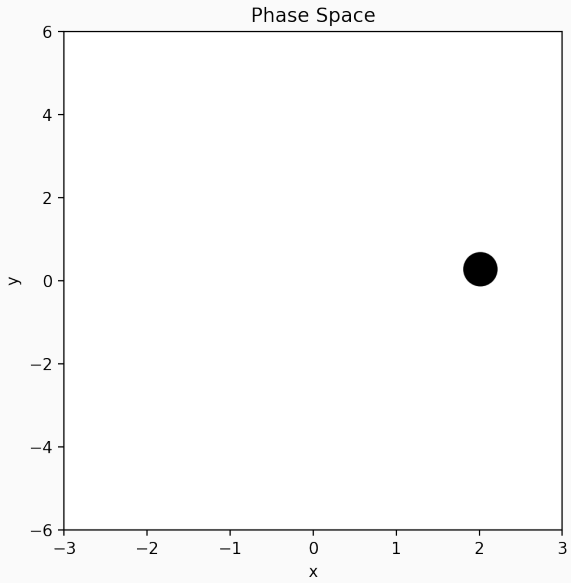
A few examples:

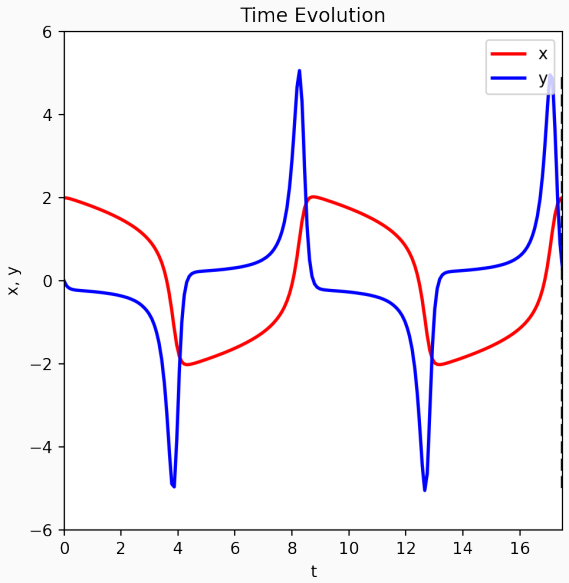


$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} -y \\ x \end{pmatrix}$$

Limitations of classical ODEs

A few examples:



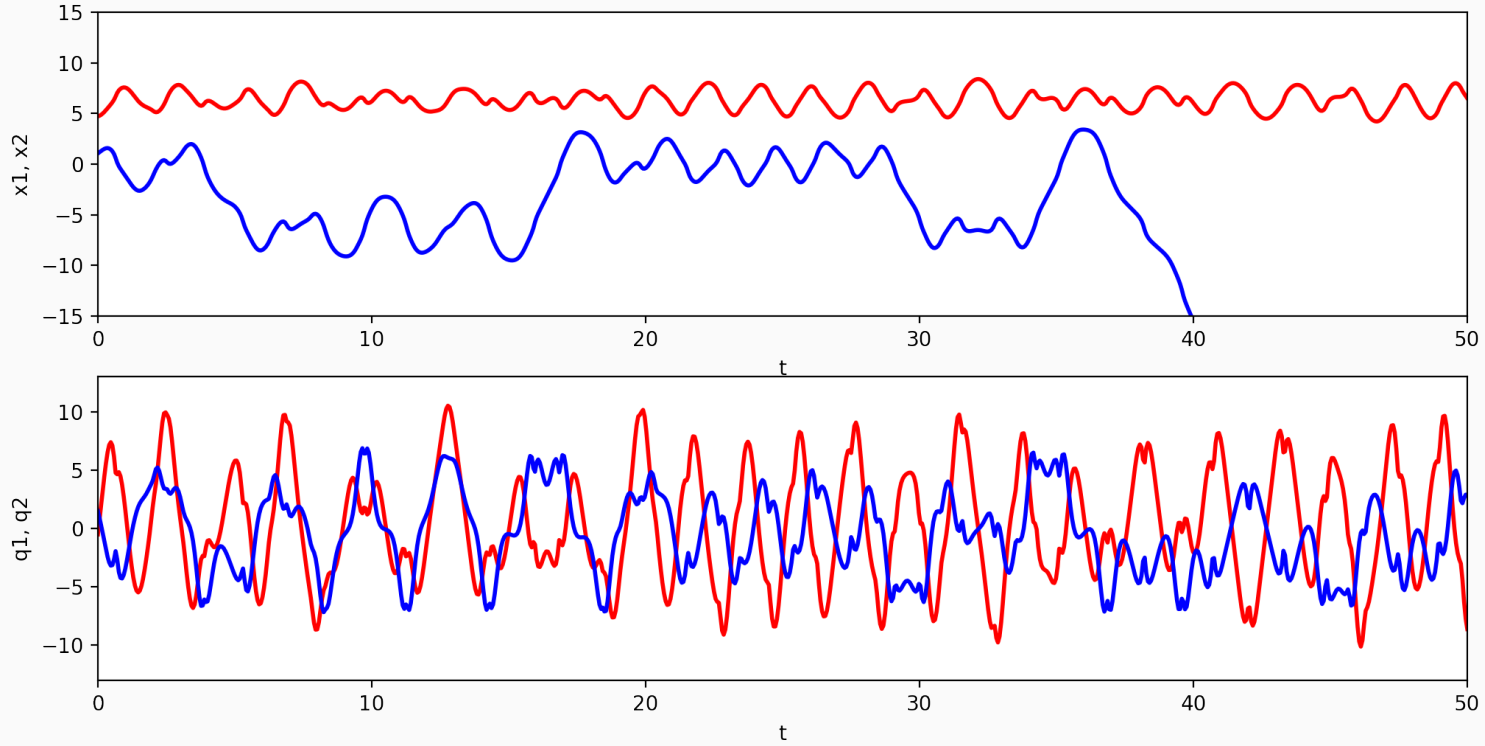


$$\ddot{x} - \epsilon(1 - x^2)\dot{x} + x = 0$$

$$\Leftrightarrow \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} y \\ \epsilon(1 - x^2)y - x \end{pmatrix}$$

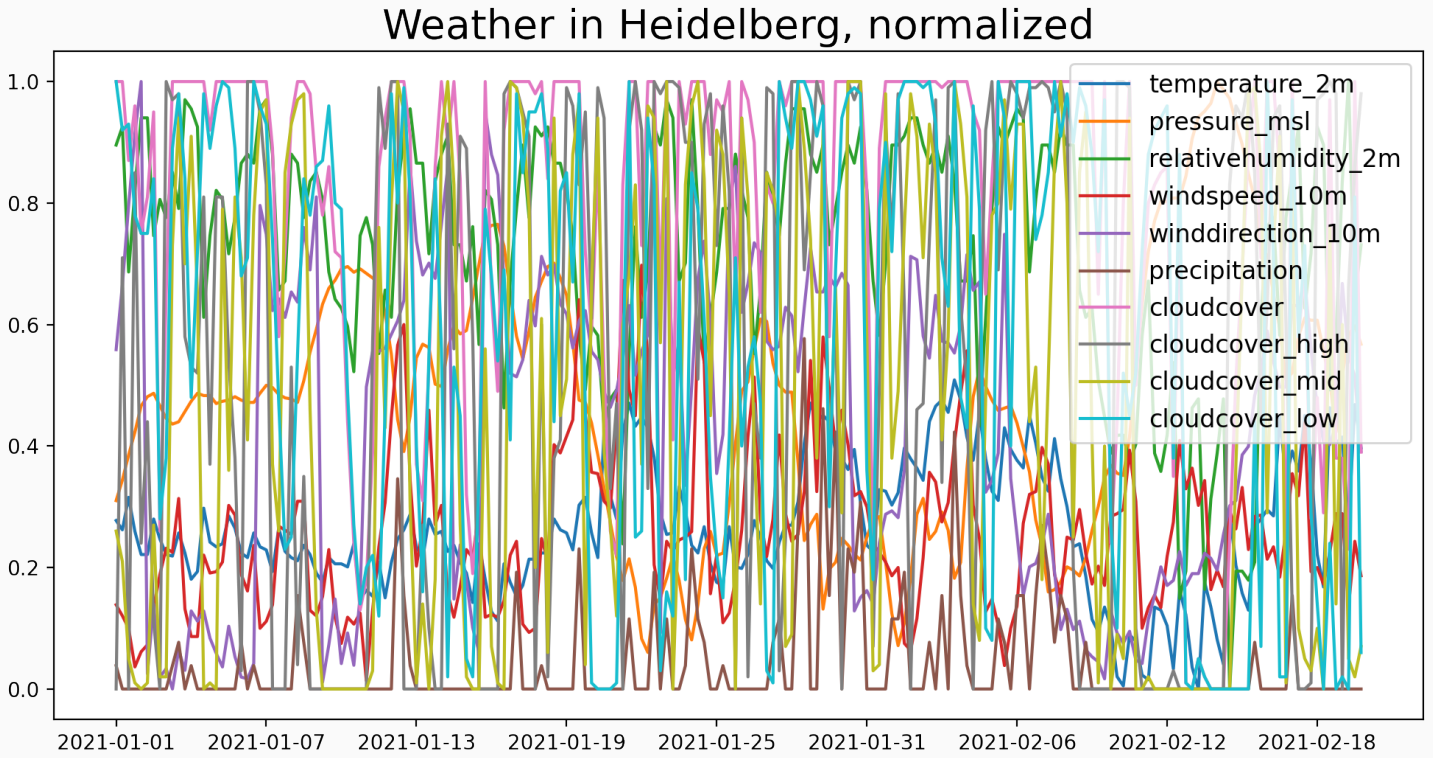
Limitations of classical ODEs

A few examples:



Limitations of classical ODEs

A few examples:



Limitations of classical ODEs

Problem

Classical ODEs are a great concept, but often not practical with real world data

How can we solve this?

~~Blackbox mystery stuff~~ Machine Learning!

Applying ML to ODEs

Applying Machine Learning to ODEs - NNs

Idea

Approximate next state $\mathbf{x}_{t+1}$ from current state $\mathbf{x}_t$ using a neural network $\mathbf{f}$:

$$\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{w})$$

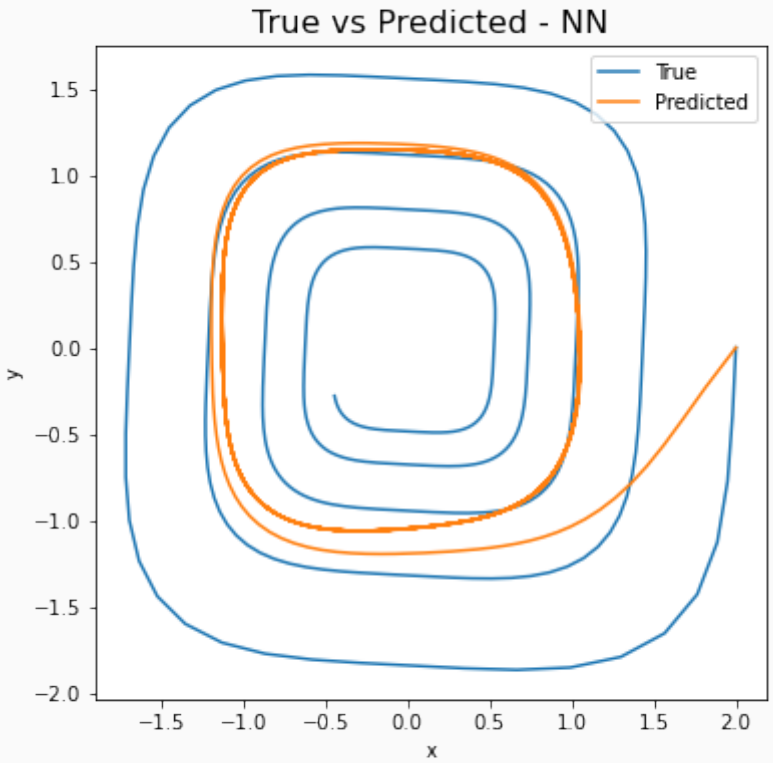
Applying Machine Learning to ODEs - NNs

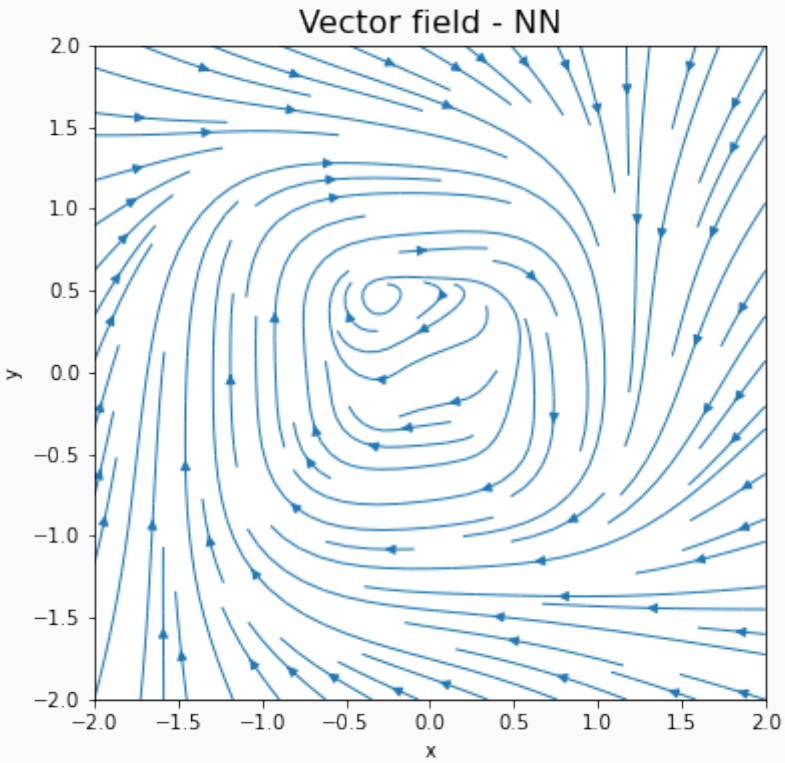
$$\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{w})$$

```python
from torch import nn

class NN(nn.Module):
    def __init__(self, dim, n, hidden_layers):
        layers = [nn.Linear(dim, n)]
        for _ in range(hidden_layers):
            layers.append(nn.Linear(n,n))
        layers += [nn.Linear(n, dim)]
        self.layers = nn.ModuleList(layers)
        self.act = nn.Tanh()

    def forward(self, x):
        for l in self.layers:
            x = self.act(l(x))
        return x
```

Applying Machine Learning to ODEs - NNs

Results:





Applying Machine Learning to ODEs - NNs

Good

Learn time evolution from data, i.e. no prior knowledge needed

Bad

Inefficient, we have to cover a wide range of possible outcomes

Applying Machine Learning to ODEs - ResNets

Idea

Approximate only the residual for next state $\mathbf{x}_{t+1}$ from current state $\mathbf{x}_t$ using a neural network $\mathbf{f}$:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{f}(\mathbf{x}_t, \mathbf{w})$$

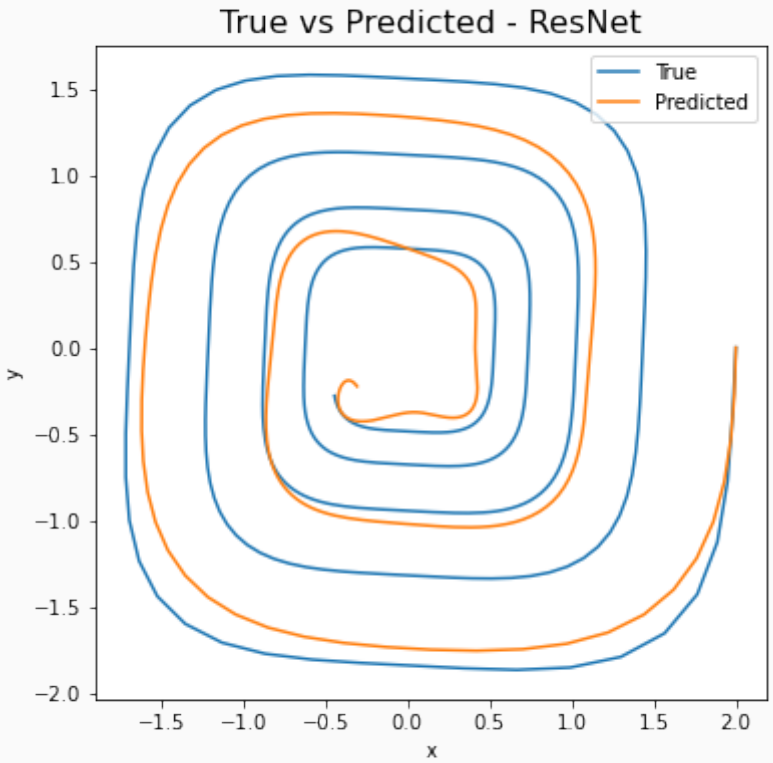
Applying Machine Learning to ODEs - ResNets

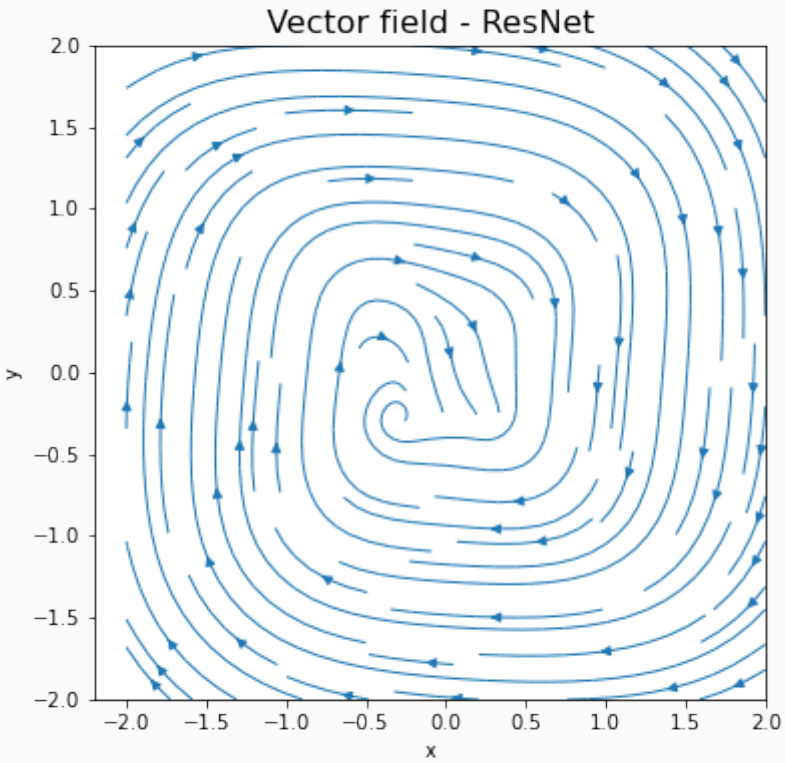
$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{f}(\mathbf{x}_t, \mathbf{w})$$

```
from torch import nn

class ResNet(nn.Module):
    def __init__(self, dim, n, hidden_layers):
        layers = [nn.Linear(dim, n)]
        for _ in range(hidden_layers):
            layers.append(nn.Linear(n,n))
        layers += [nn.Linear(n, dim)]
        self.layers = nn.ModuleList(layers)
        self.act = nn.Tanh()

    def forward(self, x):
        y = x
        for l in self.layers:
            x = self.act(l(x))
        return x + y
```

Applying Machine Learning to ODEs - ResNets

Results:





Applying Machine Learning to ODEs - ResNets

Good

More accurate representation with same number of weights (compared to plain NNs)

Bad

Time step is fixed

- What about states in between?
- What about data points at irregular times?
- We basically created maps/applied the Euler method

Neural ODEs

Neural ODEs - Concept

Let's go to the limit

Time step $\Delta t \to 0$

$$\mathbf{x}(t+1) = \mathbf{x}(t) + \mathbf{f}(\mathbf{x}(t), \mathbf{w})$$

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \mathbf{f}(\mathbf{x}(t), \mathbf{w}) \cdot \Delta t$$

$$\mathbf{f}(\mathbf{x}(t), \mathbf{w}) = \frac{\mathbf{x}(t+\Delta t) - \mathbf{x}(t)}{\Delta t}$$

$$\overset{\Delta t \to 0}{\Longrightarrow} \mathbf{f}(\mathbf{x}(t), \mathbf{w}) = \frac{d\mathbf{x}(t)}{dt}$$

Neural ODEs - Concept

Let's go to the limit

Time step $\Delta t \to 0$

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t), \mathbf{w})$$

This is amazing, we've got an ODE with our neural network ($\mathbf{f}(\mathbf{x}, \mathbf{w})$) in it!

Neural ODEs - Concept

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), \mathbf{w})$$

```python
from torch import nn, tensor

class ODEFunc(nn.Module):
    def __init__(self, dim, n, hidden_layers):
        layers = [nn.Linear(dim, n)]
        # for _ in range(hidden_layers):
        #     layers.append(nn.Linear(n,n))
        layers += [nn.Linear(n, dim)]
        self.layers = nn.ModuleList(layers)
        self.act = nn.Tanh()

    def forward(self, x):
        for l in self.layers:
            x = self.act(l(x))
        return x
```

Neural ODEs - Concept

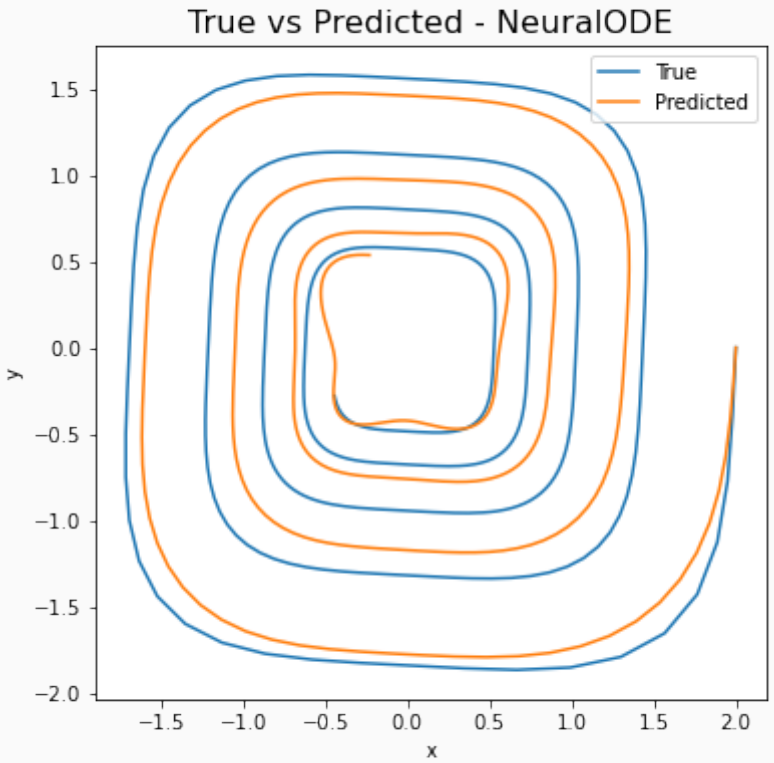
$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), \mathbf{w})$$

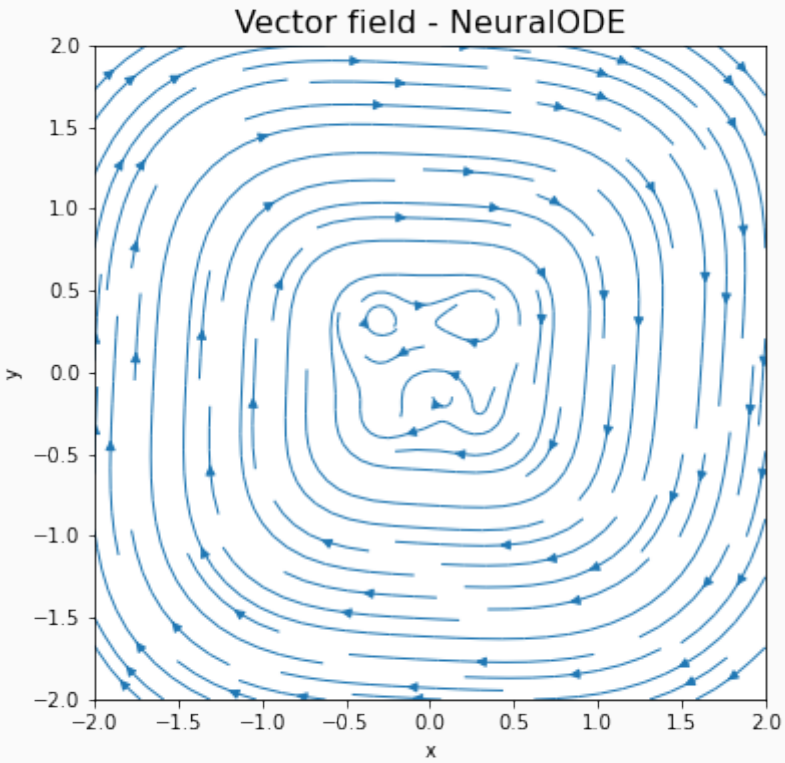
```
from torchdiffeq import odeint

x0 = torch.tensor([2.0, 0.0])
t = torch.tensor([0.0, 0.1, 0.3, 0.31, 0.8, 1.0])
x = odeint(ODEFunc, x0, t)
```

'odeint' could be any ODE solver

Neural ODEs - Concept

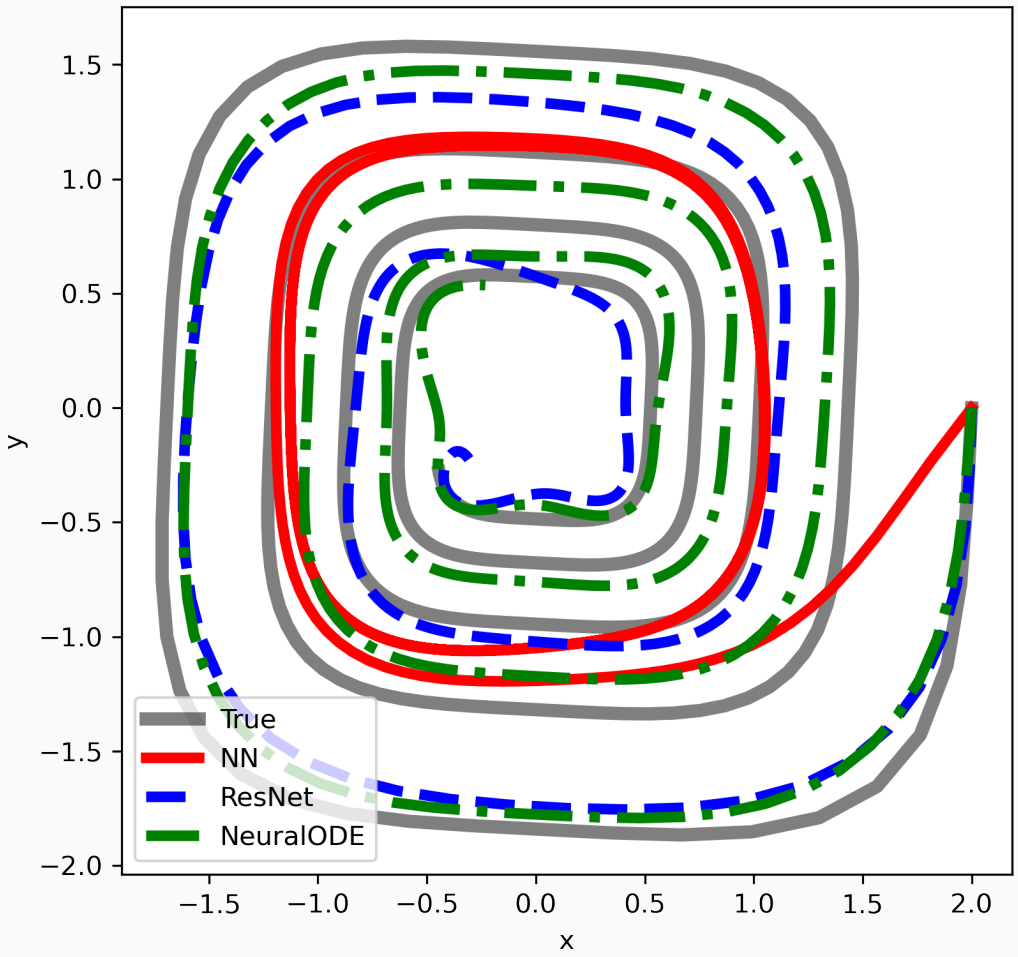
Results:





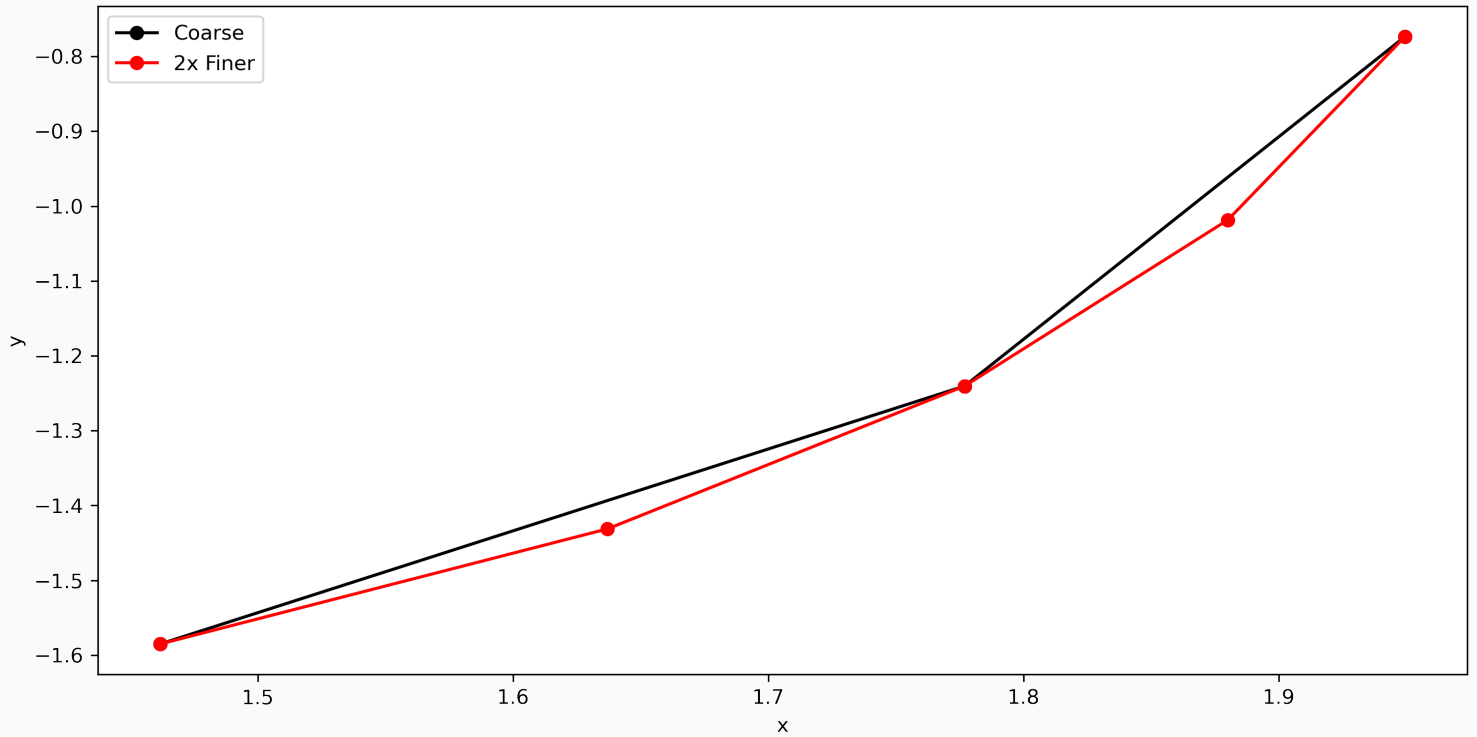
Neural ODEs - Concept

Comparison:



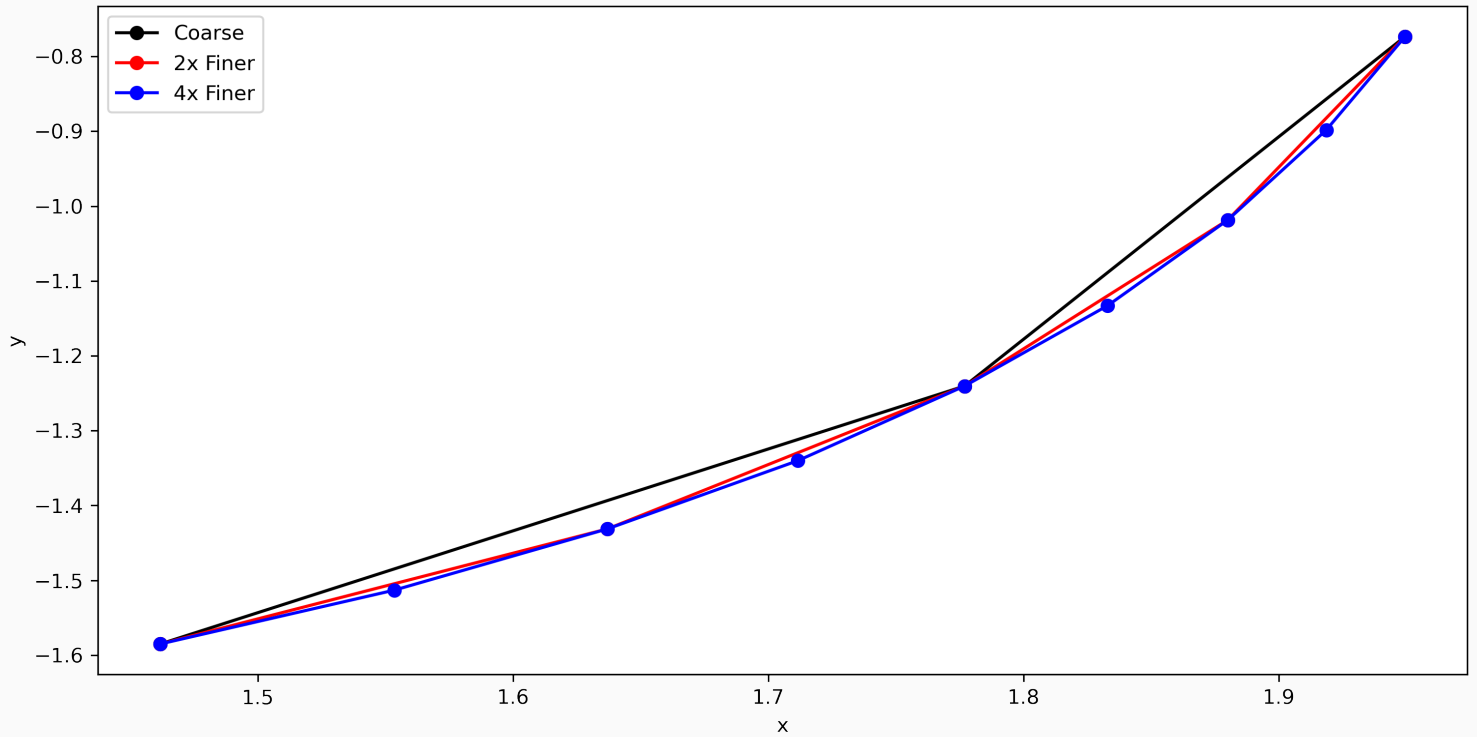
Neural ODEs - Concept

Exploiting continuous time:



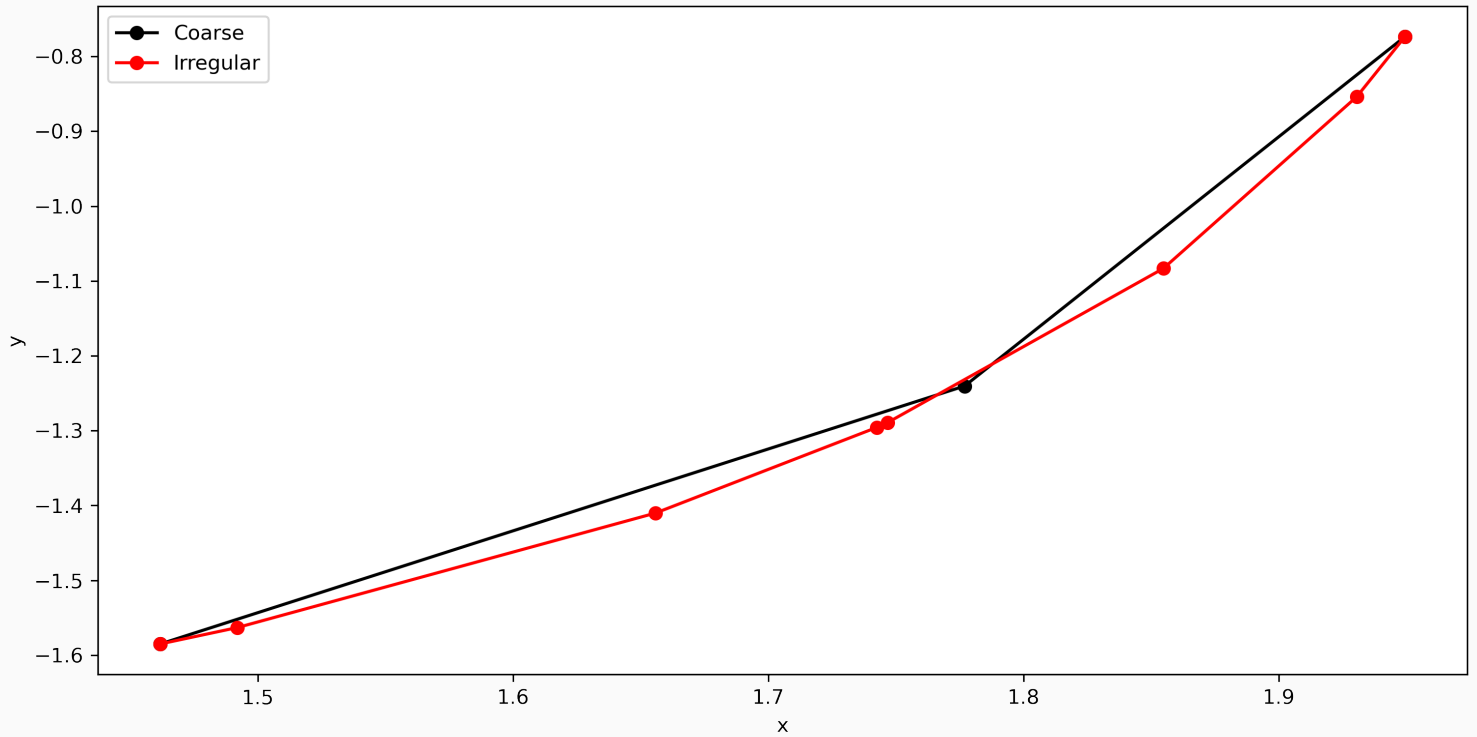
Neural ODEs - Concept

Exploiting continuous time:



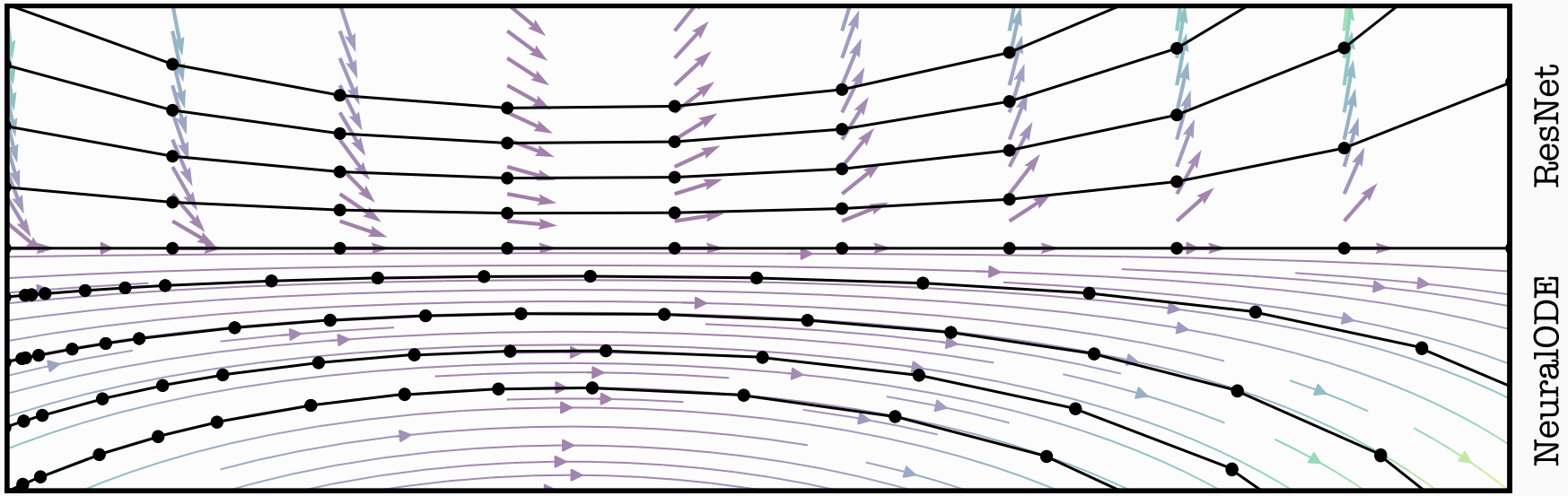
Neural ODEs - Concept

Exploiting continuous time:



Neural ODEs - Concept

Exploiting continuous time:



Neural ODEs - Training

So where's the catch?

- Training/backpropagation is not trivial in continuous time.
- Even with finite computations from ODE solver, it is intractable

Adjoint method

For loss function $L(\mathbf{x})$, define *adjoint* state $\mathbf{a}(t) = \partial L / \partial \mathbf{x}(t)$.
Loss gradient is then given by

$$\frac{dL}{d\mathbf{w}} = -\int_{t_1}^{t_0} \mathbf{a}^T(t) \frac{\partial \mathbf{f}(\mathbf{x}(t), t, \mathbf{w})}{\partial \mathbf{w}} dt$$

Neural ODEs - Training

$$\frac{dL}{d\mathbf{w}} = -\int_{t_1}^{t_0} \mathbf{a}^T(t) \frac{\partial \mathbf{f}(\mathbf{x}(t), t, \mathbf{w})}{\partial \mathbf{w}} dt$$

Components

$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{x}(t)}$: sensitivity of loss to changes in $\mathbf{x}(t)$

$\frac{\partial \mathbf{f}(\mathbf{x}, \mathbf{w})}{\partial \mathbf{w}}$: sensitivity of change of $\mathbf{x}(t)$ to changes in $\mathbf{w}$

$-\int_{t_1}^{t_0} dt$: backpropagation through time/summing up

Neural ODEs - Training

$$\frac{dL}{d\mathbf{w}} = -\int_{t_1}^{t_0} \mathbf{a}^T(t) \frac{\partial \mathbf{f}(\mathbf{x}(t), t, \mathbf{w})}{\partial \mathbf{w}} dt$$

Table 1
Comparison of the different gradient computation methods. The flow eqn. is either linear or non-linear, with P parameters and N state variables.

	Finite differences	Forward sensitivities	Adjoint
Suitability	Arbitrary	$N \gg P$	$P \gg N$
Cost	$(1 + P)$ flow eqns.	P non-linear sensitivity eqns. $+1$ flow eqn.	1 linear adjoint eqn. $+1$ flow eqn.
Key steps	1. Integrate flow eqn. 2. Parametrically perturb flow P times	1. Integrate the coupled flow and sensitivity eqns.	1. Integrate flow eqn. 2. Integrate adjoint eqn.

Figure: Comparison of gradient computing methods [Sen+14]

Neural ODEs - Training

General remarks on training with time series data

- Predicting how far?
  - One step ahead?
  - Multiple steps ahead?
  - Scheduled learning?
- How to handle missing data?

  $\rightarrow$ Not a problem for continuous time, but for discrete time
- How to measure quality of prediction?

  $\rightarrow$ Not trivial for chaotic systems
  - Error in N-step prediction
  - Kullback-Leibler divergence
  - Power Spectrum analysis
- Regularisation?
  - Jacobian norm penalty
  - Kinetic energy penalty

Neural ODEs - Applications and Examples

Where can we use Neural ODEs?

- **Time-series analysis**
- Dynamical system modelling
- Physics-informed learning
- Image and signal processing
- Generative modelling $\rightarrow$ **Continuous normalising flows**

Neural ODEs - Example: Weather simulation

Time-series: Look at weather data of Heidelberg over the span of 3 days in July 2021

→ **Parameters:** temperature, pressure, relative humidity, precipitation, cloud cover, shortwave radiation, wind speed, hour of the day

Neural ODEs - Example: CNF

Continuous normalising flows (CNF):

Concept

Using Neural ODEs to parameterise and compute the transformations.
$\Rightarrow$ **Continuous-time dynamics** described by ODEs

Advantages:
$\rightarrow$ More **flexibility** and **adaptivity** in modelling complex data distributions
$\rightarrow$ Ability to capture even **intricate transformations**

Neural ODEs - Advantages

What are the benefits?

- Continuous-time modeling
- Memory efficiency
- Adaptive computation
- Interpolation and extrapolation
- Flexibility in model design
- End-to-end learning
- Scalable and invertible normalising flows

Neural ODEs - Disadvantages

What are the limitations?

- Computational cost
- Limited control over the ODE-Solver
- Limited generalisation
- Sensitivity to training data
- Difficulty in capturing discontinuities

Neural ODEs - Outlook

Expanding to more complicated Differential Equations:

Partial Differential Equations (PDEs)

$\Rightarrow$ **Controlled Differential Equations (CDEs)**
$\rightarrow$ adaptive Framework for studying the dynamics and optimising control policies in systems governed by PDEs

Stochastic Differential Equations (SDEs)

$\Rightarrow$ **Neural SDEs**
$\rightarrow$ Framework for learning and modelling complex stochastic dynamics

Summary and Takeaways

Summary and Takeaways

- Introduction of Neural Networks and ResNets
  $\rightarrow$ universal function approximators
- Application of ResNets (and NNs) to ODEs
  $\rightarrow$ Easy to implement and good for fixed time steps
- Transition to continuous time (Neural ODEs)
  $\rightarrow$ Good for adaptive time steps
- Applications/Examples of Neural ODEs
  $\rightarrow$ Used for time-series analysis
  $\rightarrow$ Many advantages due to continuous-time modelling

Summary and Takeaways

Takeaways for working with time series data

- Know your data!
  $\rightarrow$ Make sure it's covering phase space as much as possible
- For regular data, use ResNets!
  $\rightarrow$ They're easier/faster to train
- For irregular data or intermediate states, use Neural ODEs!
  $\rightarrow$ Tweak the training length to improve your results

Thank you!

References

References

Neural Ordinary Differential Equations

Ricky T. Q. Chen*, Yulia Rubanova*, Jesse Bettencourt*, David Duvenaud
University of Toronto, Vector Institute
{rtqichen, rubanova, jessebett, duvenaud}@cs.toronto.edu

Abstract

We introduce a new family of deep neural network models. Instead of specifying a discrete sequence of hidden layers, we parameterize the derivative of the hidden state using a neural network. The output of the network is computed using a black-box differential equation solver. These continuous-depth models have constant memory cost, adapt their evaluation strategy to each input, and can explicitly trade numerical precision for speed. We demonstrate these properties in continuous-depth residual networks and continuous-time latent variable models. We also construct continuous normalizing flows, a generative model that can train by maximum likelihood, without partitioning or ordering the data dimensions. For training, we show how to scalably backpropagate through any ODE solver, without access to its internal operations. This allows end-to-end training of ODEs within larger models.

1 Introduction

Models such as residual networks, recurrent neural network decoders, and normalizing flows build complicated transformations by composing a sequence of transformations to a hidden state:

$$\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t, \theta_t) \tag{1}$$

where $t \in \{0 \dots T\}$ and $\mathbf{h}_t \in \mathbb{R}^D$. These iterative updates can be seen as an Euler discretization of a continuous transformation (Lu et al., 2017; Haber and Ruthotto, 2017; Ruthotto and Haber, 2018).

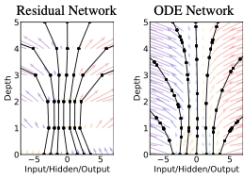


Figure 1: *Left:* A Residual network defines a discrete sequence of finite transformations. *Right:* A ODE network defines a vector field, which continuously transforms the state. *Both:* Circles represent evaluation locations.

What happens as we add more layers and take smaller steps? In the limit, we parameterize the continuous dynamics of hidden units using an ordinary differential equation (ODE) specified by a neural network:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta) \tag{2}$$

Starting from the input layer $\mathbf{h}(0)$, we can define the output layer $\mathbf{h}(T)$ to be the solution to this ODE initial value problem at some time T. This value can be computed by a black-box differential equation solver, which evaluates the hidden unit dynamics f wherever necessary to determine the solution with the desired accuracy. Figure 1 contrasts these two approaches.

Defining and evaluating models using ODE solvers has several benefits:

Memory efficiency In Section 2, we show how to compute gradients of a scalar-valued loss with respect to all inputs of any ODE solver, *without backpropagating through the operations of the solver*. Not storing any intermediate quantities of the forward pass allows us to train our models with constant memory cost as a function of depth, a major bottleneck of training deep models.

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

On Neural Differential Equations

Patrick Kidger
Mathematical Institute
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Trinity 2021

References

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Abstract

Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate residual nets with a depth of up to 152 layers—8× deeper than VGG nets [41] but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test set. This result won the 1st place on the ILSVRC 2015 classification task. We also present analysis on CIFAR-10 with 100 and 1000 layers.

The depth of representations is of central importance for many visual recognition tasks. Solely due to our extremely deep representations, we obtain a 28% relative improvement on the COCO object detection dataset. Deep residual nets are foundations of our submissions to ILSVRC & COCO 2015 competitions[1], where we also won the 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.

1. Introduction

Deep convolutional neural networks [22, 21] have led to a series of breakthroughs for image classification [21, 50, 40]. Deep networks naturally integrate low/mid/high-level features [50] and classifiers in an end-to-end multi-layer fashion, and the "levels" of features can be enriched by the number of stacked layers (depth). Recent evidence [41, 44] reveals that network depth is of crucial importance, and the leading results [41, 44, 13, 16] on the challenging ImageNet dataset [36] all exploit "very deep" [41] models, with a depth of sixteen [41] to thirty [16]. Many other non-trivial visual recognition tasks [8, 12, 7, 32, 27] have also greatly benefited from very deep models.

[1] http://image-net.org/challenges/LSVRC/2015/ and http://mscoco.org/dataset/#detections-challenge2015.

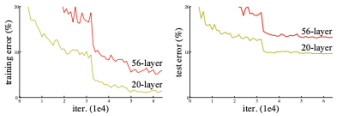


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Driven by the significance of depth, a question arises: *Is learning better networks as easy as stacking more layers?* An obstacle to answering this question was the notorious problem of vanishing/exploding gradients [1, 9], which hamper convergence from the beginning. This problem, however, has been largely addressed by normalized initialization [23, 9, 37, 13] and intermediate normalization layers [16], which enable networks with tens of layers to start converging for stochastic gradient descent (SGD) with back-propagation [22].

When deeper networks are able to start converging, a *degradation* problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is *not caused by overfitting*, and adding more layers to a suitably deep model leads to *higher training error*, as reported in [11, 42] and thoroughly verified by our experiments. Fig. 1 shows a typical example.

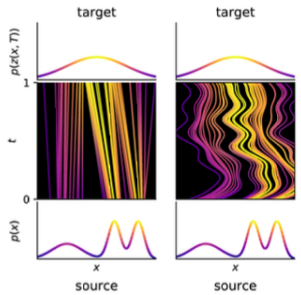
The degradation (of training accuracy) indicates that not all systems are similarly easy to optimize. Let us consider a shallower architecture and its deeper counterpart that adds more layers onto it. There exists a solution *by construction* to the deeper model: the added layers are *identity* mapping, and the other layers are copied from the learned shallower model. The existence of this constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart. But experiments show that our current solvers on hand are unable to find solutions that

How to Train Your Neural ODE: the World of Jacobian and Kinetic Regularization

Chris Finlay [1] **Jörn-Henrik Jacobsen** [2] **Levon Nurbekyan** [3] **Adam M Oberman** [1]

Abstract

Training neural ODEs on large datasets has not been tractable due to the necessity of allowing the adaptive numerical ODE solver to refine its step size to very small values. In practice this leads to dynamics equivalent to many hundreds or even thousands of layers. In this paper, we overcome this apparent difficulty by introducing a theoretically-grounded combination of both optimal transport and stability regularizations which encourage neural ODEs to prefer simpler dynamics out of all the dynamics that solve a problem well. Simpler dynamics lead to faster convergence and to fewer discretizations of the solver, considerably decreasing wall-clock time without loss in performance. Our approach allows us to train neural ODE-based generative models to the same performance as the unregularized dynamics, with significant reductions in training time. This brings neural ODEs closer to practical relevance in large-scale applications.



(a) Optimal transport map (b) generic flow

Figure 1. Optimal transport map and a generic normalizing flow.

1. Introduction

Recent research has bridged dynamical systems, a workhorse of mathematical modeling, with neural networks, the defacto function approximator for high dimensional data. The great promise of this pairing is that the vast mathematical machinery stemming from dynamical systems can be leveraged for modelling high dimensional problems in a dimension-independent fashion.

Connections between neural networks and ordinary differential equations (ODEs) were almost immediately noted after residual networks (He et al., 2016) were first proposed. Indeed, it was observed that there is a striking similarity between ResNets and the numerical solution of ordinary differential equations (E, 2017; Haber & Ruthotto, 2017; Ruthotto & Haber, 2018; Chen et al., 2018; 2019). In these works, deep networks are interepreted as discretizations of an underlying dynamical system, where time indexes the "depth" of the network and the parameters of the discretized dynamics are learned. An alternate viewpoint was taken by neural ODEs (Chen et al., 2018), where the dynamics of the neural network are approximated by an adaptive ODE solver on the fly. This latter approach is quite compelling as it does not require specifying the number of layers of the network beforehand. Furthermore, it allows the learning of homeomorphisms without any structural constraints on the function computed by the residual block.

Neural ODEs have shown great promise in the physical sciences (Köhler et al., 2019), in modeling irregular time series (Rubanova et al., 2019), mean field games (Ruthotto et al., 2019), continuous-time modeling (Yildiz et al., 2019; Kanaa et al., 2019), and for generative modeling through normalizing flows with free-form Jacobians (Grathwohl et al., 2019).

[1]Department of Mathematics & Statistics, McGill University, Montréal, Québec, Canada [2]Vector Institute, University of Toronto, Toronto, Ontario, Canada [3]Department of Mathematics, UCLA, California, USA. Correspondence to: Chris Finlay <christopher.finlay@mcgill.ca>.

Proceedings of the 37th International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020.

Recommendations

- **YouTube: Machine Learning and Simulation** for adjoint method
  `https://youtube.com/watch?v=k6s2G5MZv-I`
- **Website of Patrick Kidger** for software overview
  `https://kidger.site/`
- **Neural ODEs: breakdown of another deep learning breakthrough**
  `https://towardsdatascience.com/neural-odes-breakdown-of-another-deep-learning-breakthrough-3e78c7213795`
- **Lecture: Dynamical Systems Theory in Machine Learning**
  `https://moodle.uni-heidelberg.de/course/view.php?id=14482`