SciML - Physics Informed ML/NN

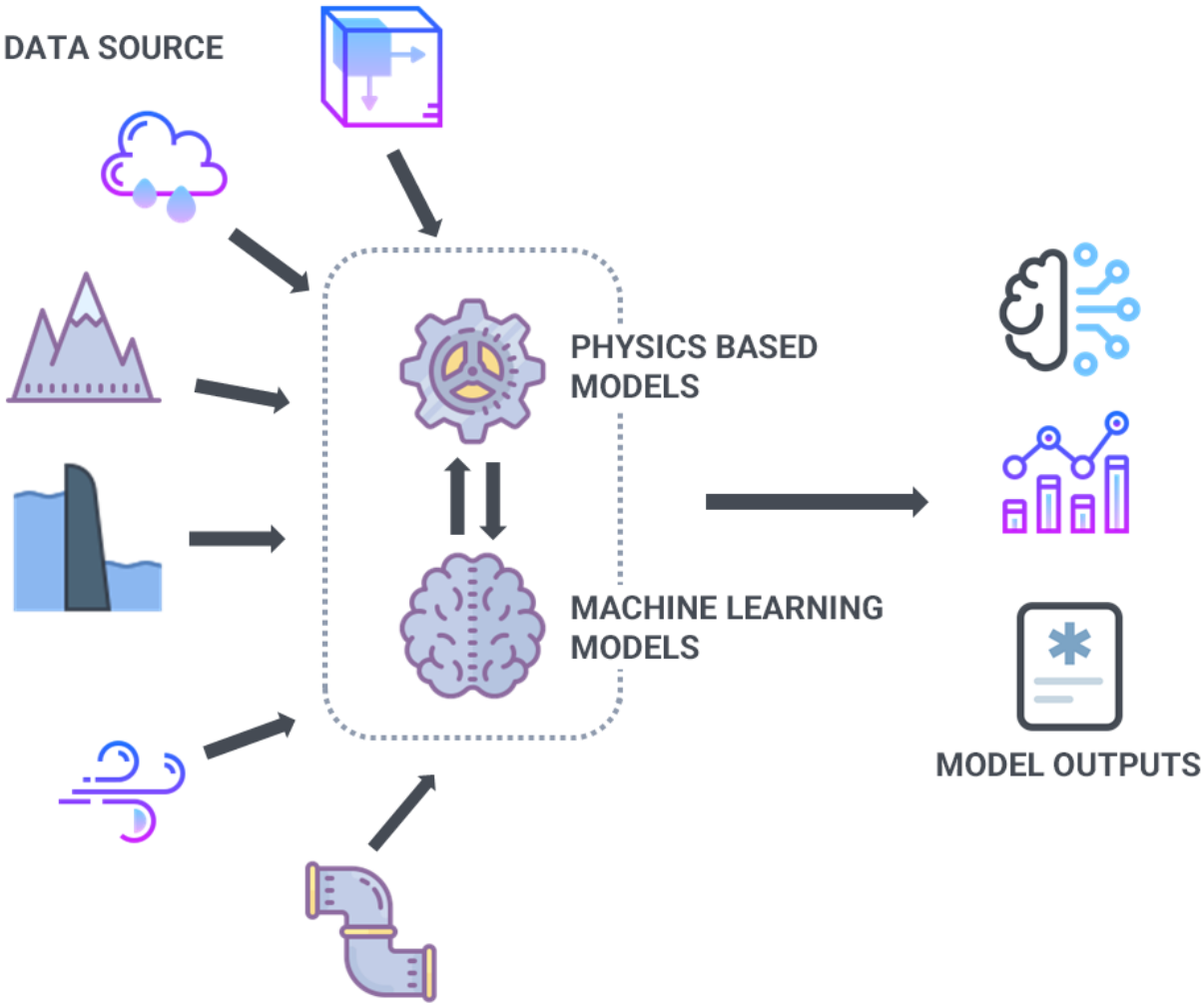
Mark Asch - IMU/VLP/CSU

2023

Program

1. Automatic differentiation for scientific machine learning:
   (a) Differentiable programming with autograd and PyTorch.
   (b) Gradients, adjoints, backpropagation.
   (c) Adjoints and inverse problems.
   (d) Neural networks for scientific machine learning.
   (e) Physics-informed neural networks.
   (f) The use of automatic differentiation in scientific machine learning.
   (g) The challenges of applying automatic differentiation to scientific applications.

Recall: What is SciML?



Recall: Differentiable programming

- Differential programming is a technique for automatically computing the derivatives of functions.

- This can be done using a variety of techniques, including:

  ⇒ **Symbolic differentiation**
  ⇒ **Numerical differentiation**
  ⇒ **Automatic differentiation**: This is a technique that combines symbolic and numerical differentiation to automatically compute the derivatives of functions. This is the most powerful technique for differential programming, and it is the most commonly used technique in scientific machine learning.

- The mathematical theory of differential programming is based on the concept of gradients.

- Differential programming can be used to solve a variety of problems in scientific machine learning, including:

⇒ Calculating the gradients of loss functions for machine learning models.
⇒ Solving differential equations.
⇒ Performing optimization.
⇒ Solving **inverse and data assimilation problems.**

Recall: Automatic Differentiation

- Automatic differentiation is an umbrella term for a variety of techniques for efficiently computing accurate derivatives of more or less general programs.
- Many algorithms in machine learning, computer vision, physical simulation, and other fields require the calculation of gradients and other derivatives.
- Practitioners across many fields have built a wide set of automatic differentiation tools, using different programming languages, computational primitives, and intermediate compiler representations.
- AD can be readily and extensively used and is thus applicable to many industrial and practical Digital Twin contexts [9].

AD for SciML

- Recent progress in machine learning (ML) technology has been spectacular.
- At the heart of these advances is the ability to obtain high-quality solutions to non-convex optimization problems for functions with billions—or even hundreds of billions—of parameters.
- Incredible opportunity for progress in classical applied mathematics problems.

Automatic Differentiation—backprop, autograd, etc.

- Backprop is a special case of autodiff.
- Autograd is a particular autodiff package.
- In practice, we will pricipally use PyTorch's autodiff functions.

Remark 1. Autodiff is NOT finite differences, nor symbolic differentiation. Finite differences are too expensive (one forward pass for each discrete point). They induce huge numerical errors (truncation/approximation and roundoff) and are very unstable in the presence of noise.

Remark 2. Autodiff is both efficient—linear in the cost of computing the value—and numerically stable.

Remark 3. The goal of autodiff is not a formula, but a procedure for computing derivatives.

Tools for AD

- New opportunities that exist because of the widespread, open-source deployment of effective software tools for automatic differentiation.

- Efficient software frameworks that natively run on hardware accelerators (GPUs).

- These frameworks inspired high-quality software libraries such as

  $\Rightarrow$ JAX,
  $\Rightarrow$ PyTorch,
  $\Rightarrow$ TensorFlow.

- The technology's key feature is: **the computational cost of computing derivatives of a target loss function is independent of the number of parameters;**

$\Rightarrow$ this trait makes it possible for users to implement gradient-based optimization algorithms for functions with staggering numbers of parameters.

AD Sayings

"Gradient descent can write code better than you, I'm sorry."

"Yes, you should understand backprop."

"I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved."

- Andrej Karpathy [~2017] (Tesla AI, OpenAI)

Why use ML/Neural Networks for SciML?

- Excellent, open-source tools and frameworks

  $\Rightarrow$ Autodiff
  $\Rightarrow$ PyTorch
  $\Rightarrow$ many, many others...

- Universal approximation property (UAP for NNs)

- Curse of dimensionality

Recall: what is a NN?

- A Neural Network is a composition of nonlinear functions (see Basic Course)

$$\mathrm{NN}(x) = W_3\sigma_2\left(W_2\sigma_1\left(W_1x + b_1\right) + b_2\right) + b_3$$

  where we can add layers, and to each layer, add neurons.

- Training a NN: given observations $y = f(x)$ of some unknown function f, find the values of W that minimize the loss function expressing the mismatch between the predictions of $\mathrm{NN}(x)$ and the corresponding values of y.

  $\Rightarrow$ hence the NN is just a function approximator

Recall: why NNs?

- Neural Networks have two important properties:

  $\Rightarrow$ Universal Approximation property, which states that for a given accuracy ϵ, one can construct a large NN such that it can approximate any (reasonable) function f, of arbitrary complexity, within the tolerance ϵ.

  $\Rightarrow$ Avoidance of the Curse of Dimensionality. If we were to make a polynomial approximation with n coefficients in each of d dimensions, then the complexity of this approximator will exponential in d. However, the growth of a NN to sufficiently approximate a d-dimensional function, only grows as a polynomial in d.

Recall: Universal Approximation for Functions

Theorem 1 (Cybenko 1989). *If σ is any continuous sigmoidal function, then finite sums*

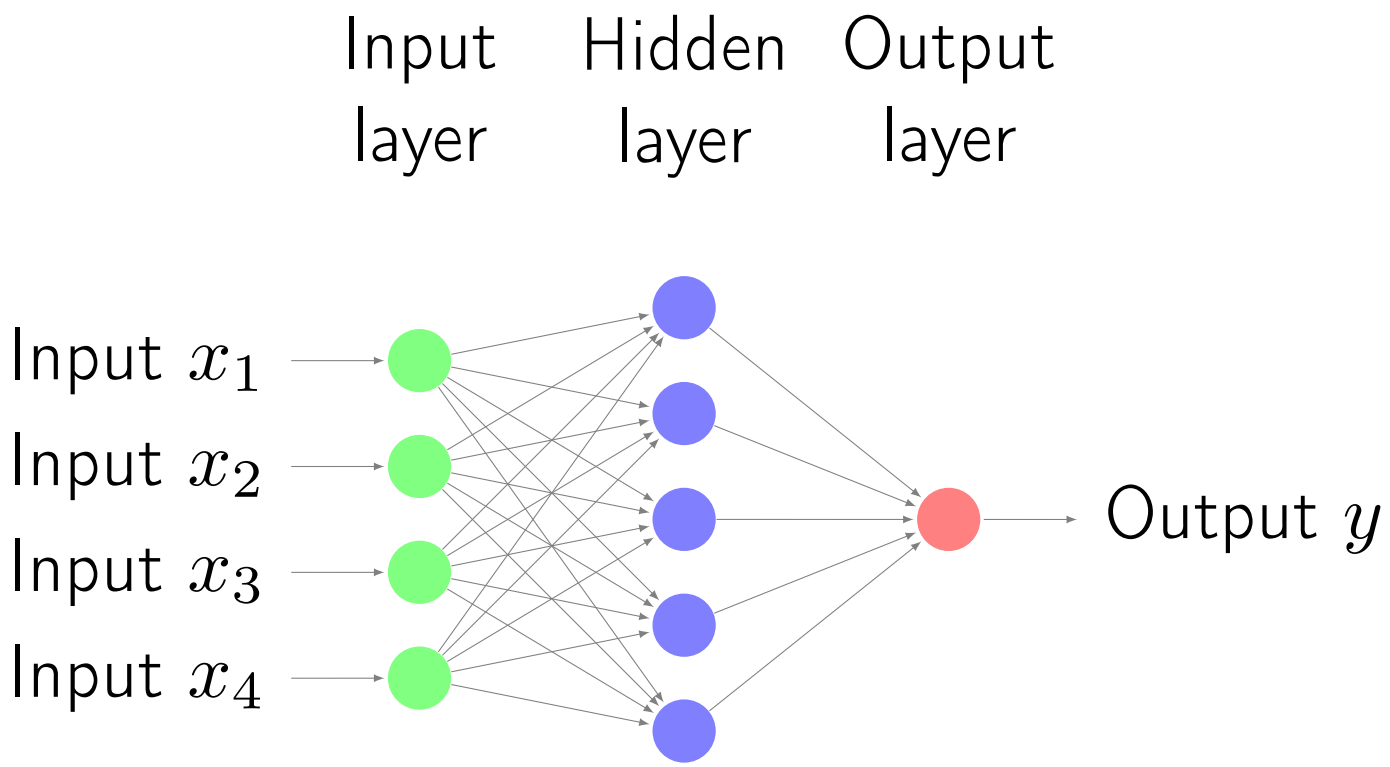
$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma \left(y_j \cdot x + \theta_j \right)$$

are dense in $C(I_d)$.

Theorem 2 (Pinkus 1999). *Let $\mathbf{m}_i \in \mathbb{Z}^d$, $i = 1, \ldots, s$, and set $m = \max_i \left| \mathbf{m}^i \right|$. Suppose that $\sigma \in C^m(\mathbb{R})$, not polynomial. Then the space of single hidden layer neural nets,*

$$\mathcal{M}(\sigma) = \operatorname{span} \left\{ \sigma(\mathbf{w} \cdot \mathbf{x} + b) \colon \mathbf{w} \in \mathbb{R}^d, \, b \in \mathbb{R} \right\},$$

is dense in $C^{\mathbf{m}^1, \ldots, \mathbf{m}^s}(\mathbb{R}^d) \doteq \cap_{i=1}^{s} C^{\mathbf{m}^i}(\mathbb{R}^d)$.


Input layer
Hidden layer
Output layer
Input x_1
Input x_2
Input x_3
Input x_4
Output y

From ML to SciML...

- In scientific machine learning, neural networks and machine learning are used as the basis to solve problems in CSE (Computational Science and Engineering)

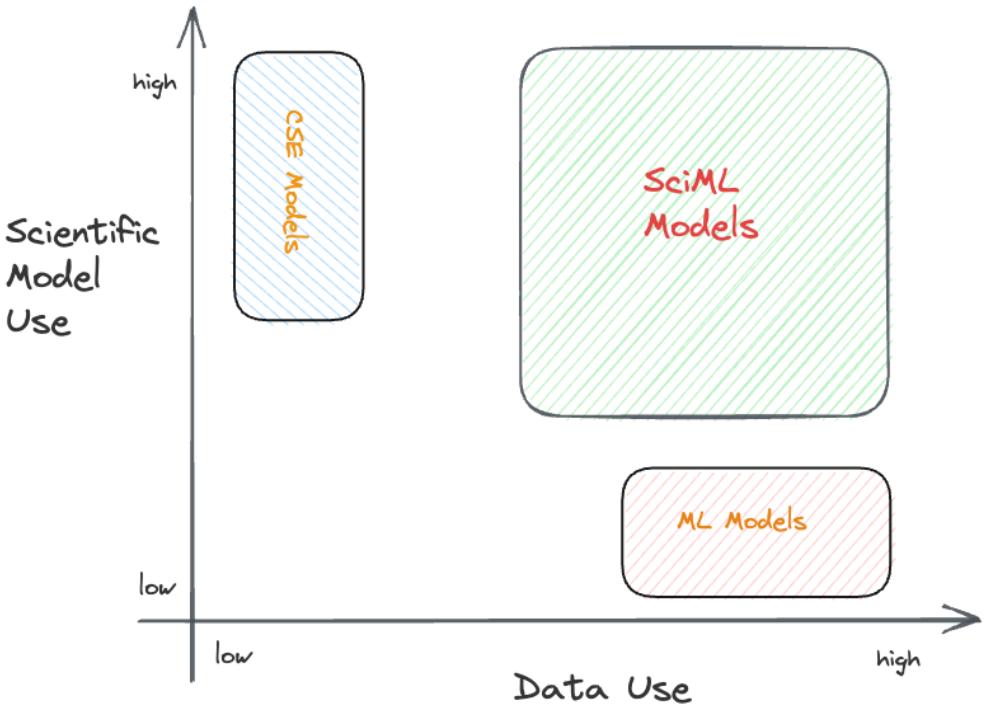
- CSE is, in majority, driven by systems of (P)DEs, since we are interested in how systems evolve/change.

  $\Rightarrow$ As a consequence, the use of ML for the solution of differential eqations is an important topic.

  $\Rightarrow$ As we will see, ML can be used in other ways too.

ML/NN Approaches

- Recall the "question of balance"



- Now, we are going,to decompose the SciML block into classes... (there are several ways of doing this)

- 3 major classes of approaches

  $\Rightarrow$ architecture-based

⇒ loss function
⇒ hybrid approaches

- 4 widely-used families of approaches:

  ⇒ SUMO - surrogate modeling
  ⇒ PCL - physics constrained learning
  ⇒ PINN - physics informed neural networks
  ⇒ DeepONet/PINO/FNO - neural operators

- Others:

  ⇒ Neural ODEs
  ⇒ Differentiable physics
  ⇒ Koopman theory

- An alternative classification:

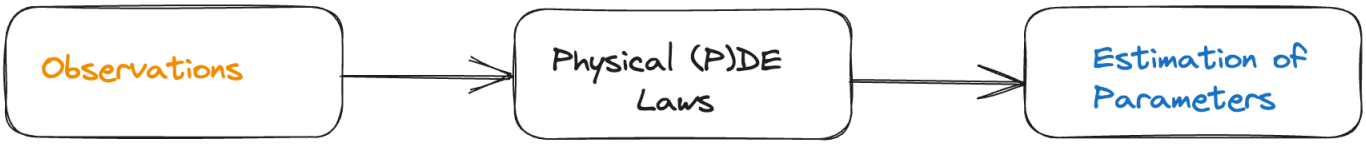
  ⇒ constrained (including PINN and PCL)
  ⇒ encoded (DL-type architectures)
  ⇒ operator-based (DeepONet, FNO)

- SUMO then falls into the "data-driven" category

Forward and Inverse Problems

Direct Problem:

Inverse Problem:



- Forward simulation is a major CSE task: predict the system's evolution, given some input conditions.

  $\Rightarrow$ Usually we solve a system of diffferential equations with some forcing and boundary conditions

  $\Rightarrow$ **Challenges**:

  $\rightarrow$ major challenge is the extremely high computational costs of simulating complex, multi-scale, multi-physics systems.

→ quantifying uncertainty for predictions/forecasts

⇒ SciML is helping to alleviate these issues by allowing the possibility to learn from previous simulations, providing more powerful computational shortcuts whilst having less impact on the simulation fidelity

- Inverse problems are tightly related to simulation and solving them is crucial for many real-world tasks.

  ⇒ Here the goal is to estimate a set of latent, hidden, or unobserved parameters of a system given a set of real-world observations of the system.

  ⇒ **Challenges**:

  → Inversion algorithms often require many forward simulations to be run in order to match the predictions of the physical model to the set of observations.

  → Given the potentially high computational costs of forward simulation stated above, this can render many IP applications unfeasible.

  → Inversion problems generally suffer from ill-posedness.

  → In such cases, sophisticated regularization schemes are required to restrict the space of possible latent

parameters the inversion algorithm can explore.

→ Finally, real-world inversion usually suffers from noise/uncertainty. This is challenging to quantify and will increase the ill-posedness of the problem.

→ Often a fully probabilistic framework is required to model such processes.

EQUATION DISCOVERY

Equation Discovery

- There are many contexts where we do not fully understand the system itself.
- We are unsure how to define the model $\mathcal{F}$—this is a type of inverse problem.
- Being able to learn about a system, for example by discovering its governing equations, is powerful as it can provide a general model of the system.
- SciML is aiding this discovery by allowing us to automate the process and/or learn about complex processes which are hard to intuit

Equation Discovery: theory

- Suppose we have a dynamic system

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t)), \tag{1}$$

  where

  $\Rightarrow$ vector $\mathbf{x}(t)$ denotes the state of a system at time t,
  $\Rightarrow$ the function $\mathbf{f}$ represents the dynamic constraints that define the equations of motion of the system, such as Newton's second law.
  $\Rightarrow$ The dynamics can be generalized to include parameterization, time dependence, and forcing.

- **Key observation**: for many systems of interest, the function $\mathbf{f}$ consists of only a few terms, making it sparse in the space of possible functions.

- To determine the function $\mathbf{f}$ from data,

$\Rightarrow$ we collect a time history of the state $\mathbf{x}(t)$

$\Rightarrow$ and either measure the derivative $\dot{\mathbf{x}}(t)$ or approximate it numerically from $\mathbf{x}(t)$

$\Rightarrow$ The data are sampled at several times $t_1, \ldots, t_m$ and arranged into two matrices:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^{\mathrm{T}}(t_1) \\ \mathbf{x}^{\mathrm{T}}(t_2) \\ \vdots \\ \mathbf{x}^{\mathrm{T}}(t_m) \end{bmatrix} = \begin{bmatrix} & \longrightarrow \text{state} \longrightarrow & & \\ x_1(t_1) & x_2(t_1) & \cdots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \cdots & x_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \cdots & x_n(t_m) \end{bmatrix} \begin{matrix} \\ \downarrow \\ \mathrm{t} \\ \downarrow \end{matrix}$$

and similarly for $\dot{\mathbf{X}}$

- Next, we construct a library $\Theta(\mathbf{X})$ made up of candidate nonlinear functions of the columns of $\mathbf{X}$

  $\Rightarrow$ For example, $\Theta(\mathbf{X})$ may consist of constant, polynomial, and trigonometric terms:

$$\Theta(\mathbf{X}) = \begin{bmatrix} | & | & | & | & & | & | & \\ 1 & \mathbf{X} & \mathbf{X}^{P_2} & \mathbf{X}^{P_3} & \cdots & \sin(\mathbf{X}) & \cos(\mathbf{X}) & \cdots \\ | & | & | & | & & | & | & \end{bmatrix}.$$

- Each column of $\Theta(\mathbf{X})$ represents a candidate function for the right-hand side of Eq. 1.

- There is tremendous freedom in choosing the entries in this matrix of nonlinearities, because we believe that only a few of these nonlinearities are active in each row of $\mathbf{f}$

- We set up a sparse regression problem to determine the sparse vectors of coefficients

$$\Xi = \begin{bmatrix} \boldsymbol{\xi}_1 & \boldsymbol{\xi}_2 & \cdots & \boldsymbol{\xi}_n \end{bmatrix}$$

that determine which nonlinearities are active,

$$\dot{\mathbf{X}} = \Theta(\mathbf{X})\Xi$$

- Each column $\boldsymbol{\xi}_k$ of Ξ is a sparse vector of coefficients determining which terms are active in the right-hand side for one of the row equations

$$\dot{\mathbf{x}}_k = \mathbf{f}_k(\mathbf{x})$$

in Eq. 1.

- Once Ξ has been determined, a model of each row of the governing equations may be constructed as follows,

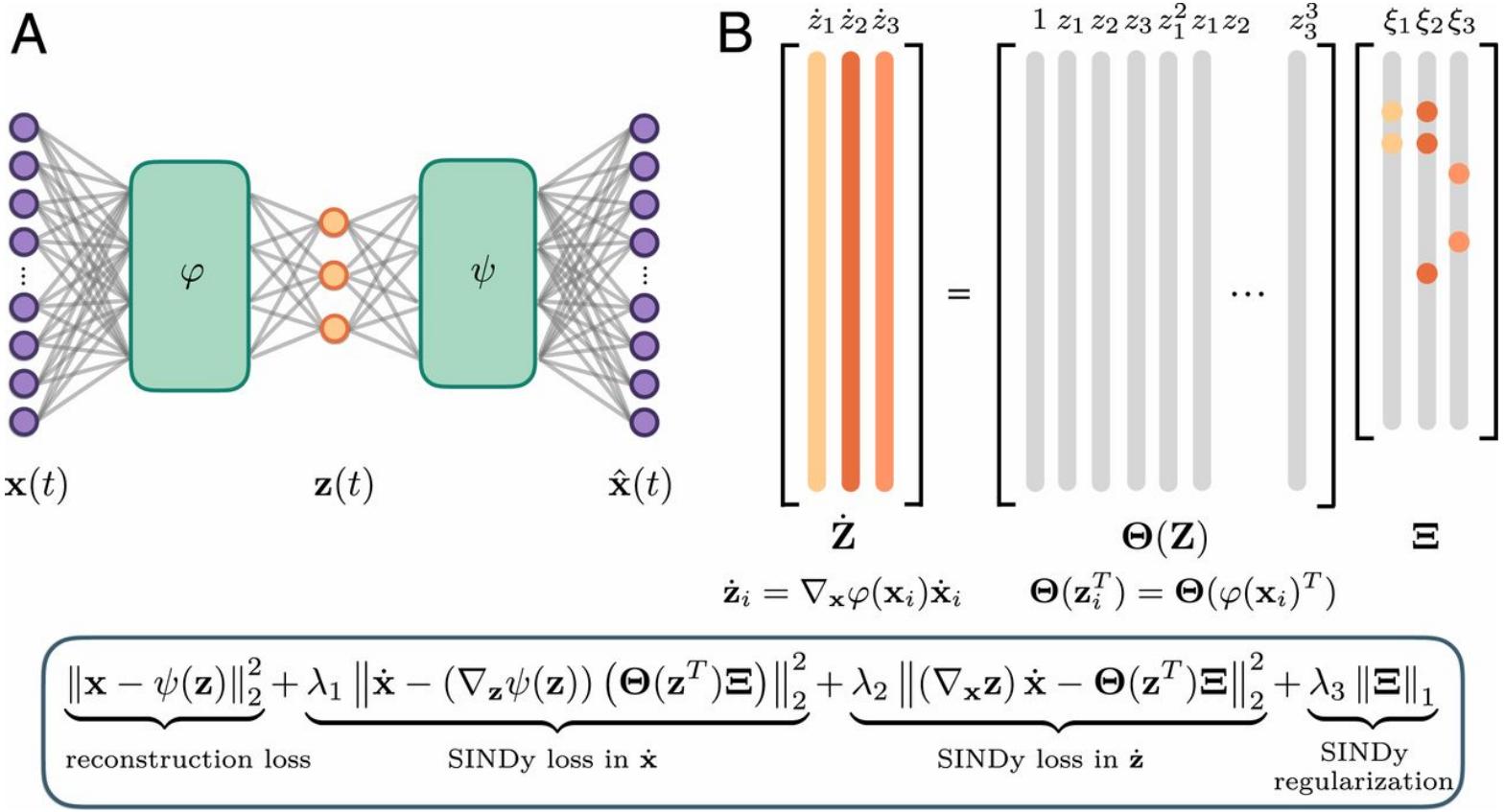
$$\dot{\mathbf{x}}_k = \mathbf{f}_k(\mathbf{x}) = \Theta(\mathbf{x}^{\mathrm{T}})\boldsymbol{\xi}_k$$

- Note that $\Theta(\mathbf{x}^{\mathrm{T}})$ is a vector of symbolic functions of elements of $\mathbf{x}$, as opposed to $\Theta(\mathbf{X})$, which is a data matrix.

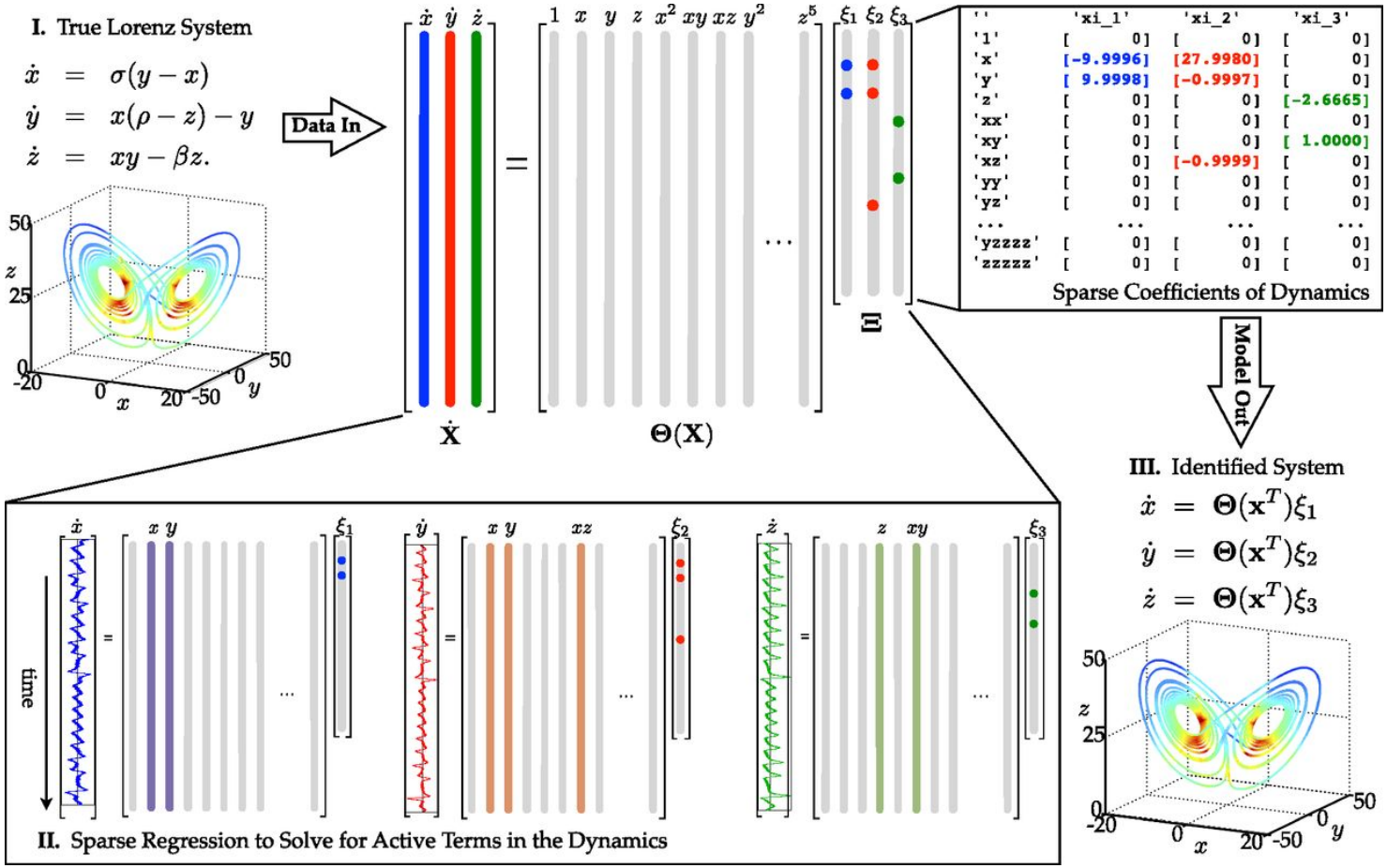
- Thus,

$$\dot{\mathbf{x}}_k = \mathbf{f}_k(\mathbf{x}) = \Xi^{\mathrm{T}} \left(\Theta(\mathbf{x}^{\mathrm{T}})\right)^{\mathrm{T}}$$

SINDy Scheme



SINDy Application



Equation Discovery: SINDy references

- Code:
  https://faculty.washington.edu/kutz/page26/
- Paper:
  https://www.pnas.org/doi/10.1073/pnas.1906995116

ARCHITECTURE BASED METHODS

Architecture-based SciML

- **Idea**: change the architecture used in the ML algorithm so that it incorporates scientific constraints.
  - ⇒ we open up the black box's design and change parts of it so that it obeys these constraints.
  - ⇒ Incorporating scientific principles in this way can restrict the range of models the algorithm can learn, and result in more generalisable and interpretable models.
  - ⇒ From a machine learning perspective, we are introducing a strong inductive bias into the model.
  - ⇒ These aspects will be more fully discussed in later lectures on Bias and Ethics of ML.

- Approaches
  - ⇒ encode certain physical variables, for example using an LSTM for intermediate variables
  - ⇒ encode symmetries, such as translational and rotational invariance—this can be easily achieved with convolutional neural networks (CNNs)

$\Rightarrow$ use Koopman theory [Brunton, Kutz]

$\Rightarrow$ use physically constrained Gaussian processes

SURROGATE MODELING

Surrogate Modelling - SUMO

- Surrogate modeling is a technique used to approximate a complex and expensive-to-evaluate function with a simpler and cheaper-to-evaluate function.
- The surrogate model, also known as a metamodel or emulator, is trained on a set of input-output data from the complex function.
- Once trained, the surrogate model can be used to predict the output of the complex function for any input value, without having to evaluate the complex function directly.

Surrogate Modelling - formulation

- The mathematical presentation of surrogate modeling depends on the specific type of surrogate model being used.

- However, there is a general framework that can be applied to most surrogate models.

  $\Rightarrow$ Let $f(x)$ be the complex function that we want to approximate, and let $s(x)$ be the surrogate model.

- The goal of surrogate modeling is to find a surrogate model$s(x)$that is as close to the complex function $f(x)$ as possible, given a limited amount of training data.

- One way to measure the similarity between the complex function and the surrogate model is to use the mean squared error (MSE):

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - s(x_i))^2$$

where N is the number of training data points, and x_i and $f(x_i)$ are the i-th input-output pair in the training dataset.

- The surrogate model can be trained using a variety of machine learning algorithms, such as kriging, radial basis functions (RBFs), support vector machines (SVMs), and artificial neural networks (ANNs).

Surrogate Modelling - formulation II

- Given a set of training data points

$$(x_1, y_1), (x_2, y_2), ..., (x_n, y_n),$$

  where x_i is the input vector and y_i is the output value, the goal of surrogate modeling is to construct a function $\hat{y}(x)$ that approximates the true function $y(x)$ as accurately as possible.

- The most common approach to surrogate modeling is to use a statistical model, such as a polynomial response surface, kriging, or support vector machine. These models can be trained on the training data points to learn the relationship between the inputs and outputs.

- Once the surrogate model is trained, it can be used to predict the output value for any input value x as follows:

$$\hat{y}(x) = \text{Model}(x)$$

where $\mathrm{Model}(x)$ is the output of the surrogate model.

Surrogate Modelling - examples

Surrogate modeling is used in a wide variety of applications, including:

- Engineering design: Surrogate models can be used to accelerate the design process by providing cheaper and faster predictions of the performance of different design alternatives.
- Scientific computing: Surrogate models can be used to reduce the computational cost of simulating complex physical systems.
- Financial modeling: Surrogate models can be used to predict the risk and return of financial investments.
- Machine learning: Surrogate models can be used to interpret the predictions of complex machine learning models.

Here are some specific examples of surrogate modeling in use:

- Aerospace engineering: Surrogate models are used to design aircraft wings, engines, and other components.
- Automotive engineering: Surrogate models are used to design car engines, suspensions, and other components.
- Chemical engineering: Surrogate models are used to design chemical reactors, pipelines, and other equipment.
- Oil and gas exploration: Surrogate models are used to predict the flow of oil and gas through reservoirs.
- Pharmaceutical development: Surrogate models are used to predict the properties of drugs and to design clinical trials.

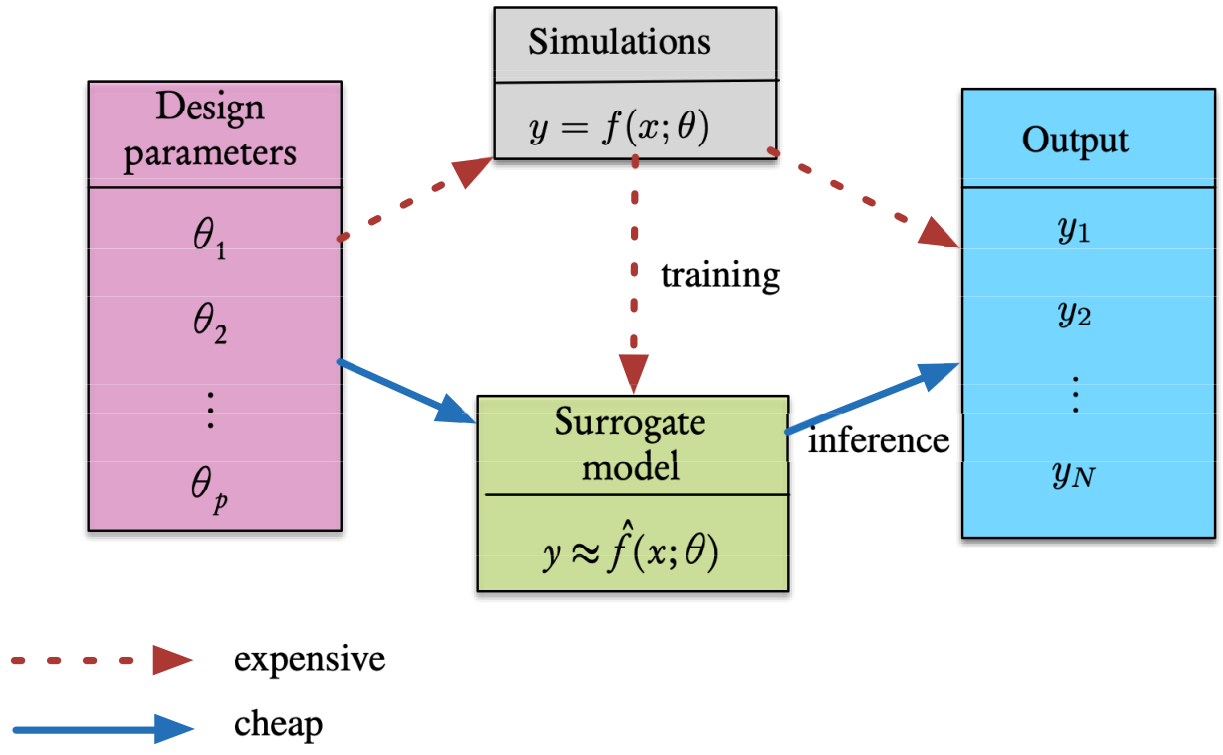
Conclusion

Surrogate modeling is a powerful technique for approximating complex and expensive-to-evaluate functions. It is

used in a wide variety of applications, including engineering design, scientific computing, financial modeling, and machine learning.

Surrogate Modelling - in practice

Definition 1. Surrogate models, also known as response surfaces, black-box models, metamodels, or emulators, are simplified approximations of more complex, higher order models. These models are used to map input-data to output-data, when the actual relationship between the two is unknown or computationally too expensive to evaluate.



- **Idea**: a ML-trained model can subsitute all or part of a

system of (P)DEs

$\Rightarrow$ learn the complete, unknown input-output relation (but no physics constraints)

$\Rightarrow$ learn some sub-parametrization that is too complicated to capture by classical methods

The surrogate modeling process, as depicted in the Figure, consists of five stages:

1. Choice of the design parameters.

2. Generation of input-output pairs of data for training, by simulations of the physics-based model, or from experiments with varying parameter values. This is the most expensive step.

3. Choice of the surrogate model, based on a suitable supervised machine learning method.

4. Training of the machine learning model, using the training data. This can be an expensive step, but is usually only performed once.

5. Finally, use of the trained surrogate model to perform computationally cheap inference for new values of the design parameters, thus permitting

   (a) an exhaustive search of the parameter space,
   (b) an optimal parameter design, and/or
   (c) a quantification of design uncertainties.

- **Questions**:

  $\Rightarrow$ But, can such a surrogate faithfully capture the complex, nonlinear relationships between input and output?
  $\Rightarrow$ And, if so, how can the surrogate do this?

- These two very important questions are fundamental for any underlying system or process that we would like to study using surrogate modeling.

  $\Rightarrow$ The answer to the first question is: "Yes, in theory," thanks to the universal approximation property of very simple machine learning models—fully-connected, feed-forward neural networks (FCNN) [Cybenko, Pinkus].

$\Rightarrow$ And the answer to the second question is: "Yes, in practice," with the aid of a large variety of supervised learning techniques, of which FCNNs are just one special case.

Tecniques for SUMO

- Both supervised and unsupervised[1] learning techniques can be used for SUMO.

- Four of these techniques are recommended, since they are robust and perform well:

  $\Rightarrow$ random forests and multi-layer perceptrons (FCNN) in the case of regression, and

  $\Rightarrow$ support vector machines (SVM) and k-means clustering in the case of classification.

- There is no single, perfect, globally applicable method that will always do the best job.

- Usually, one should try a few, and then settle for the one or two that are the simplest, but provide adequate precision and especially robustness in the face of the inherent uncertainties in the underlying processes—see also the Ethics and Bias Lectures.

[1] Adversarial and self-supervised are also possible, but far more complicated to implement.

SUMO Principles

- The principle behind the SUMO approach is the following:

  $\Rightarrow$ if a multiscale, multiphysics relationship between design parameters and output performance can be learned from data, then we can forgo—at least to some extent—the underlying ODE, PDE and population dynamics models, as well as time-consuming, expensive physical/clinical experiments and trials.

  $\Rightarrow$ Moreover, once this relationship has been learned, its use—the so-called inference phase—is very inexpensive and one can then envisage the solution of optimization and uncertainty quantification problems

  $\Rightarrow$ We are in fact, constructing a digital twin [Asch2022].

- The approach proposed here is not to seek a complex machine learning model, but to favor simpler models that are easier to interpret, trust and deploy—see Ethics and Bias Lecture.

- These low-complexity models will not suffer from brittleness and will preserve a good bias-variance trade-off. They will also have more favorable interpretabilty, ethics and bias properties.

SUMO - Training Data

- To learn a data-driven model, we need training data.
- This data is obtained from experimental observations, or model-based simulations, or some combination of the two.

  $\Rightarrow$ Here we will often, if possible, use model-based simulations, calculated for example by a SIR model.

- For any supervised machine learning method, we first need to carefully select and define response variables—or functions—that is an unknown function of the input parameters.
- When defining the output parameters for the machine learning, extreme care needs to be taken to extract reliable information describing adequately the phenomenon that we seek to explore, analyze and forecast

- Often, sampling techniques must be used to ensure a space-filling parameter range with good projection properties.

- The LHS method is a generalization to higher dimensions of the Latin square which is an $n \times n$ array filled with n different symbols, each occurring exactly once in each row and exactly once in each column.

  $\Rightarrow$ Assuming a three dimensional parameter space and n_s the number of samples of each parameter, then each sample is the only one in each axis-aligned hyperplane containing it.

  $\Rightarrow$ On the other hand, building an LHS design with the best maximin criterion on all projections provides a space filling design in the whole space and on projections.

- Hence, we obtain a training set that is well-balanced, containing a wide range of behaviors, which in turn will ensure the best possible training for the machine learning models and then, good predictions.

SUMO - EDA

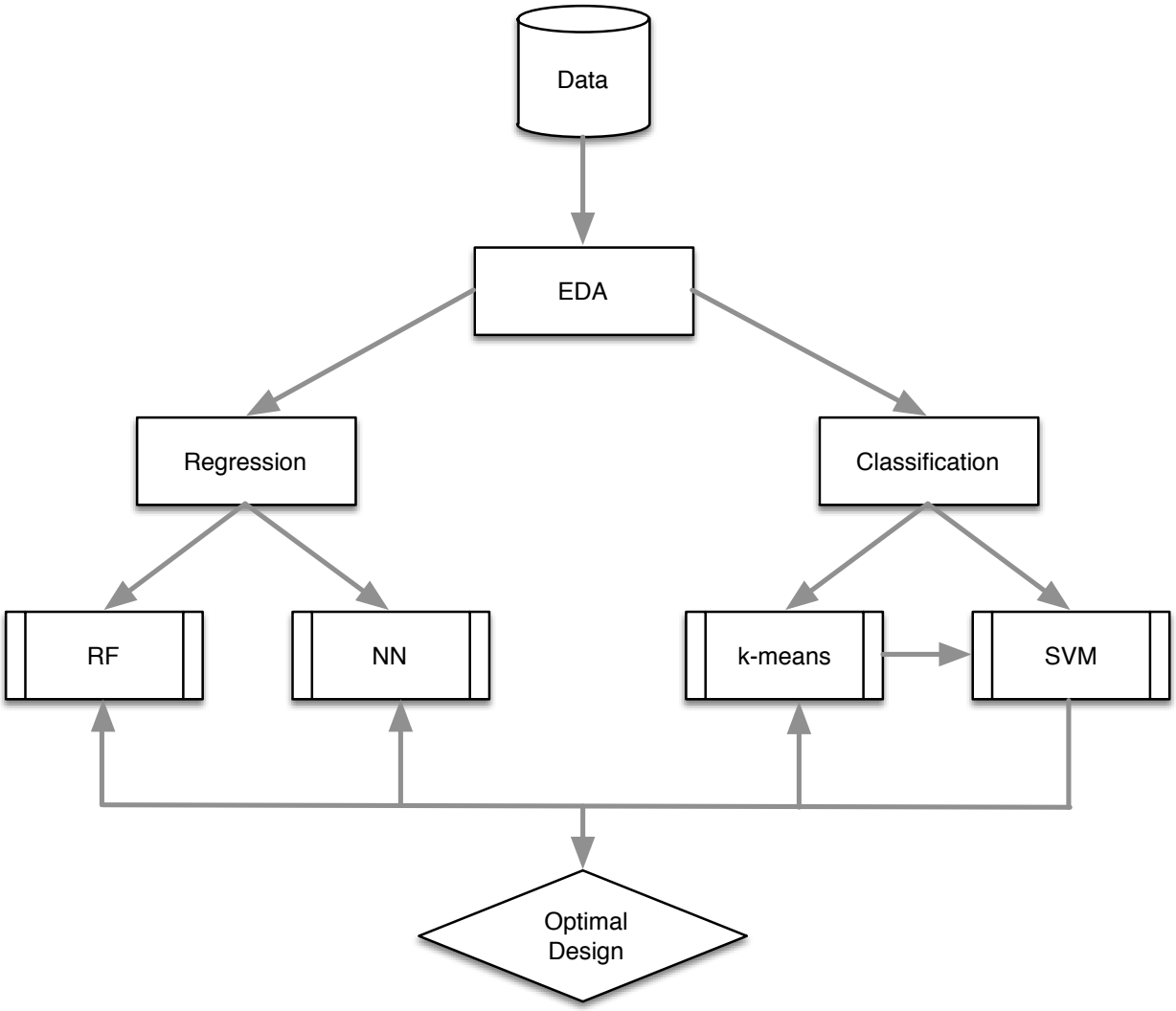
- In order to identify useful preliminary information about the data and investigate the relationships between the features and the response variables, an exploratory data analysis—see Basic Course Lectures—should always be performed, *before* attempting any surrogate modeling.

- This study can help to detect the interactions among different variables indiffering contexts, helping us to understand the importance of their effect on the desired performance of the system.

- The follwong techniques of EDA should, initially, be applied (many more are psssible):

  $\Rightarrow$ Summary statistics.
  $\Rightarrow$ Scatter plots.
  $\Rightarrow$ Correlation and partial correlation tables that ensure the non-existence of nuisance information in the database due to a confounding variable.

SUMO - Concrete Workflow Example



- The approach described above, proposes a universal workflow for data analysis and surrogate modeling in the light of optimal design choices.

- This workflow, as illustrated in the Figure, can be applied to almost *any* design/optimization problem.

  $\Rightarrow$ It suffices to replace our data by the reader's data,
  $\Rightarrow$ our underlying model by the reader's model,
  $\Rightarrow$ and then simply follow the steps of the workflow.

- Once the data is collected, exploratory data analysis (EDA) is essential for:

  1. Familiarization with the data and choice of response variables.
  2. Elimination of any unusual, or erroneous data points.
  3. Preliminary identification of the most influential features, or parameters, and the relations—or lack of relations—between the features themselves, and between the features and the response variables.
  4. Reduction of the complexity by identification of colinear variables.

- The next steps are regression *and* classification.

- Note that classification, especially unsupervised, can be considered as being a part of EDA, since it can help us in determining response variables.

- In our proposed workflow, we have purposefully selected simple approaches because of their

  $\Rightarrow$ broad applicability,
  $\Rightarrow$ ease of computation and
  $\Rightarrow$ facility of interpretation—no black boxes here.

- We highly recommend, for regression:

  1. Random forests, because of their established robustness and their capacity to rank explanatory variables by their importance.
  2. Neural networks, of FCNN type, for their extreme versatility and their universal approximation properties.

- We highly recommend, for classification:

  1. k-means for initial unsupervised clustering and identification of groups of properties.
  2. SVM for refined, supervised clustering that provides a surrogate model.

- In the final step, which of course will be context-dependent, we can exploit all the surrogate models

found above for optimal design and process planning.

- Once we have a surrogate model, or several surrogate models, at our disposition, we can address a number of outer-loop problems [Asch2022], such as:

  1. Optimization problems, where we seek to maximize, or minimize some critical response variables.
  2. Uncertainty quantification problems, where we seek some kind of confidence interval around the parameter values, given an estimation of the inherent material or process variabilities.
  3. Bayesian optimization problems, that combine the above two.

- All of these require a large number of simulations, or large volume of experimental data, that can now be completely replaced by the surrogate model.

  $\Rightarrow$ Recall that the evaluation of a surrogate model, whose training has already been done in an offline computation, can be done in quasi-real time.

- For the surrogate models themselves, other ML techniques can be envisaged that could provide further in-

sight and better predictive power. Among these, we can mention:

⇒ SVM regression that is well-adapted to highly nonlinear relationships.
⇒ Functional data analysis (FDA) that is particularly well-suited to time series data.
⇒ Further exploration of more sophisticated neural network architectures. This is particularly indicated in the presence of multi-physics, or multi-modal data.

- The key findings of the approach presented here can be summarized as follows.

⇒ Firstly, exploratory data analysis, including correlations, partial correlations and unsupervised classification by k-means, leads to the judicious choice of a few design parameters, and response variables.
⇒ Then, supervised classification and regression methods, used together in synergy, reveal the optimal parameter choices and propose hitherto unforeseen operating regimes.
⇒ We can identify the most influential design parameters and we can characterize the ranges of these

parameters and identify optimal clusterings of these.

PHYSICS CONSTRAINED LEARNING

Physics Constrained Learning - PCL

- **Idea**: use a NN as part of the (P)DE

- Given a physical relation

$$F(u; \boldsymbol{\theta}) = 0 \tag{2}$$

represented by an IBVP, or other functional relationship, with

$\Rightarrow$ u the physical quantity
$\Rightarrow$ $\boldsymbol{\theta}$ the (material/medium) properties/parameters

- Inverse Problem is defined as:

$\Rightarrow$ **Given** observations/measurements of u at the locations $\mathbf{x} = \{x_i\}$

$$\mathbf{u}^{\text{obs}} = \{u(x_i)\}_{i \in \mathcal{I}}$$

⇒ **Estimate** the parameters $\boldsymbol{\theta}$ by minimizing a loss/objective/cost function

$$L(\boldsymbol{\theta}) = \left\| u(\mathbf{x}) - \mathbf{u}^{\text{obs}} \right\|_2^2$$

subject to (2).

PCL - use of a NN

- If $\boldsymbol{\theta} = \boldsymbol{\theta}(x)$, model it by a NN

$$\boldsymbol{\theta}(x) \approx \mathrm{NN}(x)$$

- Express the numerical scheme, such as RK, finte differences, finite, elements, etc., for approximating the (P)DE (2) as a computational graph $G(\boldsymbol{\theta})$.

- Use reverse-mode AD (aka. backpropagation) to compute the gradient of L with respect to $\boldsymbol{\theta}$ and the NN coefficients (weights and biases).

- Minimize by a suitable gradient algorithm

  $\Rightarrow$ Adam, SGD (1st order)
  $\Rightarrow$ L-BFGS (quasi-Newton)
  $\Rightarrow$ trust-region (2nd order)

PCL - Recall: use of AD

- Optimization problem: $\min_\theta L(u)$ subject to $F(\theta, u) = 0$.
- Suppose we have a computational graph for $u = G(\theta)$.
- Then $\tilde{L}(\theta) = L\left(G(\theta)\right)$ and by the IFT we can compute the gradient with respect to θ,

  $\Rightarrow$ first of F,

$$\frac{\partial F}{\partial \theta} + \frac{\partial F}{\partial u}\frac{\partial G}{\partial \theta} = 0, \quad \Rightarrow \quad \frac{\partial G}{\partial \theta} = -\left[\frac{\partial F}{\partial u}\right]^{-1}\frac{\partial F}{\partial \theta}$$

  $\Rightarrow$ then of $\tilde{L}$, by the chain rule,

$$\frac{\partial \tilde{L}}{\partial \theta} = \frac{\partial L}{\partial u}\frac{\partial G}{\partial \theta} = -\frac{\partial L}{\partial u}\left[\frac{\partial F}{\partial u}\right]^{-1}\frac{\partial F}{\partial \theta}$$

- The first derivative is obtained directly from the loss function, the second and third by reverse-mode AD

PCL - applications

- PCL has been successfully applied to numerous academic and real-life problems.

  $\Rightarrow$ Geomechanics, solid mechanics, fluid dynamics, seismic inversion

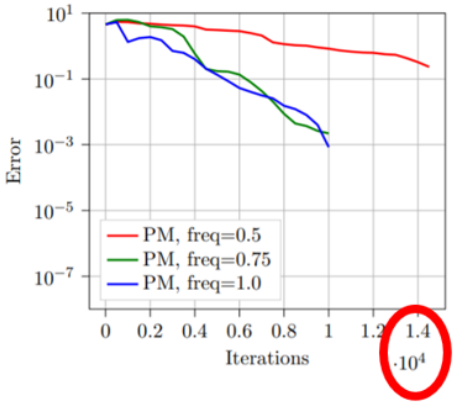
  $\Rightarrow$ General approach to inverse modeling: ADCME `https://kailaix.github.io/ADCME.jl/latest/` [Xu, Darve]

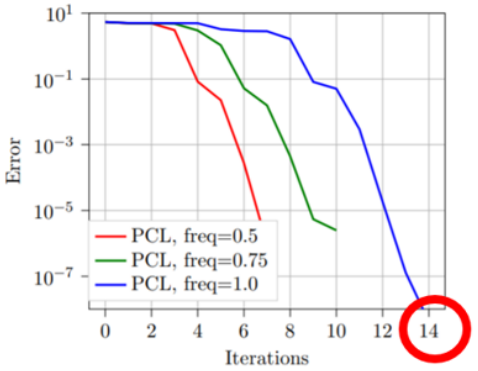
PCL - applications to stiff (P)DEs

Parameter Inverse Problem

$$\Delta u + k^2 g(x) u = 0$$
$$g(x) = 5x^2 + 2y^2$$

$$g_\theta(x) = \theta_1 x^2 + \theta_2 y^2 + \theta_3 xy + \theta_4 x + \theta_5 y + \theta_6$$

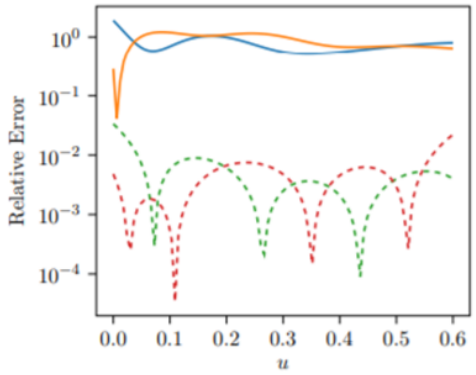


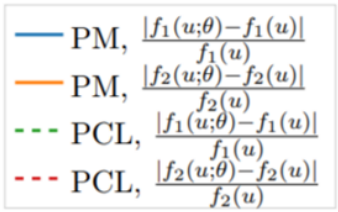


Approximate Unknown Functions using DNNs

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

$$\boldsymbol{f}(u) = \begin{bmatrix} NN\,(u;\theta_1) & 0 \\ 0 & NN(u;\theta_2) \end{bmatrix}$$





PM = Penalty Method — [Xu, Darve]

PHYSICS INSPIRED LEARNING

Physics Inspired Neural Networks - PINN background

- **Idea**: put the (P)DE into the ML algorithm, via the cost function (among others)

- Please review the Optimization Lecture for

  $\Rightarrow$ unconstrained optimization with penalization/regularization
  $\Rightarrow$ constraned optimization with Lagrange Multipliers

- Physics-informed neural network (PINN) models. [Gholami, et al. NeurIPS, 2021]—see also the Introductory Lecture.

  $\Rightarrow$ The typical approach is to incorporate physical domain knowledge as soft constraints on an empirical loss function and use existing machine learning methodologies to train the model.
  $\Rightarrow$ It can be shown that, while existing PINN methodologies can learn good models for relatively trivial problems, they can easily fail to learn relevant physical phenomena even for simple PDEs.

- → analyze several distinct situations of widespread physical interest, including learning differential equations with convection, reaction, and diffusion operators.
- → provide evidence that the soft regularization in PINNs, which involves differential operators, can introduce a number of subtle problems, including making the problem ill-conditioned.

⇒ Importantly, it can be shown that these possible failure modes are not due to the lack of expressivity in the NN architecture, but that the PINN's setup makes the loss landscape very hard to optimize.

⇒ Two promising solutions to address these failure modes.

- → The first approach is to use curriculum regularization, where the PINN's loss term starts from a simple PDE regularization, and becomes progressively more complex as the NN gets trained.
- → The second approach is to pose the problem as a sequence-to-sequence learning task, rather than learning to predict the entire space-time at once.
- → And there are many, many more "fixes" - this implies the necessity to treat each case with particular

attention, and not entertain the "magic wand" illusion...

PINN - solving (P)DEs with NNs

Definition 2. The process of solving a differential equation with a neural network, or using a differential equation as a regularizer in the loss function, is known as a physics-informed neural network (PINN), since this allows for physical equations to guide the training of the neural network in circumstances where data might be lacking.

- Idea: use the neural network to approximate the solution to the differential equation, while also satisfying any other physical constraints of the problem.

  $\Rightarrow$ For a scalar ODE, the neural network would have one input, which is the independent variable, and one output, which is the dependent variable. The neural network would be trained to minimize a loss function that includes both
  - $\rightarrow$ the error between the neural network's output and the known solution at some data points, and
  - $\rightarrow$ the error between the neural network's output and the differential equation itself.

⇒ For a PDE, the neural network would have multiple inputs, which would represent the independent variables in the PDE, and multiple outputs, which would represent the dependent variables in the PDE. The neural network would be trained to minimize a loss function that includes both the error between the neural network's output and the known solution at some data points, and the error between the neural network's output and the PDE.

⇒ PINNS can solve both direct and inverse problems.

⇒ **Warning**: As higher frequencies and more multi-scale features are added, more collocation points and a larger neural network with significantly more free parameters are typically required to accurately approximate the solution. This creates a significantly more complex optimization problem when training the PINN—see below for pros and cons.

SciML - hard vs. soft constraint

- Recall: there are two possible optimization strategies for constraining the NN (ML) to respect the physics

  1. Hard constraints
  2. Soft constraints

- Suppose we have a (P)DE of the form

$$\mathcal{F}(u(\mathrm{x},t)) = 0, \quad \mathrm{x} \in \Omega \subset \mathbb{R}^d, \quad t \in [0,T]\,,$$

  where

  $\Rightarrow$ $\mathcal{F}$ is a differential operator representing the (P)DE
  $\Rightarrow$ $u(\mathrm{x},t)$ is the state variable (i.e., quantity of interest), with $\mathrm{x},t)$ the space-time variables
  $\Rightarrow$ T is the time horizon and Ω is the spatial domain (empty for ODEs)
  $\Rightarrow$ initial and boundary conditions must be added for the problem to be well-posed

- Hard constraint: solve the contrained optimization problem

$$\min_{\theta} \mathcal{L}(u) \quad \text{s.t.} \quad \mathcal{F}(u) = 0,$$

where

$\Rightarrow$ $\mathcal{L}(u)$ is the data (mismatch) loss term

$\Rightarrow$ $\mathcal{F}$ is the constraint on the residual of the (P)DE under consideration

$\Rightarrow$ as was amply discussed in the DA/inverse problem context, this type of (P)DE-constrained optimization is usually quite difficult to code and to solve

- Soft constraint: solve the regularized/penalized uncontrained optimization problem

$$\min_{\theta} \mathcal{L}(u) + \alpha_{\mathcal{F}} \mathcal{F}(u), \tag{3}$$

$$\mathcal{L}(u) = \mathcal{L}_{u_0} + \mathcal{L}_{u_b},$$

where

$\Rightarrow$ $\mathcal{L}_{u_0}$ represents the misfit of the NN predictions

$\Rightarrow$ $\mathcal{L}_{u_b}$ represents the misfit of the initial/boundary conditions

$\Rightarrow$ θ represents the NN parameters

$\Rightarrow$ $\alpha_{\mathcal{F}}$ is a regularization parameter that controls the emphasis on the PDE-based residual (which we ideally want to be zero)

- Finally, we use ML methods (stochastic optimization, etc.) to train the NN model to minimize the loss.

PINN - warnings

1. Even with a large training set, this approach does not guarantee that the NN will obey the conservation/governing equations in the constraint (3).

2. In many SciML problems, these sorts of constraints on the system matter, as they correspond to physical mechanisms of the system. For example, if the conservation of energy equation is only approximately satisfied, then the system being simulated may behave qualitatively differently or even result in unrealistic solutions.

3. This approach of incorporating physics-based regularization, where the reglarization constraint, $\mathcal{L}_{\mathcal{F}}$, corresponds to a differential operator, is very different than incorporating much simpler norm-based regularization (such as L1 or L2 regularization), as is common in ML more generally. Here, the regularization operator, $\mathcal{L}_{\mathcal{F}}$, is non-trivially structured—it involves a differential operator that could actually be ill-conditioned, and it does

not correspond to a nice convex set as is the case for a norm ball.

4. Moreover, $\mathcal{L}_{\mathcal{F}}$ corresponds to actual physical quantities, and there is often an important distinction between satisfying the constraint exactly versus satisfying the constraint approximately—the soft constraint approach doing only the latter.

5. Adding/increasing the PDE-based soft constraint regularization makes it more complex and harder to optimize, especially for cases with non-trivial coefficients.

6. The loss landscape changes as the regularization parameter $\alpha_{\mathcal{F}}$ is changed. Reducing the regularization parameter can help alleviate the complexity of the loss landscape, but this in turn leads to poor solutions with high errors that do not satisfy the PDE/constraint.

PINN - steps

1. The **first step** is to define a neural network architecture that can be used to approximate the solution to the differential equation.
   (a) The neural network should have an input layer, an output layer, and one or more hidden layers.
   (b) The number of neurons in each layer and the activation function used in each layer must be chosen by the user.

2. The **second step** is to collect a dataset of known solutions to the differential equation.
   (a) This dataset can be generated using numerical methods, or
   (b) originate from experimental data.

3. The **third step** is to define the loss function.
   (a) The loss function is a measure of the error between the neural network's output and the known solutions to the differential equation.

(b) The loss function should be chosen so that it penalizes the neural network for making errors in both the spatial and temporal domains.

4. The **fourth step** is to train the neural network.

(a) The training process is done using an optimization algorithm, such as gradient descent.
(b) The optimization algorithm minimizes the loss function by adjusting the weights and biases of the neural network.

Once the neural network has been trained, and validated, it can be used to approximate the solution to the differential equation at any point in the domain. The accuracy of the approximation will depend on the size and quality of the training dataset, as well as the architecture of the neural network.

PINN - formulation

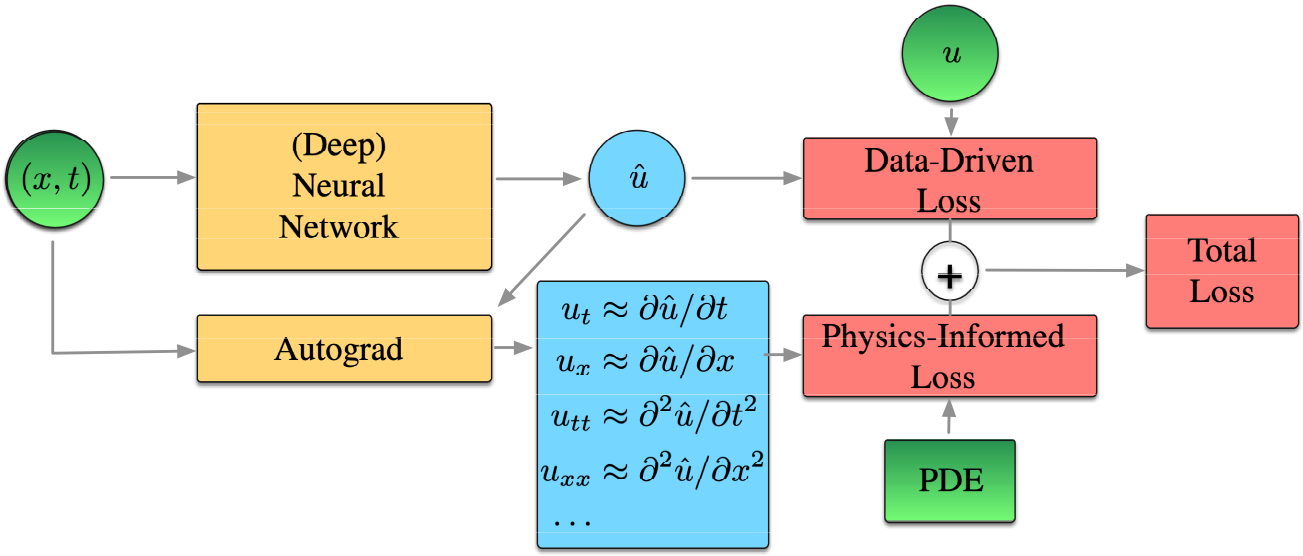
The mathematical formulation of a typical PINN loss function is as follows:

$$L = \phi_u(X_u) + \phi_b(X_b) + \phi_r(X_r),$$

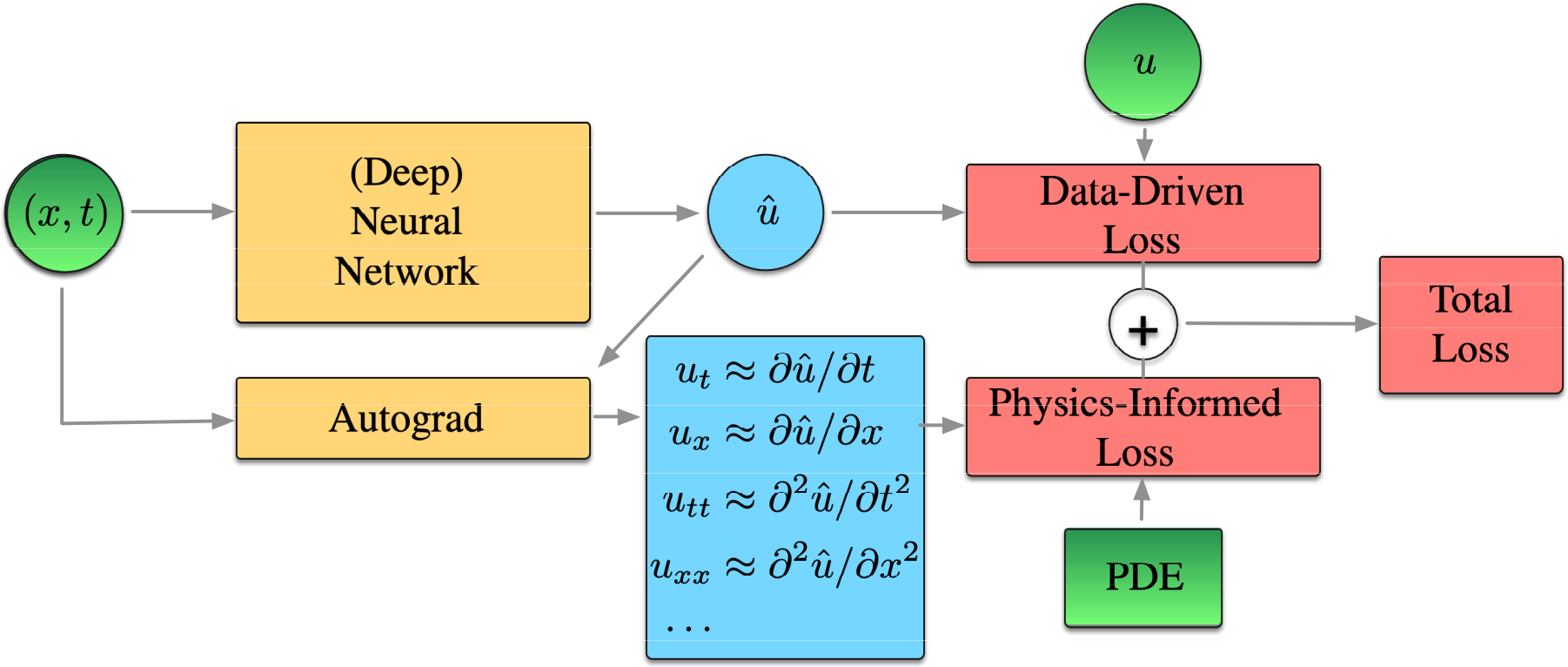
where,

- L is the global loss function.
- $\phi_u(X_u)$ is the loss term that penalizes the error between the neural network's output and the known solution at the training points X_u.
- $\phi_b(X_b)$ is the loss term that penalizes the error between the neural network's output and the boundary conditions at the training points X_b.
- $\phi_r(X_r)$is the loss term that penalizes the error between the neural network's output and the residual of the differential equation at the training points X_r.

The diagram below depicts how a PINN works:



PINN - Diagram



PINN Formulation - Neural Network

- **Neural Network**:

  $\Rightarrow$ a basic/adequate definition is to simply consider a NN as a mathematical function with some learnable parameters

  $\Rightarrow$ more mathematically, let the network be defined as

$$\mathrm{NN}(\mathbf{x}, \theta) : \mathbb{R}^{d_x} \times \mathbb{R}^{d_\theta} \times \mathbb{R}^{d_u}$$

  where

  $\rightarrow$ $\mathbf{x}$ are the inputs to the network

  $\rightarrow$ θ are a set of learnable paramaters (usually, weights)

  $\rightarrow$ d_x, d_θ and d_u are the dimensions of the network's inputs, parameters and outputs, respectively.

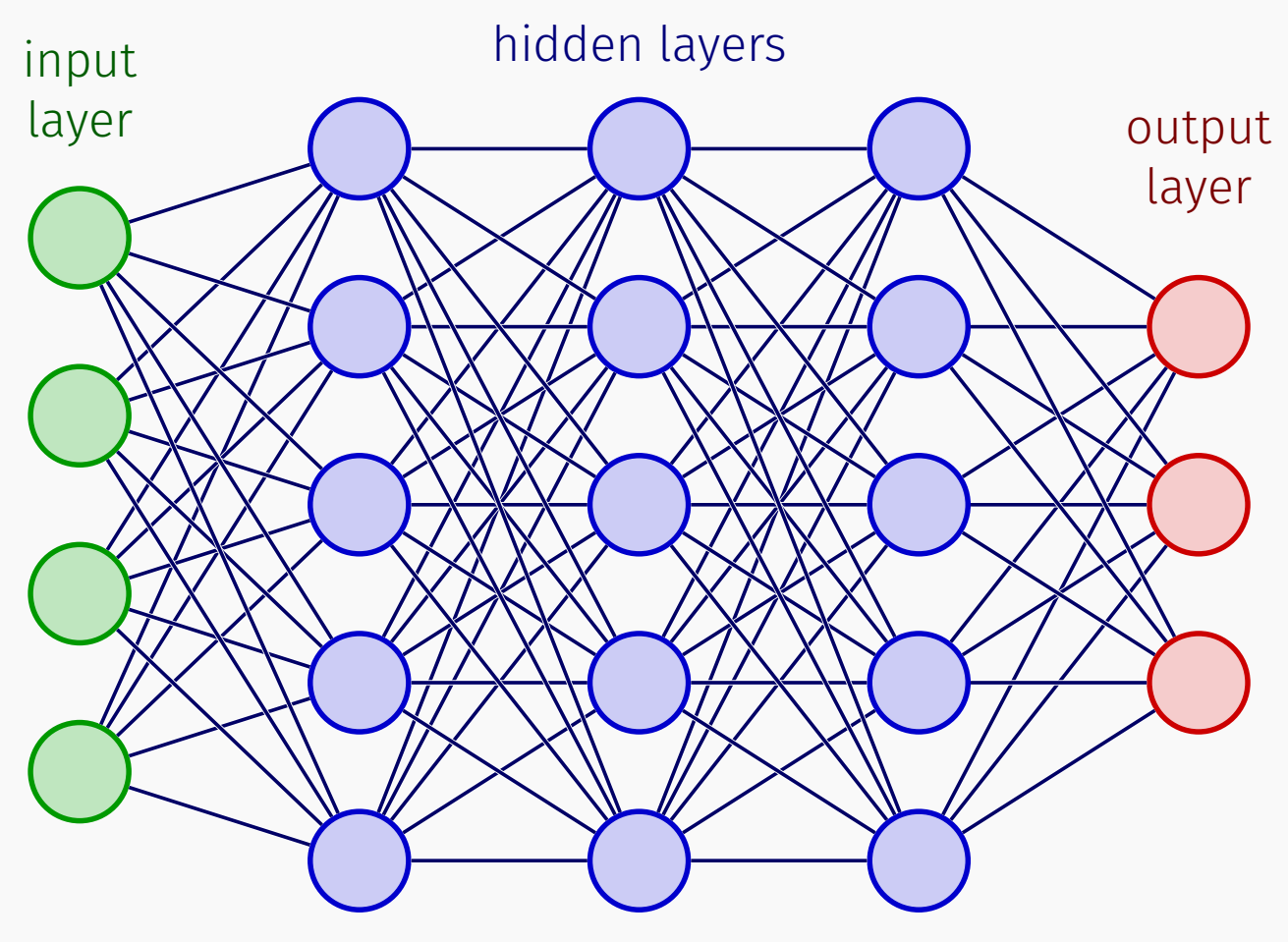
  $\Rightarrow$ The exact form of the network function is determined by the neural network's architecture. Here we use feedforward fully-connected networks (FCNNs), defined as

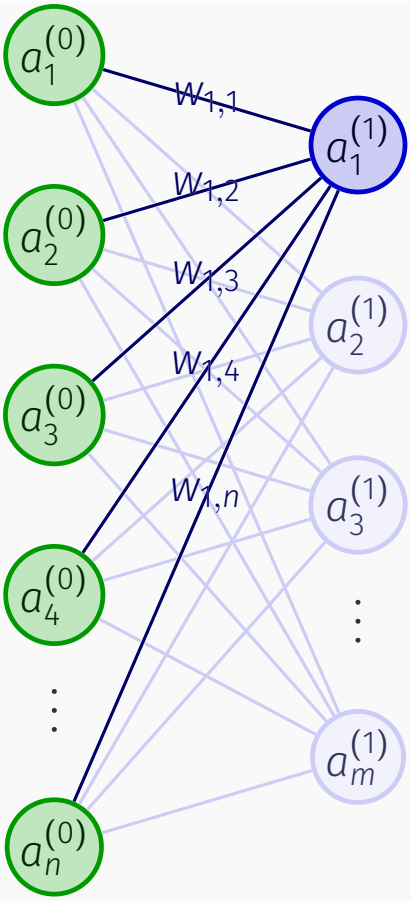
$$\mathrm{NN}(\mathbf{x}, \theta) = f_n \circ \cdots \circ f_i \circ \cdots \circ f_1(\mathbf{x}, \theta),$$

where

- $\mathbf{x} \in \mathbb{R}^{d_0}$ is the input to the FCNN
- $\mathrm{NN} \in \mathbb{R}^{d_n}$ is the output of the FCNN
- n is the number of layers (depth) of the FCNN
- $f_i(\mathbf{x}, \theta) = \sigma_i(W_i\mathbf{x} + \mathbf{b})$ are element-wise, nonlinear activation functions (usually ReLU or hyperbolic tangent)
- with $\theta_i = (W_i, \mathbf{b}_i)$,
- $W_i \in \mathbb{R}^{d_i \times d_{i-1}}$ weight matrices, $\mathbf{b} \in \mathbb{R}^{d_i}$ bias vectors, and
- $\theta = (\theta_1, \ldots, \theta_i, \ldots, \theta_n$ are the set of learnable parameters/weights of the network.

Recall: FCNN Architecture and Activation





$$= \sigma\left(w_{1,0}a_0^{(0)} + w_{1,1}a_1^{(0)} + \ldots + w_{1,n}a_n^{(0)} + b_1^{(0)}\right)$$

$$= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{(0)} + b_1^{(0)}\right)$$

$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,0} & \ldots & w_{1,n} \\ w_{2,0} & \ldots & w_{2,n} \\ \vdots & \ddots & \vdots \\ w_{m,0} & \ldots & w_{m,n} \end{pmatrix}\begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix}\right]$$

$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{a}^{(0)} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{y} = \sigma\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right), \quad \text{for a single hidden layer.}$$

9/49

PINN Formulation - (P)DE

- Recall: PINNs use neural networks to solve problems related to differential equations

- Consider a general boundary-value problem (I)BVP of the form

$$\begin{aligned} \mathcal{D}[u](\mathrm{x}) &= f(\mathrm{x}), \quad \mathrm{x} \in \Omega \subset \mathbb{R}^d, \\ \mathcal{B}_k[u](\mathrm{x}) &= g_k(\mathrm{x}), \quad \mathrm{x} \in \Gamma_k \subset \partial\Omega, \end{aligned} \tag{4}$$

  where

  $\Rightarrow$ $\mathcal{D}[u](\mathrm{x})$ is a differential operator
  $\Rightarrow$ $u(\mathrm{x})$ is the solution
  $\Rightarrow$ $\mathcal{B}_k(\cdot)$ are a set of boundary and/or initial conditions that ensure uniqueness of the solution
  $\Rightarrow$ the variable x represents/includes both spatial and time variables
  $\Rightarrow$ the full equation describes many possible contexts: linear and nonlinear, time-dependent and independent, irregular higher-order, cyclic BCs, etc.

$\Rightarrow$ To solve (4), PINNs use a neural network to directly approximate the solution,

$$\mathrm{NN}(\mathbf{x}, \theta) \approx u(\mathbf{x})$$

- PINN provides a functional approximation to the solution, and not a discretized solution similar to that provided by traditional methods such as finite difference methods

  $\Rightarrow$ as such PINNs are a mesh-free approach for solving differential equations

PINN Formulation - Loss Function

- **Loss Function**: Let $F = 0$ be the PDE, $B = 0$ the boundary/initial conditions, $I = 0$ the inversion conditions, then the PINN loss is

$$\mathcal{L}(\theta, \lambda; \mathcal{T}) = w_f \mathcal{L}_f(\theta, \lambda; \mathcal{T}_f) + w_b \mathcal{L}_b(\theta\lambda; \mathcal{T}_b) + w_i \mathcal{L}_i(\theta, \lambda; \mathcal{T}_i)$$

where

$$\mathcal{L}_f(\theta; \mathcal{T}_f) = \|F(\hat{u}, x, \lambda)\|_2^2$$
$$\mathcal{L}_b(\theta; \mathcal{T}_b) = \|B(\hat{u}, x)\|_2^2$$
$$\mathcal{L}_i(\theta, \lambda, \mathcal{T}_i) = \frac{1}{|\mathcal{T}_i|} \sum_{x \in \mathcal{T}_i} \|I(\hat{u}, x)\|_2^2$$

and

$\Rightarrow$ x are the training points,
$\Rightarrow$ $\hat{u}$ the approximate solution,
$\Rightarrow$ λ the inversion coefficients,

$\Rightarrow$ w the weights that ensure balance among the different loss function terms

- The solution is then given by,

$$\{\theta^*, \lambda^*\} = \underset{\theta,\lambda}{\operatorname{argmin}} \mathcal{L}(\theta, \lambda; \mathcal{T})$$

- **Note**: solving the inverse problems requires only the addition of one term in the loss function, and nothing more!

PINN Formulation - Error Analysis

- Error analysis can been derived[2][3], in terms of

  $\Rightarrow$ optimization error $e_o = \|\hat{u}_{\mathcal{T}} - u_{\mathcal{T}}\|$
  $\Rightarrow$ generalization error $e_g = \|u_{\mathcal{T}} - u_{\mathcal{F}}\|$
  $\Rightarrow$ approximation error $e_a = \|u_{\mathcal{F}} - u\|$

- then

$$e \doteq \|\hat{u}_{\mathcal{T}} - u\| \leq e_o + e_g + e_a$$

[2] Lu, Karniadakis, SIAM Review, 2021.

[3] Mishra, Molinaro; arXiv:2006.16144v2 and IMA J. of Numerical Analysis, Volume 43, Issue 1, January 2023, Pages 1–43.

PINN Formulation - Loss Function (II)

- The values for the loss function are available, in general, at discrete points, often called collocation points.
- We will write down the terms explicitly in this case, for the direct problem, with composite loss function,

$$\mathcal{L}(\theta) = \mathcal{L}_D(\mathbf{x}, \theta) + \mathcal{L}_B(\mathbf{x}, \theta), \tag{5}$$

where

$\Rightarrow$ (P)DE residual is defined as

$$\mathcal{L}_D(\mathbf{x}, \theta) = \frac{\alpha_I}{N_I} \sum_{i=1}^{N_I} \left(\mathcal{D}[u](\mathrm{x}_i, \theta) - f(\mathrm{x}_i)\right)^2$$

$\Rightarrow$ (I)BC residual is defined as

$$\mathcal{L}_B(\mathbf{x}, \theta) = \sum_{k=1}^{N_k} \frac{\alpha_B^k}{N_B^k} \sum_{i=1}^{N_B^k} \left(\mathcal{B}_k[u](\mathrm{x}_i^k, \theta) - g_k(\mathrm{x}_i^k)\right)^2,$$

where

→ $\{\mathrm{x}_i\}_{i=1}^{N_I}$ is a set of collocation points sampled in the interior of the domain

→ $\left\{\mathrm{x}_j^k\right\}_{j=1}^{N_B^k}$ is a set of points sampled along each boundary condition (where k permits to separate Dirichlet, Neumann, mixed, and initial conditions)

→ α_I and α_B^k are well-chosen scalar weights, chosen by suitable tuning methods, that ensure the terms in the loss function are well-balanced.

PINN Formulation - Loss Function (III)

We can see, intuitively, that

- by minimizing the (P)DE residual, the method tries to ensure that the solution learned by the network obeys the underlying PDE, and
- by minimizing the (I)BC residual, the method tries to ensure that the learned solution is unique by matching it to the BCs.
- **Note**: a sufficient number of collocation and boundary points must be chosen such that the PINN is able to learn a consistent solution across the domain.

As usual (see Optimization Lecture), iterative schemes are typically used to optimize this loss function

- variants of the stochastic gradient descent (SGD) method, such as the Adam optimizer, or

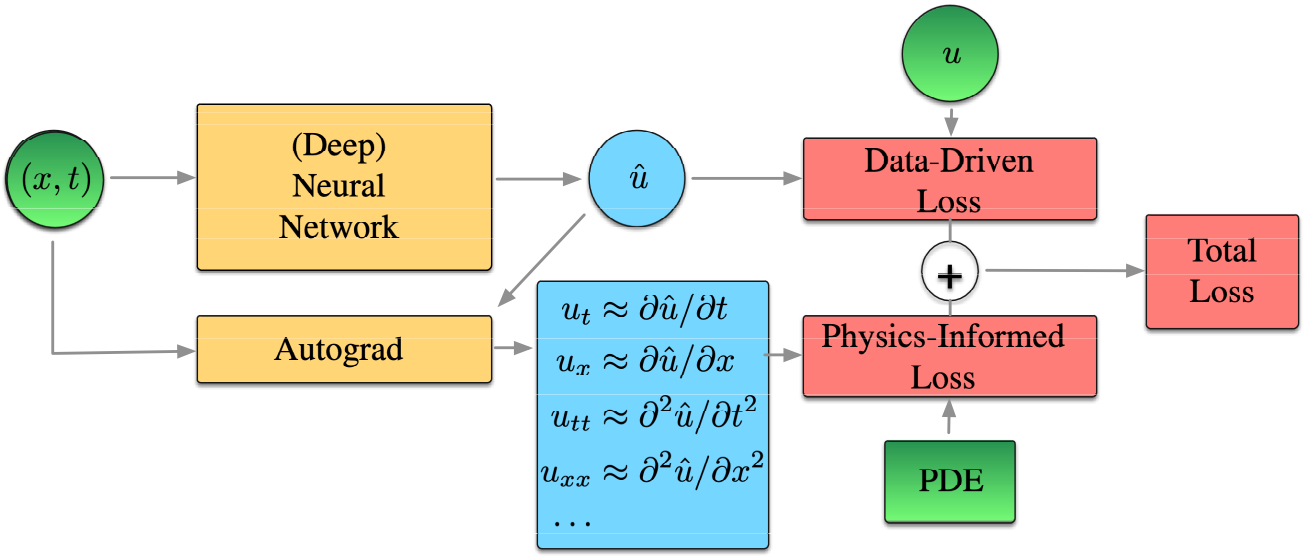
- quasi-Newton methods, such as the limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm are employed.

Note:

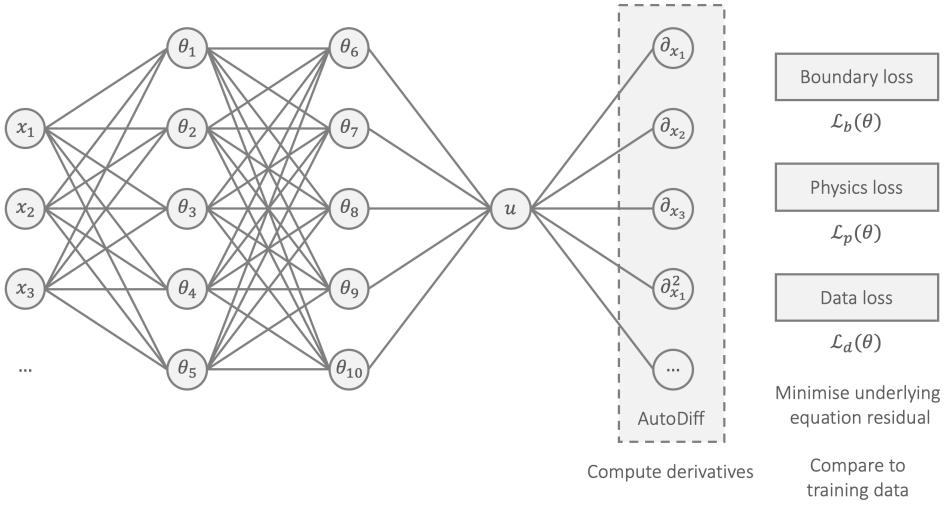
- These methods require the computation of the gradient of the loss function with respect to the network parameters, which can computed easily and efficiently using automatic differentiation provided systematically in all modern deep learning libraries

- Note also that gradients of the network output with respect to its inputs are also typically required to evaluate the PDE residual in the loss function, and can similarly be obtained and further differentiated through to update the network's parameters using automatic differentiation once again.

Recall the global flowchart for PINN:



Here is a case with 2 hidden layer NN:



PINN - pros and cons

- Here are some of the advantages of using PINNs to solve differential equations:

  ✔ They can be used to solve a wide variety of differential equations, including ODEs and PDEs.
  ✔ They can be used to solve problems with complex geometries and non-linear behavior.
  ✔ They are essentially mesh-free.
  ✔ They can be trained to be very accurate, even with limited data and noisy data.
  ✔ They are relatively easy to implement, leveraging AD capabilities.
  ✔ When they work, they can provide impressive speed-ups of 3 to 4 orders of magnitude.

- Here are some of the disadvantages of using PINNs to solve differential equations:

  ✘ They can produce a horrendous optimization problem.

- ✘ They can be computationally expensive to train.
- ✘ They have difficulty with high frequencies and multiple scales.
- ✘ They can be sensitive to the choice of hyperparameters, in particular to the network architecture and size.
- ✘ They can be difficult to interpret, as the neural network may learn a complex relationship between the inputs and outputs that is not easily understood.

- Overall, PINNs are a promising new approach to solving differential equations.

  $\Rightarrow$ They provide powerful tools that can be used to solve a wide variety of problems, but they also have some severe limitations.

PINN - remedies

- A downside of training PINNs with the loss function given by (5) is that the BCs are softly enforced.

  $\Rightarrow$ This means the learned solution may deviate from the BCs because the BC term may not be fully minimized.

  $\Rightarrow$ Furthermore, it can be challenging to balance the different objectives of the PDE and BC terms in the loss function, which can lead to poor convergence and solution accuracy.

- One possibility is to enforce BCs in a hard fashion by using the neural network as part of a solution *ansatz*. This will be shown in some of the examples below.

- Many other "fixes" have been formulated (see references, in particular arXiv, where new solutions appear almost daily...)

PINN Remedies - enforcing hard BCs

- **Idea**: use the NN as part of a solution ansatz, that by definition satisfies the BC, thus avoiding the soft constraint on $\mathcal{L}_B$ in (5)

- More precisely, we approximate the solution of the (P)DE by

$$\mathcal{C}[u](\mathrm{x}, \theta) \approx u(\mathrm{x}, \theta)$$

  where $\mathcal{C}$ is an appropriately selected constraining operator that analytically/exactly enforces the BCs

- **Example**:

  $\Rightarrow$ suppose we want to enforce

$$u(x = 0) = 0$$

  in a scalar ODE

⇒ The constraining operator and solution ansatz could be chosen as

$$\mathcal{C}[u](\mathrm{x}, \theta) = (\tanh x)\, u(\mathrm{x}, \theta)$$

or any other function whose value at $x = 0$ is zero

→ the function $\tanh(x)$ is zero at 0, forcing the BC to always be obeyed, but non-zero away from 0, allowing the network to learn the solution away from the BC.

- With this approach, the BCs are always satisfied and therefore the BC term in the loss function (5) can be removed,

⇒ the PINN can be trained using the simpler unconstrained loss function,

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(\mathcal{D}[\mathcal{C}u](\mathrm{x}_i, \theta) - f(\mathrm{x}_i)\right)^2$$

where $\{\mathrm{x}_i\}_{i=1}^{N}$ is a set of collocation points sampled in the interior of the domain

- **Notes**:

  $\Rightarrow$ There is no unique way of choosing the constraining operator, and the definition of a suitable constraining operator for complex geometries and/or complex BCs may be difficult or sometimes even impossible, i.e., this strategy is problem dependent; in this case, one may resort to the soft enforcement of boundary conditions (5) instead.

  $\Rightarrow$ A promising approach for alleviating the difficulties when higher frequencies and multi-scale features are added to the solution, is to use a finite basis PINN approach (FBPINN) [Dolean, Heinlein, Mishra, Moseley 2023], where instead of using a single neural network to represent the solution, many smaller neural networks are confined in overlapping subdomains and summed together to represent the solution.

Example: IBVP for Diffusion Equation

Compute $u(\mathbf{x}, t)$: $\Omega \times [0, T] \to \mathbb{R}$ such that

$$\frac{\partial u(\mathbf{x}, t)}{\partial t} - \nabla \cdot (\lambda(x) \nabla u(\mathbf{x}, t)) = f(\mathbf{x}, t) \quad \text{in } \Omega \times (0, \mathrm{T}), \tag{6}$$

$$u(\mathbf{x}, t) = g_D(\mathbf{x}, t) \quad \text{on } \partial\Omega_{\mathrm{D}} \times (0, \mathrm{T}),$$

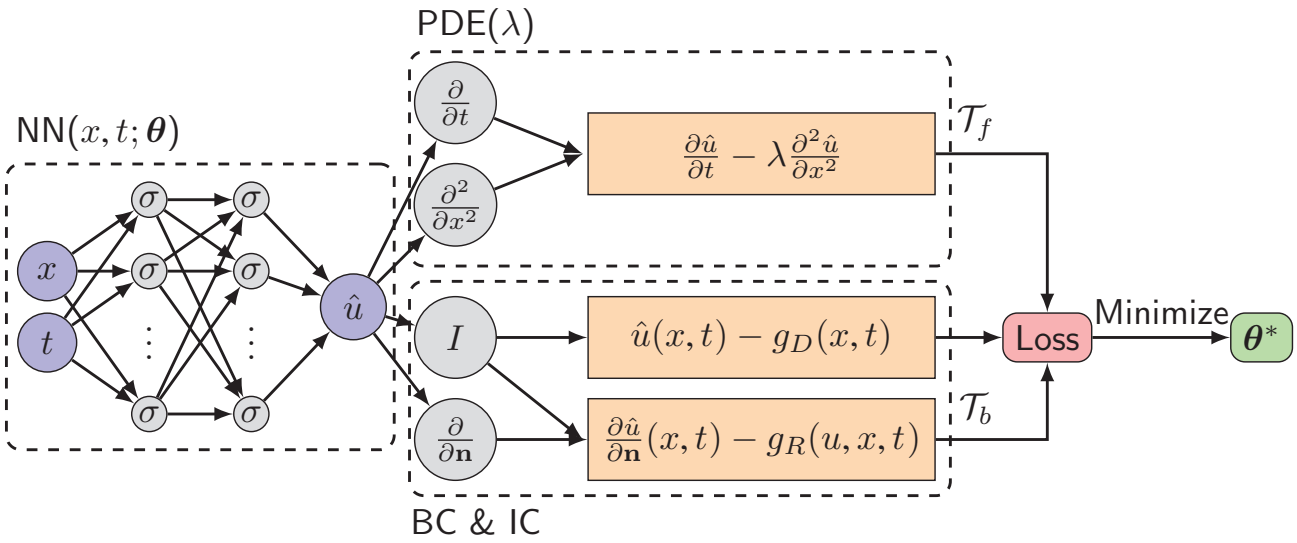
$$-\lambda(x) \nabla u(\mathbf{x}, t) \cdot \mathbf{n} = g_R(\mathbf{x}, t) \quad \text{on } \partial\Omega_{\mathrm{R}} \times (0, \mathrm{T}),$$

$$u(\mathbf{x}, 0) = u_0(\mathbf{x}) \quad \text{for } \mathrm{x} \in \Omega.$$

Note that $\lambda(x)$ is, in general, a tensor (matrix) with elements λ_{ij}.

- Direct **problem**: given λ, compute u.
- Inverse **problem**: given u, compute λ.

PINN for the Diffusion Equation



[Credit: Lu, Karniadakis, SIAM Review, 2021]

- Use FCNN to approximate u at the selected points x, with training data at residual points $\mathcal{T}_f$ and $\mathcal{T}_b$
- Use AD to compute derivatives for the PDE and the boundary/initial conditions
- Minimize the augmented, weighted loss function

PHYSICS OPERATOR LEARNING

Recall: Universal Approximation for Operators

Theorem 3 (Chen, Chen 1995). *Suppose σ is continuous, non-polynomial, X is a Banach space, $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are compact sets, V is compact in $C(K_1)$, G is continuous operator from V into $C(K_2)$. Then, for any $\epsilon > 0$, there exist positive integers m, n, p, constants c_k^i, ξ_{ij}^k, θ_i^k, $\zeta_k \in \mathbb{R}$, $w_k \in \mathbb{R}^d$, $x_j \in K_1$, such that*

$$\left| G(u)(y) - \sum_{k=1}^{p} \sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right) \sigma \left(w_k \cdot y + \zeta_k \right) \right| < \epsilon$$

for all $u \in V$, $y \in K_2$.

Operator Learning - DeepONet, FNO, PINO

- **Idea**: train a NN to learn the (P)DE operator

  $\Rightarrow$ used by NVIDIA in FourCastNet

- A related set of approaches which incorporate governing equations into their loss function are physics-informed neural operators (PINO).

- These are neural networks which are similar to PINNs in that they are designed to learn the solution to differential equations, but instead of learning a single solution they learn an entire family of solutions by adding certain inputs of the differential equation as inputs to the network.

- Thus, they do not need to be retrained to carry out new simulations, and during inference they offer a fast surrogate model.

- From a mathematical standpoint, the goal is to learn an operator to map function spaces to function spaces, rather than just a single function.

- DeepONet consisted of two sub-networks,

  ⇒ a "branch" network that encodes the input function (by taking discretised samples of the input function at fixed locations as input), and
  ⇒ a "trunk" network that encodes a set of input coordinates.
  ⇒ The solution of the differential equation is then approximated by merging the outputs of both of these sub-networks.
  ⇒ The network is trained using a loss function that extends the PINN loss function (5) by averaging over many random samples of the input function.

- FNO (Fourier Neural Operator) uses a physics-informed loss function when training Fourier neural operators

  ⇒ Similar to DeepONets, FNOs learn an operator to map between function spaces.
  ⇒ This is achieved by using a series of stacked Fourier

layers, where the input to each layer is Fourier transformed and truncated to a fixed number of Fourier modes.

$\Rightarrow$ This truncation allows the model to learn mappings that are invariant to the number of discrete points used in its inputs and outputs.

Operator Nets

- Use the Universal Operator Approximation Theorem...

$$\left| G(u)(y) - \sum_{k=1}^{p} \sum_{i=1}^{n} c_i^k \underbrace{\sigma\left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right)}_{\text{branch}} \underbrace{\sigma\left(w_k \cdot y + \zeta_k\right)}_{\text{trunk}} \right|$$
$$< \epsilon,$$

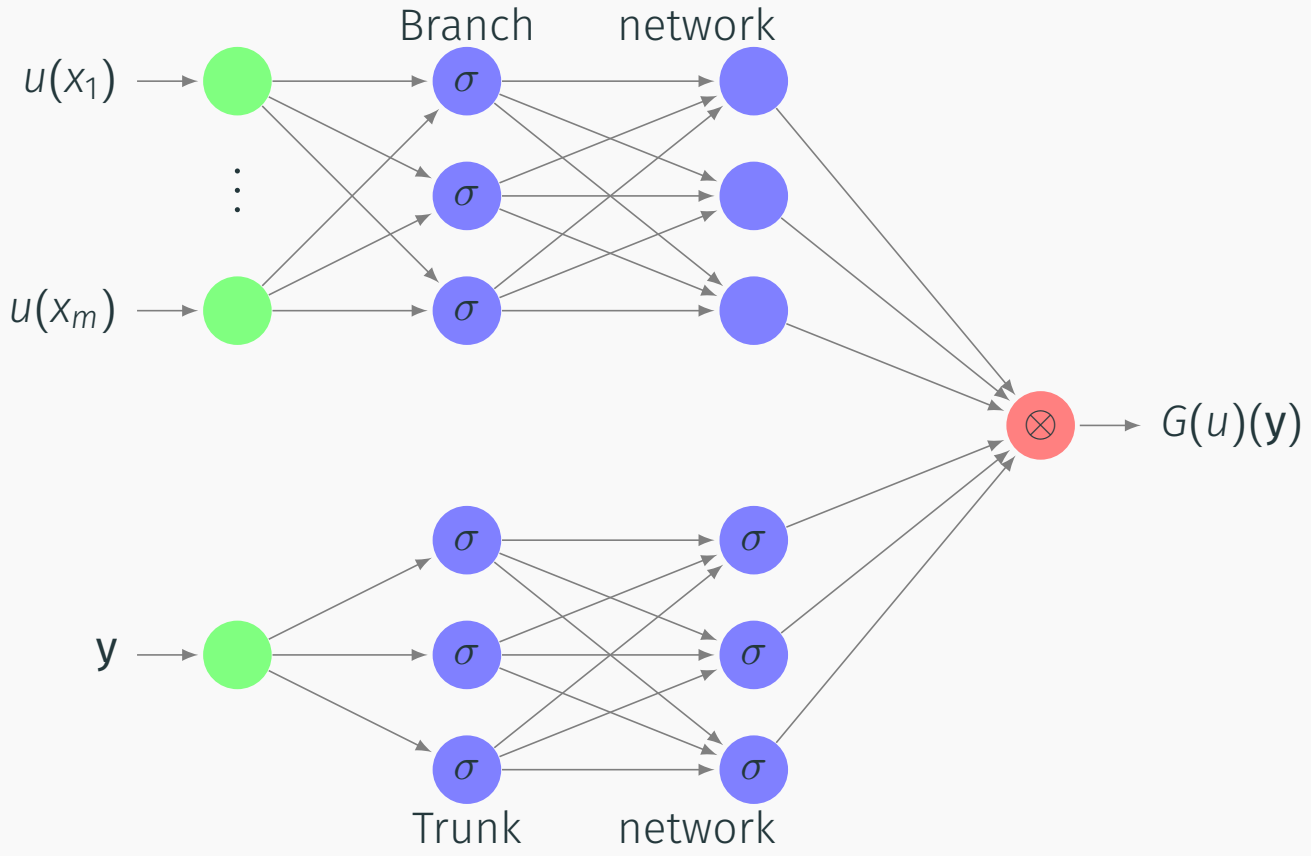
  where

  $\Rightarrow$ G is the solution operator,
  $\Rightarrow$ u is an input function,
  $\Rightarrow$ x_i are "sensor" points,
  $\Rightarrow$ y are random points where we evaluate the output function $G(u)$.

- 2 main contenders:

  $\Rightarrow$ DeepONet
  $\Rightarrow$ Fourier Neural Operators (FNO)—a special case of DeepONet

DeepONet Architecture



Directly copied from the Theorem!

DeepONet Loss Function

- Branch (FCNN, ResNET, CNN, etc.) and trunk networks (FCNN) are merged by an inner product.
- Prediction of a function u evaluated at points $\mathbf{y}$ is then given by

$$G_\theta(u)(y) = \sum_{k=1}^{q} \underbrace{b_k(u(x))}_{\text{branch}} \underbrace{t_k(y)}_{\text{trunk}} + b_0$$

- Training weights and biases, θ, computed by minimizing the loss (mini-batch by Adam, single-batch by L-BFGS)

$$\mathcal{L}_o(\theta) = \frac{1}{NP} \sum_{i=1}^{N} \sum_{j=1}^{P} \left| G_\theta(u^{(i)})(y_j^{(i)}) - G(u^{(i)})(y_j^{(i)}) \right|^2$$

DeepONet: Pros and Cons

- Pros:
  - ✔ relatively fast training (compared to PINN)
  - ✔ can overcome the curse of dimensionality (in some cases...)
  - ✔ suitable for multiscale and multiphysics problems
- Cons:
  - ✘ no guarantee that physics is respected
  - ✘ require large training sets of paired input-output observations (expensive!)

DeepONet Formulation (I)

- Parametric, linear/nonlinear operator plus IBC (IBVP)

$$\mathcal{O}(u, s) = 0,$$
$$\mathcal{B}(u, s) = 0,$$

- where

  $\Rightarrow$ $u \in \mathcal{U}$ is the input function (parameters),
  $\Rightarrow$ $s \in \mathcal{S}$ is the hidden, solution function

- If $\exists!$ solution $s = s(u) \in \mathcal{S}$ to the IBVP, then we can define the solution operator $G \colon \mathcal{U} \mapsto \mathcal{S}$ by

$$G(u) = s(u).$$

DeepONet Formulation (II)

- Approximate the solution map G by a DeepONet G_θ

$$G_\theta(u)(y) = \sum_{k=1}^{q} \underbrace{b_k(u(x))}_{\text{branch}} \underbrace{t_k(y)}_{\text{trunk}} + b_0$$

where θ represents all the trainable weights and biases, computed by minimizing the loss at a set of P random output points $\{y_j\}_{j=1}^{p}$

$$\mathcal{L}(u, \theta) = \frac{1}{P} \sum_{j=1}^{P} |G_\theta(u)(y_j) - s(y_j)|^2,$$

and $s(y_j)$ is the PDE solution evaluated at P locations in the domain of $G(u)$

DeepONet Formulation (III)

- To obtain a vector output, a stacked version is defined by repeated sampling over $i = 1, \ldots, N$, giving the overall operator loss

$$\mathcal{L}_o(\theta) = \frac{1}{NP} \sum_{i=1}^{N} \sum_{j=1}^{P} \left| G_\theta(u^{(i)})(y_j^{(i)}) - s^{(i)}(y_j^{(i)}) \right|^2$$

DeepONet + PINN = PI-DeepONet

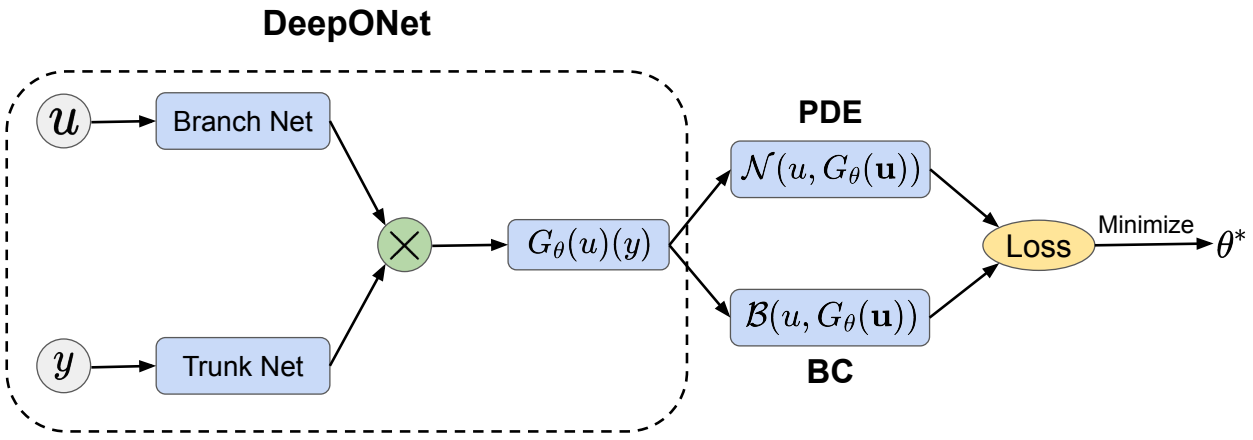
- We can combine the two, to get the best of both worlds

$$\mathcal{L}(\theta) = w_f \mathcal{L}_f(G_\theta(u)(y)) + w_b \mathcal{L}_b(G_\theta(u)(y)) + w_o \mathcal{L}_o(G_\theta(u)(y)$$

- Results:[4]

  ⇒ no need for paired input-ouput observations, just samples of the input function and BC/IC (self-supervised learning)
  ⇒ respects the physics
  ⇒ improved predictive accuracy
  ⇒ ideal for parametric PDE studies—optimization, parameter estimation, screening, etc.

[4] Wang, Wang, Bhouri, Perdikaris. arXiv:2103.10974v1, arXiv:2106.05384, arXiv:2110.01654, arXiv:2110.13297



[Credit: Wang, Wang, Perdikaris; arXiv, 2021]

- Train by minimizing the composite loss

$$\mathcal{L}(\theta) = \mathcal{L}_o(\theta) + \mathcal{L}_\phi(\theta),$$

where

$\Rightarrow$ the operator loss is as above for deepOnet, or using the IBC

$$\mathcal{L}_o(\theta) = \frac{1}{NP} \sum_{i=1}^{N} \sum_{j=1}^{P} \left| \mathcal{B}\left(u^{(i)}(x_j^{(i)}), G_\theta(u^{(i)})(y_j^{(i)}) \right) \right|^2$$

$\Rightarrow$ the physics loss is computed using the operator network approximate solution

$$\mathcal{L}_\phi(\theta) = \frac{1}{NQ} \sum_{i=1}^{N} \sum_{j=1}^{Q} \left| \mathcal{O}\left(u^{(i)}(x_j^{(i)}), G_\theta(u^{(i)})(y_j^{(i)}) \right) \right|^2$$

- This is self-supervised, and does not require paired input-output observations!

OTHER METHODS

Differentiable Physics

A potentially powerful hybrid approach is to open up the black box of a traditional algorithm and tightly integrate ML models within it.

- This allows a more granular way of balancing the two paradigms;

  $\Rightarrow$ ML can be inserted where we are unsure how to solve a problem, or where the traditional workflow is computationally expensive, and

  $\Rightarrow$ traditional components can be kept where we require robust and interpretable outputs.

- Often, the performance of the traditional workflow is improved whilst the ML components are easier to train, are more interpretable, and require less parameters and training data compared to a naive ML approach.

- A general approach for doing so is to use concepts from the field of differentiable physics

⇒ many traditional scientific algorithms can be written as a composition of basic and differentiable mathematical operations (such as matrix multiplication, addition, subtraction, etc), and that

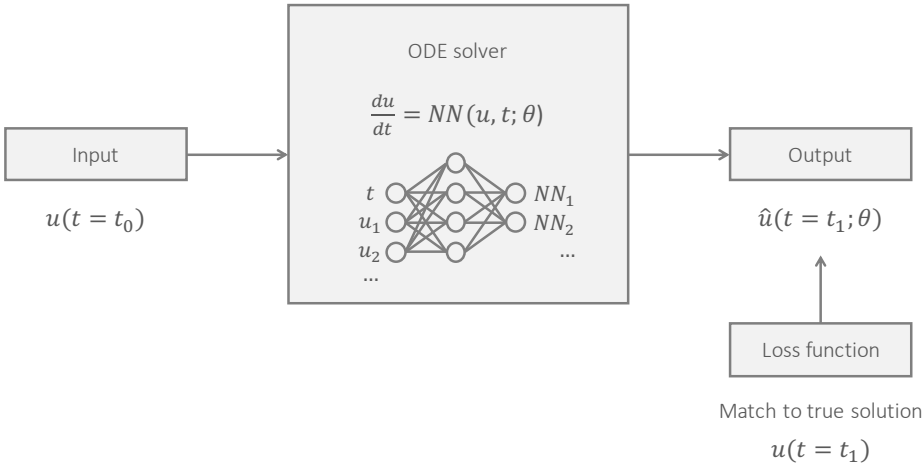
⇒ modern automatic differentiation and differential programming languages [Baydin et al., 2018] make it easy to track and backpropagate the gradients of these outputs with respect to their inputs.

⇒ This unlocks the possibility of inserting and training gradient-based ML components (such as neural networks) within traditional workflows, whereas otherwise it may have been difficult to do so.

- A simple approach to start with is to re-implement a traditional workflow inside a modern differentiable programming language, such as PyTorch, TensorFlow or JAX.

- Once a traditional algorithm is implemented, its design can be altered by treating certain parameters as learnable, or by inserting new learned components.

Neural ODEs



Schematic of a neural ordinary differential equation (ODE) [Moseley2022].

- The goal of a neural ODE is to learn the right-hand side term of an unknown ODE.

- A neural network $\mathrm{NN}(u, t; \theta)$ is used to represent this term, which is trained by using many examples of the solution of the ODE at two times, $u(t = t_0)$ and $u(t = t_1)$.

- More specifically, a standard ODE solver is used to model the solution of the ODE, $u(t = t_1)$, at time $t = t_1$ given the solution at time $t = t_0$ and evaluations of the network where needed.

- Then, the network's free parameters, θ, are updated by matching this estimated solution with the true solution and differentiating through the entire ODE solver

Other Approaches

- Recurrent NNs - see FIDL example
- Material design (META) uses Graph NNs
- GP and Ridge regression - used by Mendez
- LSTM
- Encoder-decoder
- in fact the list is never-ending...
- and now there is generative learning!

GENERATIVE PHYSICS LEARNING

GPT

- GPT = Genrative Pre-trained Transformer
- Theory:
  - $\Rightarrow$ ingest huge volumes of data
  - $\Rightarrow$ "fill in the gaps" using Markov Chains on tokens
- Applications
  - $\Rightarrow$ NWP + Climatology (ClimaX by Microsoft)
  - $\Rightarrow$ healthcare and drug-design (Alpha-Fold)
  - $\Rightarrow$ etc.
- Quo Vadimus??? See Introductory and Ethics Lectures for more details.

APPLICATIONS

Applications of ML for (P)DEs

- Literally, from ALL domains...
- See next lecture.

Bibliography-Reviews

References

[1] S A Faroughi, N Pawar, C Fernandes, M. Raissi, S. Das, N K Kalantari, S K Mahjour. Physics-Guided, Physics-Informed, and Physics-Encoded Neural Networks in Scientific Computing. arXiv, 2023. https://arxiv.org/pdf/2211.07377

[2] S. Cuomo, V. Schiano Di Cola, F. Giampaolo, G. Rozza, M. Raissi, F. Piccialli. Scientific Machine Learning Through Physics–Informed Neural Networks: Where we are and What's Next. *Journal of Scientific Computing* (2022) 92:88.

[3] L Lu, P Jin, G Pang, Z Zhang, GE Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence* 3 (3), 218-229 (2021).

[4] L. Lu, X. Meng, Z. Mao, G. Karniadakis. DeepXDE: A Deep Learning Library for Solving Differential Equations. *SIAM Review,* 63, 1 (2021).

[5] E. Darve, K. Xu. Physics constrained learning for data-driven inverse modeling from sparse observations. *J. of Computational Physics*, 453. (2022).

[6] M. Raissi, P. Perdikaris, G. E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. of Computational Physics*, 378, pp. 686-707, 2021.

[7] N. Kovachki, Z. Li, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkulmar. Neural Operator: Learning Maps Between Function Spaces With Applications to PDEs. *J. of Machine Learning Research* 24 (2023).

[8] M. Asch, M. Bocquet, M. Nodet. *Data Assimilation: Theory, Algorithms and Applications.* SIAM, 2016.

[9] M. Asch. *Digital Twins: from Model-Based to Data-Driven.* SIAM, 2022.