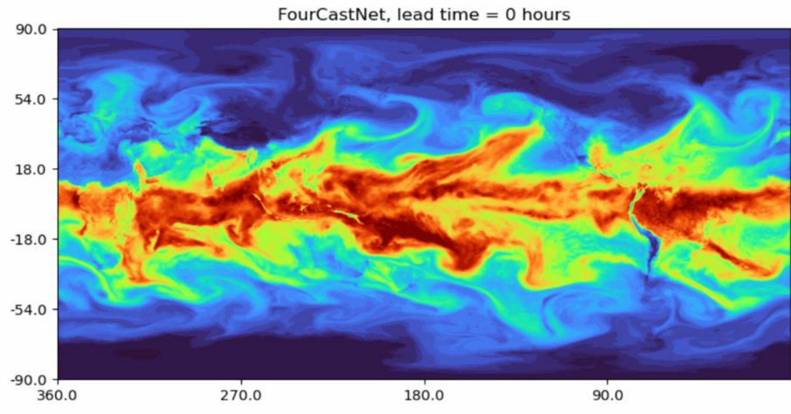
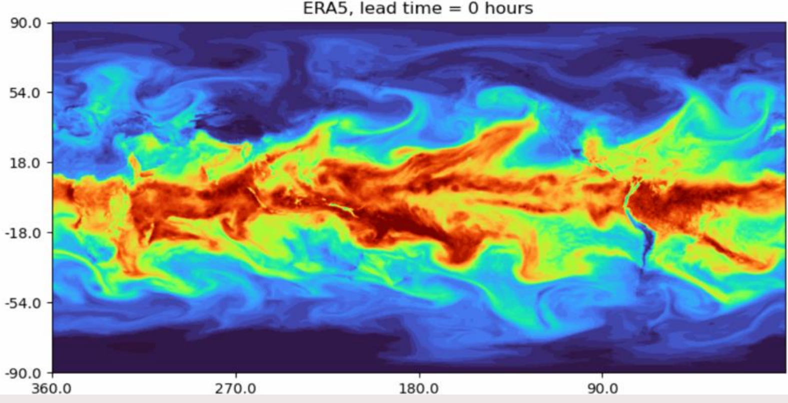
An introduction to SciML

VICTORITA DOLEAN

CASA – M&CS

AI for science: a revolution?

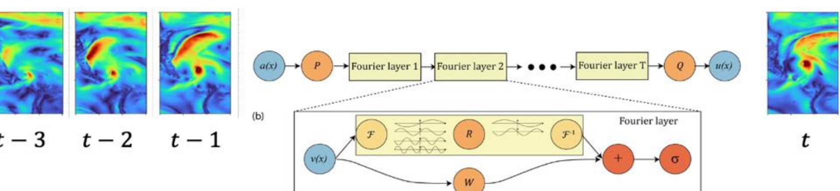




WEATHER Extreme Weather Climate Capital Weather Gang Environment Climate Lab

How Big Tech AI models nailed forecast for Hurricane Lee a week in advance

U.S. and European weather agencies are escalating their engagement with artificial intelligence as the technology rapidly advances



Pathak et al, FourCastNet: A Global Data-driven High-resolution Weather Model using Adaptive Fourier Neural Operators, ArXiv (2022)

AI for science: a revolution?

Droplets

Negative Triangularity

ITER-like shape

Snowflake

Elongated Plasma

a Learning loop

Learner

Control policy parameters

Replay buffer

(a, m, t, r)

Actor

Measurements

Environment

Control policy

Voltage commands

Targets

b Simulated environment

Sensor model

Physical parameters

Power supply

Terminate

Forward Grad–Shafranov solver (FGE)

Reward

Simulation state

c Control policy

Inputs: $m = 92$, $t \leq 132$
Neural net: MLP = 3 × 256
Outputs: $a = 19$

d Deployment

e Our architecture

Targets

Control policy

f Conventional control

Observers

Plasma shape

R

Z

Ip

Ip, R, Z, shape targets

Controllers

Offline feedforward generation

Coil-current control

Currents

Voltages

Real-time control system

e or f

g TCV

16 Poloidal field coils

Ohmic coils

Fast coil

h Vessel cross section

X-point in vacuum

Vessel

Plasma

Active X-point

Strike points

Isoflux line

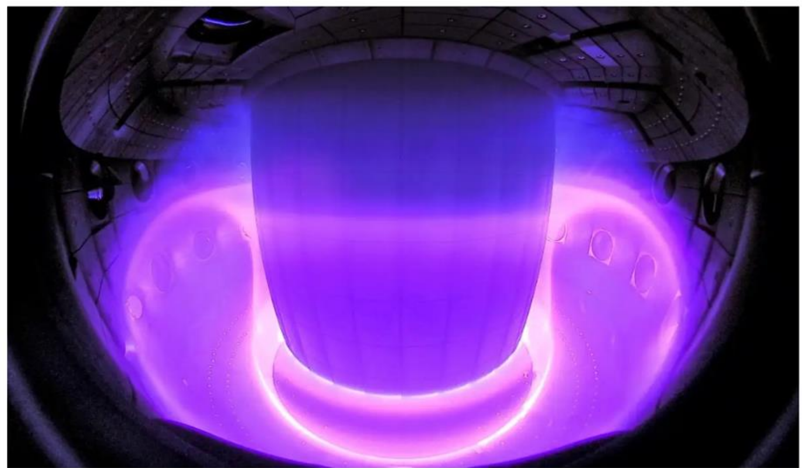
Plasma boundary

Axis R, Z position

Baffle

Legs

Limiter



Variable Configuration Tokamak (TCV) in Lausanne, Switzerland
Source: DeepMind & SPC/EPFL

nature

Explore content ⌄ About the journal ⌄ Publish with us ⌄

nature > articles > article

Article | Open access | Published: 16 February 2022

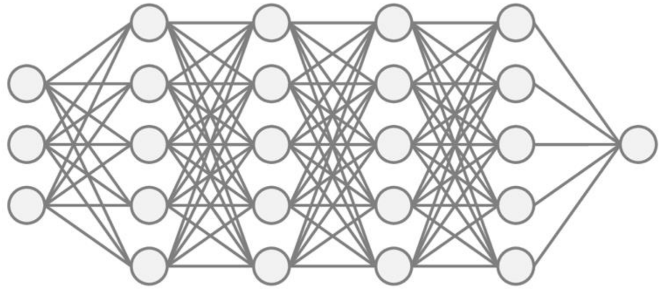
Magnetic control of tokamak plasmas through deep reinforcement learning

Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de las Casas, Craig Donner, Leslie Fritz, Cristian Galperti, Andrea Huber, James Keeling, Maria Tsimpoukelli, Jackie Kay, Antoine Merle, Jean-Marc Moret, Seb Noury, Federico Pesamosca, David Pfau, Olivier Sauter, Cristian Sommariva, ... Martin Riedmiller + Show authors

Nature 602, 414–419 (2022) | Cite this article

206k Accesses | 223 Citations | 2430 Altmetric | Metrics

What is deep learning?



Probability(Dog) = 1

Input	Model	Output
$\boldsymbol{x}$	$NN(\boldsymbol{x}, \boldsymbol{\theta})$	$\boldsymbol{y} = NN(\boldsymbol{x}, \boldsymbol{\theta})$

Neural networks are simply **flexible functions** fit to data

For example:

$$\boldsymbol{y} = W_2 \sigma(W_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$

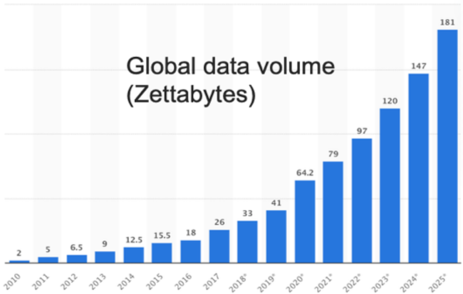
Trained ($\boldsymbol{\theta}$ learned) using:
- Many examples of inputs and outputs
- A loss function
- An optimisation algorithm, e.g. stochastic gradient descent

Deng et al, ImageNet: A large-scale hierarchical image database, CVPR (2009)

Why now?

Neural networks date back to the 1950's – so why is deep learning so popular today?

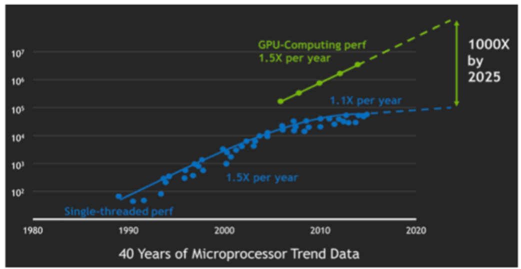
Rapidly increasing amounts of data



Source: Statista

IMAGENET

Hardware improvements



Source: NVIDIA

- Graphical processing units (GPUs)
- Highly optimised for deep learning (massively parallel)

Software improvements

- Mature deep learning frameworks
- Better training algorithms
- Deeper and more sophisticated architectures

Scientific machine learning (SciML)

Major problem

Naively using deep learning for scientific tasks usually leads to:

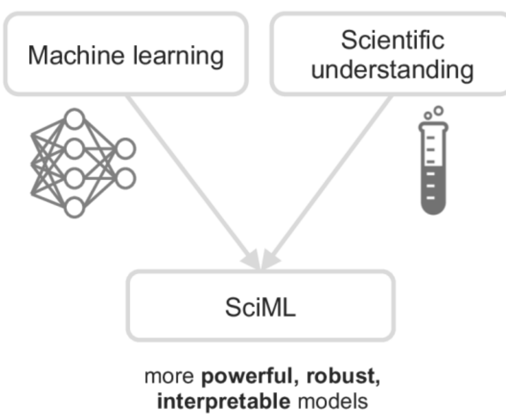
- Lack of interpretability
- Poor generalisation
- Lots of training data required

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

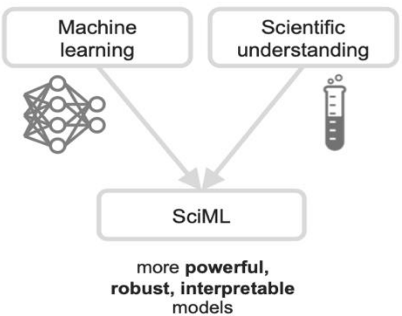
Solution



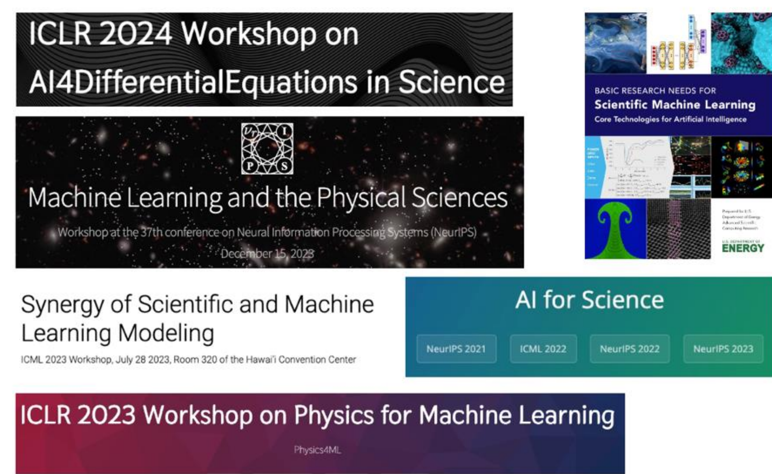
more **powerful**, **robust**, **interpretable** models

Scientific machine learning

Hamiltonian neural networks
Solver-in-the-loop
Physics-constrained Gaussian processes
Learned sub-grid processes
Hidden physics models
Physics-informed neural networks
AI Feynman
DeepONets
PDE-Net
Algorithm unrolling
AlphaFold
Learned regularisation
Differentiable simulation
Physics-informed neural operators
Fourier neural operators
Encoding conservation laws
Encoding physical symmetries
Neural ODEs

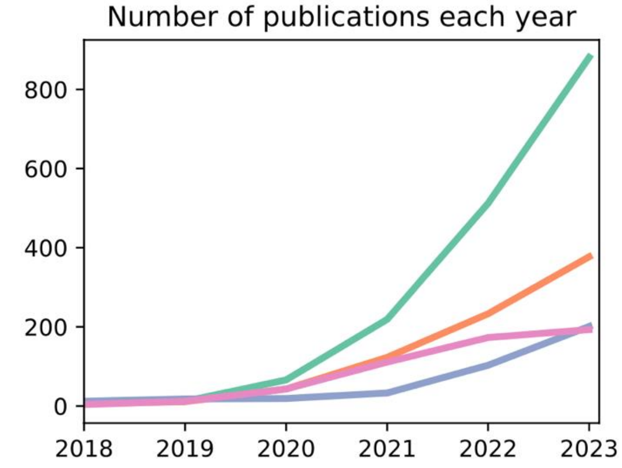


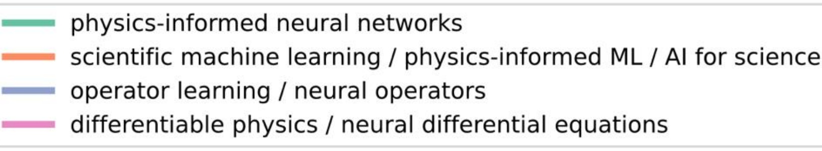
A rapidly growing field



The Symbiosis of Deep Learning and Differential Equations (DLDE)

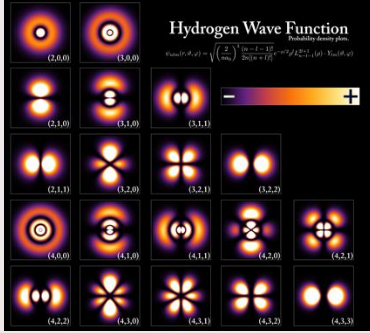
NeurIPS 2022 Workshop





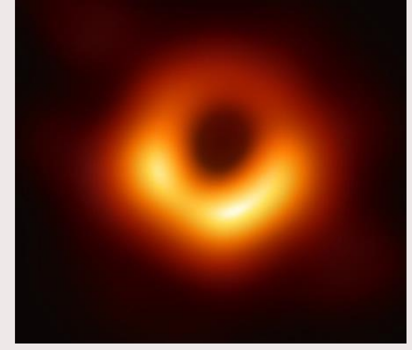
Source: Scopus keyword search (Feb 2024)

Importance of PDEs



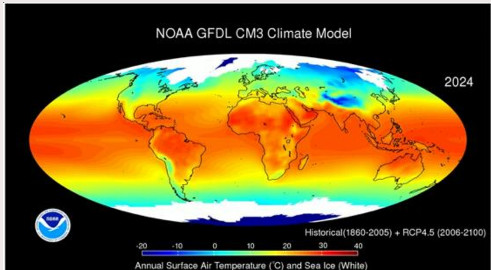
Source: Wikipedia

Schrödinger equation



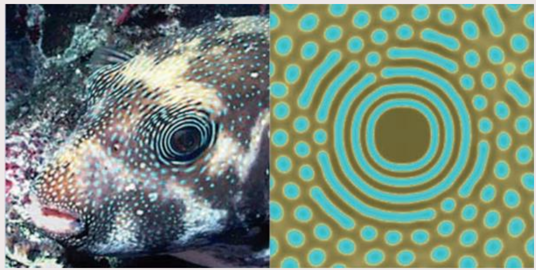
Source: The Event Horizon Telescope (2019)

Einstein field equations



Source: NOAA

Navier-Stokes equations



Source: Kondo and Miura, Science (2010)

Reaction-diffusion equation

Key scientific tasks: simulation

$$b = F(a)$$

$a =$ set of input conditions

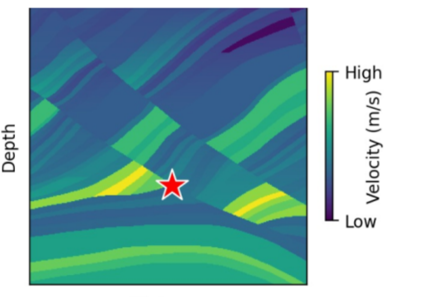
$F =$ physical model of the system (usually a PDE)

$b =$ resulting properties given F and a

Key scientific tasks: simulation

Simulation is:

- Crucial for practically all domains of science
- Essential for understanding the behaviour of complex phenomena
- Usually used as a starting point for other tasks (e.g. inverse / control / design problems)

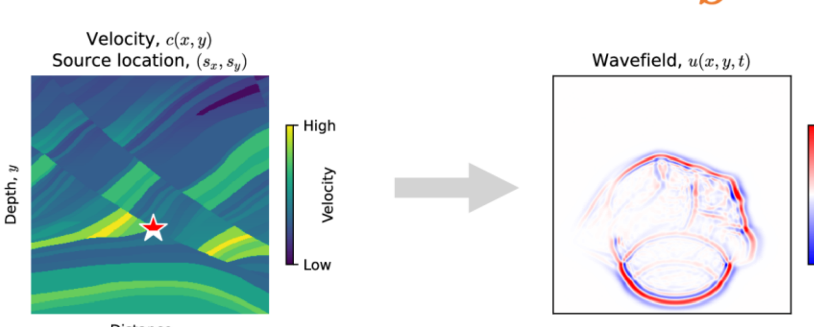


$$\nabla^2 u - \frac{1}{c(x)^2}\frac{\partial^2 u}{\partial t^2} = s$$

Wave equation

Key scientific tasks: simulation

$$b = F(a)$$



$$\nabla^2 u - \frac{1}{c(x)^2}\frac{\partial^2 u}{\partial t^2} = \delta(x = s, t = 0)$$
$$u(x, t = 0) = 0$$
$$u'(x, t = 0) = 0$$

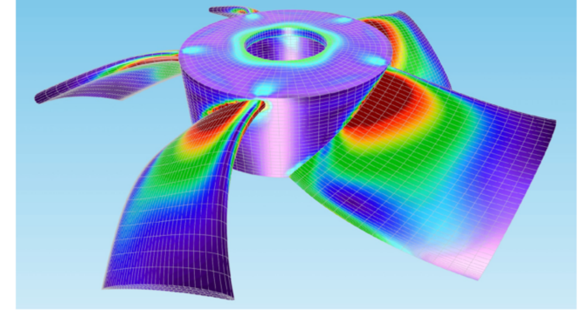
a = velocity model, $c(x, y)$, source location, s

b = wavefield, $u(x, t)$

F = A method for solving the wave equation (e.g. finite difference simulation)

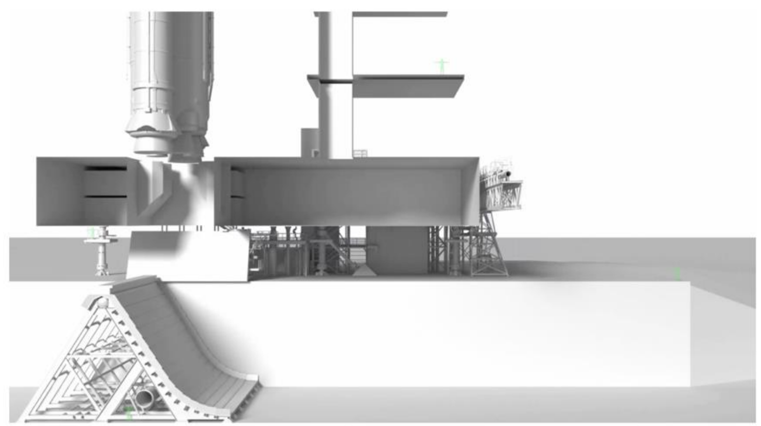
How can we solve simulation problems?

- Usually, analytical solutions do not exist, and we must resort to **numerical** modelling
- Many approaches exist, the most appropriate choice highly depends on the specific problem and scientific domain
- Popular methods (for systems modelled by PDEs) include:
  - finite difference methods, finite element methods, finite volume methods, spectral methods, domain decomposition, mesh-free methods, …



Mesh for finite element method
Source: COMSOL

Challenges of simulation



Angel et al, Predicting SLS Launch Environment using a Novel Multiphase Formulation (NASA) SC22 (2022) Source: NASA

Required: 500 million grid cells, ran for several weeks on 8,000 cores, generating 400 TB (!)

- Typically, **computational cost** is the main challenge
- Simulation can require **elaborate** parallel software implementations (especially for multi-scale, multi-physics systems) with 10,000s of lines of code
- Significant **human effort** is often required, e.g. in defining high-quality meshes for finite element simulations

How to Contribute to OpenFOAM

OpenFOAM is a large piece of software (of the order of 1 million lines of code) in a complex area of scientific application. Since its open source release in 2004, it has become the CFD software of choice for many thousands of people from industry, government laboratories, academic institutions, etc., who download OpenFOAM and use it for free. With such a large user base, sometimes working on mission-critical applications, we have a responsibility to maintain OpenFOAM as a robust, efficient, and scalable CFD software package for a wide range of applications.

Source: OpenFOAM

Key scientific tasks: inverse problems

$$b = F(a)$$

a = set of input conditions

F = physical model of the system (usually a PDE)

b = resulting properties given F and a

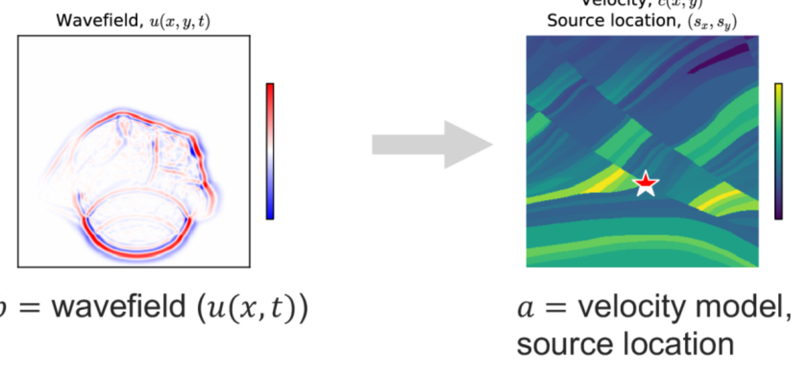
Key scientific tasks: inverse problems

$$b = F(a)$$

Inverse problems are **pervasive** across all domains of science and solving them is essential for many real-world tasks

Example inverse problems:

- Seismic imaging
- Magnetic resonance imaging
- Image denoising
- Estimating infection rates
- Design problems
- …



$b =$ wavefield $(u(x, t))$

$a =$ velocity model, source location

$F =$ A method for solving the PDE (e.g. finite difference simulation)

How can we solve inverse problems?

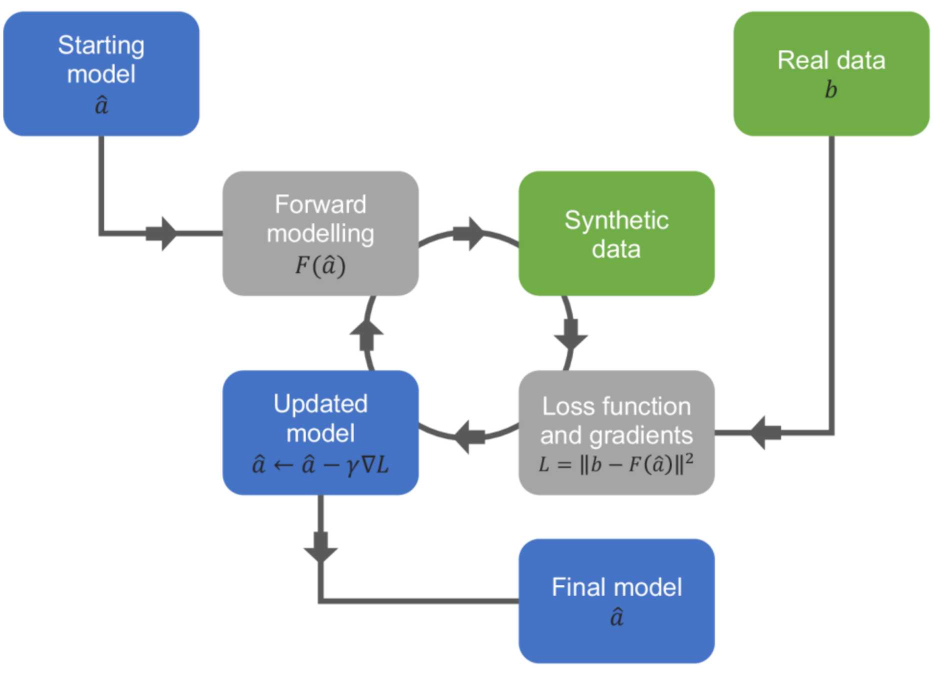
$$b = F(a)$$

- Fundamentally, inverse problems are **search** problem
- It is often useful to frame them as an optimisation problem, for example:

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

- If F is differentiable, one option is to use gradient-based methods (e.g. **gradient descent**)
- Otherwise, we can use gradient-free methods (e.g. evolutionary algorithms, Bayesian optimisation, brute-force search, …)

Solving inverse problems with gradient descent



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

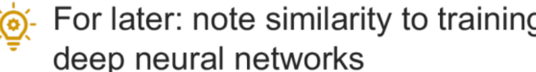
Loss function is:

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

Gradient descent step:

$$\hat{a} \leftarrow \hat{a} - \gamma \nabla L$$

- Requires F to be differentiable



For later: note similarity to training deep neural networks

Solving inverse problems with gradient descent

Starting model
$\hat{a}$

$\hat{a}$ = starting velocity model

F = (differentiable) finite-difference solver for the wave equation

Real data
b

Wavefield, $u(x, y, t)$

b = wavefield

Forward modelling
$F(\hat{a})$

Synthetic data

Updated model
$\hat{a} \leftarrow \hat{a} - \gamma \nabla L$

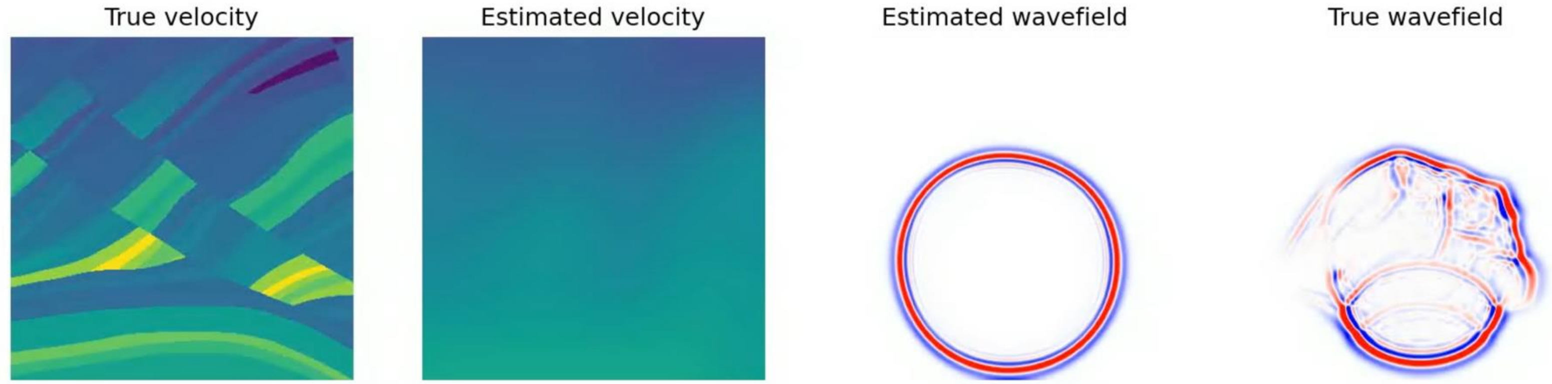
Loss function and gradients
$L = \|b - F(\hat{a})\|^2$

Final model
$\hat{a}$

Velocity, $c(x, y)$
Source location, (s_x, s_y)

a = true velocity model

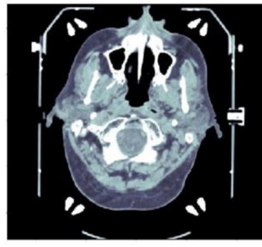
Solving inverse problems with gradient descent



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

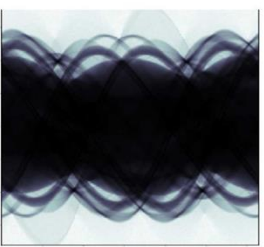
- In geophysics, this inverse problem is known as **full waveform inversion**

Challenges of inverse problems



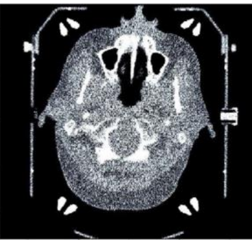
Ground truth computed tomography image

$a(x)$



Resulting tomographic data (sinogram)

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



Result of inverse algorithm (filtered back-projection)

$\hat{a}(x)$

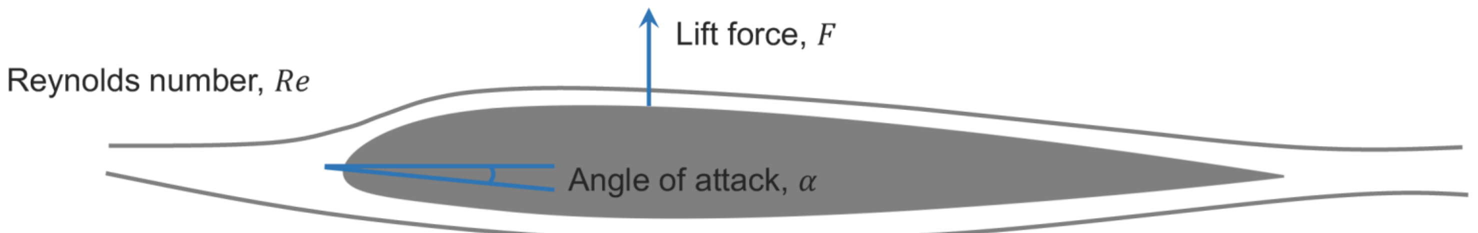
Adler et al, Solving ill-posed inverse problems using iterative deep neural networks, Inverse Problems (2017)

Typically, inverse problems are **incredibly challenging** to solve because:

- They are usually **ill-posed** (not enough information for a unique solution)
- Observed real-world data is usually **noisy** and **sparse**
- Often require forward modelling to be carried out thousands of times – making them extremely **computationally demanding**

Key scientific tasks: design

- Also a type of inverse problem



Aerofoil shape (set of points), $\{x_i, y_i\}_{i=1}^{N}$

Design task:

Find $\{x_i, y_i\}_{i=1}^{N}$, α which maximize F

Key scientific tasks: discovering unknown physics

$$b = F(a)$$

$a =$ set of input conditions

$F =$ physical model of the system (usually a PDE)

$b =$ resulting properties given F and a

Challenges of discovering unknown physics

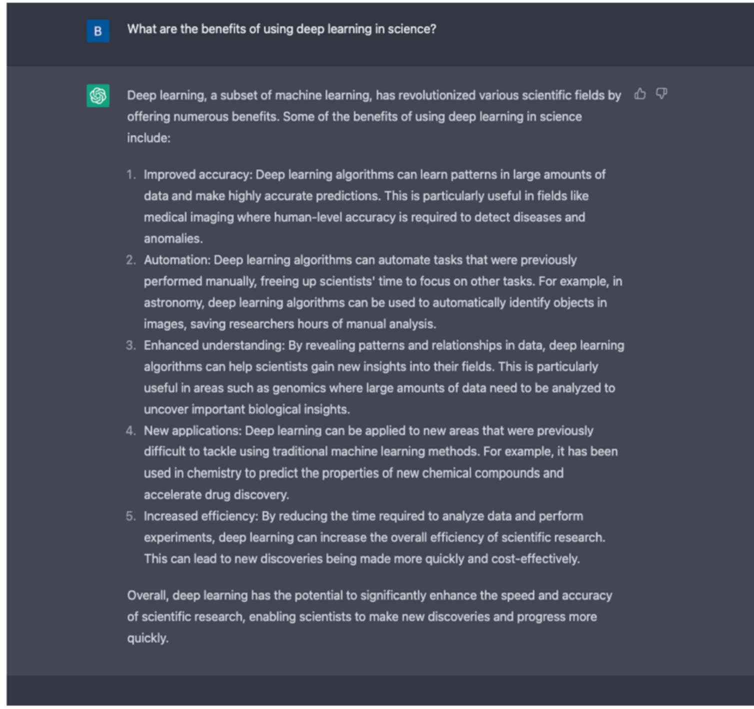
$$b = F(a)$$

$E = mc^2$

- Historically, F (= laws of physics) has been found through remarkable human **intuition**
- Constant interplay between theory and experiment
- From a computational standpoint, discovering physics is like solving an inverse problem (trying to fit a model to observed data)
- But the model should be **explainable**, **generalisable** and make **novel** predictions

How can deep learning help?

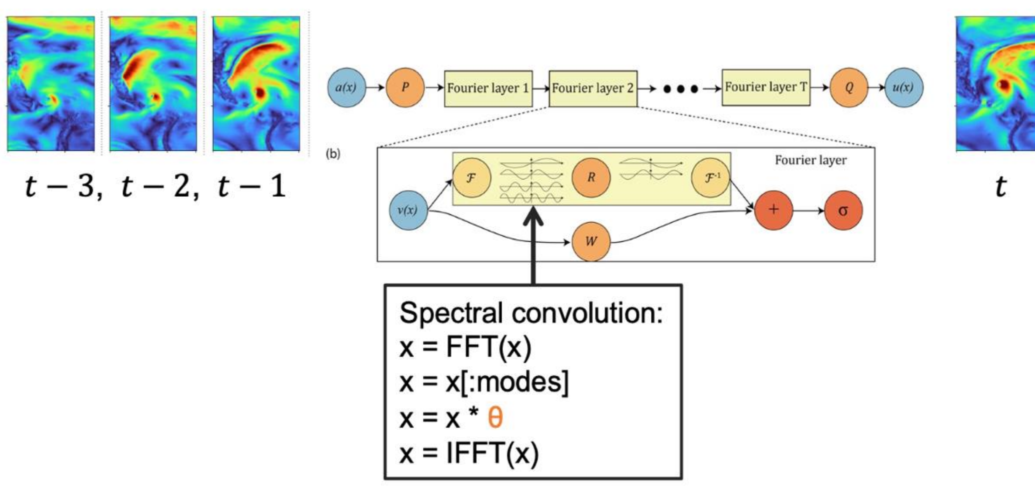
How can deep learning help?

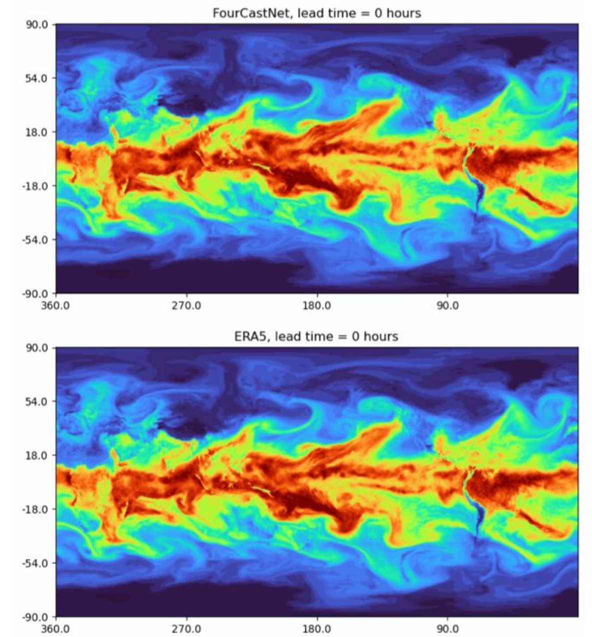


- Accuracy
- Efficiency
- Automation
- Discovery

How Deep Learning can help?

Fourier Neural Operator





Idea: use deep neural network to predict next timestep of multiple atmospheric variables, given previous timesteps

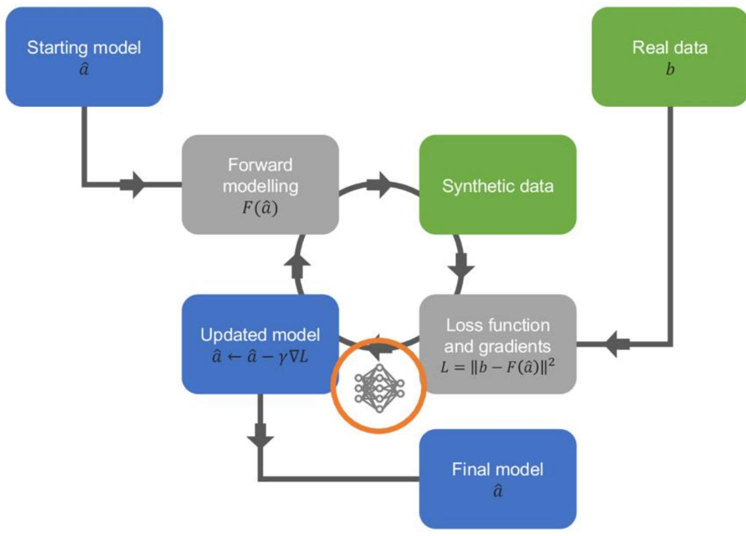
- Use a Fourier neural operator architecture
- **4-5 orders of magnitude** faster than numerical simulation

Pathak et al, FourCastNet: A Global Data-driven High-resolution Weather Model using Adaptive Fourier Neural Operators, ArXiv (2022)

Li et al, Fourier Neural Operator for Parametric Partial Differential Equations, ICLR (2021)

Example - inverse problems

Learned gradient descent

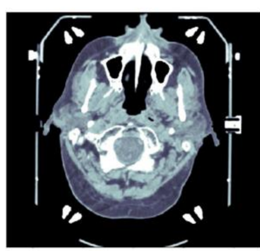


Adler et al, Solving ill-posed inverse problems using iterative deep neural networks, Inverse Problems (2017)

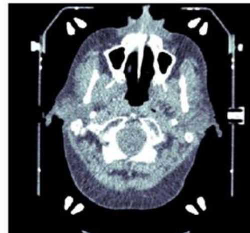
Idea: use a neural network to **update** gradient before descent step

1. Initialise $\hat{a}$
2. Loop:
   1. Compute gradient, $g = \frac{\partial L(\hat{a})}{\partial \hat{a}}$
   2. Update gradient, $g \leftarrow NN(g, \hat{a}, b, \theta)$
   3. Take step, $\hat{a} \leftarrow \hat{a} - \gamma g$

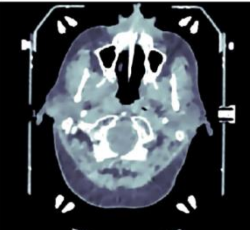
Network is trained using many example inverse problems, and differentiating through entire algorithm **end-to-end**



Ground truth

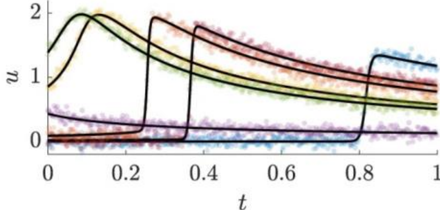


Traditional inversion



Learned gradient descent

Example - equation discovery



Ground truth: $u_t + uu_x - 0.0032u_{xx} = 0$

Discovered: $u_t + 1.002uu_x - 0.0032u_{xx} = 0$

$$u_t = \Lambda\phi$$

$$\Lambda = (\lambda_1 \;\; \lambda_2 \;\; \lambda_3 \;\; \lambda_4 \;\; \lambda_5 \; \dots) \qquad \phi = \begin{pmatrix} \hat{u}_x \\ \hat{u}_{xx} \\ \hat{u}_t \\ \hat{u}_{tt} \\ \hat{u}_{xt} \\ \dots \end{pmatrix}$$

Idea: fit a neural network to observed data, then regress over a **library** of gradients to “discover” underlying equations

Input: $\tilde{u}(x,t)$ = (noisy) observational data

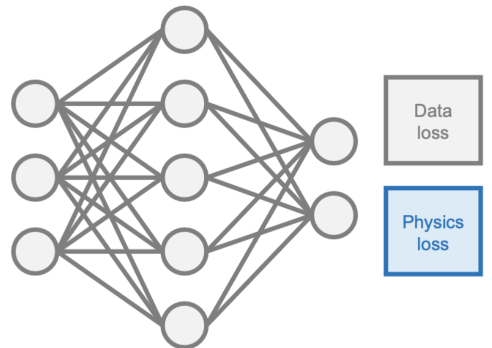
1. Fit a neural network to data, $\hat{u} = NN(x,t;\theta) \approx \tilde{u}$, using supervised learning
2. Compute various gradients of network, for example $\hat{u}_x, \hat{u}_{xx}, \hat{u}_t, \hat{u}_{tt}, \hat{u}_{xt}$ ..., at many random (x,t) locations
3. Carry out a (sparse) **linear regression** over combinations of these gradients to “discover” underlying equation

Chen et al, Physics-informed learning of governing equations from scarce data, Nature communications (2021)

The state-of-the-art: scientific machine learning

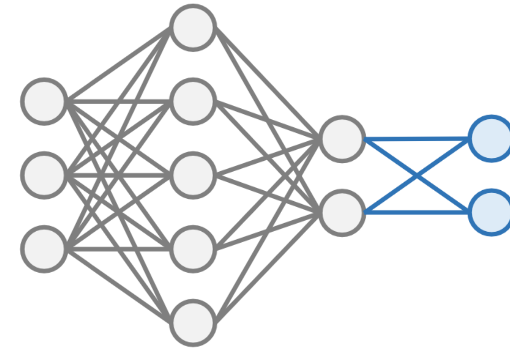
Ways to incorporate scientific principles into machine learning

Loss function



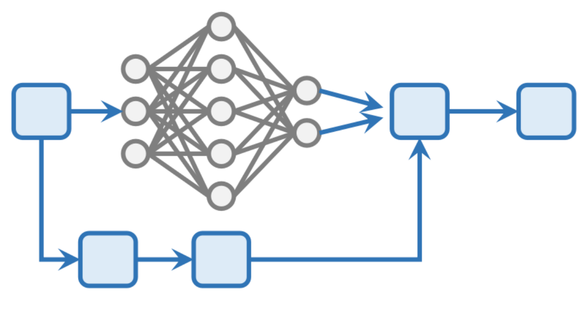
Example:
Physics-informed neural networks
(add governing equations to loss function)

Architecture



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

Hybrid approaches



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

A plethora of SciML techniques

Scientific task

SciML technique

	Forward simulation	Inversion	Equation discovery
Architecture			
Adding physical variables	Daw et al.		
Encoding symmetries	Ling et al., Wang et al., Anderson et al., Schütt et al.		Udrescu et al.
Physics-inspired NAS	Ba et al., Panju and Ghodsi		
ML inspired by Koopman theory	Geneva and Zabaras, Lusch et al.		
Physically constrained GPs		Raissi et al., Raissi and Karniadakis	
Other approaches	Jumper et al., Mohan et al.		
Loss function			
Constraining physical quantities	Karpatne et al., Zhang et al., Benjamin Erichson et al., Xie et al., Brehmer et al.		
Encoding conservation laws	Beucler et al., Zeng et al.		Greydanus et al., Toth et al., Cranmer et al.
Auxiliary tasks	de Oliveira et al.		
Encoding governing equations	Raissi et al., Jin et al., Kharazmi et al., Yang et al., Wang et al., Wang et al., Li et al., Zhu et al., Geneva and Zabaras, Gao et al.	Jin et al., Chen et al., Yang et al.	Chen et al., Champion et al.
Hybrid approaches			
Residual modelling	Pawar et al.	Jiang et al.	
Differentiable physics		Ren et al., Minkov et al., Würfl et al., Zhang et al.	
Neural differential equations			Chen et al., Rackauckas et al., Long et al.
In-the-loop methods	Um et al., Rasp et al.	Adler and Öktem, Morningstar et al., Hammernik et al., Li et al., Lunz et al., Bora et al., Mosser et al.	

Hamiltonian neural networks
Solver-in-the-loop
Learned sub-grid processes
Physics-constrained Gaussian processes
Hidden physics models
Physics-informed neural networks
AI Feynman
DeepONets
PDE-Net
Algorithm unrolling
AlphaFold
Learned regularisation
Differentiable simulation
Physics-informed neural operators
Fourier neural operators
Encoding conservation laws
Encoding physical symmetries
Neural ODEs

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

A plethora of SciML techniques

Naive ML

Constraining physical quantities
Encoding conservation laws
Auxiliary tasks
Encoding governing equations

Loss function

Residual modelling
Differentiable physics
Neural differential equations
In-the-loop methods

Hybrid approaches

Adding physical variables
Encoding symmetries
Physics-inspired NAS
ML inspired by Koopman theory
Physically constrained GPs

Architecture

Traditional workflows

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

A plethora of SciML techniques

Constraining physical quantities
Encoding conservation laws
Auxiliary tasks
Encoding governing equations
Loss function

Residual modelling
Differentiable physics
Neural differential equations
In-the-loop methods
Hybrid approaches

Adding physical variables
Encoding symmetries
Physics-inspired NAS
ML inspired by Koopman theory
Physically constrained GPs
Architecture

Naive ML

Fully learned

Fully data-driven

No physics constraints

Traditional workflows

No learning

No data required

Hard physics constraints

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

A plethora of SciML techniques

Constraining physical quantities
Encoding conservation laws
Auxiliary tasks
Encoding governing equations
Loss function

Residual modelling
Differentiable physics
Neural differential equations
In-the-loop methods
Hybrid approaches

Adding physical variables
Encoding symmetries
Physics-inspired NAS
ML inspired by Koopman theory
Physically constrained GPs
Architecture

Naive ML

Fully learned

Fully data-driven

No physics constraints

Traditional workflows

No learning

No data required

Hard physics constraints

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

Introduction to DL

VICTORITA DOLEAN

Deep learning vs AI

Artificial intelligence
= Mimic human behaviour

Reasoning
Knowledge representation
Logic
Value alignment
Theorem proving
Planning
Turing test
Search
Memory
…
…

Machine learning
= Learn about the world

Support vector machines
Logistic regression
Symbolic learning
K-means
Gaussian processes
Decision trees
MCMC
Principle component analysis
Bayesian modelling
…
…

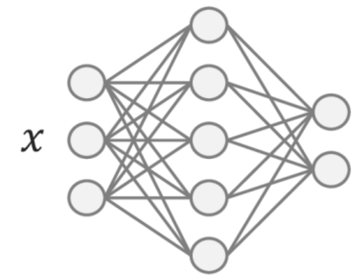
Deep learning
= Extract patterns using neural networks

Transformers
RNNs
ResNets
MLPs
GANs
CNNs
VAEs
Diffusion models
…

For a wide introduction to AI, see for example:
Russell & Norvig, Artificial Intelligence: A Modern Approach

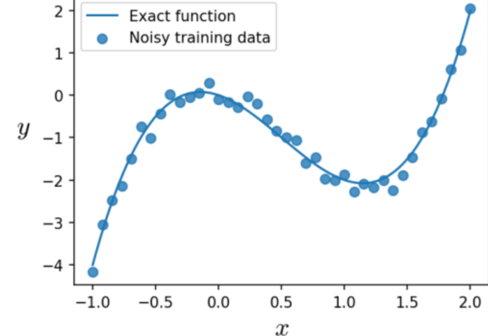
What is a neural network?

Neural networks are simply **flexible functions** fit to data



x $\hat{y} = NN(x; \theta)$

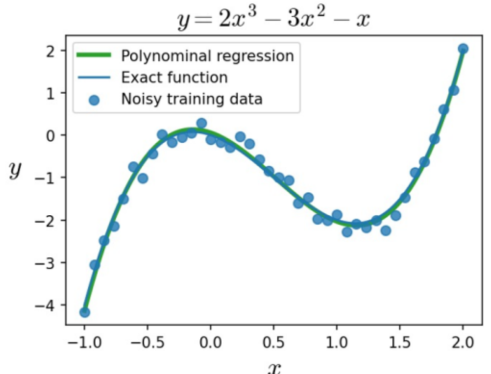
Example dataset:



Goal: given training data, find a function (with flexible parameters θ) which approximates the true function,

$$\hat{y} = NN(x; \theta) \approx y(x)$$

Function fitting



Simple polynomial regression

$$\hat{y}(x;\theta) = \theta_4 x^3 + \theta_3 x^2 + \theta_2 x + \theta_1$$

To fit, use least-squares:

$$\theta^* = \min_\theta \sum_i^N (\hat{y}(x_i;\theta) - y_i)^2 \qquad (1)$$

Re-write using linear algebra:

$$\begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \dots \end{pmatrix} = \begin{pmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{pmatrix} \text{ or } \hat{Y} = \Phi^T \theta$$

$$\theta^* = \min_\theta \|\Phi^T\theta - Y\|^2$$

In this case, it can be shown (1) has an analytical solution:

$$\theta^* = (\Phi^T\Phi)^{-1}\Phi^T Y$$

Neural network regression

$$\hat{y}(x;\theta) = NN(x;\theta)$$

To fit, use least-squares:

$$\theta^* = \min_\theta \sum_i^N (NN(x_i;\theta) - y_i)^2 \qquad (2)$$

In general, no analytical solution to (2) exists, so we must use **optimisation**

For example, gradient descent:

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial \sum_i^N (NN(x_i;\theta) - y_i)^2}{\partial \theta_j}$$

or equally

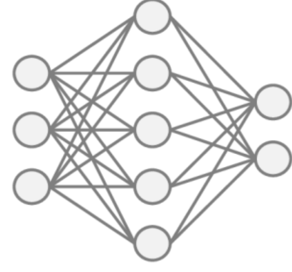
$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L(\theta)}{\partial \theta_j}$$

Where γ is the learning rate and $L(\theta)$ is the **loss function**

Neural network architecture

So, what exactly is $\hat{y} = NN(x; \theta)$?

This depends on the network **architecture** you choose (CNN, ResNet, Transformer, … etc)



2-layer MLP

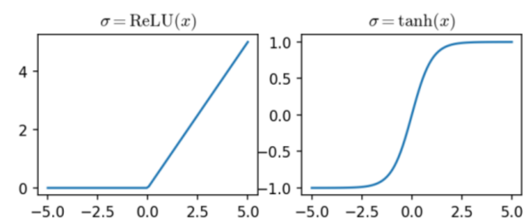
The most basic architecture is the **multilayer perceptron** (MLP) (aka **fully connected network**)

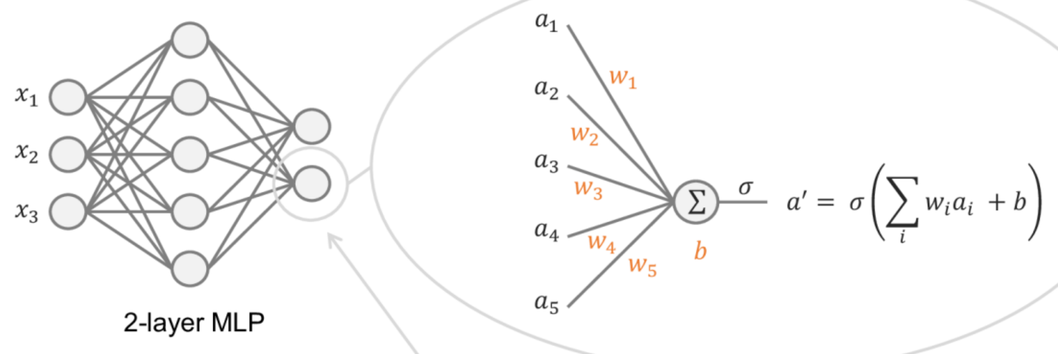
For example, a 2-layer MLP is defined as:

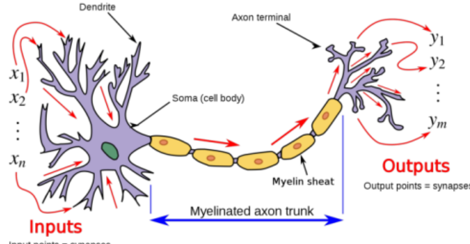
$$NN(x; \theta) = W_2 \sigma(W_1 x + b_1) + b_2$$

Where x is an input vector, W_1 and W_2 are learnable weight matrices, b_1 and b_2 are learnable bias vectors, and σ is an activation function, for example, $\sigma = \tanh(\cdot)$

Biological inspiration







Biological neuron
(Source: Wikipedia)

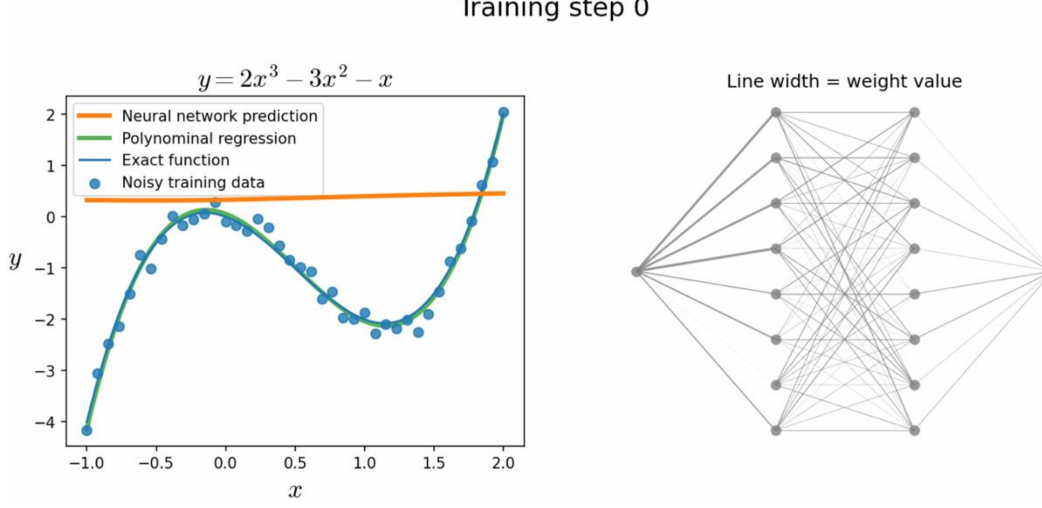
For last layer:

$$\begin{pmatrix} a'_1 \\ a'_2 \end{pmatrix} = \sigma\left(\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \end{pmatrix}\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)$$

Entire network:

$$NN(\boldsymbol{x};\theta) = \sigma(W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}) = \boldsymbol{f} \circ \boldsymbol{g}\,(\boldsymbol{x};\theta)$$

Polynomial regression example



$$NN(x;\theta) = W_3(\sigma(W_2\sigma(W_1 x + b_1) + b_2) + b_3$$

Trained using gradient descent

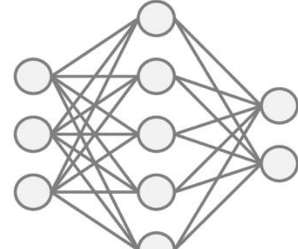
$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial \sum_i^N (NN(x_i;\theta) - y_i)^2}{\partial \theta_j}$$

Universal approximation

So why not just use linear regression?

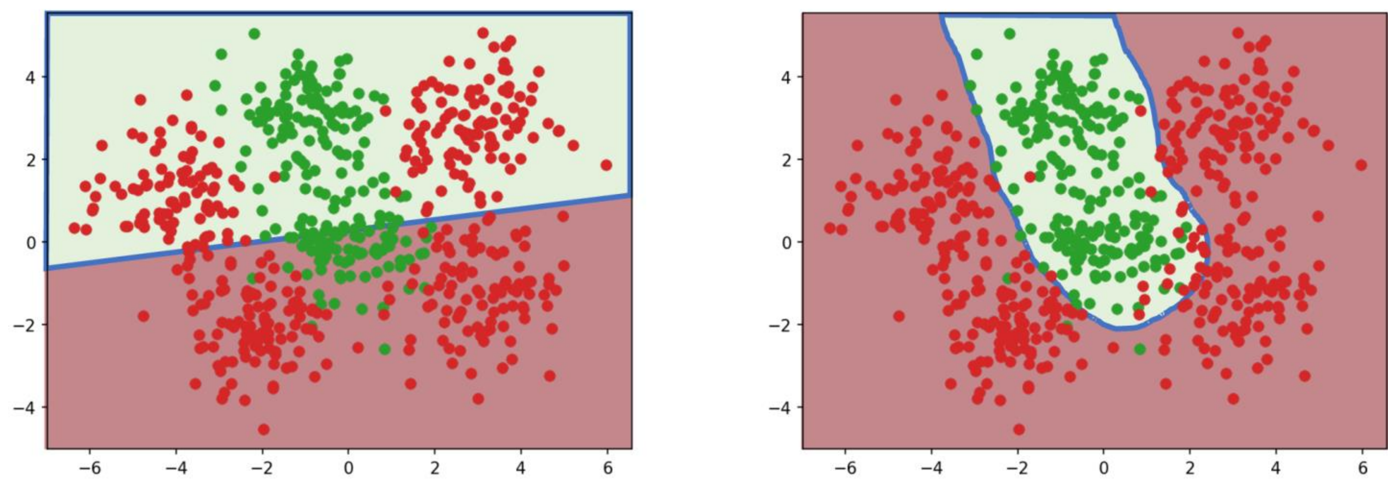
Neural networks are simply **flexible functions** fit to data

With enough parameters, neural networks can approximate any* arbitrarily complex function
= **universal approximation**



$\hat{y} = P(\text{dog} \mid x) = 1$

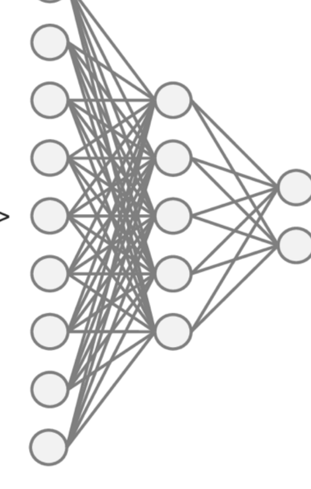
Importance of activation functions



Non-linearities allow us to approximate arbitrary **non-linear** functions

MLPs use lots of parameters

=> Flatten to 1D =>



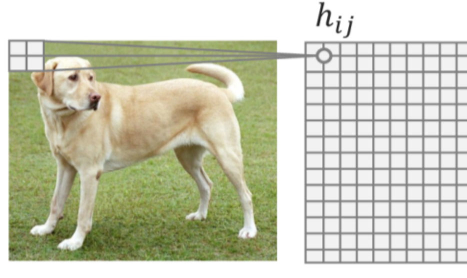
$$NN(x;\theta) = W_3(\sigma(W_2\sigma(W_1 x + b_1) + b_2) + b_3$$

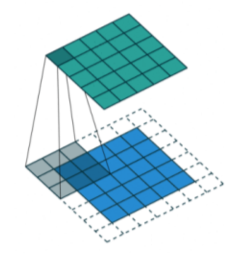
Assume the image has shape 128 x 128, and we have 100 hidden units in the first layer, then W_1 has shape (100 x (128 x 128)) = (100 x 16,384)

= 1.6M parameters!

=> A simple MLP image classifier can have millions of parameters

Convolutional neural network (CNN)





$$NN(x;\theta) = W_3 \star (\sigma(W_2 \star \sigma(\underbrace{W_1 \star x + b_1}_{h}) + b_2) + b_3$$

$$h_{ij} = \sum_{i'}^{l} \sum_{j'}^{m} W_{i'j'} x_{i+i',j+j'} + b$$

Convolutional neural networks honor the **spatial correlations** in their inputs

Each neuron;

- Has a **limited** field of view
- **Shares** the same weights as the other neurons in the layer
- Mathematically, CNNs use cross-correlation

CNNs have translation equivariance (an inductive bias)

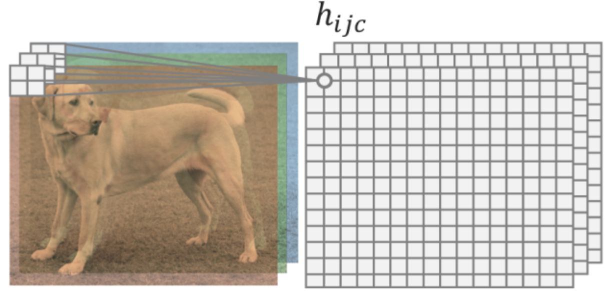
Let the size of the convolutional filter be 3 x 3

Then W_1 has shape (3 x 3)

= 9 parameters! (much, much smaller than a MLP)

Image source:
github/vdumoulin/conv_arithmetic

Convolutional neural network (CNN)



In practice, CNNs are usually extended so they can have multiple **channels** in the inputs and outputs of each layer

e.g. (R,G,B) image as input, where each channel is a color

Also:
- 1D and 3D CNNs follow analogously
- And we can add dilations and strides too

Then the convolutional layer is defined by:

$$h_{ijc} = \sum_{i'}^{l} \sum_{j'}^{m} \sum_{c'}^{C_{\text{in}}} W_{i'j'c'c} x_{i+i',j+j',c'} + b_c$$

Let the size of the convolutional filter be 3 x 3

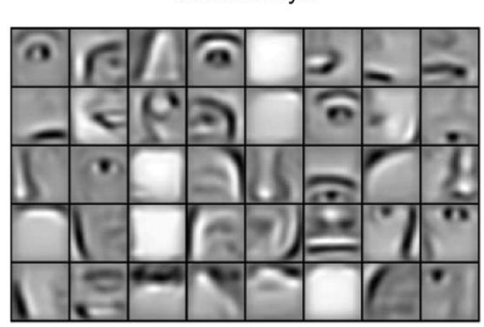
Then W has shape (3 x 3 x C_{in} x C_{out})

= 81 parameters for 3 input and 3 output channels

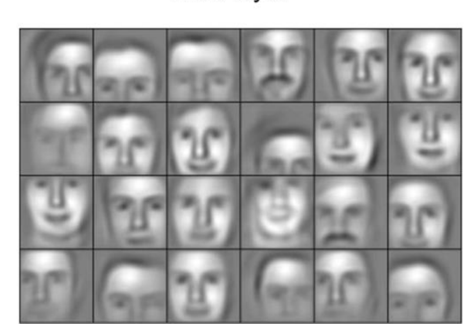
Deep CNNs

First layer

Second layer



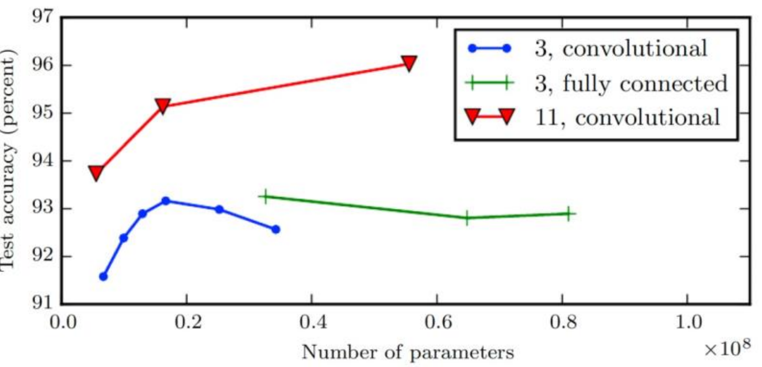
Third layer



Deep CNNs learn **hierarchical** features

Lee et al, Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks, Communications of the ACM (2011)

Depth is key



Goodfellow et al, Multi-digit number recognition from street view imagery using deep convolutional neural networks, ICLR (2014)

Empirically, deep neural networks perform better than shallow neural networks

=> encode a very general belief that the true function is **composed** of simpler functions

How do we train neural networks?

Gradient descent

$$\hat{y}(x;\theta) = NN(x;\theta)$$

To fit, use least-squares:

$$\theta^* = \min_\theta \sum_i^N (NN(x_i;\theta) - y_i)^2 \qquad (2)$$

In general, no analytical solution to (2) exists, so we must use **optimisation**

For example, gradient descent:

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial \sum_i^N (NN(x_i;\theta) - y_i)^2}{\partial \theta_j}$$

or equally

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L(\theta)}{\partial \theta_j}$$

Where γ is the learning rate and $L(\theta)$ is the **loss function**

Note that

$$\frac{\partial L(\theta)}{\partial \theta_j} = \sum_i^N 2(NN(x_i;\theta) - y_i)\frac{\partial NN(x_i;\theta)}{\partial \theta_j}$$

Let's consider a fully connected network

$$NN(\boldsymbol{x};\theta) = W_3(\sigma(\overbrace{W_2\sigma(\underbrace{W_1\boldsymbol{x} + \boldsymbol{b_1}}_{\boldsymbol{h}}) + \boldsymbol{b_2}}^{\boldsymbol{g}}) + \boldsymbol{b_3} = \boldsymbol{f} \circ \boldsymbol{g} \circ \boldsymbol{h}(\boldsymbol{x};\theta)$$

How do we calculate $\frac{\partial NN(\boldsymbol{x}_i;\theta)}{\partial W_1}$?

Note $\boldsymbol{f}, \boldsymbol{g}$, and $\boldsymbol{h}$ are vector functions =>

Use the **multivariate chain rule** (= matrix multiplication of **Jacobians**)

$$\frac{\partial NN}{\partial W_1} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}}\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{h}}\frac{\partial \boldsymbol{h}}{\partial W_1}$$

$$J = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{g}} = \begin{pmatrix} \frac{\partial f_1}{\partial g_1} & \cdots & \frac{\partial f_1}{\partial g_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial g_1} & \cdots & \frac{\partial f_m}{\partial g_n} \end{pmatrix}$$

Backpropagation

Forward pass:

$$\boldsymbol{x}_i \rightarrow \boldsymbol{h}_i = W_1\boldsymbol{x}_i + \boldsymbol{b}_1 \rightarrow \boldsymbol{g}_i = W_2\sigma(\boldsymbol{h}_i) + \boldsymbol{b}_2 \rightarrow f_i = W_3\sigma(\boldsymbol{g}_i) + \boldsymbol{b}_3$$

Backward pass:

In practice:

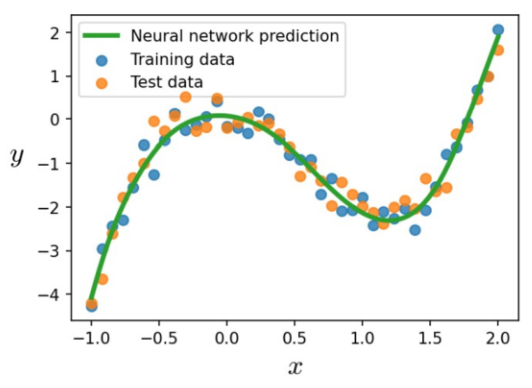
Autodifferentiation tracks all your forward computations and their gradients and applies the chain rule automatically for you, so you don't have to worry!

Train loss vs Train error vs Test error

Training loss:

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

$$D_{\text{train}} = \{(x_1, y_1), \dots, (x_N, y_N)\}$$
$$x, y \sim p(x, y)$$



Train error:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}|NN(x_i;\theta) - y_i|$$

$$D_{\text{train}} = \{(x_1, y_1), \dots, (x_N, y_N)\}$$
$$x, y \sim p(x, y)$$

Test error:

$$L(\theta) = \frac{1}{M}\sum_{i}^{M}|NN(x_i;\theta) - y_i|$$

$$D_{\text{test}} = \{(x_1, y_1), \dots, (x_M, y_M)\}$$
$$x, y \sim p(x, y)$$

Overfitting vs underfitting

1 layers, 0 hidden units

2 layers, 16 hidden units

3 layers, 32 hidden units

Training step 10000

Training step 10000

Training step 10000

Underfit

Ideal model

Overfit

Understanding error sources

Training loss:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$
$$D = \{(x_1, y_1), \dots, (x_N, y_N)\}$$

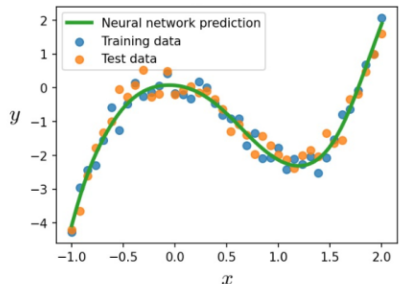
This is also known as the **empirical loss**, and can be more generally written as:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N} l(NN(x_i;\theta), y_i), \qquad x, y \sim p(x, y)$$

But what we really want to minimise is the **expected loss**:

$$\mathcal{L}(\theta) = \iint l(NN(x;\theta), y)\, p(x, y) dx dy$$
$$= E_{(x,y)\sim p}[l(x, y; \theta)]$$

Why is this impossible in practice?
Because, usually, we only have access to a finite number of samples from $p(x, y)$ (our training dataset, D)



This leads to

1) **Estimation error** (finite amount of training data) (aka generalisation error)

$$\mathcal{E}_{\text{est}} = \mathcal{L}(NN) - \mathcal{L}(NN^*)$$

NN which minimises empirical loss — NN which minimises expected loss

We also have

2) **Approximation error** (limited expressivity of neural network)

$$\mathcal{E}_{\text{app}} = \mathcal{L}(NN^*) - \mathcal{L}(y^*)$$

NN which minimises expected loss — True function which minimises expected loss

Function space

Function space, $f\colon \mathbb{R}^u \rightarrow \mathbb{R}^v$

True function, $y^*(x)$

Manifold true function lies on

Neural network, $NN(x;\theta)$, which minimises expected loss $\mathcal{L}(\theta)$

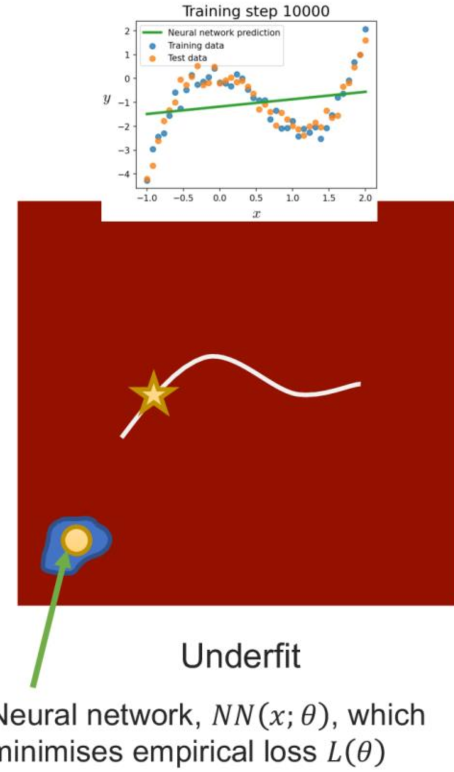
Approximation error, $\mathcal{E}_{\mathrm{app}}$

Neural network, $NN(x;\theta)$, which minimises empirical loss $L(\theta)$

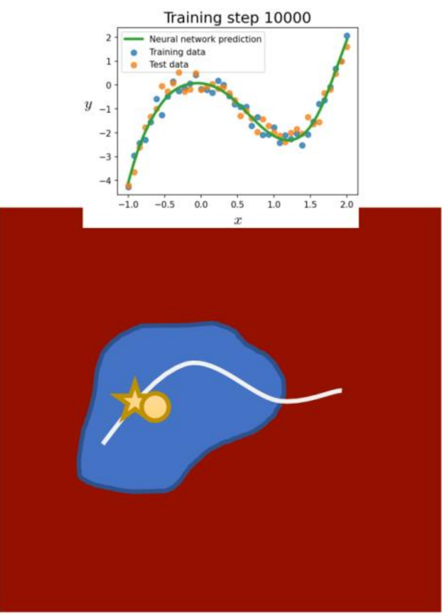
Estimation error, $\mathcal{E}_{\mathrm{est}}$

Space of all possible functions that neural network can represent

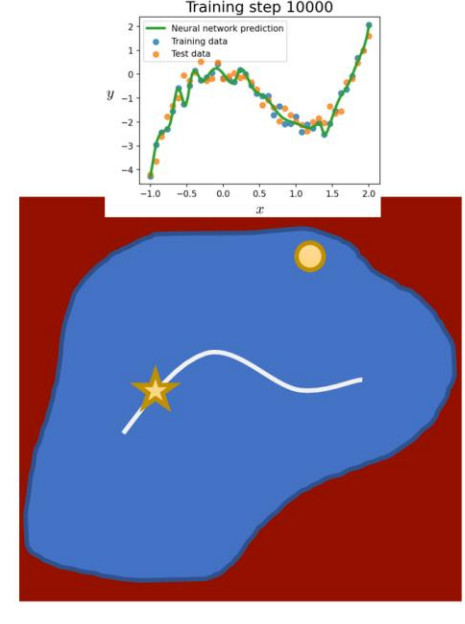
Function space



Underfit



Ideal model



Overfit

Neural network, $NN(x;\theta)$, which minimises empirical loss $L(\theta)$

Decomposing expected loss

Let $h(x, D) = NN(x; \theta)$ be the function learned by the neural network using a **particular** training dataset $D = \{(x_1, y_1), \dots, (x_N, y_N)\}$

Then we would like to understand what the expected loss is averaged over **all possible training datasets**, i.e.

$$E_{(x,y)\sim p,\, D\sim p^N}[l(x, y, D)]$$

Assume a simple least-squares loss function, e.g.

$$E_{x,y,D}[(h(x, D) - y)^2]$$

Then it can be shown*:

$$E_{x,y,D}[(h(x, D) - y)^2] = \underbrace{E_x\left[\left(\bar{h}(x) - \bar{y}(x)\right)^2\right]}_{\textbf{Bias}^2} + \underbrace{E_{x,D}\left[\left(h(x, D) - \bar{h}(x)\right)^2\right]}_{\textbf{Variance}} + \underbrace{E_{x,y}[(\bar{y}(x) - y)^2]}_{\textbf{Irreducible noise}}$$

Where

$\bar{y}(x) = E_{y|x}[y] = \int y\, p(y|x) dy$ = Average observation

$\bar{h}(x) = E_D[h(x, D)] = \int h(x, D)\, p(D) dD$ = Average model

*for full derivation, see e.g. https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote12.html

Decomposing expected loss

From previous slide:

$$E_{x,y,D}[(h(x,D) - y)^2] = \underbrace{E_x\left[\left(\bar{h}(x) - \bar{y}(x)\right)^2\right]}_{\textbf{Bias}^2} + \underbrace{E_{x,D}\left[\left(h(x,D) - \bar{h}(x)\right)^2\right]}_{\textbf{Variance}} + \underbrace{E_{x,y}[(\bar{y}(x) - y)^2]}_{\textbf{Irreducible noise}}$$

Where

$\bar{y}(x) = E_{y|x}[y] = \int y\, p(y|x)dy$ = Average observation

$\bar{h}(x) = E_D[h(x,D)] = \int h(x,D)\, p(D)dD$ = Average model

Bias = "average" model error

Variance = sensitivity of model prediction to training data

Irreducible noise = inherent noise in the training observations; you can never beat this ($\bar{y}(x)$ is the best possible predictor of y)

Decomposing expected loss

From previous slide:

$$E_{x,y,D}[(h(x,D)-y)^2] = \underbrace{E_x\left[\left(\bar{h}(x)-\bar{y}(x)\right)^2\right]}_{\textbf{Bias}^2} + \underbrace{E_{x,D}\left[\left(h(x,D)-\bar{h}(x)\right)^2\right]}_{\textbf{Variance}} + \underbrace{E_{x,y}[(\bar{y}(x)-y)^2]}_{\textbf{Irreducible noise}}$$

Where

$\bar{y}(x) = E_{y|x}[y] = \int y\, p(y|x)dy$ = Average observation

$\bar{h}(x) = E_D[h(x,D)] = \int h(x,D)\, p(D)dD$ = Average model

Bias = “average” model error

Variance = sensitivity of model prediction to training data

Irreducible noise = inherent noise in the training observations; you can never beat this ($\bar{y}(x)$ is the best possible predictor of y)

We want $E_{x,y,D}[(h(x,D)-y)^2]$ to be as low as possible.
But, typically, there is a bias-variance **tradeoff**!

Bias-variance tradeoff

Low Variance
High Variance
Low Bias
High Bias

Total Error
Optimum Model Complexity
Variance
$Bias^2$
Error
Model Complexity

Source: http://scott.fortmann-roe.com/docs/BiasVariance.html

Bias = “average” model error

Variance = sensitivity of model prediction to training data

Overfitting vs underfitting

1 layers, 0 hidden units	2 layers, 16 hidden units	3 layers, 32 hidden units
High bias Low variance	Low bias Low variance	Low bias High variance

≈ Variance

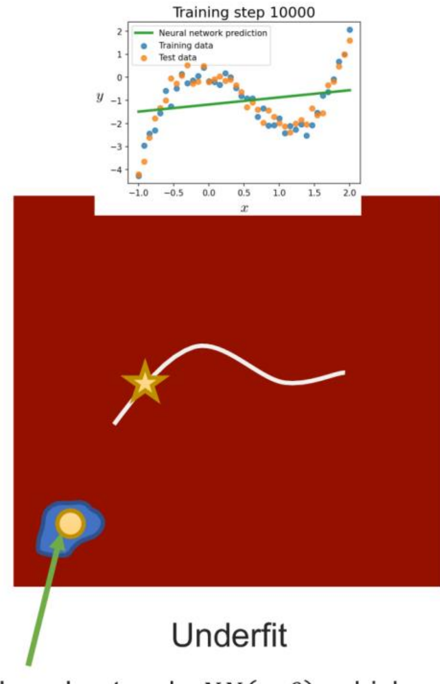
≈ Bias

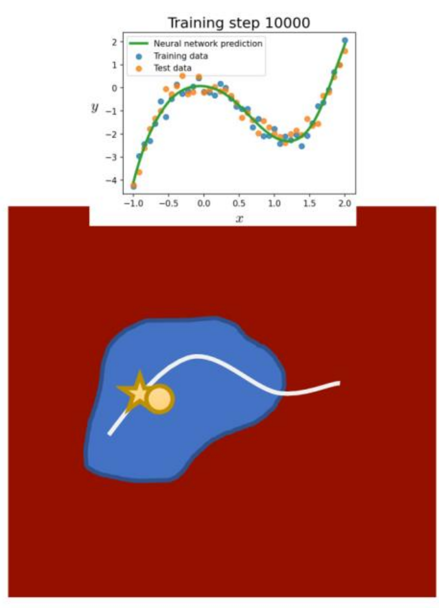
Improving model performance

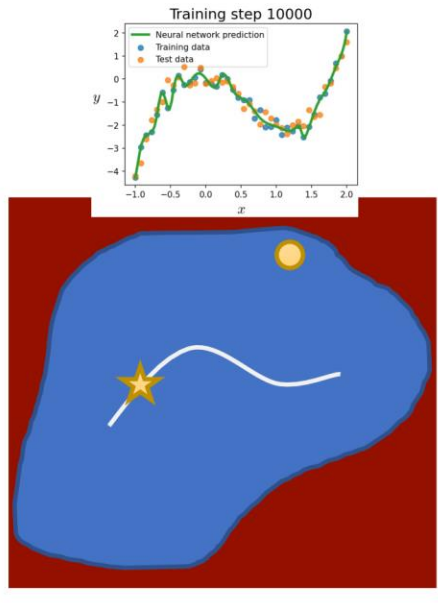
If you have high bias (or underfitting)
- Increase model **complexity** (e.g. number of free parameters)
- Or modify model architecture (shift location of hypothesis space)

If you have high variance (or overfitting)
- Increase the amount of training **data**
- Or **regularise** the neural network in some other way

Function space



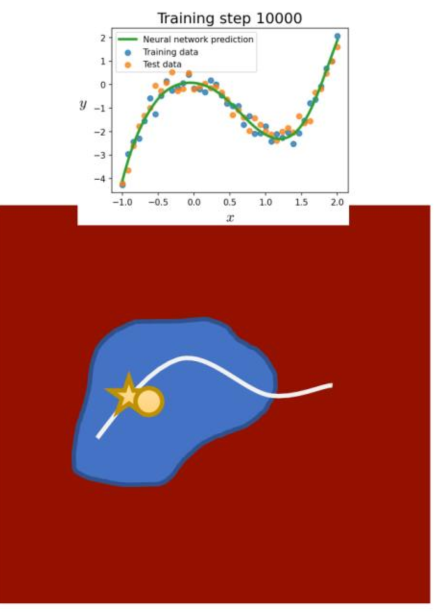




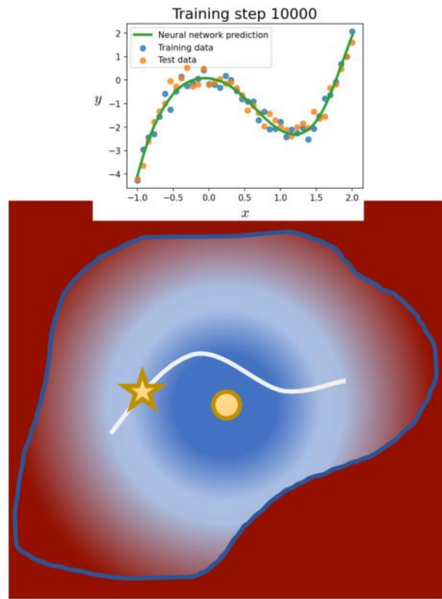
Neural network, $NN(x;\theta)$, which minimises empirical loss $L(\theta)$

Regularisation

- As well as changing the boundaries of the function space (hard constraint)
- We can **prefer** certain regions of the function space over others (soft constraint)



Ideal model



Regularised

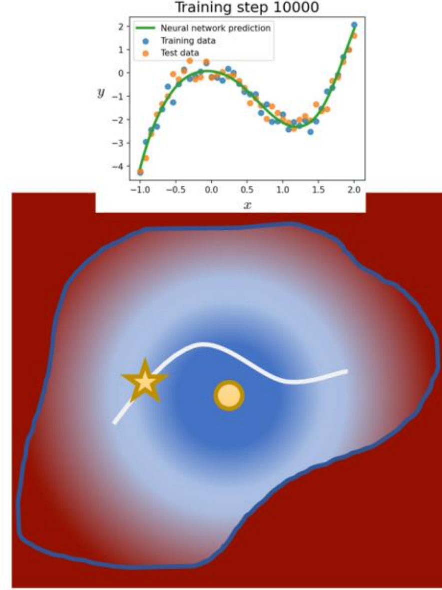
Regularisation

Regularisation = **restrict** the space of possible functions a neural network can learn / **prefer** certain regions of the function space / impart a **prior** on the learning algorithm

Ways to regularise neural networks:

- Reduce model complexity
- Modify model architecture

- Increase amount of (or augment) training data
- Weight regularization / pruning
- Additional loss terms
- Dropout
- Early stopping



Regularised

Additional loss terms

Weight regularisation:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2 + \lambda\|\theta\|^2$$

Alternatively, we can use any **additional** arbitrary loss term as a regulariser:

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2 + R(D,\theta)$$

Examples include:

Structural similarity index measure (SSIM) loss (perceptual image quality measure)

$$\mathrm{SSIM}(y,\hat{y}) = \frac{(2\mu_y\mu_{\hat{y}} + c_1)(2\sigma_{y\hat{y}} + c_2)}{(\mu_y^2 + \mu_{\hat{y}}^2 + c_1)(\sigma_y^2 + \sigma_{\hat{y}}^2 + c_2)}$$

Smoothness loss (e.g. in frequency domain)

$$L(\hat{y}) = w(\omega)\mathrm{FFT}(\hat{y})$$

Feature space loss (e.g. using a pretrained CNN)

$$F(y,\hat{y}) = \|\mathrm{CNN}(y)_{\mathrm{hidden}} - \mathrm{CNN}(\hat{y})_{\mathrm{hidden}}\|^2$$

Early stopping

3 layers, 32 hidden units

Mean L1 error

10^0

10^{-1}

Train error

Test error

Stop!

0 2000 4000 6000 8000 10000

Training step

Training step 100

Neural network prediction

Training data

Test data

y

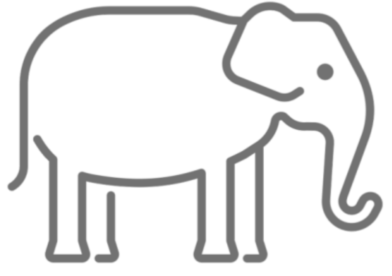
2 1 0 −1 −2 −3 −4

−1.0 −0.5 0.0 0.5 1.0 1.5 2.0

x

Related to the **spectral bias** of neural networks

Another source of error…



Up until now, we have assumed that the learned neural network parameters **globally** minimise the empirical loss, i.e.

$$\theta^* = \min_{\theta} L(\theta)$$

But, in practice, finding the global minima is **extremely** hard

So, we must also consider the effect of **optimisation error** on the performance of neural networks

=> Even though a large enough neural network can represent any arbitrary function (universal approximation theorem), there is **no guarantee** finding this network is tractable!

All error sources

Function space, $f: \mathbb{R}^u \rightarrow \mathbb{R}^v$

True function, $y^*(x)$

Manifold true function lies on

Neural network, $NN(x;\theta)$, which minimises expected loss $\mathcal{L}(\theta)$

Approximation error, $\mathcal{E}_{\text{app}}$ => Choose larger hypothesis space

Neural network, $NN(x;\theta)$, which minimises empirical loss $L(\theta)$

Estimation error, $\mathcal{E}_{\text{est}}$ => More training data / use regularisation

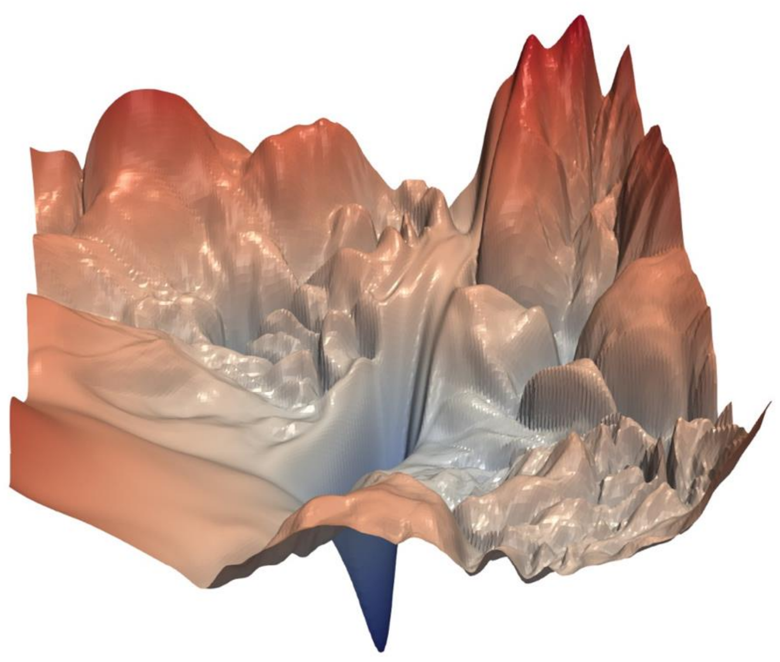
Optimisation error, $\mathcal{E}_{\text{opt}}$ => Improve optimiser

Trained neural network, $NN(x;\theta)$

Space of all possible functions that neural network can represent

$$\mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}}$$

Loss landscape



Li et al, Visualizing the Loss Landscape of Neural Nets, NeurIPS (2018)

2D visualization of the loss function

$$l(\alpha, \beta) = L(\theta^* + \alpha u + \beta v)$$

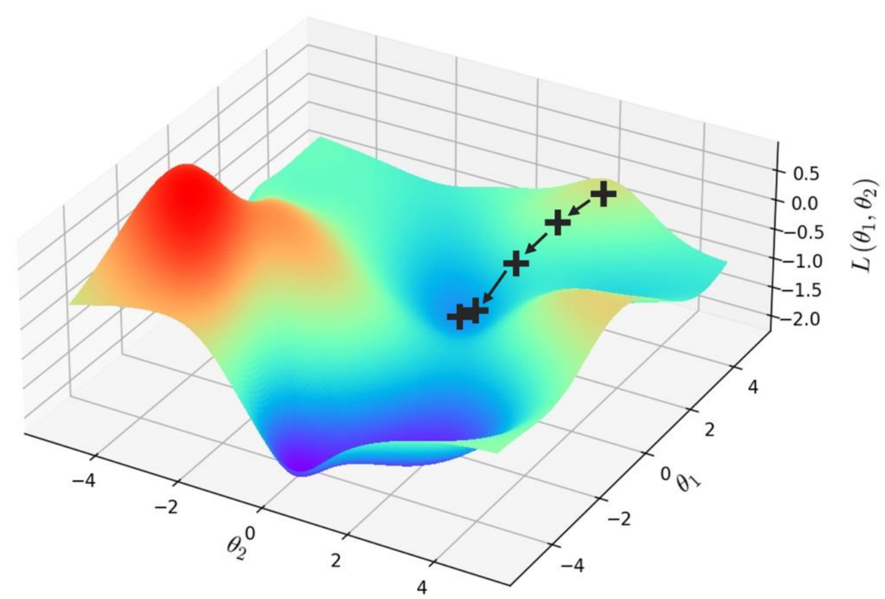
Where u and v are 2 randomly sampled directional vectors in the parameter space

Loss surface for ResNet-56 without skip connections trained on CIFAR-10 (natural images)

Loss function is very high dimensional and contains many **local minima**

Gradient descent

- Problem 2): Loss function has lots of **local** minima



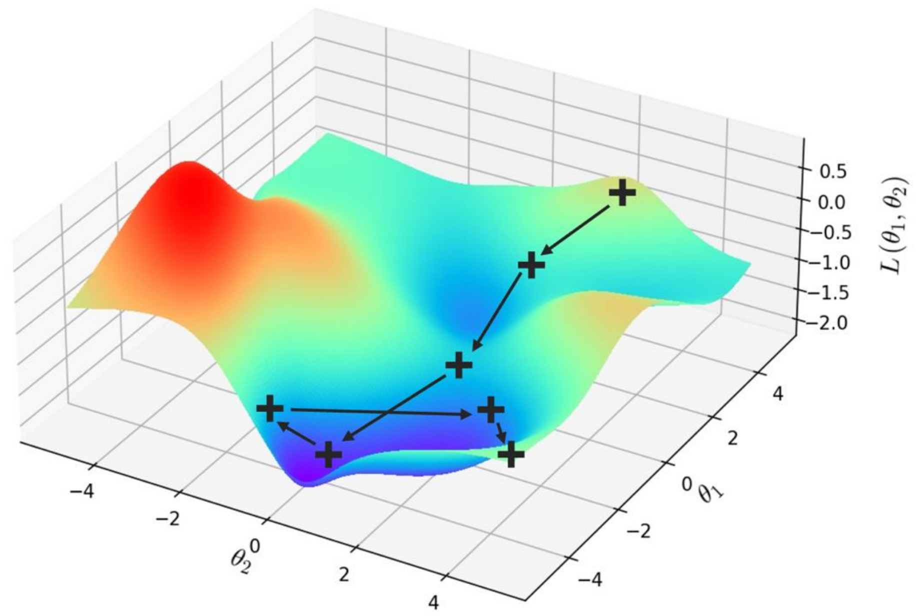
$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$

1. Initialise weights randomly
2. Loop:
    1. Compute gradient, $\frac{\partial L(\theta)}{\partial \theta_j}$
    2. Update weights,

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L(\theta)}{\partial \theta_j}$$

Gradient descent with large learning rate

- Can escape local minima (and global minima too!)
- Typically reduce learning rate with training step (e.g. exponentially decaying)

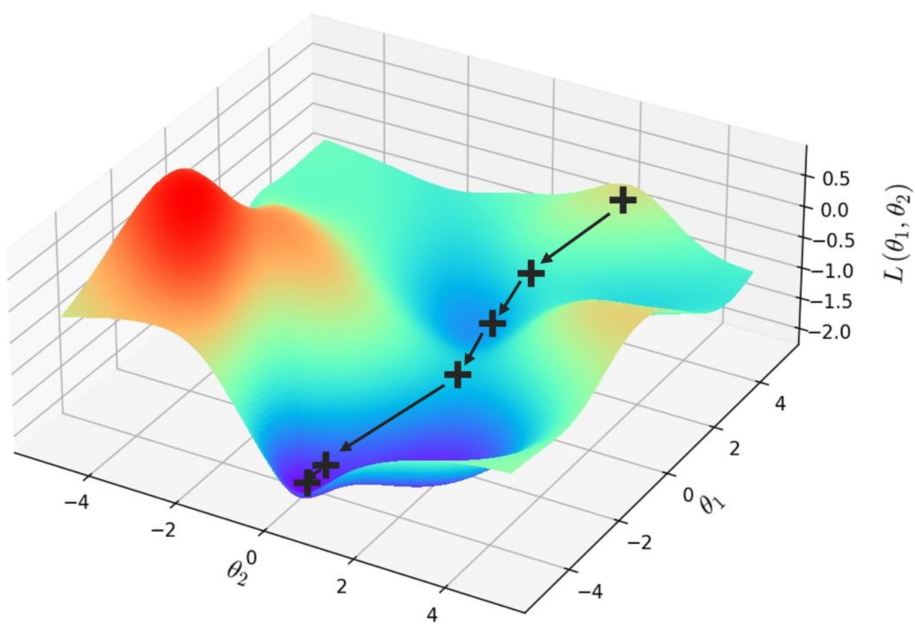


$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$

1. Initialise weights randomly
2. Loop:
   1. Compute gradient, $\frac{\partial L(\theta)}{\partial \theta_j}$
   2. Update weights,

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L(\theta)}{\partial \theta_j}$$

Gradient descent with adaptive learning rate

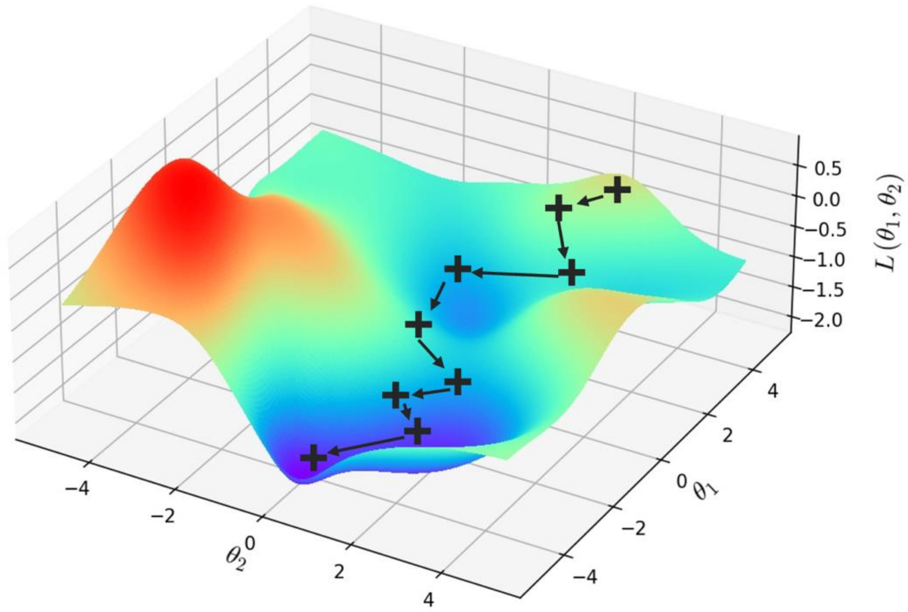


"Intelligent" learning rate

Can adapt learning rate based on

- How large gradient is
- Local curvature
- How fast learning is happening
- Values of weights
- .. many other ideas

(Mini-batch) stochastic gradient descent



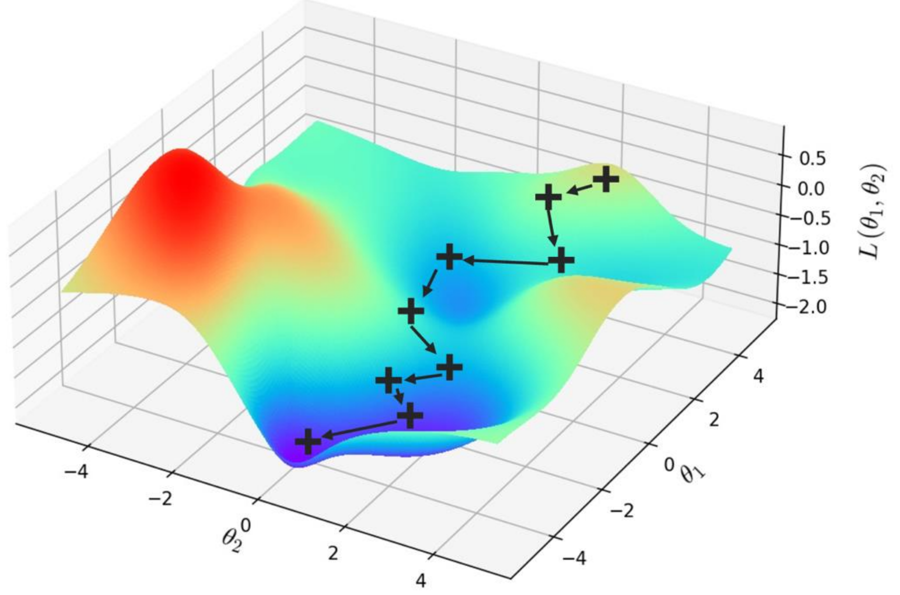
$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(x_i;\theta) - y_i)^2$$

1. Initialise weights randomly
2. Loop:
   1. **Randomly** sample mini-batch (subset) of B training points
   2. Compute **mini-batch** gradient, $\frac{\partial L_B(\theta)}{\partial \theta_j}$
   3. Update weights,

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L_B(\theta)}{\partial \theta_j}$$

(Mini-batch) stochastic gradient descent

- Produces noisy (approximate) gradient estimates ✓
- Can escape local minima ✓
- Much **more** efficient gradient computation (batch size of 100s vs millions) ✓
- Widely used in practice



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

1. Initialise weights randomly
2. Loop:
   1. **Randomly** sample mini-batch (subset) of B training points
   2. Compute **mini-batch** gradient, $\frac{\partial L_B(\theta)}{\partial \theta_j}$
   3. Update weights,

$$\theta_j \leftarrow \theta_j - \gamma \frac{\partial L_B(\theta)}{\partial \theta_j}$$

Adam optimiser

Adaptive moment estimation

Kingma et al, Adam: A Method for Stochastic Optimization, ICLR, (2015)

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

1. Initialise θ randomly, and $m_0 \leftarrow 0$ (first moment) $v_0 \leftarrow 0$ (second moment)
2. For $t = 1$ to … :
    1. Compute gradient, $\frac{\partial L}{\partial \theta_{t-1}}$
    2. Update moments, $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial \theta_{t-1}}$, $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial L}{\partial \theta_{t-1}}\right)^2$
    3. Apply bias correction, $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$, $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
    4. Update weights,

$$\theta_t \leftarrow \theta_{t-1} - \gamma\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Typically, $\beta_1 = 0.9, \;\; \beta_2 = 0.999$

Adam optimiser

Idea 1: keep some **momentum** when moving in parameter space

$$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L}{\partial \theta_{t-1}}$$

β_1 = "forgetting factor"
Essentially computes (exponentially-weighted) moving average of gradient

Idea 2: **adapt** the step size (per parameter) based on the **magnitude** of the gradient, again using some momentum

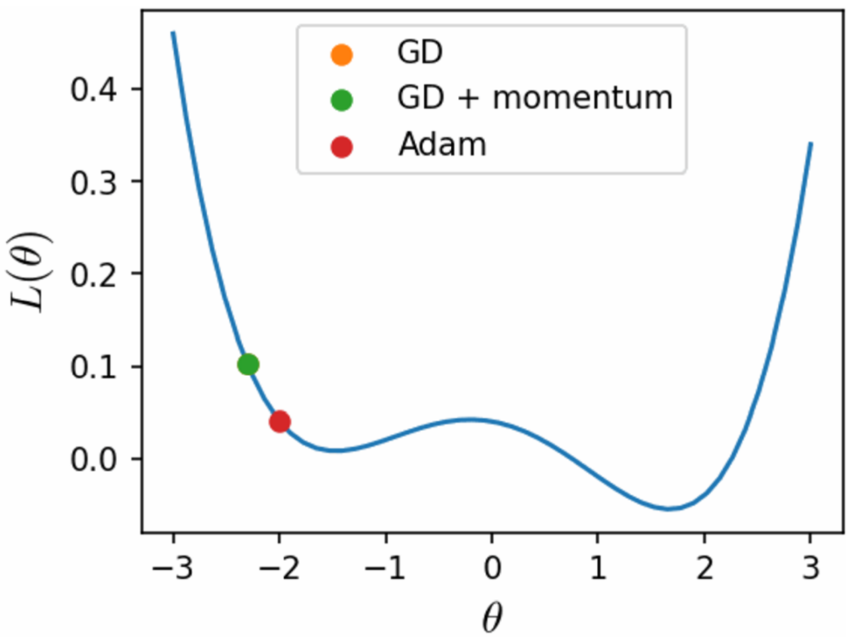
$$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) \left(\frac{\partial L}{\partial \theta_{t-1}} \right)^2$$

"Normalises" step size

$$\theta_t \leftarrow \theta_{t-1} - \gamma \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}$$

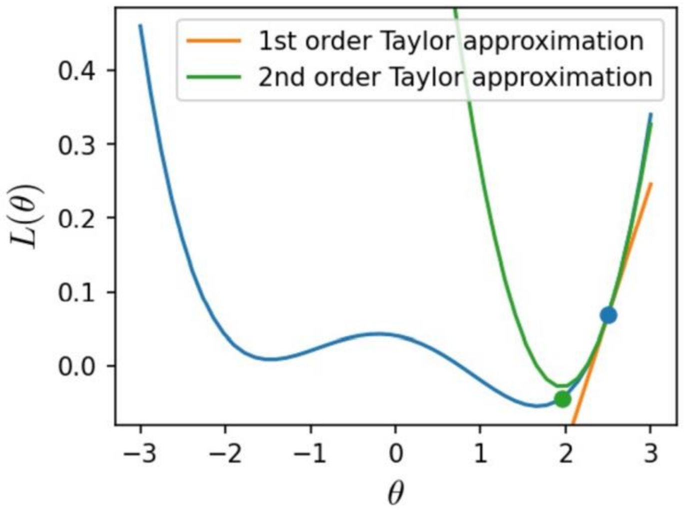
Note - the bias correction gives a more accurate estimate of the moving average (removes effect of initial value)

Adam optimiser Widely used in practice!



Higher-order optimisation

Make use of **higher-order** derivatives (e.g. curvature) of loss function to make bigger steps



Taylor approximation:

$$L(\theta) = L(a) + L'(a)(\theta - a) + \frac{1}{2}L''(a)(\theta - a)^2 + \ ...$$

Newton's method:
Instead of following gradient, step towards the local **minima** of the 2nd order approximation

$$\theta \leftarrow \theta - \gamma \frac{L'(\theta)}{L''(\theta)}$$

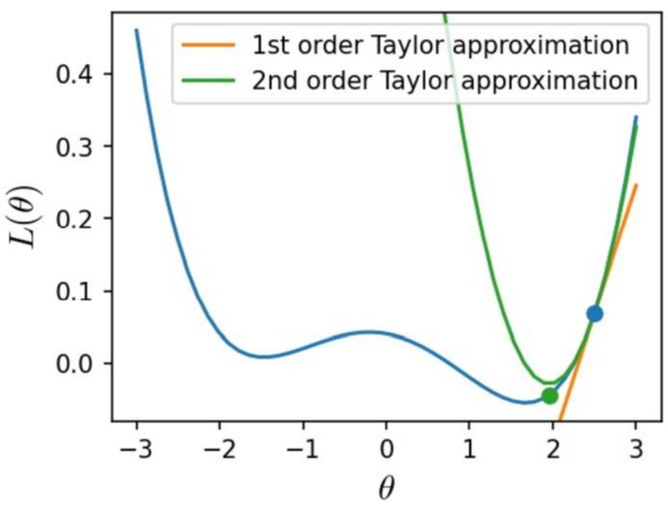
For a multivariate function:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \big(H(L(\boldsymbol{\theta}))\big)^{-1} \nabla L(\boldsymbol{\theta})$$

Where $H(L(\boldsymbol{\theta}))$ is the **Hessian** matrix

Higher-order optimisation

Make use of **higher-order** derivatives (e.g. curvature) of loss function to make bigger steps



$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma\left(H(L(\boldsymbol{\theta}))\right)^{-1} \nabla L(\boldsymbol{\theta})$$

Note

- It is usually prohibitively expensive to compute the Hessian $(p \times p)$ and its inverse
- So most modern methods (e.g. L-BFGS) use approximations
- These are called **quasi-Newton** methods

PINNs and their limitations

VICTORITA DOLEAN

What is a PINN?

Damped harmonic oscillator:

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

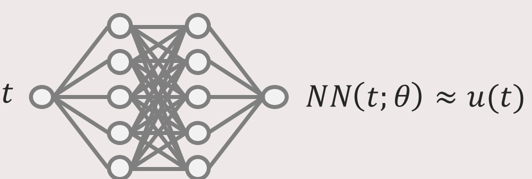
Initial conditions:

$$u(t=0)=1$$
$$u_t(t=0)=0$$

$u=$ displacement
$m=$ mass of oscillator
$\mu=$ coefficient of friction
$k=$ spring constant

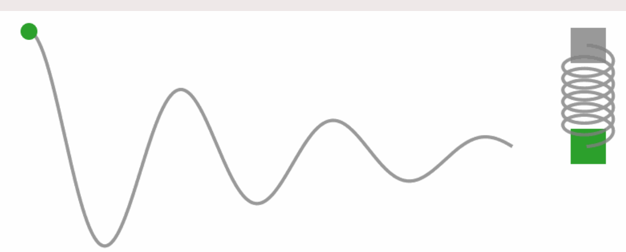
Key idea: use a neural network to directly approximate the solution

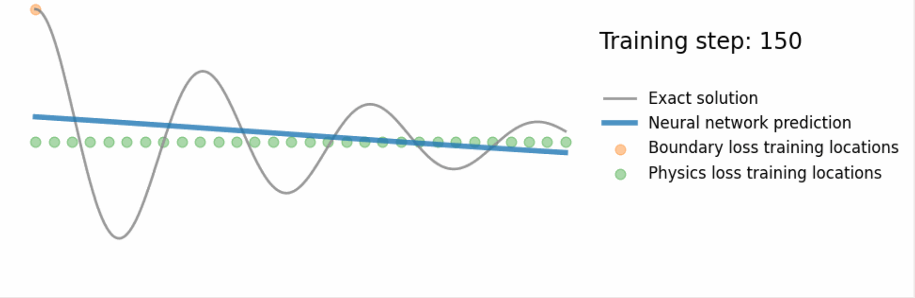
$$NN(t;\theta)\approx u(t)$$



Train the network using the loss function:

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





Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)
Lagaris et al, Artificial neural networks for solving ordinary and partial differential equations, IEEE (1998)

PINNs are a general framework for solving PDEs

Given a PDE and its boundary/initial conditions

$$\mathcal{D}[u(x)] = f(x), \qquad x \in \Omega \subset \mathbb{R}^d$$
$$\mathcal{B}_k[u(x)] = g_k(x), \qquad x \in \Gamma_k \subset \partial\Omega$$

Where $\mathcal{D}$ is some differential operator, $\mathcal{B}_k$ are a set of boundary operators, and $u(x)$ is the solution to the PDE

PINNs train a neural network to **approximate** the solution to the PDE $NN(x;\theta) \approx u(x)$ using the following loss function:

$$L(\theta) = L_b(\theta) + L_p(\theta)$$

$$L_b(\theta) = \sum_k \frac{\lambda_k}{N_{bk}} \sum_j^{N_{bk}} \left\| \mathcal{B}_k[NN(x_{kj};\theta)] - g_k(x_{kj}) \right\|^2$$

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \left\| \mathcal{D}[NN(x_i;\theta)] - f(x_i) \right\|^2$$

For example, the 1+1D viscous Burgers' equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0$$

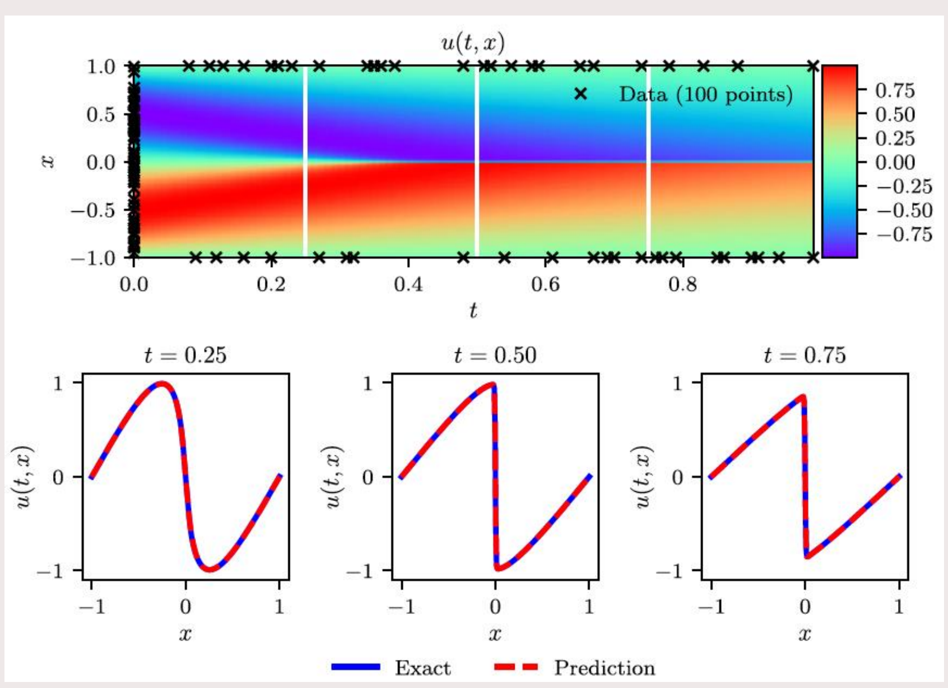
$$u(x,0) = -\sin(\pi x)$$
$$u(-1,t) = u(+1,t) = 0$$

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

$$L_b(\theta) = \frac{\lambda_1}{N_{b1}} \sum_j^{N_{b1}} \left(NN(x_j,0;\theta) + \sin(\pi x_j)\right)^2 + \frac{\lambda_2}{N_{b2}} \sum_k^{N_{b2}} (NN(-1,t_k;\theta) - 0)^2 + \frac{\lambda_3}{N_{b3}} \sum_l^{N_{b3}} (NN(+1,t_l;\theta) - 0)^2$$

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \left(\left(\frac{\partial NN}{\partial t} + NN\frac{\partial NN}{\partial x} - \nu\frac{\partial^2 NN}{\partial x^2} \right)(x_i,t_i;\theta) \right)^2$$

PINNs for solving viscous Burgers' equation

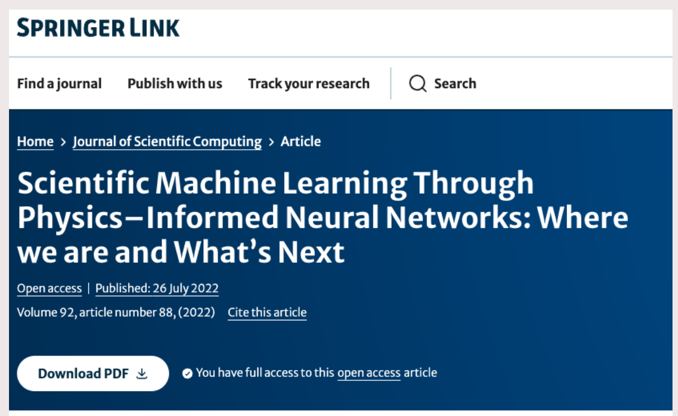


$$L_b(\theta) = \frac{\lambda_1}{N_{b1}} \sum_{j}^{N_{b1}} \left(NN(x_j, 0; \theta) + \sin(\pi x_j)\right)^2 + \frac{\lambda_2}{N_{b2}} \sum_{k}^{N_{b2}} (NN(-1, t_k; \theta) - 0)^2 + \frac{\lambda_3}{N_{b3}} \sum_{l}^{N_{b3}} (NN(+1, t_l; \theta) - 0)^2$$

$$L_p(\theta) = \frac{1}{N_p} \sum_{i}^{N_p} \left(\left(\frac{\partial NN}{\partial t} + NN \frac{\partial NN}{\partial x} - \nu \frac{\partial^2 NN}{\partial x^2} \right)(x_i, t_i; \theta) \right)^2$$

$\nu = 0.01/\pi$
$N_p = 10{,}000$ (Latin hypercube sampling)
$N_{b1} + N_{b2} + N_{b3} = 100$
Fully connected network with 9 layers, 20 hidden units (3021 free parameters)
Tanh activation function
L-BFGS optimiser

PINNs – an entire research field



SPRINGER LINK

Find a journal Publish with us Track your research Search

Home > Journal of Scientific Computing > Article

Scientific Machine Learning Through Physics–Informed Neural Networks: Where we are and What's Next

Open access | Published: 26 July 2022

Volume 92, article number 88, (2022) Cite this article

Download PDF

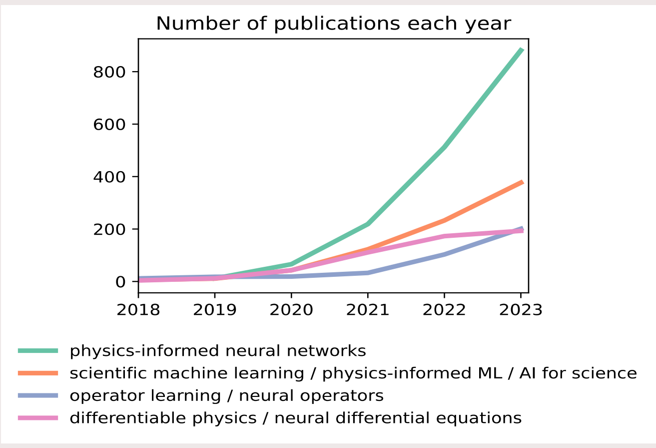
You have full access to this open access article

Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi & Francesco Piccialli

67k Accesses 274 Citations 7 Altmetric Explore all metrics

Abstract

Physics-Informed Neural Networks (PINN) are neural networks (NNs) that encode model equations, like Partial Differential Equations (PDE), as a component of the neural network itself. PINNs are nowadays used to solve PDEs, fractional equations, integral-differential equations, and stochastic PDEs. This novel methodology has arisen as a multi-task learning framework in which a NN must fit observed data while reducing a PDE residual. This article provides a comprehensive review of the literature on PINNs: while the primary



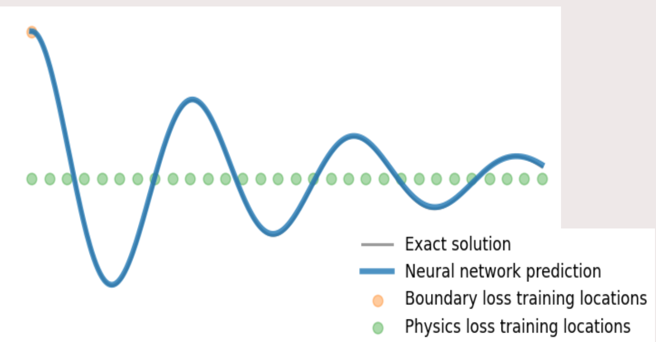
Source: Scopus keyword search (Feb 2024)

- The basic concepts behind PINNs were introduced in the 1990s (Lagaris et al, IEEE (1998) and others)
- Raissi et al, JCP (2019) is the seminal paper which reimplemented and extended PINNs using a modern deep learning framework

Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)

Lagaris et al, Artificial neural networks for solving ordinary and partial differential equations, IEEE (1998)

PINN training loop



Boundary loss $L_b(\theta)$

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

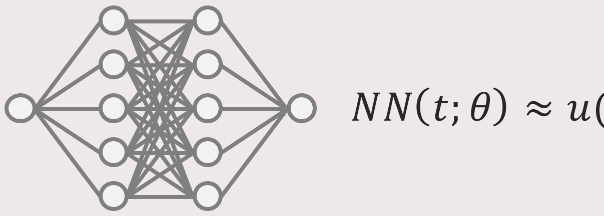
Physics loss $L_p(\theta)$

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

Training loop:

1. Sample boundary/ physics training points
2. Compute network outputs
3. Compute 1st and 2nd order gradient of network output **with respect to network input**
4. Compute loss
5. Compute gradient of loss function **with respect to network parameters**
6. Take gradient descent step

How can we compute the gradients (e.g. $\frac{dNN}{dt}$ and $\frac{dL}{d\theta}$) required in steps 3 and 5?



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

Gradient computation for PINNs

Assume network is an MLP, e.g.:

$$NN(t;\theta) = W_3(\sigma(W_2\sigma(W_1 t + \boldsymbol{b}_1) + \boldsymbol{b}_2) + b_3 \approx u(t)$$

Analogous to backpropagation, we can use the multivariate **chain rule** to compute gradients with respect to network **inputs**:

$$\begin{aligned} NN(t;\theta) &= f \circ \boldsymbol{g} \circ \boldsymbol{h}(t;\theta) \\ f &= W_3\sigma(\boldsymbol{g}) + b_3 \\ \boldsymbol{g} &= W_2\sigma(\boldsymbol{h}) + \boldsymbol{b}_2 \\ \boldsymbol{h} &= W_1 t + \boldsymbol{b}_1 \end{aligned}$$

$$\frac{\partial NN}{\partial t} = \frac{\partial f}{\partial \boldsymbol{g}} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial t}$$

$$\frac{\partial NN}{\partial t} = W_3 \operatorname{diag}(\sigma'(\boldsymbol{g}))W_2 \operatorname{diag}\big(\sigma'(\boldsymbol{h})\big)W_1$$

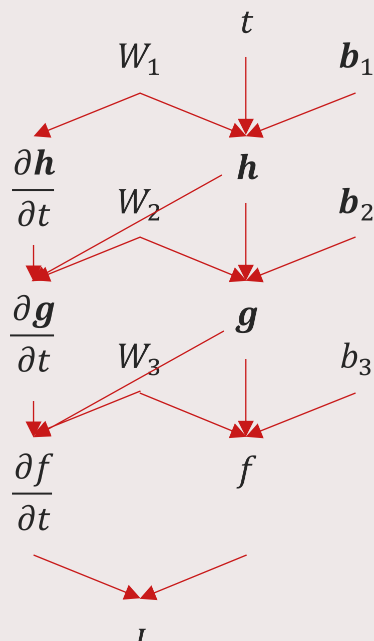
Extending computational graph

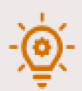
Computing gradients can simply be thought of as **extending** the network's computational graph:

$$NN(t;\theta) = W_3(\sigma(W_2\sigma(W_1 t + \boldsymbol{b}_1) + \boldsymbol{b}_2) + b_3$$

$$\begin{aligned} f &= W_3\sigma(\boldsymbol{g}) + b_3 \\ \boldsymbol{g} &= W_2\sigma(\boldsymbol{h}) + \boldsymbol{b}_2 \\ \boldsymbol{h} &= W_1 t + \boldsymbol{b}_1 \end{aligned}$$

$$\frac{\partial NN}{\partial t} = W_3 \operatorname{diag}(\sigma'(\boldsymbol{g}))W_2 \operatorname{diag}\big(\sigma'(\boldsymbol{h})\big)W_1$$





We can recursively apply **autodifferentiation** to compute gradients and extend the graph

PINN training loop

Training loop:

1. Sample boundary/ physics training points
2. Compute network outputs
3. Compute 1st and 2nd order gradient of network output **with respect to network input** **<= (recursively) apply autodiff, extending graph**
4. Compute loss
5. Compute gradient of loss function **with respect to network parameters** **<= apply autodiff on extended graph**
6. Take gradient descent step

We can recursively apply **autodifferentiation** to compute gradients and extend the graph

```
# PINN training psuedocode

#2.
t.requires_grad_(True)# tells PyTorch to start tracking graph
theta.requires_grad_(True)
u = NN(t, theta)

#3.
dudt = torch.autograd.grad(u, t, torch.ones_like(u),
                           create_graph=True)[0]
d2udt2 = torch.autograd.grad(dudt, t, torch.ones_like(u),
                             create_graph=True)[0]

#4.
physics_loss = torch.mean((m*d2udt2 + mu*dudt + k*u)**2)
loss = physics_loss + lambda_*boundary_loss

#5.
dtheta = torch.autograd.grad(loss, theta)[0]
```

Computational cost of higher order derivatives

Note, gradient computation roughly **doubles** the size of the computational graph*

⇒ The cost of evaluating $\frac{\partial^n NN}{\partial t^n}$ grows **exponentially** with n (!)

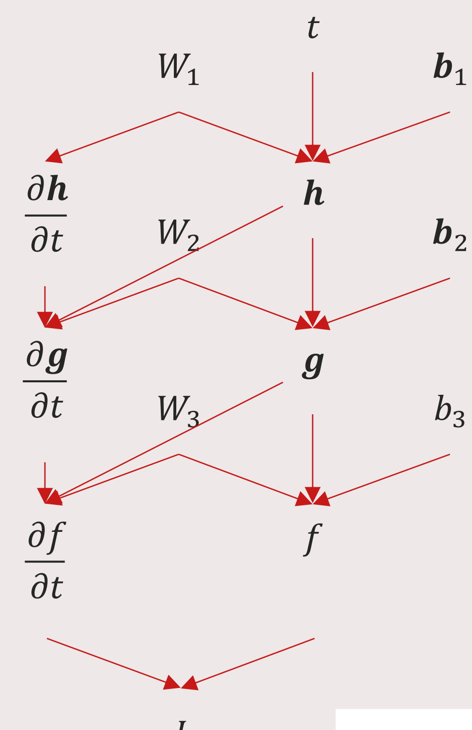
⇒ Most time is spent computing gradients, not the forward pass, when training PINNs

*More precisely, given some $f: \mathbb{R}^n \to \mathbb{R}^m$, its Jacobian $J \in \mathbb{R}^{m \times n}$ and some vector $v \in \mathbb{R}^n$

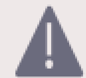
$$TIME\{f, Jv\} \leq \omega\, TIME\{f\}$$

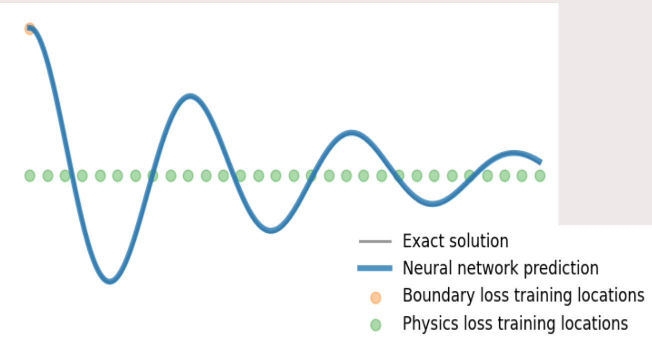
With a constant $\omega \in [2, 2.5]$ using autodifferentiation

For detailed derivation, see eg: Griewank and Walther, Evaluating Derivatives, Ch 3.1, SIAM (2008))



Other important considerations





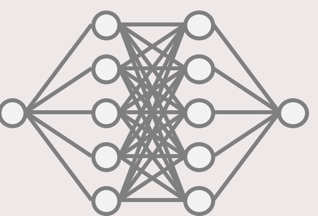
- λ values often significantly affect convergence
- "Enough" collocation points $\{t_i\}_{i=1}^{N_p}$ must be chosen so that the learned solution is accurate across the full domain
- Similarly, "enough" boundary points must be chosen such that the learned solution is unique
- Training points are usually sampled uniformly, randomly or quasi randomly
- PINNs still suffer from approximation, estimation and optimisation error (just like normal neural networks.

Boundary loss $L_b(\theta)$

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

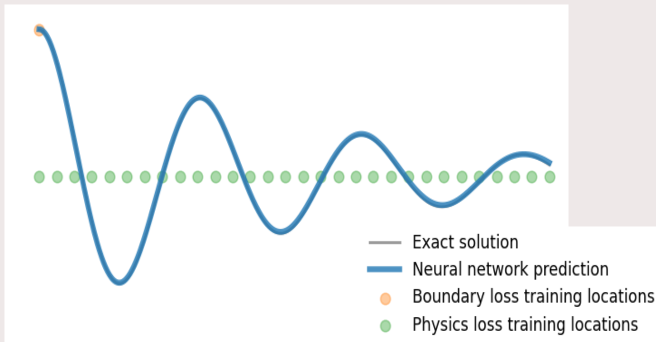
Physics loss $L_p(\theta)$

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



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

Other important considerations



From a mathematical perspective:

- PINNs provide a way to solve PDEs:
  - Neural network is a mesh-free, **functional** approximation of PDE solution
  - Physics loss is used to assert solution is **consistent** with PDE
  - Boundary loss is used to assert boundary/initial conditions, to ensure solution is **unique**

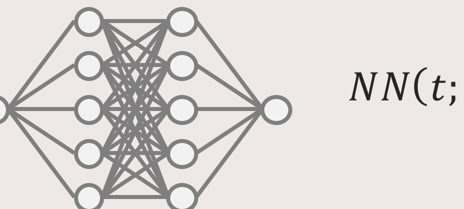
Boundary loss $L_b(\theta)$

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

Physics loss $L_p(\theta)$

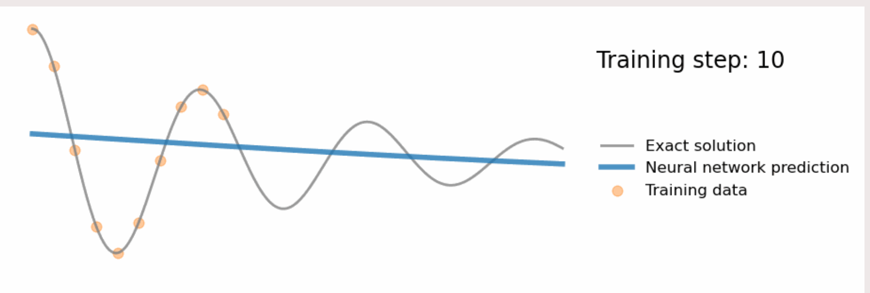
$$+ \frac{1}{N_p} \sum_i^{N_p} \left(\left[m \frac{d^2}{dt^2} + \mu \frac{d}{dt} + k \right] NN(t_i;\theta) \right)^2$$

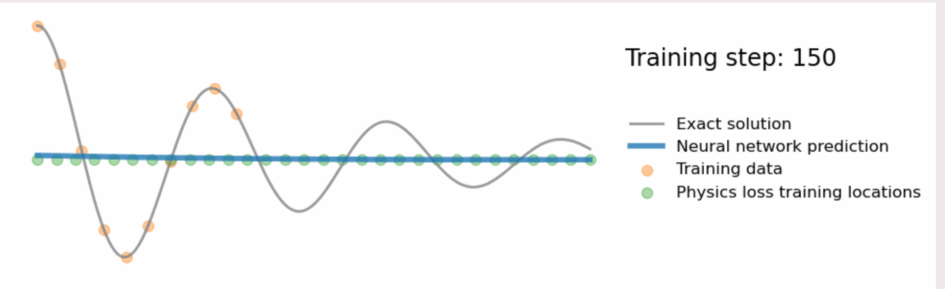
What can we say about PINNs from a ML perspective?
Physics loss is an **unsupervised** regulariser, which adds prior knowledge



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

PINNs from a ML perspective





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

$$L(\theta) = \frac{1}{N}\sum_{i}^{N}(NN(t_i;\theta) - u_i)^2 \quad \text{Supervised loss}$$

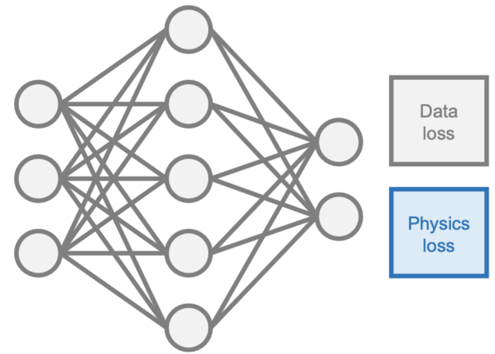
$$+ \frac{\lambda}{M}\sum_{j}^{M}\left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k\right]NN(t_j;\theta)\right)^2 \quad \text{Physics loss}$$

From a ML perspective:

- Physics loss is an **unsupervised** regulariser, which adds prior knowledge

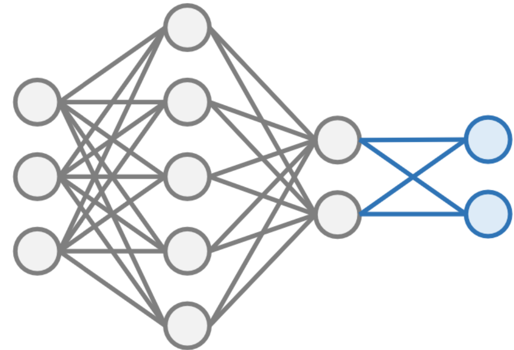
Ways to incorporate scientific principles into machine learning

Loss function



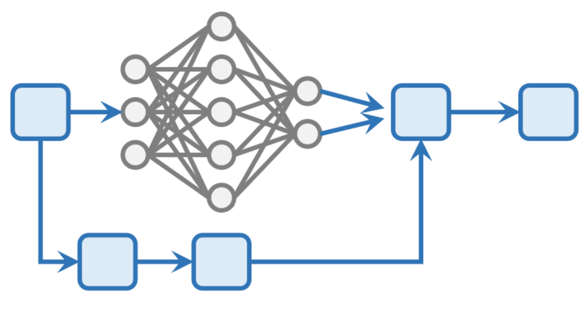
Example:
Physics-informed neural networks
(add governing equations to loss function)

Architecture



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

Hybrid approaches



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

Key scientific tasks

Velocity, $c(x, y)$
Source location, (s_x, s_y)

Depth, y
Distance, x
High
Velocity
Low

Wavefield, $u(x, y, t)$

Forward simulation

Estimate wavefield $u(x, y, t)$
Given velocity $c(x, y)$
and source location (s_x, s_y)

Wavefield, $u(x, y, t)$

Velocity, $c(x, y)$
Source location, (s_x, s_y)

Inversion

Estimate velocity $c(x, y)$
and source location (s_x, s_y)
Given wavefield $u(x, y, t)$

Wavefield, $u(x, y, t)$
Velocity, $c(x, y)$
Source location, (s_x, s_y)

Wave equation

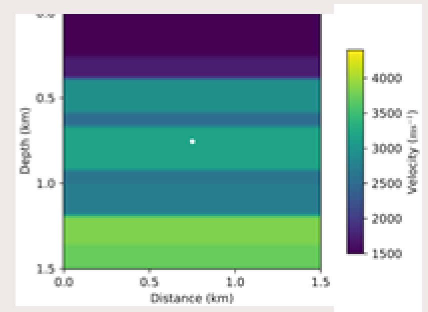
$$\nabla^2 u - \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = f$$

Equation discovery

Estimate governing equation
Given wavefield $u(x, y, t)$,
velocity $c(x, y)$,
and source location (s_x, s_y)

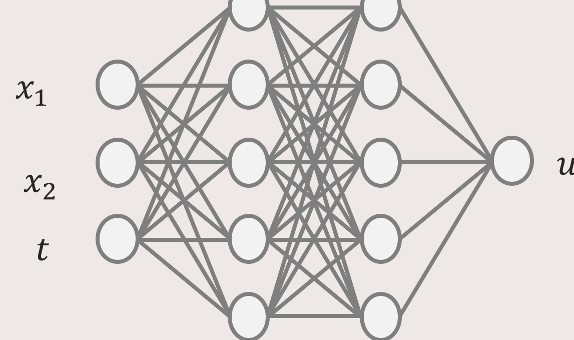
1. PINNs for solving wave equation – forward problem

Velocity model, $c(x)$



Ground truth FD simulation

t = 0.00 s | t = 0.05 s | t = 0.10 s | t = 0.15 s | t = 0.20 s



Moseley et al, Solving the wave equation with physics-informed deep learning, ArXiv (2020)

$$L_b(\theta) = \frac{\lambda}{N_b} \sum_{j}^{N_b} \left(NN(x_j, t_j; \theta) - u_{FD}(x_j, t_j) \right)^2$$

$$L_p(\theta) = \frac{1}{N_p} \sum_{i}^{N_p} \left(\left[\nabla^2 - \frac{1}{c(x_i)^2} \frac{\partial^2}{\partial t^2} \right] NN(x_i, t_i; \theta) \right)^2$$

Boundary data from FD simulation (first 0.02 seconds)

Collocation points randomly sampled over entire domain (up to 0.2 seconds)

PINNs for solving wave equation

t = 0.00 s | t = 0.05 s | t = 0.10 s | t = 0.15 s | t = 0.20 s

Ground truth (FD)

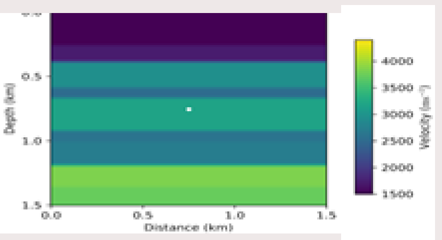
Prediction (NN)

Prediction (NN + physics loss)

Difference (NN)

Difference (NN + physics loss)

Velocity model, $c(x)$



Moseley et al, Solving the wave equation with physics-informed deep learning, ArXiv (2020)

Mini-batch size $N_b = N_p = 500$ (random sampling)

Fully connected network with 10 layers, 1024 hidden units

Softplus activation

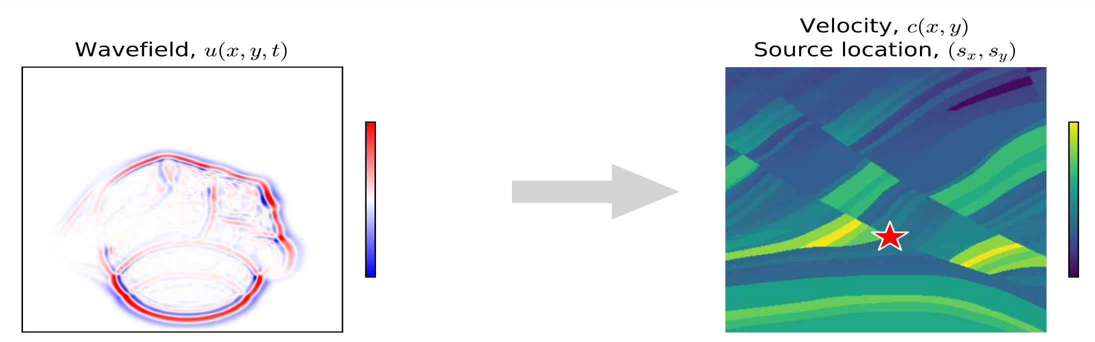
Adam optimiser

Training time: ~**1 hour**

What is an inverse problem?

Wave equation:

$$\nabla^2 u - \frac{1}{c(x)^2}\frac{\partial^2 u}{\partial t^2} = 0$$



b = observed wavefield $u(x,t)$

$$b = F(a)$$

a = (velocity model $c(x)$, source location)

- Fundamentally, inverse problems are **search** problems
- It is often useful to frame them as an optimisation problem, for example:

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

a = set of input conditions

F = physical model of the system

b = resulting properties given F and a

2. PINNs for inversion

PINNs for solving **forward** simulation:

$$L(\theta) = L_b(\theta) + L_p(\theta)$$

$$L_b(\theta) = \sum_k \frac{\lambda_k}{N_{bk}} \sum_j^{N_{bk}} \left\| \mathcal{B}_k[NN(x_{kj};\theta)] - g_k(x_{kj}) \right\|^2$$ Boundary loss

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \| \mathcal{D}[NN(x_i;\theta)] - f(x_i) \|^2$$ Physics loss

For example:

$$D = \left[\nabla^2 - \frac{1}{c(x)^2} \frac{\partial^2}{\partial t^2} \right]$$
$$f = 0$$

PINNs for solving **inverse** problems:

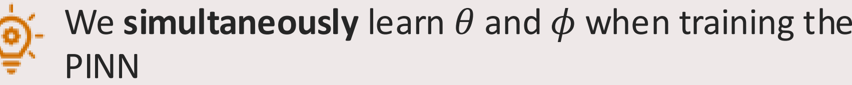
$$L(\theta, \phi) = L_p(\theta, \phi) + L_d(\theta)$$

Physics loss

$$L_p(\theta, \phi) = \frac{1}{N_p} \sum_i^{N_p} \| \mathcal{D}[NN(x_i;\theta);\phi] - f(x_i) \|^2$$

$$L_d(\theta) = \frac{\lambda}{N_d} \sum_l^{N_d} \| NN(x_l;\theta) - u_l \|^2$$ **Data loss**

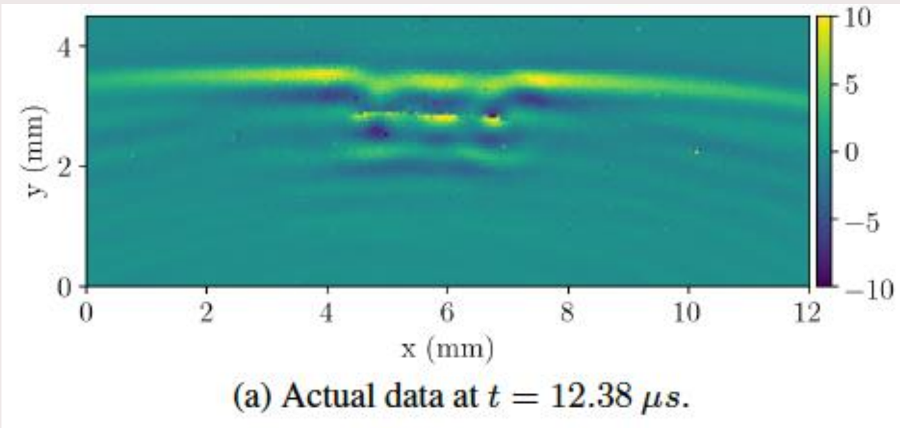
where ϕ are unknown, underlying PDE parameters we wish to invert for, and $\{x_l, u_l\}$ are a set of (potentially noisy) observational data



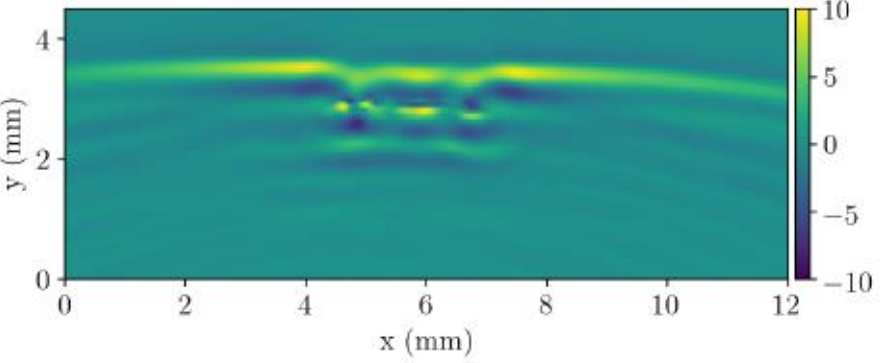
We **simultaneously** learn θ and ϕ when training the PINN

Wave equation

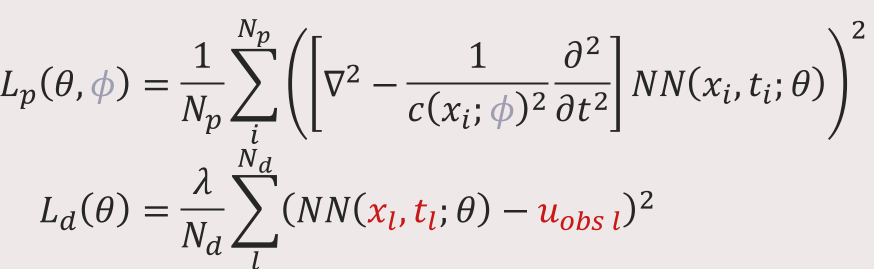
Shukla et al, Physics-Informed Neural Network for Ultrasound Nondestructive Quantification of Surface Breaking Cracks, Journal of Nondestructive Evaluation (2020)



(a) Actual data at $t = 12.38\ \mu s$.

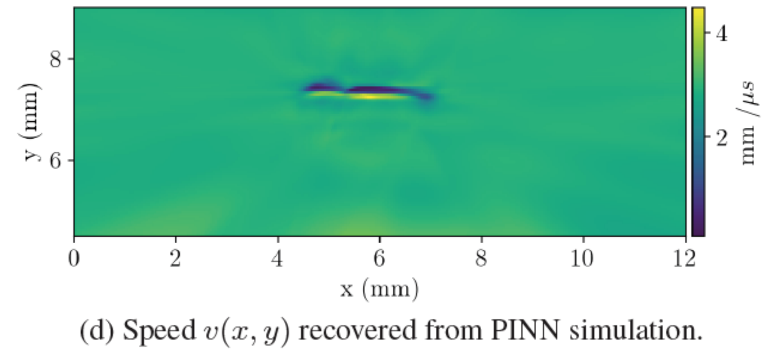


(b) Data recovered from PINN simulation at $t = 12.38\ \mu s$.



$$L_p(\theta, \phi) = \frac{1}{N_p} \sum_i^{N_p} \left(\left[\nabla^2 - \frac{1}{c(x_i; \phi)^2} \frac{\partial^2}{\partial t^2} \right] NN(x_i, t_i; \theta) \right)^2$$

$$L_d(\theta) = \frac{\lambda}{N_d} \sum_l^{N_d} (NN(x_l, t_l; \theta) - u_{obs\ l})^2$$



(d) Speed $v(x, y)$ recovered from PINN simulation.

Treat velocity model as **another** neural network, and simultaneously learn it

3. PINNs for equation discovery

PINNs for solving **inverse** problems:

$$L(\theta, \phi) = L_p(\theta, \phi) + L_d(\theta)$$

$$L_p(\theta, \phi) = \frac{1}{N_p}\sum_{i}^{N_p} \| \mathcal{D}[NN(x_i;\theta);\phi] - f(x_i) \|^2 \quad \text{Physics loss}$$

$$L_d(\theta) = \frac{\lambda}{N_d}\sum_{l}^{N_d} \| NN(x_l;\theta) - u_l \|^2 \quad \text{Data loss}$$

Where $\{x_l, u_l\}$ are a set of (potentially noisy) observational data

But how do we learn the **entire** differential operator $\mathcal{D}$, rather than its parameters ϕ?

How do we learn an **entire** differential operator $\mathcal{D}$?

Build a **library** of n operators, such as:

$$\phi = (1, \partial_x, \partial_t, \partial_{xx}, \partial_{tt}, \partial_{xt})^T$$

Then assume the differential operator can be represented as

$$\mathcal{D} = \Lambda \phi$$

Where Λ is a (sparse) matrix of shape (d_u, n)

E.g. for 1D damped harmonic oscillator:

$$\mathcal{D} = \begin{pmatrix} k & \mu & m & 0 \end{pmatrix} \begin{pmatrix} 1 \\ d_t \\ d_{tt} \\ d_{ttt} \end{pmatrix}$$

$$= m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k$$

3. PINNs for equation discovery

How do we learn an **entire** differential operator $\mathcal{D}$?

Build a **library** of n operators, such as:

$$\phi = (1, \partial_x, \partial_t, \partial_{xx}, \partial_{tt}, \partial_{xt})^T$$

Then assume the differential operator can be represented as

$$\mathcal{D} = \Lambda\phi$$

Where Λ is a (sparse) matrix of shape (d_u, n)

E.g. for 1D damped harmonic oscillator:

$$\mathcal{D} = (k \quad \mu \quad m \quad 0)\begin{pmatrix} 1 \\ d_t \\ d_{tt} \\ d_{ttt} \end{pmatrix}$$

$$= m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k$$

PINNs for **equation discovery**:

$$L(\theta, \Lambda) = L_p(\theta, \Lambda) + L_d(\theta)$$

$$L_p(\theta, \Lambda) = \frac{1}{N_p}\sum_{i}^{N_p} \|\Lambda\phi[NN(x_i; \theta)]\|^2 + \|\Lambda\|^2$$

$$L_d(\theta) = \frac{\lambda}{N_d}\sum_{l}^{N_d} \|NN(x_l; \theta) - u_l\|^2$$

Where Λ are treated as **learnable** parameters and $\{x_l, u_l\}$ are a set of (potentially noisy) observational data

Typically, some regularization / prior on Λ (e.g. sparsity) is needed, as this optimisation problem can be very **ill-posed**

PINNs for equation discovery

DNN with Unknown Parameters $\boldsymbol{\theta}$

Layer 1 … Layer l … Layer L

t, x, y, z → $\mathcal{N}(\boldsymbol{\theta})$ → $u(\mathbf{x}, t; \boldsymbol{\theta})$, $v(\mathbf{x}, t; \boldsymbol{\theta})$, $w(\mathbf{x}, t; \boldsymbol{\theta})$

$\mathbf{u} = \{u, v, w\}$, $\mathbf{x} = \{x, y, z\}$

Physical Law with Unknown Parameters $\boldsymbol{\Lambda}$

AutoDiff: 1, ∂_t, ∂_x, ∂_y, ∂_z — *derivatives*

Candidate Function Selection ϕ: $\{1, \mathbf{u}, \mathbf{u}_t, \mathbf{u}^2, \mathbf{u}_x, \mathbf{u}_y, \mathbf{u}^3 \odot \mathbf{u}_{xy}, \dots\}$

PDE Construction

$\mathcal{R}: \mathbf{u}_t - \phi\boldsymbol{\Lambda} \to \mathbf{0}$

where $\mathbf{u}_t = \{u_t, v_t, w_t\}$

Note: $\boldsymbol{\Lambda}$ is sparse

Sparse Regression: $\dot{\mathbf{U}}(\boldsymbol{\theta}) = \boldsymbol{\Phi}(\boldsymbol{\theta})\boldsymbol{\Lambda}$ (collocation points; Sparse)

Data Loss: $\mathcal{L}_d(\boldsymbol{\theta}; \mathcal{D}_u) = \frac{1}{N_m}\|\mathbf{u}^\theta - \mathbf{u}^m\|_2^2$ (measurement)

$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\Lambda}; \mathcal{D}_u, \mathcal{D}_c) = \mathcal{L}_d(\boldsymbol{\theta}; \mathcal{D}_u) + \alpha\mathcal{L}_p(\boldsymbol{\theta}, \boldsymbol{\Lambda}; \mathcal{D}_c) + \beta\|\boldsymbol{\Lambda}\|_0$ (total loss = data loss + physics loss + regularization)

Residual Loss: $\mathcal{L}_p(\boldsymbol{\theta}, \boldsymbol{\Lambda}; \mathcal{D}_c) = \frac{1}{N_c}\|\dot{\mathbf{U}}(\boldsymbol{\theta}) - \boldsymbol{\Phi}(\boldsymbol{\theta})\boldsymbol{\Lambda}\|$ (collocation points)

Solution by ADO: $\hat{\boldsymbol{\Lambda}}_{k+1} := \arg\min_{\boldsymbol{\Lambda}} \left[\|\dot{\mathbf{U}}(\hat{\boldsymbol{\theta}}_k) - \boldsymbol{\Phi}(\hat{\boldsymbol{\theta}}_k)\boldsymbol{\Lambda}\|_2^2 + \beta\|\boldsymbol{\Lambda}\|_0\right]$ by STRidge; $\hat{\boldsymbol{\theta}}_{k+1} := \arg\min_{\boldsymbol{\theta}} \left[\mathcal{L}_d(\boldsymbol{\theta}; \mathcal{D}_u) + \alpha\mathcal{L}_p(\boldsymbol{\theta}, \hat{\boldsymbol{\Lambda}}_{k+1}; \mathcal{D}_c)\right]$ by DNN training

- Trains by alternating between updating Λ and θ

Chen et al, Physics-informed learning of governing equations from scarce data, Nature communications (2021)

PINNs for equation discovery

Ground truth: $u_t + uu_x - 0.0032u_{xx} = 0$

Discovered: $u_t + 1.002uu_x - 0.0032u_{xx} = 0$

- PINN "discovering" Burgers' equation
- By combining datasets sampled under three different I/BCs with 10% noise

Limitations of PINNs?

Solution time

t = 0.00 s | t = 0.05 s | t = 0.10 s | t = 0.15 s | t = 0.20 s

Ground truth (FD)

Prediction (NN)

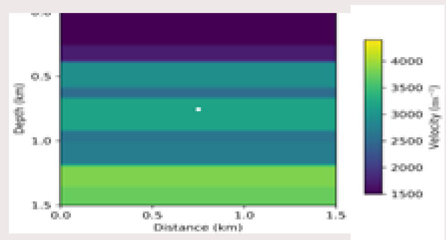
Prediction (NN + physics loss)

Difference (NN)

Difference (NN + physics loss)

PINNs need to be retrained for each IC/BC!

Velocity model, $c(x)$

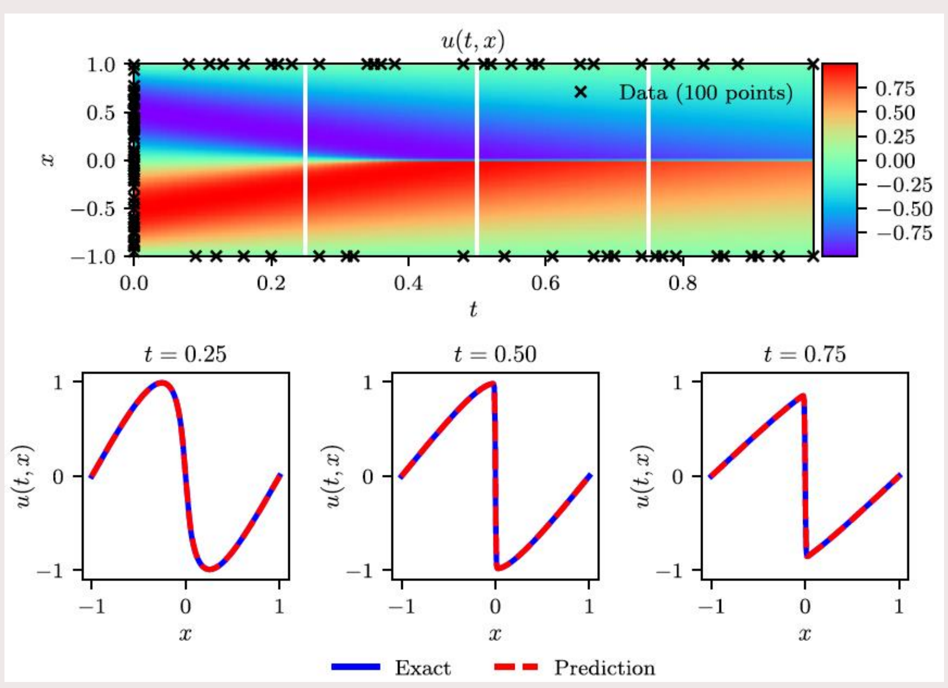


Moseley et al, Solving the wave equation with physics-informed deep learning, ArXiv (2020)

Mini-batch size $N_b = N_p = 500$ (random sampling)
Fully connected network with 10 layers, 1024 hidden units
Softplus activation
Adam optimiser

Training time: ~**1 hour**

Competing loss terms



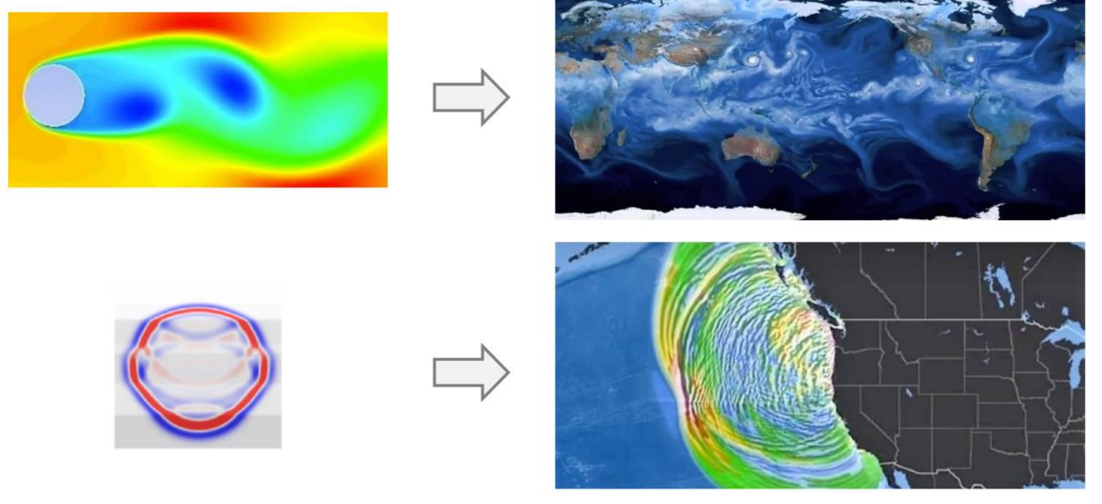
$$L_b(\theta) = \frac{\lambda_1}{N_{b1}} \sum_{j}^{N_{b1}} \left(NN(x_j, 0; \theta) + \sin(\pi x_j)\right)^2 + \frac{\lambda_2}{N_{b2}} \sum_{k}^{N_{b2}} \left(NN(-1, t_k; \theta) - 0\right)^2 + \frac{\lambda_3}{N_{b3}} \sum_{l}^{N_{b3}} \left(NN(+1, t_l; \theta) - 0\right)^2$$

$$L_p(\theta) = \frac{1}{N_p} \sum_{i}^{N_p} \left(\left(\frac{\partial NN}{\partial t} + NN \frac{\partial NN}{\partial x} - \nu \frac{\partial^2 NN}{\partial x^2} \right)(x_i, t_i; \theta) \right)^2$$

How do we choose $\lambda_1, \lambda_2, \lambda_3$? Their relative size can influence the quality of the solution and reinforcement of the BC.

There can be competing terms in the loss functions!

Scaling to more complex problems



It is often challenging to **scale** traditional scientific algorithms to:

- More complex phenomena (**multi-scale**, **multi-physics**)
- Large domains / higher frequency solutions
- Incorporate **real, noisy** and **sparse** data

How do PINNs cope in this setting?

Majority of PINN research focuses on **toy/simplified** problems, as proof-of-principle studies

Why SciML models do not scale? ChatGPT says:

Scaling Scientific Machine Learning (SciML) models presents several challenges. These issues arise from the unique characteristics of scientific data and models, which differ from those typically used in traditional machine learning. Here are key reasons why SciML models may not scale well:

1. Complexity of Scientific Data and Models

- **High Dimensionality**: Scientific datasets are often high-dimensional, involving numerous variables and parameters. This can lead to the "curse of dimensionality," where the computational complexity grows exponentially with the number of variables.
- **Non-linear Dynamics**: Many scientific models, such as those in physics, chemistry, or biology, involve highly non-linear relationships, which are challenging for machine learning models to capture efficiently at larger scales.
- **Coupled Systems**: SciML models often represent coupled systems (e.g., climate models, fluid dynamics), where different subsystems interact with each other. This introduces additional layers of complexity that make it difficult to scale models across multiple systems.

Scaling PINNs to higher frequencies

321 free parameters

(a) PINN ($\omega = 1$, 2 layers, 16 hidden units)

(b) PINN ($\omega = 15$, 2 layers, 16 hidden units)

(c) PINN ($\omega = 15$, 4 layers, 64 hidden units)

(d) PINN ($\omega = 15$, 5 layers, 128 hidden units)

Legend: Exact solution; PINN

66,433 free parameters

(e) Test loss

Legend: PINN ($\omega = 1$, 2 layers, 16 hidden units); PINN ($\omega = 15$, 2 layers, 16 hidden units); PINN ($\omega = 15$, 4 layers, 64 hidden units); PINN ($\omega = 15$, 5 layers, 128 hidden units)

Axes: L1 loss; Training step

PINN solving:

$$\frac{du}{dx} = \cos(\omega x)$$
$$u(0) = 0$$

Problem: PINNs **struggle** to solve high-frequency / multiscale problems

Moseley et al, Finite Basis Physics-Informed Neural Networks (FBPINNs): a scalable domain decomposition approach for solving differential equations, ACM (2023)

Spectral bias issue

(a) Iteration 100

(b) Iteration 1000

(c) Iteration 10000

(d) Iteration 80000

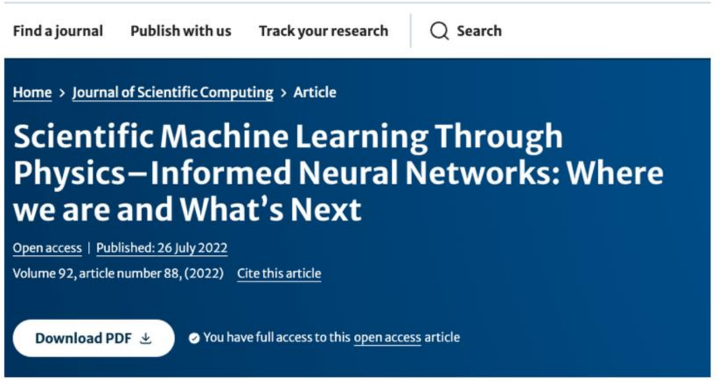
NNs prioritise learning **lower** frequency functions first

Under certain assumptions can be proved via neural tangent kernel theory

Rahaman, N., et al, On the spectral bias of neural networks. 36th International Conference on Machine Learning, ICML (2019)

This behaviour can limit the model's ability to scale effectively, especially when the goal is to approximate complex scientific data with both large-scale and fine-scale dynamics.

PINNs – an entire research field


SPRINGER LINK

Find a journal | Publish with us | Track your research | Search

Home > Journal of Scientific Computing > Article

Scientific Machine Learning Through Physics–Informed Neural Networks: Where we are and What's Next

Open access | Published: 26 July 2022

Volume 92, article number 88, (2022) Cite this article

Download PDF

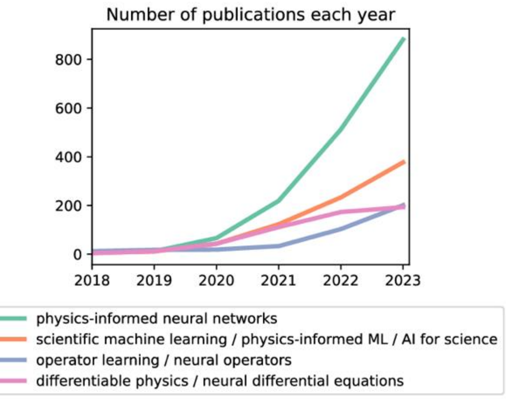
You have full access to this open access article

Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi & Francesco Piccialli

67k Accesses 274 Citations 7 Altmetric Explore all metrics →

Abstract

Physics-Informed Neural Networks (PINN) are neural networks (NNs) that encode model equations, like Partial Differential Equations (PDE), as a component of the neural network itself. PINNs are nowadays used to solve PDEs, fractional equations, integral-differential equations, and stochastic PDEs. This novel methodology has arisen as a multi-task learning framework in which a NN must fit observed data while reducing a PDE residual. This article provides a comprehensive review of the literature on PINNs: while the primary



Source: Scopus keyword search (Feb 2024)

[HTML] **Physics-informed neural networks**: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations

M Raissi, P Perdikaris, GE Karniadakis - Journal of Computational physics, 2019 - Elsevier

… We introduce **physics-informed neural networks** – **neural networks** that are trained to solve supervised learning tasks while respecting any given laws of physics described by general …

Save Cite Cited by 8017 Related articles All 7 versions

100-1000s of PINN applications / extensions!

Advantages / limitations of PINNs

Advantages

- **Mesh-free**
- Can jointly solve **forward** and **inverse** problems
- Often performs well on “messy” problems (where some observational data is available)
- Tractable, analytical solution gradients (e.g. for sensitivity analysis)
- Mostly **unsupervised**

Limitations

- **Computational cost** often high (especially for forward-only problems)
- Can be hard to **optimise** (and convergence properties less well understood)
- Challenging to **scale** to larger domains, multi-scale, multi-physics problems

But.. many PINN extensions exist!

PINNs and their extensions

VICTORITA DOLEAN

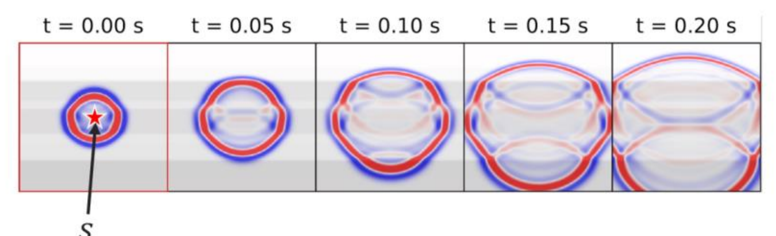
CASA – M&CS

Limitation 1) – computational cost

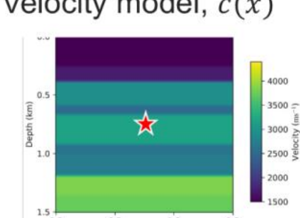
Conditioned PINNs

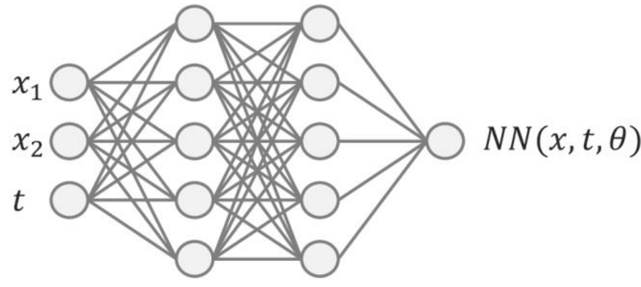
Idea: add I/BCs / other PDE parameters as **additional** network input parameters

Ground truth FD simulation

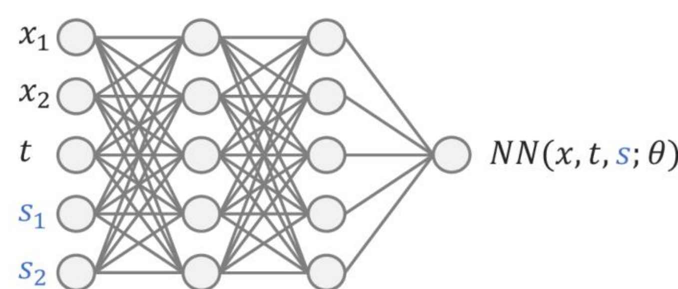


Velocity model, $c(x)$





PINN

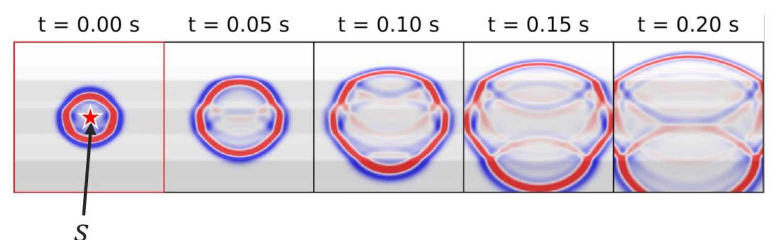


Conditioned PINN

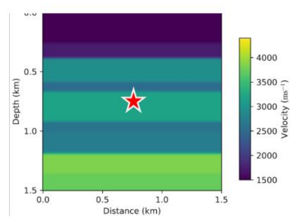
Conditioned PINNs

Idea: add I/BCs / other PDE parameters as **additional** network input parameters

Ground truth FD simulation



Velocity model, $c(x)$



PINN

$$L_b(\theta) = \frac{\lambda}{N_b} \sum_{j}^{N_b} \left(NN(x_j, t_j; \theta) - u_{FD}(x_j, t_j) \right)^2$$

$$L_p(\theta) = \frac{1}{N_p} \sum_{i}^{N_p} \left(\left[\nabla^2 - \frac{1}{c(x_i)^2} \frac{\partial^2}{\partial t^2} \right] NN(x_i, t_i; \theta) \right)^2$$

Conditioned PINN

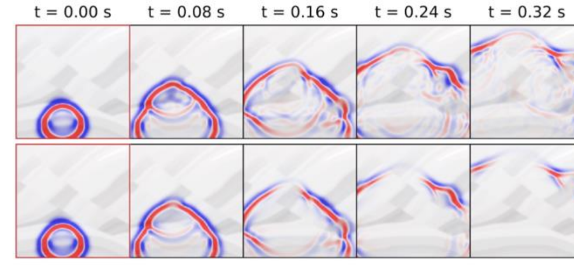
Boundary examples from many **different** source locations

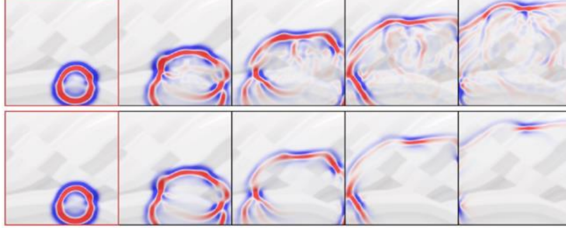
$$L_b(\theta) = \frac{\lambda}{N_b} \sum_{j}^{N_b} \left(NN(x_j, t_j, s_j; \theta) - \underline{u_{FD}(x_j, t_j, s_j)} \right)^2$$

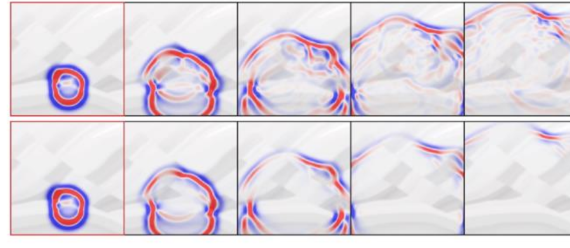
$$L_p(\theta) = \frac{1}{N_p} \sum_{i}^{N_p} \left(\left[\nabla^2 - \frac{1}{c(x_i)^2} \frac{\partial^2}{\partial t^2} \right] NN(\underline{x_i, t_i, s_i}; \theta) \right)^2$$

Randomly sampled input coordinates **and** source locations

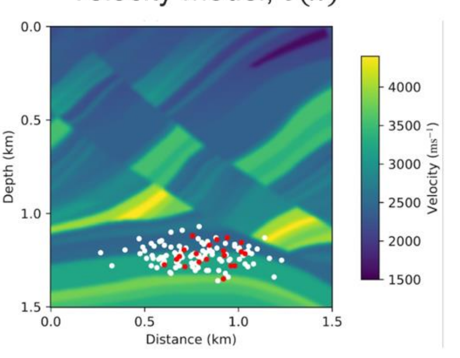
Conditioned PINNs





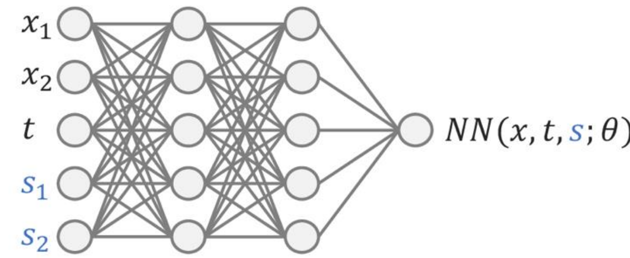


Velocity model, $c(x)$



White = random source locations used for training

Red = source locations used for testing



Means the network **does not** need to be retrained for each simulation => much faster!

Aka a **surrogate** model

Moseley et al, Solving the wave

Physics-informed deep operator networks (DeepONets)

Conditioned PINN for solving reaction-diffusion equation:

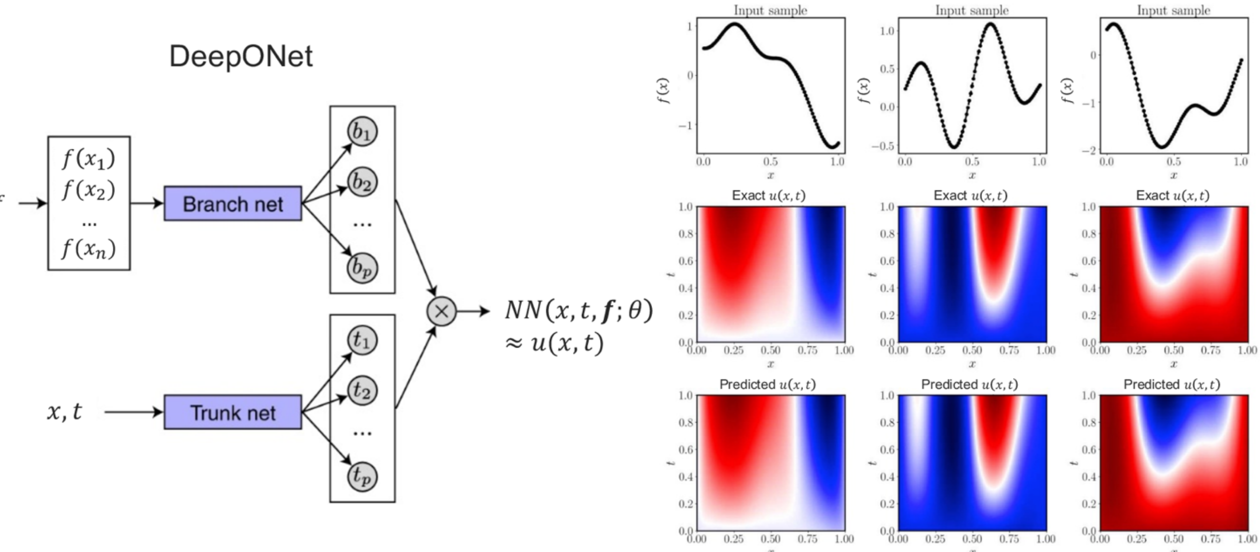
$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} + ku^2 + f(x)$$

Input:
n discretised values of $f(x)$

Output:

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

Trained using many examples of $f(x)$



Wang et al, Learning the solution operator of parametric partial differential equations with physics-informed DeepONets, Science Advances (2021)

Lu et al, Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators, Nature Machine Intelligence (2021)

Discretised PINNs

Idea: **discretise** solution and use e.g. convolutional network to learn spatial/temporal correlations

Input: 0
(fixed value)

Architecture:
(Transposed) convolutional layers

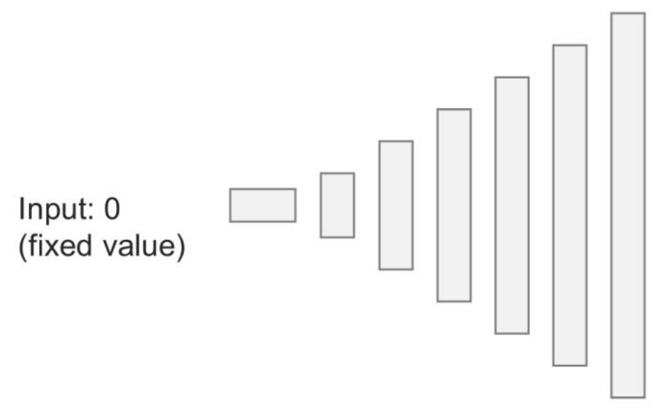
Output: **discretised** solution
(a matrix of values)
e.g. 128 x 256

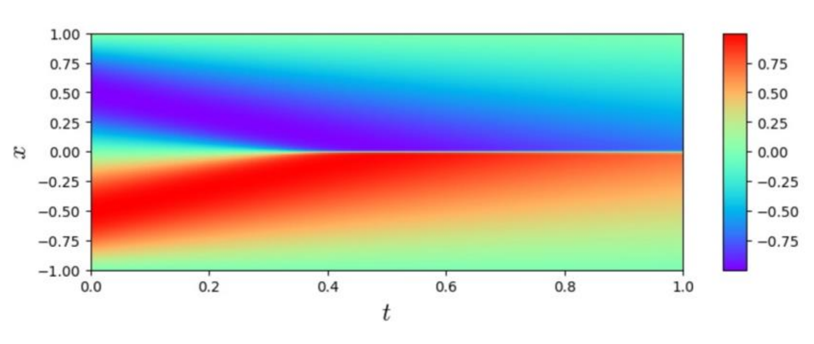
$$NN(\theta)_{ij} \approx u(x = x_i, t = t_j)$$

Instead of: $NN(x, t; \theta) \approx u(x, t)$

Discretised PINNs

Idea: **discretise** solution and use e.g. convolutional network to learn spatial/temporal correlations





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

$$NN(\theta)_{ij} \approx u(x = x_i, t = t_j)$$

$$L_p(\theta) = \frac{1}{N_p}\sum_{i}^{N_p}\left(\left(\frac{\partial NN}{\partial t} + NN\frac{\partial NN}{\partial x} - \nu\frac{\partial^2 NN}{\partial x^2}\right)(x_i, t_i; \theta)\right)^2$$

PINN

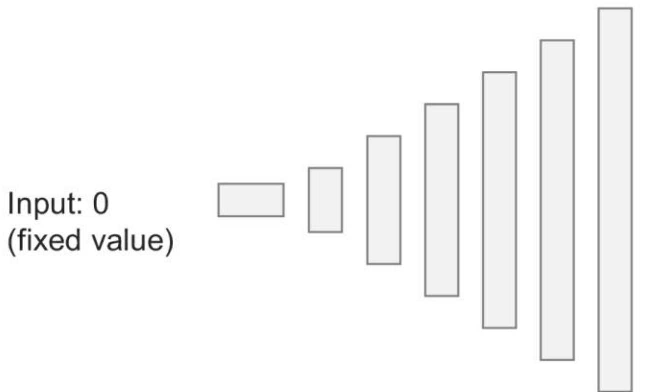
$$L_p(\theta) = \frac{1}{N_x N_t}\sum_{i}^{N_x}\sum_{j}^{N_t}\left(\frac{\delta NN(\theta)_{ij}}{\delta t} + NN(\theta)_{ij}\frac{\delta NN(\theta)_{ij}}{\delta x} - \nu\frac{\delta^2 NN(\theta)_{ij}}{\delta x^2}\right)^2$$

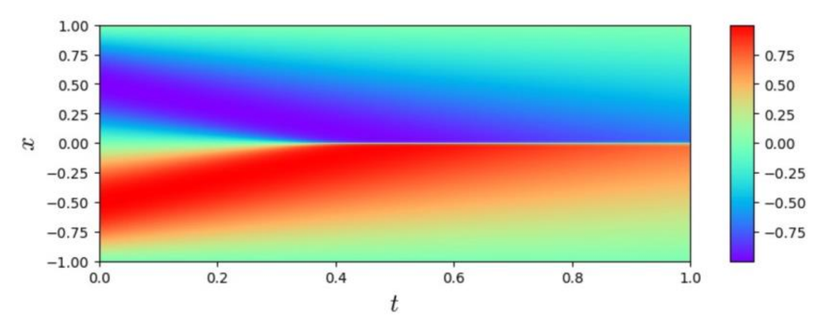
Discretised PINN

Discretised PINNs

Idea: **discretise** solution and use e.g. convolutional network to learn spatial/temporal correlations

Input: 0
(fixed value)





Derivatives in loss function are **approximated** using **finite difference** filters, e.g.

$$\frac{\delta NN(\theta)_{ij}}{\delta t} = \frac{NN(\theta)_{ij} - NN(\theta)_{i\,j-1}}{t_j - t_{j-1}}$$

$$NN(\theta)_{ij} \approx u(x = x_i, t = t_j)$$

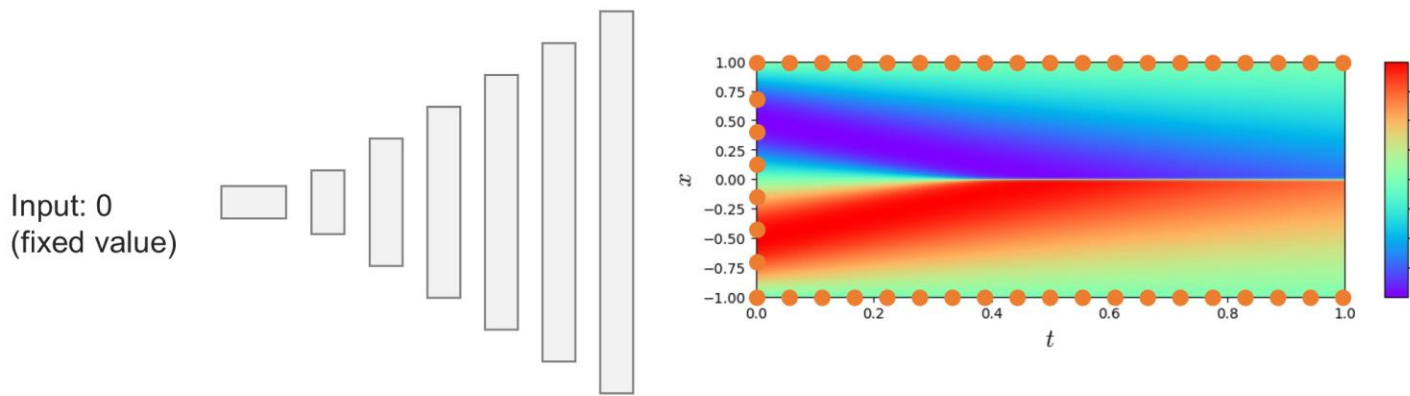
$$L_p(\theta) = \frac{1}{N_x N_t} \sum_i^{N_x} \sum_j^{N_t} \left(\frac{\delta NN(\theta)_{ij}}{\delta t} + NN(\theta)_{ij} \frac{\delta NN(\theta)_{ij}}{\delta x} - \nu \frac{\delta^2 NN(\theta)_{ij}}{\delta x^2} \right)^2$$

Discretised PINN

Autodiff is still used to update θ

Discretised PINNs

Idea: **discretise** solution and use e.g. convolutional network to learn spatial/temporal correlations



Initial/boundary conditions are asserted by **padding** the edges of the output solution with **appropriate** values (=hard constraint), e.g.

$$NN(\theta)_{i\ j=0} = -\sin(\pi x_i)$$
$$NN(\theta)_{i=0\ j} = NN(\theta)_{i=128\ j} = 0$$

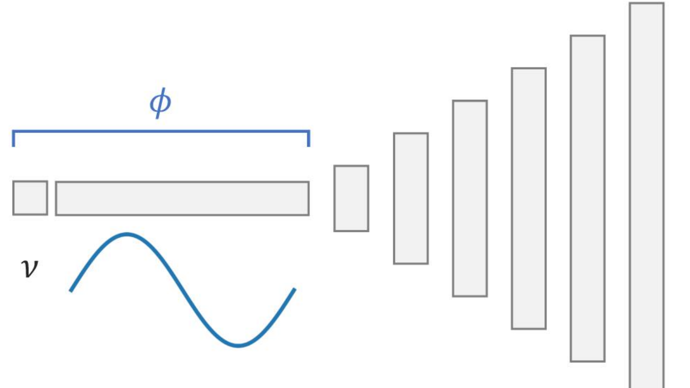
$$NN(\theta)_{ij} \approx u(x = x_i, t = t_j)$$

$$L_p(\theta) = \frac{1}{N_x N_t} \sum_i^{N_x} \sum_j^{N_t} \left(\frac{\delta NN(\theta)_{ij}}{\delta t} + NN(\theta)_{ij} \frac{\delta NN(\theta)_{ij}}{\delta x} - \nu \frac{\delta^2 NN(\theta)_{ij}}{\delta x^2} \right)^2$$

Discretised PINN

Conditioned discretised PINNs

Idea: **discretise** solution and use e.g. convolutional network to learn spatial/temporal correlations
And condition on I/BCs / other PDE parameters



Input: (discretised) I/BCs / other PDE parameters

$$NN(\phi;\theta)_{ij} \approx u(x = x_i, t = t_j)$$

$$L_p(\theta) = \frac{1}{N_x N_t N_\phi} \sum_{k}^{N_\phi} \sum_{i}^{N_x} \sum_{j}^{N_t} \left(\frac{\delta NN(\phi_k;\theta)_{ij}}{\delta t} + NN(\phi_k;\theta)_{ij} \frac{\delta NN(\phi_k;\theta)_{ij}}{\delta x} - \phi_{k0} \frac{\delta^2 NN(\phi_k;\theta)_{ij}}{\delta x^2} \right)^2$$

Conditioned discretised PINN

Conditioned discretised PINNs

Input:
permeability field

PDE: steady-state flow in heterogeneous media (Darcy flow)

Output: fluid pressure and flux field

$$V(\hat{\mathbf{y}}; \mathbf{x}) = \frac{1}{n}\left(\|\boldsymbol{\tau} + \mathbf{x} \odot \nabla \mathbf{u}\|_2^2 + \|\nabla \cdot \boldsymbol{\tau} - \mathbf{f}\|_2^2\right)$$
$$B(\hat{\mathbf{y}}) = \|\mathbf{u}[:,0] - \mathbf{1}\|_2^2 + \|\mathbf{u}[:,-1]\|_2^2 + \|\boldsymbol{\tau}_2[[0,-1],:]\|_2^2$$
$$L(\hat{\mathbf{y}}; \mathbf{x}) = V(\hat{\mathbf{y}}; \mathbf{x}) + \lambda B(\hat{\mathbf{y}})$$

Physics-constrained Loss

$\mathbf{x}$

$\hat{\mathbf{y}} = [\mathbf{u}, \boldsymbol{\tau}]$

Sobel Filter

Conv

$48 \times 32 \times 32$

Dense Block

$144 \times 32 \times 32$

Encoding

$72 \times 16 \times 16$

Dense Block

$200 \times 16 \times 16$

Decoding

$100 \times 32 \times 32$

Dense Block

$196 \times 32 \times 32$

Decoding

Pressure

Horizontal Flux

Vertical Flux

- Fully **unsupervised** training (no simulation data required)
- Only requires many input examples to train

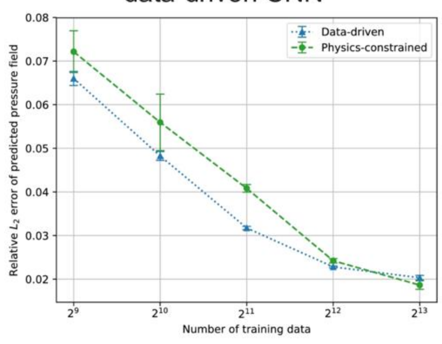
Zhu, Y et al, Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. Journal of Computational Physics (2019)

Conditioned discretised PINNs

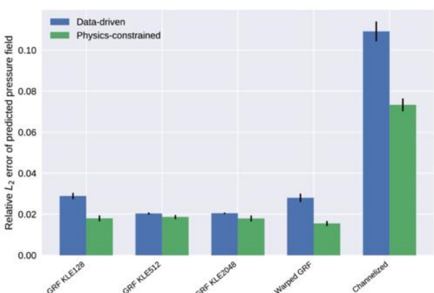
(c) Channelized, test 1.

(d) Channelized, test 2.

Physics-informed vs fully data-driven CNN



Generalisation to different input distributions



Zhu, Y et al, Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. Journal of Computational Physics (2019)

Advantages / limitations of discretised PINNs

Advantages

- Allows the use of CNNs to exploit spatial correlations between inputs/outputs of PDE
- Can be extended to:
  - Irregular geometries (Graph NNs)
  - Explicit time dependence (RNNs)
  - Mixed continuous/discrete input coordinates

Limitations

- Relies on approximate FD gradients
- Only outputs discretised solution; needs to be retrained to output on larger domains / finer grids

Geneva et al, Modeling the dynamics of PDE systems with physics-constrained deep auto-regressive networks, JCP (2020)

Gao et al, PhyGeoNet: Physics-informed geometry-adaptive convolutional neural networks for solving parameterized steady-state PDEs on irregular domain, JCP (2021)

Training PINNs with finite differences

Most time is spent **computing gradients**, not the forward pass, when training PINNs

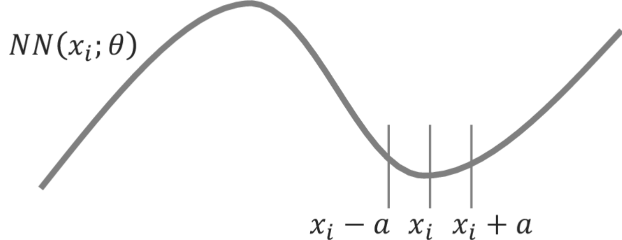
$$L_p(\theta) = \frac{1}{N_p} \sum_{i}^{N_p} \|\mathcal{D}[NN(x_i;\theta)] - f(x_i)\|^2$$

Idea: instead of using exact gradients from autodifferentiation, use **approximate** gradients from **finite differences**

For each collocation point x_i:

1. Sample a stencil of input points around x_i
2. Run forward pass of network with all these points
3. Approximate derivatives in $\mathcal{D}$ using finite differences
4. Compute loss function using these derivatives

Autodiff is still used to update θ



$$\frac{\partial NN(x_i;\theta)}{\partial x} \approx \frac{NN(x_i + a;\theta) - NN(x_i - a;\theta)}{2a}$$

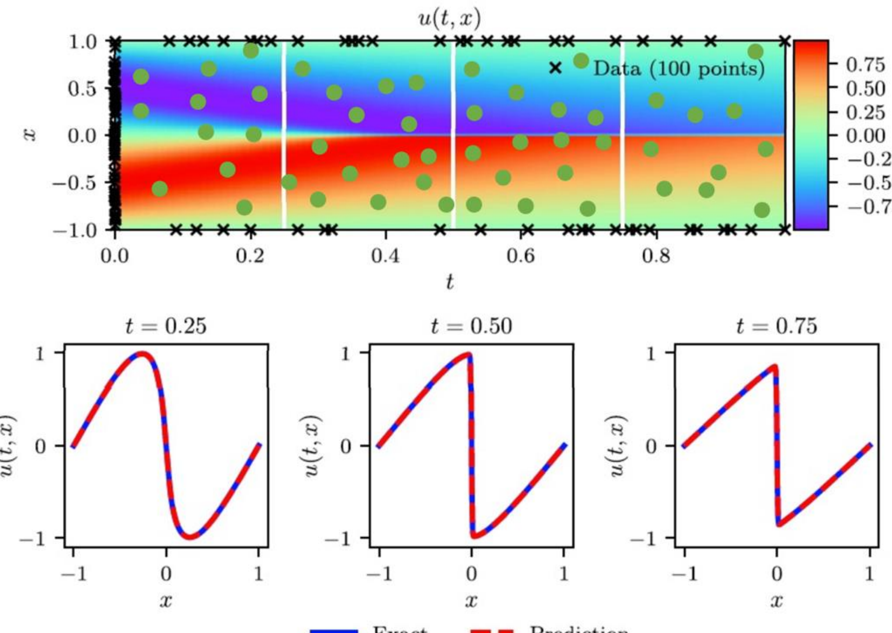
Can offer 2-4x speedups depending on PDE

But choosing a suitable value of a is critical

Sharma et al, Accelerated Training of Physics-Informed Neural Networks (PINNs) using Meshless Discretizations, NeurIPS (2022)

Limitation 2) – poor convergence

Competing loss terms



Raissi et al, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, JCP (2018)

$$L_b(\theta) = \frac{\lambda_1}{N_{b1}} \sum_j^{N_{b1}} \left(NN(\underline{x_j}, 0; \theta) + \underline{\sin(\pi x_j)}\right)^2 + \frac{\lambda_2}{N_{b2}} \sum_k^{N_{b2}} \left(NN(-1, \underline{t_k}; \theta) - \underline{0}\right)^2 + \frac{\lambda_3}{N_{b3}} \sum_l^{N_{b3}} \left(NN(+1, \underline{t_l}; \theta) - \underline{0}\right)^2$$

$$L_p(\theta) = \frac{1}{N_p} \sum_i^{N_p} \left(\left(\frac{\partial NN}{\partial t} + NN \frac{\partial NN}{\partial x} - \nu \frac{\partial^2 NN}{\partial x^2} \right) \underline{(x_i, t_i; \theta)} \right)^2$$

How do we choose λ_1, λ_2, and λ_3?

λ too small => doesn't learn unique solution
λ too large => only learns boundary condition

Thus, there can be **competing** terms in the loss function

Hard initial / boundary conditions

Idea: use neural network as part of a solution **ansatz**

Burgers' problem above:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0$$

$$u(x,0) = -\sin(\pi x)$$
$$u(-1,t) = u(+1,t) = 0$$

Then let the solution approximated by

$$\hat{u}(x,t;\theta) = (x-1)(x+1)(t-0)NN(x,t;\theta) - \sin(\pi x) \approx u(x,t)$$

It is easy to verify that the initial / boundary conditions are now satisfied by **construction**

Lagaris et al, Artificial neural networks for solving ordinary and partial differential equations, IEEE (1998)

Hard initial / boundary conditions

Idea: use neural network as part of a solution **ansatz**

Burgers' problem above:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0$$

$$u(x,0) = -\sin(\pi x)$$
$$u(-1,t) = u(+1,t) = 0$$

Then let the solution approximated by

$$\hat{u}(x,t;\theta) = (x-1)(x+1)(t-0)NN(x,t;\theta) - \sin(\pi x) \approx u(x,t)$$

It is easy to verify that the initial / boundary conditions are now satisfied by **construction**

Lagaris et al, Artificial neural networks for solving ordinary and partial differential equations, IEEE (1998)

Thus, the boundary loss $L_b(\theta)$ is not needed and the problem is turned into an **unconstrained** optimisation problem, where we minimize

$$L(\theta) = L_p(\theta) = \frac{1}{N_p}\sum_i^{N_p}\left(\left(\frac{\partial \hat{u}}{\partial t} + \hat{u}\frac{\partial \hat{u}}{\partial x} - \nu\frac{\partial^2 \hat{u}}{\partial x^2}\right)(x_i,t_i;\theta)\right)^2$$

Note:

- You can think of the ansatz as a custom final layer of the network, asserting a **hard** constraint on the network's output

- Autodifferentiation lets us differentiate through ansatz

- Training is now fully **unsupervised** (only need collocation points)

- Can be challenging to use this approach for complex boundary conditions

Hard vs soft PINN

(a) CFD benchmark

(b) DNN hard

(c) DNN soft $\lambda = 1$

(d) DNN soft $\lambda = 10$

(e) DNN soft $\lambda = 100$

(f) DNN soft $\lambda = 1000$

Sun et al, Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data, Computer Methods in Applied Mechanics and Engineering (2020)

Soft PINN with varying λ vs hard PINN
When solving incompressible Navier-Stokes equations

Adaptive lambdas

Idea: Specify **separate** λ for each training point
And treat λ as **learnable**

$$L(\theta) = L_b(\theta) + L_p(\theta)$$

$$L_b(\theta, \lambda) = \sum_k \sum_j \lambda_{kj} \left\| \mathcal{B}_k[NN(x_{kj};\theta)] - g_k(x_{kj}) \right\|^2$$

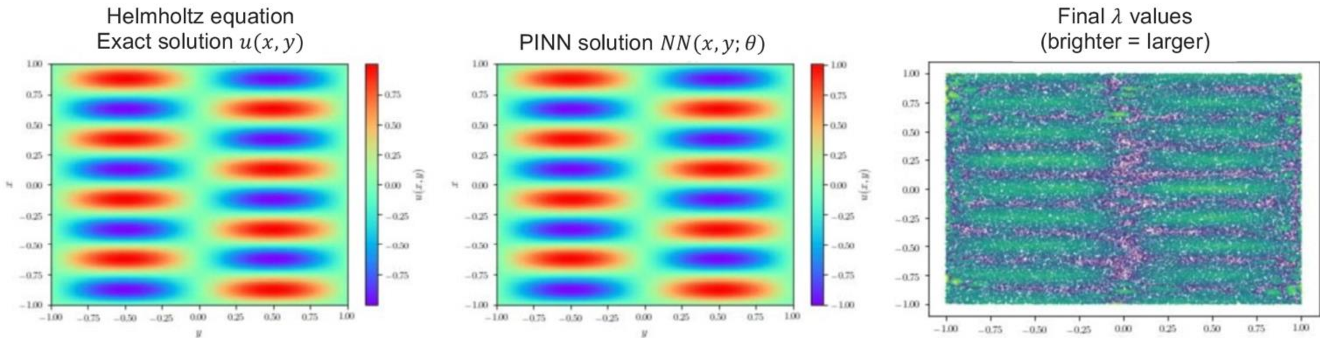
$$L_p(\theta, \lambda) = \sum_i \lambda_i \| \mathcal{D}[NN(x_i;\theta)] - f(x_i) \|^2$$

Loop:

1. Compute gradients $\frac{\partial L}{\partial \theta_l}, \frac{\partial L}{\partial \lambda_l}$
2. Update weights,
   $$\theta_l \leftarrow \theta_l - \gamma \frac{\partial L}{\partial \theta_l} \text{ (minimise } L)$$
3. Update lambdas,
   $$\lambda_l \leftarrow \lambda_l + \gamma \frac{\partial L}{\partial \lambda_l} \text{ (\textbf{maximise} } L)$$

Note, for example, $\frac{\partial L}{\partial \lambda_i} = \| \mathcal{D}[NN(x_i;\theta)] - f(x_i) \|^2$

λ can be thought of as a soft attention layer



McClenny et al, Self-adaptive physics-informed neural networks, JCP (2023)

Adaptive collocation points

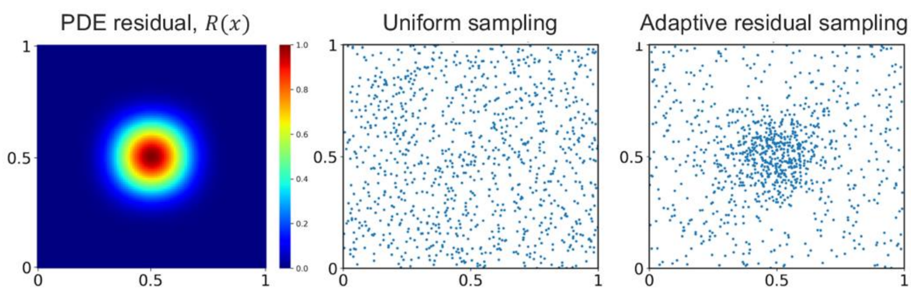


Table 2. L^2 relative error of the PINN solution for the forward problems.

	Diffusion	Burgers'	Allen–Cahn	Wave
No. of residual points	30	2000	1000	2000
Grid	0.66±0.06%	13.7±2.37%	93.4±6.98%	81.3±13.7%
Random	0.74±0.17%	13.3±8.35%	22.2±16.9%	68.4±20.1%
LHS	0.48±0.24%	13.5±9.05%	26.6±15.8%	75.9±33.1%
Halton	0.24±0.17%	4.51±3.93%	0.29±0.14%	60.2±10.0%
Hammersley	0.17±0.07%	3.02±2.98%	**0.14±0.14**%	58.9±8.52%
Sobol	0.19±0.07%	3.38±3.21%	0.35±0.24%	57.5±14.7%
Random-R	**0.12±0.06**%	1.69±1.67%	0.55±0.34%	**0.72±0.90**%
RAR-G [3]	0.20±0.07%	**0.12±0.04**%	0.53±0.19%	0.81±0.11%
RAD	**0.11±0.07**%	**0.02±0.00**%	**0.08±0.06**%	**0.09±0.04**%
RAR-D	**0.14±0.11**%	**0.03±0.01**%	**0.09±0.03**%	**0.29±0.04**%

Idea: sample collocation points according to the **probability distribution**

$$x \sim p(x) = \frac{R(x)}{A}$$

Where R is the PDE residual, $R(x) = \|\mathcal{D}[NN(x;\theta)] - f(x)\|^2$

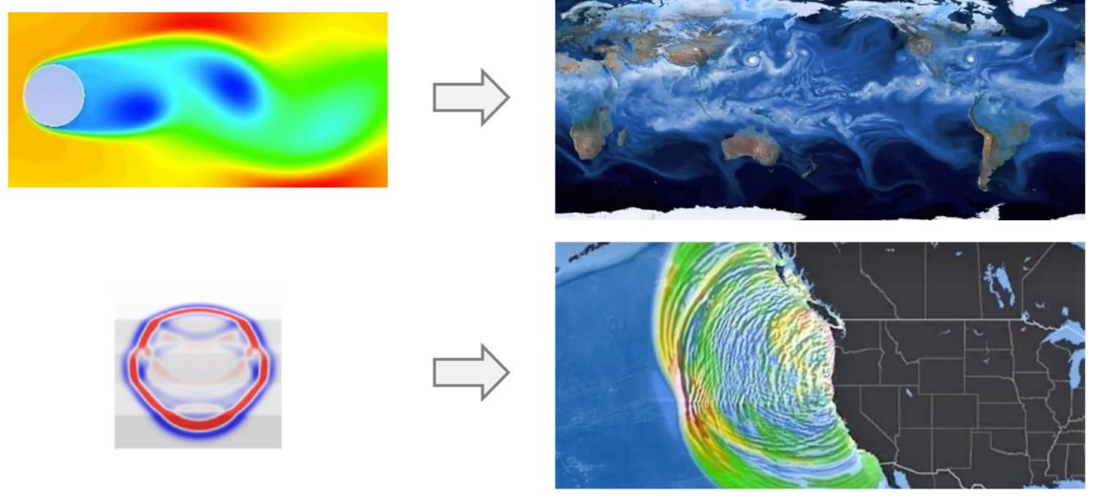
And A is a normalising constant

Inspired by adaptive mesh refinement in FEM

Wu et al, A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks, Computer Methods in Applied Mechanics and Engineering (2023)

Limitation 3) – scaling to more complex problems

Scaling to more complex problems



Majority of PINN research focuses on **toy/simplified** problems, as proof-of-principle studies

It is often challenging to **scale** traditional scientific algorithms to:

- More complex phenomena (**multi-scale**, **multi-physics**)
- Large domains / higher frequency solutions
- Incorporate **real, noisy** and **sparse** data

How do PINNs cope in this setting?

Scaling PINNs to higher frequencies

321 free parameters

(a) PINN ($\omega = 1$, 2 layers, 16 hidden units)

(b) PINN ($\omega = 15$, 2 layers, 16 hidden units)

(c) PINN ($\omega = 15$, 4 layers, 64 hidden units)

(d) PINN ($\omega = 15$, 5 layers, 128 hidden units)

Exact solution
PINN

66,433 free parameters

(e) Test loss

L1 loss
Training step

PINN ($\omega = 1$, 2 layers, 16 hidden units)
PINN ($\omega = 15$, 2 layers, 16 hidden units)
PINN ($\omega = 15$, 4 layers, 64 hidden units)
PINN ($\omega = 15$, 5 layers, 128 hidden units)

PINN solving:

$$\frac{du}{dx} = \cos(\omega x)$$
$$u(0) = 0$$

Problem: PINNs **struggle** to solve high-frequency / multiscale problems

Moseley et al, Finite Basis Physics-Informed Neural Networks (FBPINNs): a scalable domain decomposition approach for solving differential equations, ACM (2023)

Spectral bias issue

(a) Iteration 100

(b) Iteration 1000

(c) Iteration 10000

(d) Iteration 80000

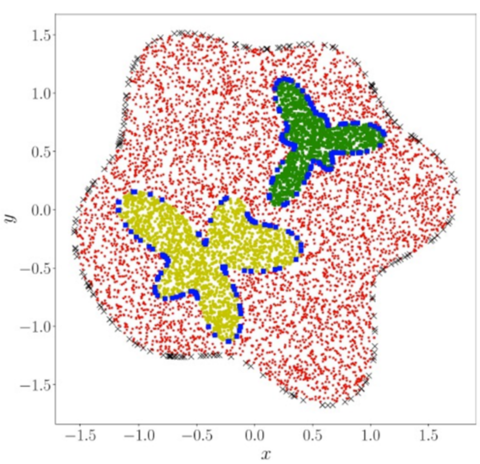
NNs prioritise learning **lower** frequency functions first

Under certain assumptions can be proved via neural tangent kernel theory

Rahaman, N., et al, On the spectral bias of neural networks. 36th International Conference on Machine Learning, ICML (2019)

This behaviour can limit the model's ability to scale effectively, especially when the goal is to approximate complex scientific data with both large-scale and fine-scale dynamics.

PINNs + domain decomposition



Idea:

Take a “**divide-and-conquer**” strategy to model more complex problems:

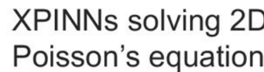
1. Divide modelling domain into many smaller **subdomains**
2. Use a separate neural network in each subdomain to model the solution

Hypothesis:

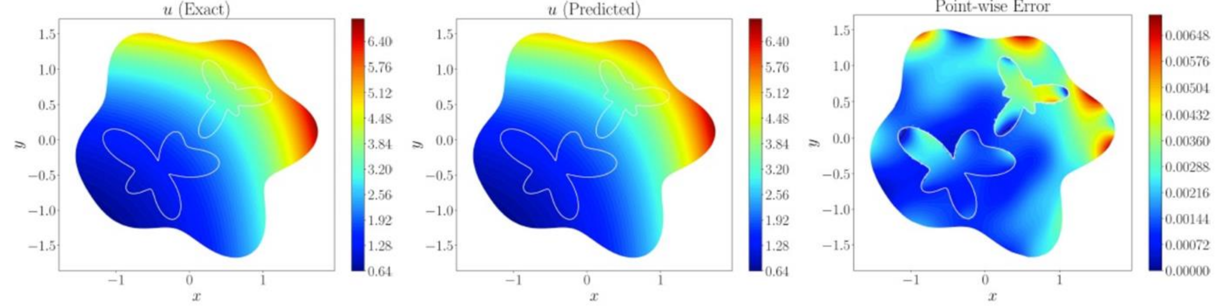
The resulting (coupled) local optimization problems are easier to solve than a single global problem

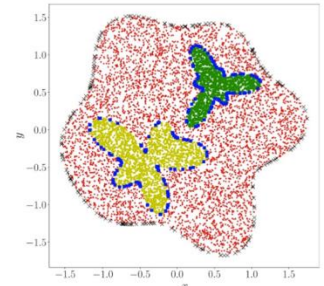
Jagtap, A., et al., Extended physics-informed neural networks (XPINNs): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. Communications in Computational Physics (2020)

Extended PINNs (XPINNs)



XPINNs solving 2D Poisson's equation





$$L(\theta_m) = \frac{\lambda_b}{N_b}\sum_{i}^{N_b}\left(NN(x_i;\theta_m) - u_i\right)^2$$

Boundary loss

$$+ \frac{\lambda_p}{N_p}\sum_{j}^{N_p}\left(R\left(NN(x_j;\theta_m)\right)\right)^2$$

Physics loss

$$+ \sum_{l}^{N_n}\frac{\lambda_l}{N_l}\sum_{k}^{N_l}\left(NN(x_k;\theta_l) - NN(x_k;\theta_m)\right)^2$$

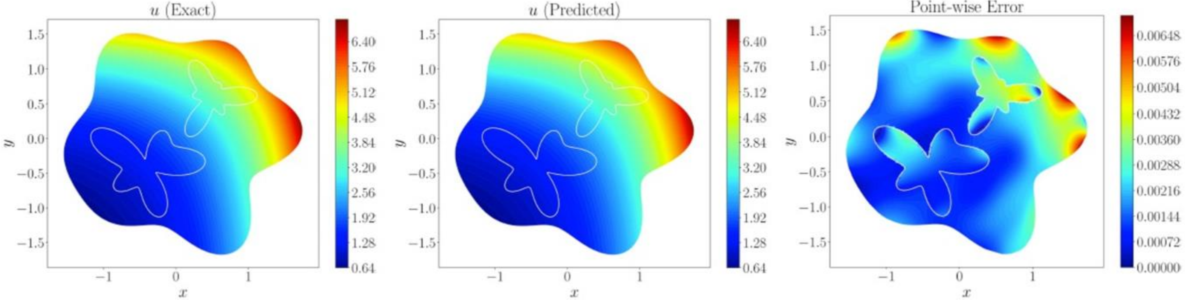
Interface conditions

$l \in \text{Neighbours}(m)$

Jagtap, A., et al., Extended physics-informed neural networks (XPINNs): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. Communications in Computational Physics (2020)

Extended PINNs (XPINNs)

XPINNs solving 2D Poisson's equation



Limitations:

- Introduces discontinuities in solution at subdomain interfaces
- Requires extra loss terms

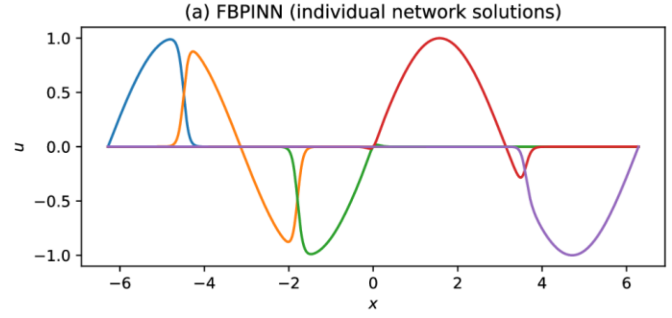
$$L(\theta_m) = \frac{\lambda_b}{N_b}\sum_{i}^{N_b}(NN(x_i;\theta_m) - u_i)^2$$ Boundary loss

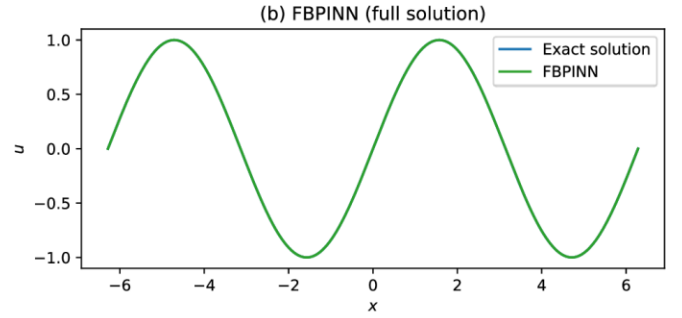
$$+ \frac{\lambda_p}{N_p}\sum_{j}^{N_p}\left(R\left(NN(x_j;\theta_m)\right)\right)^2$$ Physics loss

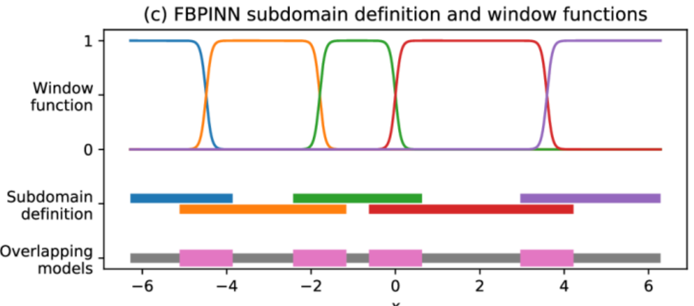
$$+ \sum_{\substack{l \\ l \in \text{Neighbours}(m)}}^{N_n}\frac{\lambda_l}{N_l}\sum_{k}^{N_l}\left(NN(x_k;\theta_l) - NN(x_k;\theta_m)\right)^2$$ Interface conditions

Jagtap, A., et al., Extended physics-informed neural networks (XPINNs): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. Communications in Computational Physics (2020)

Finite basis PINNs (FBPINNs)







$$\hat{u}(x;\theta) = \mathcal{C}\left[\sum_{i}^{n} w_i(x) \cdot \mathrm{unnorm} \circ NN_i \circ \mathrm{norm}_i(x)\right]$$

Window function

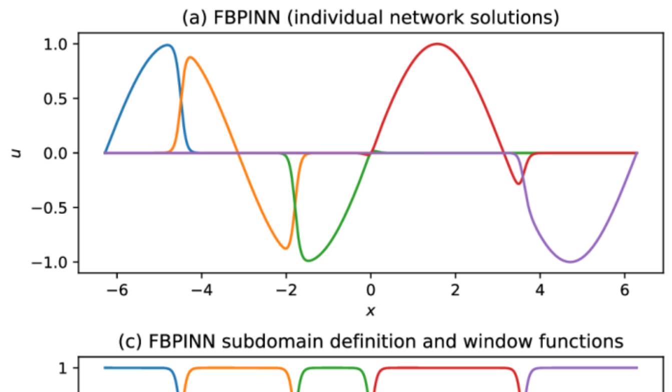
Subdomain network

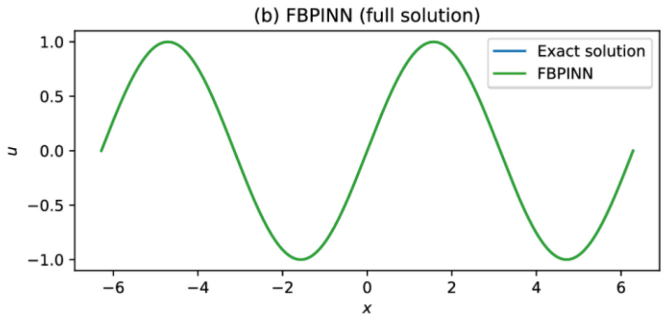
Individual subdomain normalisation

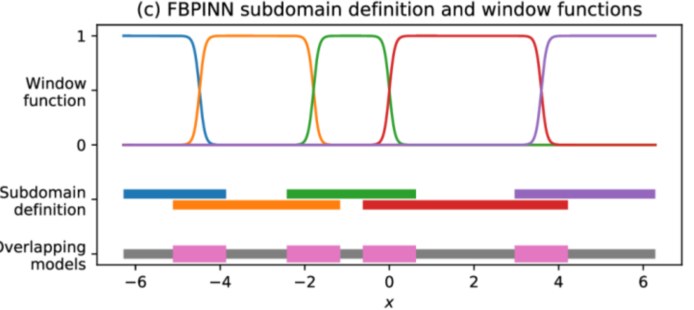
Moseley et al, Finite Basis Physics-Informed Neural Networks (FBPINNs): a scalable domain decomposition approach for solving differential equations, ACM (2023)

Idea: use **overlapping** subdomains and a **globally** defined solution ansatz

Finite basis PINNs (FBPINNs)





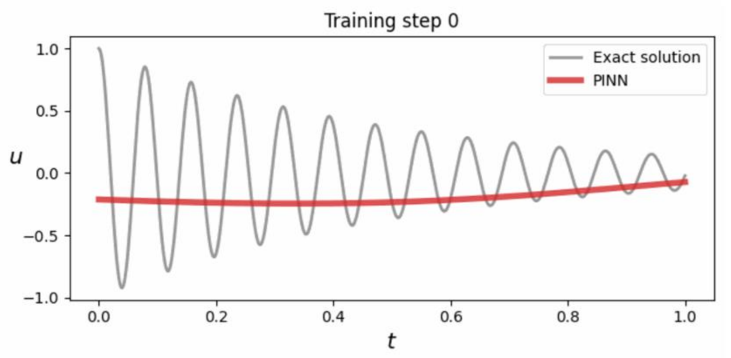


Note:

- By construction, FBPINN solution is continuous across subdomain interfaces
- Can be trained with same loss function as PINNs
- FBPINNs can simply be thought of as a custom NN architecture for PINNs

Moseley et al, Finite Basis Physics-Informed Neural Networks (FBPINNs): a scalable domain decomposition approach for solving differential equations, ACM (2023)

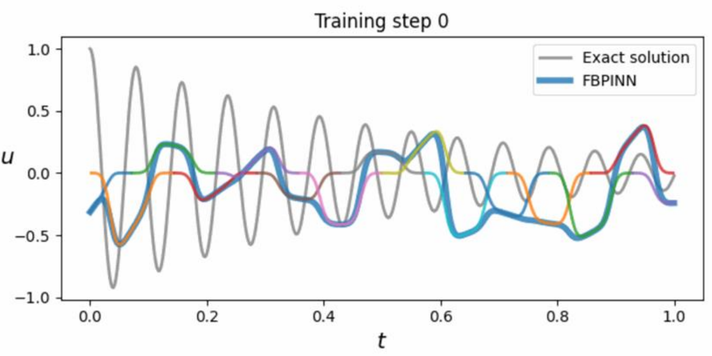
FBPINNs vs PINNs



PINN solution

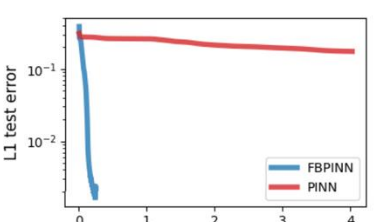
Network size: 2 hidden layers, 64 hidden units

Damped harmonic oscillator



FBPINN solution

Number of subdomains: 15
Subdomain network size: 1 hidden layer, 32 hidden units



Other PINN extensions

- Different differential equations:
  - Fractional equations, stochastic PDEs, variational formulations (VPINNs), mixed formulations, …
- Uncertainty estimation:
  - Bayesian PINNs (B-PINNs), GAN PINNs, …
- Architectures:
  - Adaptive activation functions, extreme learning machines, SIRENs, wavelet layers, LSTMs, …
- Other optimisation strategies:
  - NTK weighting, casual training, specialised optimisers
- PINN software libraries:
  - NVIDIA Modulus, DeepXDE, NeuroDiffEq, PyDENs, …
- And much more!

SPRINGER LINK

Find a journal   Publish with us   Track your research   Search

Home › Journal of Scientific Computing › Article

Scientific Machine Learning Through Physics–Informed Neural Networks: Where we are and What's Next

Open access | Published: 26 July 2022

Volume 92, article number 88, (2022)   Cite this article

Download PDF   You have full access to this open access article

Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi & Francesco Piccialli

67k Accesses   274 Citations   7 Altmetric   Explore all metrics →

Abstract

Physics-Informed Neural Networks (PINN) are neural networks (NNs) that encode model equations, like Partial Differential Equations (PDE), as a component of the neural network itself. PINNs are nowadays used to solve PDEs, fractional equations, integral-differential equations, and stochastic PDEs. This novel methodology has arisen as a multi-task learning framework in which a NN must fit observed data while reducing a PDE residual. This article provides a comprehensive review of the literature on PINNs: while the primary

1. Training DNN is slow and do not scale

- Deep neural networks (DNNs) are used to solve learning tasks with “state-of-the-art accuracies” …but the training time and cost is increasing!
- **GPT-4** contain hundreds of billions of parameters, far exceeding the scale of earlier models like GPT-3. Training a model like GPT-4 can take months on **thousands of GPUs**…
- All sophisticated deep learning architectures also have long training times and models do not scale!

The **future of AI** may depend on developing more **resource-efficient architectures** and **training paradigms!**

Why SciML models do not scale? ChatGPT says:

Scaling Scientific Machine Learning (SciML) models presents several challenges. These issues arise from the unique characteristics of scientific data and models, which differ from those typically used in traditional machine learning. Here are key reasons why SciML models may not scale well:

1. Complexity of Scientific Data and Models

- **High Dimensionality**: Scientific datasets are often high-dimensional, involving numerous variables and parameters. This can lead to the "curse of dimensionality," where the computational complexity grows exponentially with the number of variables.
- **Non-linear Dynamics**: Many scientific models, such as those in physics, chemistry, or biology, involve highly non-linear relationships, which are challenging for machine learning models to capture efficiently at larger scales.
- **Coupled Systems**: SciML models often represent coupled systems (e.g., climate models, fluid dynamics), where different subsystems interact with each other. This introduces additional layers of complexity that make it difficult to scale models across multiple systems.

2. Divide and conquer approaches tackle spectral bias

Chat GPT says: Partition of Unity Networks (POU Nets) offer a promising approach to scaling Scientific Machine Learning (SciML) models:

- **Localized learning** reduces computational complexity and improves model stability.
- **Parallel training** of local models enhances scalability.
- **Piecewise approximation** handles non-linearities more effectively.
- **Adaptive refinement** allows models to scale in specific regions where higher accuracy is needed.
- **Preservation of physical constraints** ensures model accuracy, even when scaled.

Partition of Unity Networks (POUNets) offer several mechanisms to mitigate spectral bias, especially in the context of scaling SciML models.

- **Partitioning the domain** into smaller regions,
- **Allowing localized learning** of high-frequency features.
- **Leveraging multiscale representations**, and
- **Ensuring smooth transitions across regions**

One-level divide and conquer approach based on POU nets

(a) FBPINN (individual network solutions)

(b) FBPINN (full solution)

Exact solution

FBPINN

(c) FBPINN subdomain definition and window functions

Window function

Subdomain definition

Overlapping models

(d) Test loss

(e) Test loss

FBPINN

PINN

L1 loss

Training step

FLOPS

1e13

FBPINN by VD, B. Moseley et al.

Hu, Jagtap, Karniadakis, Kawaguchi, APINNs: A gating network-based soft DDM, 2024

3. One-level architectures are not enough

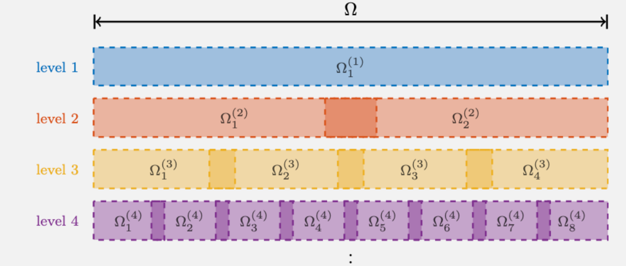
L-level network architecture

Let L **overlapping domain decompositions** $\Omega = \cup_{j=1}^{J^{(l)}} \Omega_j^{(l)}$ and **window functions** $\omega_j^{(l)}$ with

$$\sum_{j=1}^{J^{(l)}} \omega_j^{(l)} \equiv 1, \ \mathrm{supp}(\omega_j^{(l)}) \subset \Omega_j^{(l)}.$$

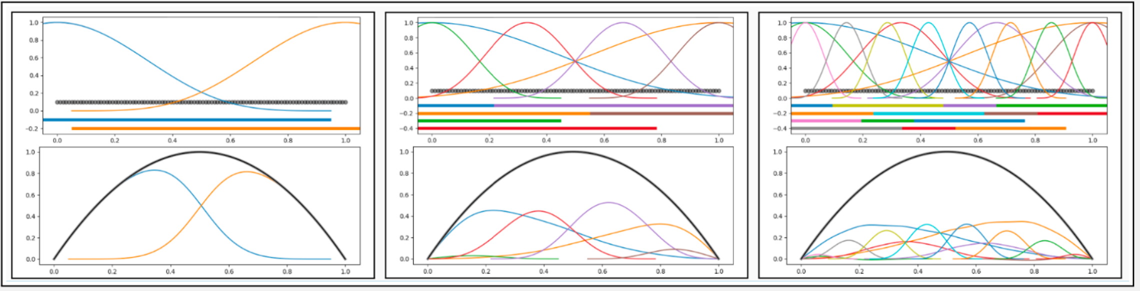
Then the multilevel architecture is

$$u(\mathbf{x}, \boldsymbol{\theta}) = \mathcal{C} \sum_{l=1}^{L} \sum_{j=1}^{J^{(l)}} \omega_j^{(l)} u_j^{(l)}(\mathbf{x}, \boldsymbol{\theta}_j^{(l)})$$



Loss function

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} (\mathcal{N}[\mathcal{C} \sum_{l=1}^{L} \sum_{\mathbf{x}_i \in \Omega_j^{(l)}} \omega_j^{(l)} u_j^{(l)}](\mathbf{x}_i, \boldsymbol{\theta}_j^{(l)}) - f(\mathbf{x}_i))^2.$$



Dolean, Heinlein, Mishra, Moseley, Multilevel DDM (CMAME, 2024)

Final thoughts and conclusions

There is far more than that:

- Layer parallelism can be achieved in a similar fashion (comes in 1 and 2-levels flavours):

  Kopanicacova, Kothari, Karniadakis, Krause, SISC, 2023
  Lee, Kopanicakova, Karniadakis, arXiv, 2024

- Combining POU networks with ELM (extreme learning machines) and add a second level: RFM (Random Feature Method)

  Chi, Chen, E, Yang, JML, 2022
  Chen, E, Sun, CAMC, 2024

- .... (literature growing at a fast pace as we speak)

Conclusions

Home > Computational Science and Engineering > Article

Machine learning and domain decomposition methods – a survey

Research | Open access | Published: 23 September 2024
Volume 1, article number 2, (2024) Cite this article

Download PDF
You have full access to this open access article

Axel Klawonn, Martin Lanser & Janine Weber

791 Accesses Explore all metrics

Abstract

Hybrid algorithms, which combine black-box machine learning methods with experience from traditional numerical methods and domain expertise from diverse application areas, are progressively gaining importance in scientific machine learning and various industrial domains, especially in computational science and engineering. In the present survey, several promising avenues of research will be examined which focus on the combination of machine learning (ML) and domain decomposition methods (DDMs). The aim of this survey is to provide an overview of existing work within this field and to structure it into

A few take aways:

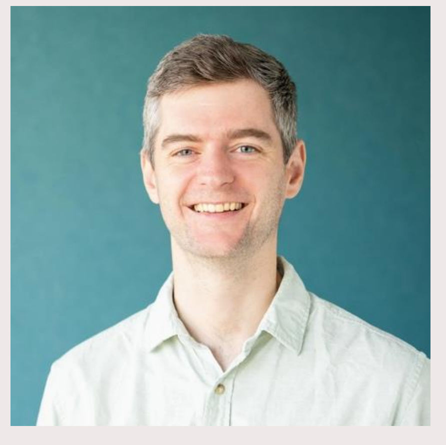
- Data parallelism is not enough
- Clever algorithms require some change of architecture

We use a partition of unity (PoU) to piece together different local representations as well as a global representation for the large- scale components of the solution. This strategy has proven to be vital in practice in order to achieve good accuracy. (Chi, Chen, E, Yang, *2022)*

- POU networks **are akin to divide and conquer approaches** (like domain decomposition) -> DD+ML is also now a research field!

- Old HPC and DD wisdom says that **mere splitting won't guarantee scalability** (e.g. w.r.t frequency) or robustness (in multiscale contexts) -> **we need more levels**…

Acknowledgements



Ben Moseley (Imperial College London):
https://benmoseley.blog

GitHub: https://github.com/benmoseley/

YouTube channel (course on Deep Learning for Scientific Computing)
https://www.youtube.com/@CAMLabETHZurich