Deep Learning in Scientific Computing

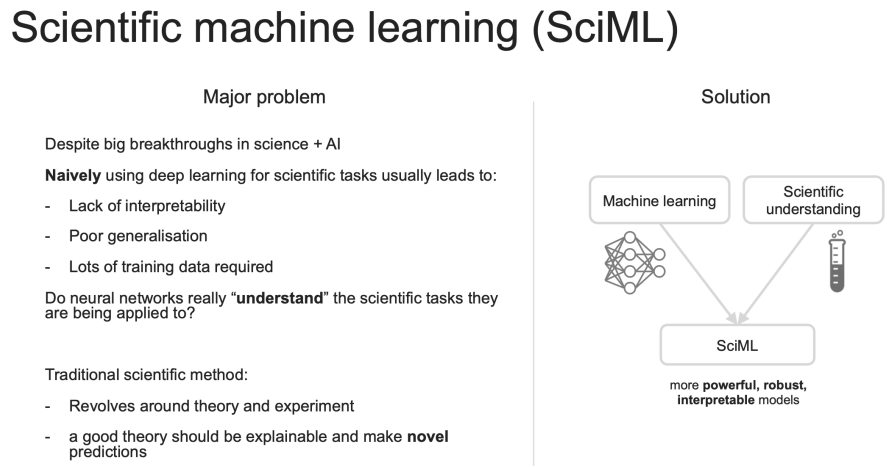
Shizheng Wen

shiwen@student.ethz.ch

Table of contents

1 Introduction

Definition of Scientific machine learning (SciML)



Key scientific tasks and how machine learning can help

In science, there are mainly three types of question. We will see how ML can help us.

- Simulation
  - Usually used as a starting point for other tasks (e.g. inverse / control / design problems)
  - $$b = F(a)$$

    a = set of input conditions
    F = physical model of the system
    b = resulting properties given F and a
  - Analytical solutions do not exist, and we must resort to numerical modeling. By the way, most of the questions based on the system modelled by PDEs.
- Inverse, control, and data assimilation problems
  - $$b = F(a)$$

    a = set of input conditions
    F = physical model of the system
    b = resulting properties given F and a
  - Tricks for solving 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, …)

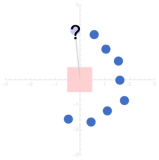
    - If we want to derive the parameters that gives rise to the solution of b, then we treat it as a optimization problem and use gradient-based or gradient-free methods to solve it. Gradient-based means that the update of a is based on gradient of the function.

  – For control, a is a function.

    - $a = f(t)$, force applied to cart
      b = pendulum stays balanced (i.e. $\theta(t) \to 0$)
      F = method for solving equations of motion

  – For data assimilation

    - **Data assimilation**



      $a = x(t=0), \theta(t=0)$,
      b = Noisy measurements of $x(t_i), \theta(t_i)$
      F = Method for solving equations of motion and noise model

    - We have a bunch of noisy measurements of data, we want to know where we start of the system.

- Equation discovery and anomaly detection

  –

$$b = F(a)$$

a = set of input conditions
F = physical model of the system
b = resulting properties given F and a

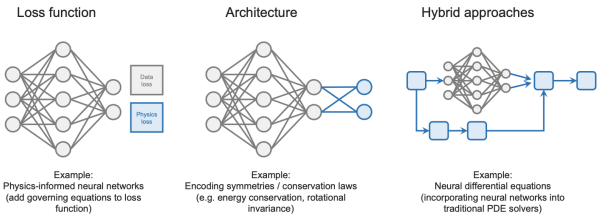
  – Historically, F (= laws of physics) has been found through remarkable human **intuition**. From a computational standpoint, discovering physics is like solving an inverse problem (trying to fit a model to observed data). But the model should be **explainable**, **generalisable** and make **novel**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

Ways to incorporate scientific principles into deep learning

- Example - Simulation: Fourier Neural Operator (Application: FourCastNet)

- Example - Inverse problems: use a neural network to update gradients. It's hard to understand such problems. View this paper for more information: Solving ill-posed inverse problems using iterative deep neural networks, Inverse Problems (2017)
- Example - Equation Discovery: Fit a neural network to observed data, then regress over a library of gradients to "discover" underlying equations. Here, we don't need to use complex regression methods, only linear regression is enough for finding the underlying equation.

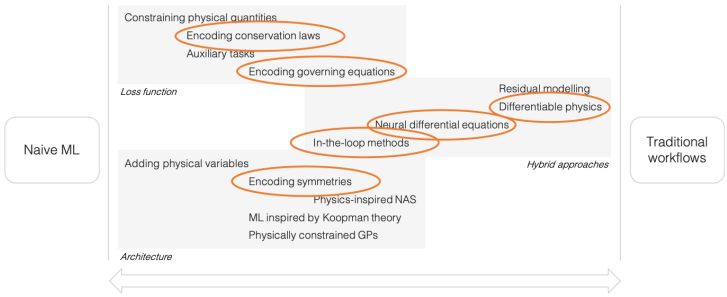
Ways to incorporate scientific principles into machine learning



A plethora of SciML techniques

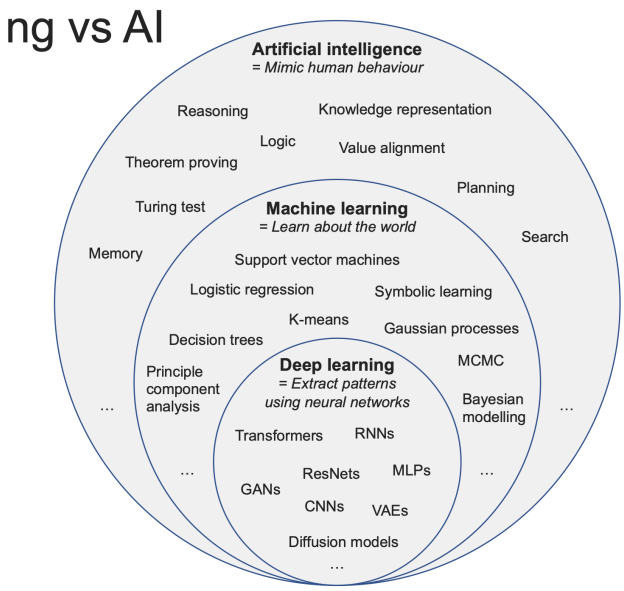
Scientific task

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., Jin et al., Chen et al., Kharazmi et al., Yang Yang et al. et al., Wang et al., Wang et al., Li et al., Zhu et al., Geneva and Zabaras, Gao 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.	

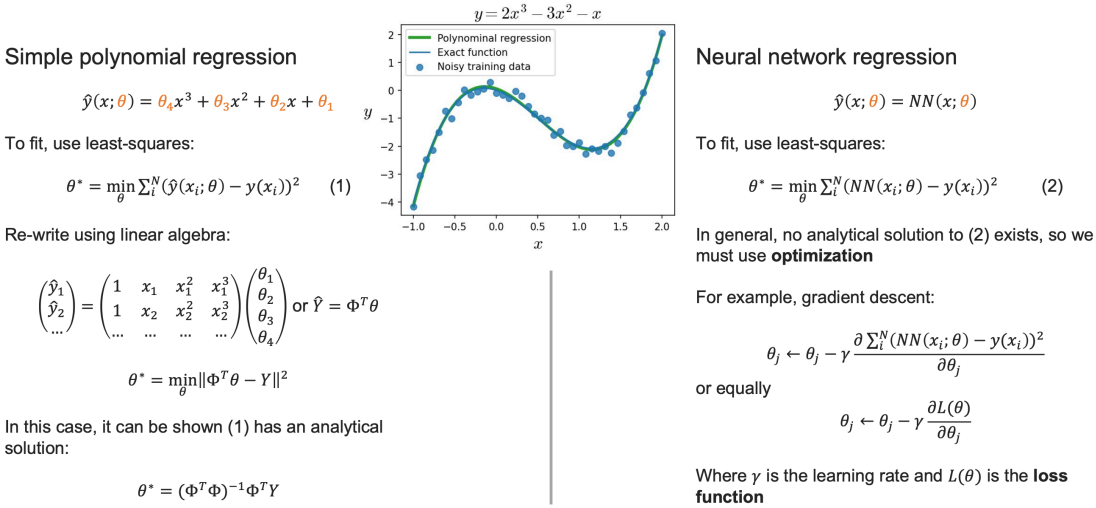


2 Introduction to Deep Learning

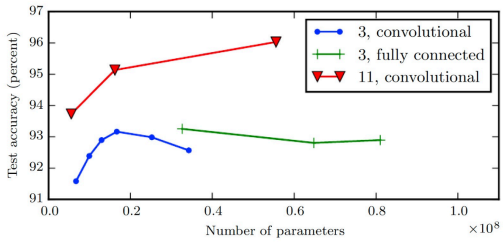
In this chapter, firstly, the lecturer introduce the concept of Machine learning and neural network. Unlike the polynomial regression having closed-form solution, the only way to train neural network is by optimization method. Then the lecturer introduce the CNN, which has advantages for processing spatial correlations input.



Multilayer perceptrons



- Function fitting between polynomial regression and neural network. For polynomial regression, we have the closed form.



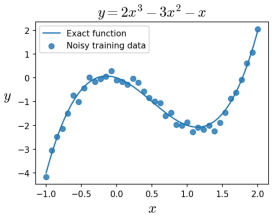
Goodfellow et al, Multi-digit number recognition from street view imagery using deep convolutional neural networks, ICLR (2014)

- 当模型复杂度上升，模型有过拟合的迹象，在深度学习中，model parameters > training datset的时候，模型的accuracy又会进一步上升，效果又会进一步变好。
- Empirically, deep neural networks perform better than shallow neural networks => encode a very general belief that the true function is **composed** of simpler functions

Popular deep learning tasks

Supervised learning

- Regression



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

Loss function (mean squared error)

$$L(\theta) = \frac{1}{N}\sum_i^N (NN(x_i;\theta) - y(x_i))^2$$

Supervised learning - regression:

Given a set of example inputs and outputs (labels) $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ from some true function $y(x)$ where $x \in \mathbb{R}^n, y \in \mathbb{R}^m$

Find

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

Probabilistic perspective:

Given a set of example inputs and outputs (labels) $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ drawn from the probability distribution $p(y|x)$

Find

$$\hat{p}(y|x,\theta) \approx p(y|x)$$

Probabilistic perspective:

Assume $\hat{p}(y|x,\theta)$ is a **normal** distribution:

$$\hat{p}(y|x,\theta) = \mathcal{N}(y;\mu = NN(x;\theta), \sigma = 1)$$

Then, assume each training datapoint is independently and identically distributed (iid), then the **data likelihood** can be written as:

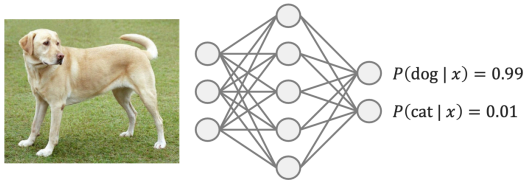
$$\hat{p}(D|\theta) = p(x_1, y_1, \ldots, x_N, y_N|\theta) = \prod_i^N \hat{p}(y_i|x_i,\theta)$$

Then use **maximum likelihood estimation** (MLE) to estimate θ^*:

$$\begin{aligned}\theta^* &= \max_\theta \hat{p}(D|\theta) \\ &= \max_\theta \prod_i^N e^{-\frac{1}{2}\left(\frac{y(x_i) - NN(x_i;\theta)}{1}\right)^2} \\ &= \min_\theta \sum_i^N \left(NN(x_i;\theta) - y(x_i)\right)^2\end{aligned}$$

  - Here, due to the reason that the observed data will be influenced by noise. It is better to use probabilistic perspective.
  - 最大似然估计是一种统计方法，它用来求一个样本集的相关概率密度函数的参数。要在数学上实现最大似然估计法，我们首先要定义可能性：$lik(\theta) = f_D(x_1, x_2, \ldots, x_n|\theta)$。并且在θ的所有取值上，使这个函数最大化。这个使可能性最大的值即被称为θ的最大似然估计。
  - Regression problem corresponds to Continuous PDF, while classfication corresponds to the discrete PDF.

- Classification



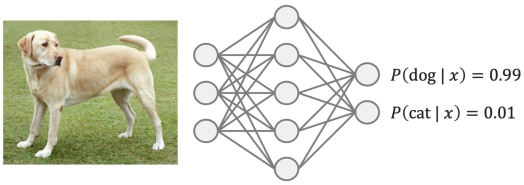
Supervised learning - classification:

Given a set of example inputs and outputs (labels) $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ drawn from the discrete probability distribution $P(y|x)$

where $y \in Y$, for example, $Y = \{\text{dog}, \text{cat}\}$

Find

$$\hat{P}(y|x,\theta) \approx P(y|x)$$



Let each class be encoded as a one-hot vector of length C, e.g

$$y = (0,0,1,0)$$

Typically, we use a softmax output layer to assert $\sum_j^C NN(x;\theta)_j = 1$;

$$\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j^C e^{z_j}}$$

Then assume

$$\hat{P}(y|x,\theta) = \prod_j^C NN(x;\theta)_j^{y_j}, \qquad \sum_j^C NN(x;\theta)_j = 1$$

Then, assume each training datapoint is independently and identically distributed (iid), then the **data likelihood** can be written as:

$$\hat{P}(D|\theta) = \hat{P}(x_1, y_1, \dots, x_n, y_n|\theta) = \prod_i^N \hat{P}(y_i|x_i,\theta)$$

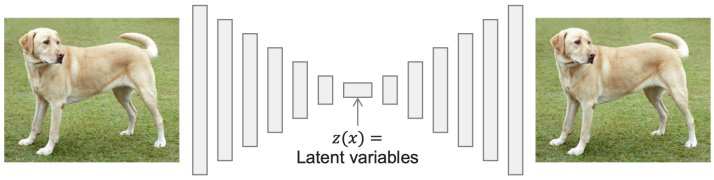
Then use **maximum likelihood estimation** (MLE) to estimate θ^*:

$$\begin{aligned}\theta^* &= \max_\theta \hat{P}(D|\theta) \\ &= \max_\theta \prod_i^N \prod_j^C NN(x_i;\theta)_j^{y_{ij}} \\ &= \min_\theta -\sum_i^N \sum_j^C y_{ij} \log NN(x_i;\theta)_j\end{aligned}$$

Also known as the **cross-entropy loss**

Unsupervised learning

- Feature learning
  -



Loss function

Many different possibilities, a simple choice is

$$L(\theta) = \sum_i^N (NN(x_i;\theta) - x_i))^2$$

Unsupervised learning – feature learning

Given a set of examples $\{x_1, \dots, x_N\}$, find some features $z(x)$

Which are salient descriptors of x, where $x \in \mathbb{R}^n, z \in \mathbb{R}^d$

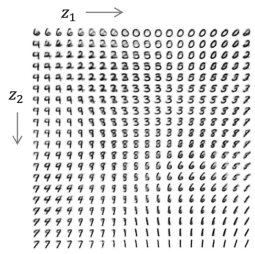
Typically, $d \ll n$ (= compression)

z can be used for downstream tasks, e.g. clustering / classification

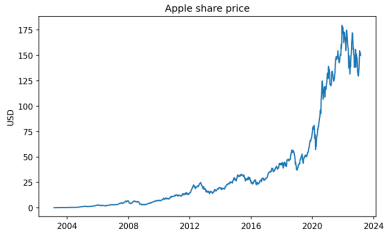
For example:

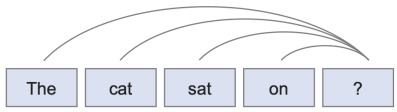
Variational autoencoders (VAEs)

Kingma et al, 2014



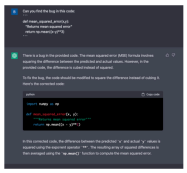
- Autoregression

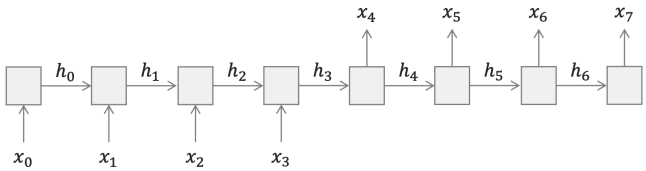




For example:

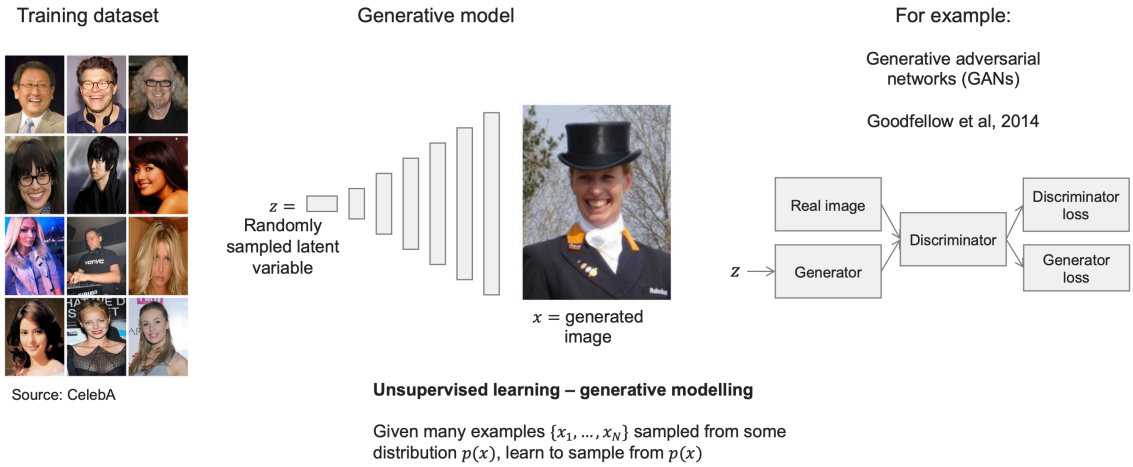
ChatGPT





  - Given many examples sequences, train a model to predict future values from past values

- Generative models



– ◦ High-dimensional Statistics

Training a deep neural network

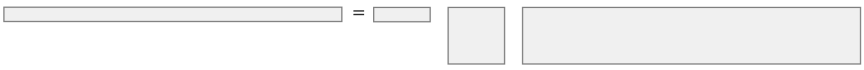
Gradient descent

We have discussed the gradient and chain rule, let's think about dimensionality:

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

Let's think about **dimensionality**. Let $\boldsymbol{f}$ have 1 element (scalar output), $\boldsymbol{g}$ and $\boldsymbol{h}$ both have 100 elements (100 hidden units) and $\boldsymbol{x}$ be a flattened 128 x 128 image, then W_1 has shape (100 x (128 x 128)) = 1.6M elements and

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



(1 x 1.6M) = (1 x 100) (100 x 100) (100 x 1.6M)

⚠ The last Jacobian is extremely large! = 160M elements (or 0.7 GB!)

Forward- versus reverse-mode differentiation

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

(1 x 1.6M) = (1 x 100) (100 x 100) (100 x 1.6M)

Consider evaluating the chain rule (RHS):

1) From right to left **(forward-mode)**

(100 x 100) (100 x 1.6M) → (100 x 1.6M)
(1 x 100) (100 x 1.6M) → (1 x 1.6M)

= lots of computation (large matrix-matrix multiply)

2) From left to right **(reverse-mode)**

(1 x 100) (100 x 100) → (1 x 100)
(1 x 100) (100 x 1.6M) → (1 x 1.6M)

= much less computation (vector-matrix multiplies)

=> Order matters! Typically, reverse mode differentiation is more efficient for ***scalar*** *loss functions with large numbers of parameters ("wide" Jacobians), whilst forward mode is more efficient for evaluating "tall" Jacobians*

- – 这就是自动微分方法的引入背景，我们在前向传播时，会记录下当前参数下各层的数值，则我们可以进行left to right的计算。

- 接着，我们计算发现其实有些层是可以用outer product来进行表示。
  -

But note:

$$\frac{\partial \boldsymbol{h}}{\partial W_1} = \frac{\partial (W_1 \boldsymbol{x} + \boldsymbol{b_1})}{\partial W_1} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n & 0 & 0 & \cdots & 0 \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & \cdots & 0 & x_1 & x_2 & \cdots & x_n \end{pmatrix}$$

Then

$$\frac{\partial NN}{\partial W_1} = \frac{df}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial W_1} = \left(\frac{df}{\partial h_1}(x_1, \cdots x_N), \cdots, \frac{df}{\partial h_{100}}(x_1, \cdots, x_N) \right)$$
$$= \frac{\partial f}{\partial \boldsymbol{h}}^T \otimes \boldsymbol{x}$$

This is just the (flattened) **outer product** of two vectors (100 x 1) ⊗ (16,384 x 1)

⇒ We don't have to fully populate the last Jacobian ($\frac{\partial \boldsymbol{h}}{\partial W_1}$) when computing its vector-Jacobian product

⇒ Computing vector-Jacobian products is usually simpler and **more** efficient than directly computing Jacobians

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

(1 x 1.6M) = (1 x 100) (100 x 100) (100 x 1.6M)

Note:
Training neural networks = Lots of matrix – matrix multiplications

These types of parallelisable operations are perfect for GPUs

10x to 100x speed-ups compared to CPUs

2) From left to right **(reverse-mode)**

(1 x 100) (100 x 100) → (1 x 100)
~~(1 x 100) (100 x 1.6M) → (1 x 1.6M)~~
(100 x 1) ⊗ (16,384 x 1) → (1 x 1.6M)

= Efficient training code

Allows us to train neural networks with **billions** of parameters

  - 这告诉我们不需要完全计算出Jacobian之后再去做矩阵乘法，计算的时候就可以算vector-Jacobian。这会更加简单和高效。

- 因此我们策略如下，反向传播算法

Forward pass:

$$\boldsymbol{x_i} \rightarrow \boldsymbol{h_i} = W_1 \boldsymbol{x_i} + \boldsymbol{b_1} \rightarrow \boldsymbol{g_i} = W_2 \sigma(\boldsymbol{h_i}) + \boldsymbol{b_2} \rightarrow f_i = W_3 \sigma(\boldsymbol{g_i}) + \boldsymbol{b_3}$$

Save layer outputs in forward pass

Backward pass:

$$\frac{\partial L}{\partial W_1} = \sum_i^N 2(f_i - y(\boldsymbol{x_i}))\ W_3\ \text{diag}(\sigma'(\boldsymbol{g_i}))\ W_2\ \text{diag}(\sigma'(\boldsymbol{h_i})) \otimes \boldsymbol{x_i}$$

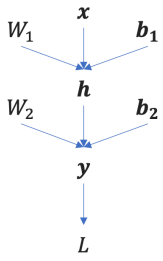
Evaluate from left to right (reverse-mode)

Batch matrix multiplies over training examples $\boldsymbol{x_i}$

Similar equations for other weight matrices and bias vectors

  - 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!
    -

$$y = W_2\sigma(W_1x + b_1) + b_2$$



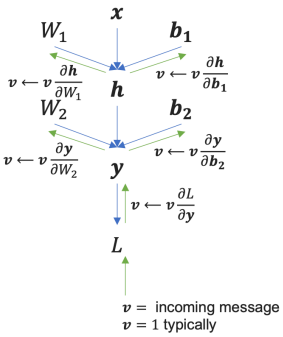
Autodifferentiation = gradients of **arbitrary** programs

Fundamentally, autodifferentiation builds a directed **graph** of **primitive operations**

Each primitive operation defines:
1) **Forward operation**
2) **vector-Jacobian product** (for reverse mode autodiff)
3) **Jacobian-vector product** (for forward-mode autodiff)

◦

$$y = W_2\sigma(W_1x + b_1) + b_2$$

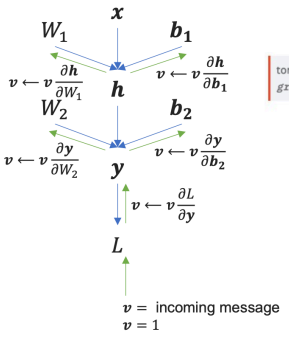


Then, autodifferentiation can be thought of as **message-passing** between the graph's nodes

Each message computes the vjp or jvp

Combined, evaluates the **chain rule**

$$y = W_2\sigma(W_1x + b_1) + b_2$$



loss.backward()

PyTorch

torch.autograd.backward(*tensors*, *grad_tensors=None*, *retain_graph=None*, *create_graph=False*, *grad_variables=None*, *inputs=None*) [SOURCE]

Computes the sum of gradients of given tensors with respect to graph leaves.

The graph is differentiated using the chain rule. If any of `tensors` are non-scalar (i.e. their data has more than one element) and require gradient, then the Jacobian-vector product would be computed, in this case the function additionally requires specifying `grad_tensors`. It should be a sequence of matching length, that contains the "vector" in the Jacobian-vector product, usually the gradient of the differentiated function w.r.t. corresponding tensors (`None` is an acceptable value for all tensors that don't need gradient tensors).

Note autodiff is **not**
- Symbolic differentiation
- Finite differences

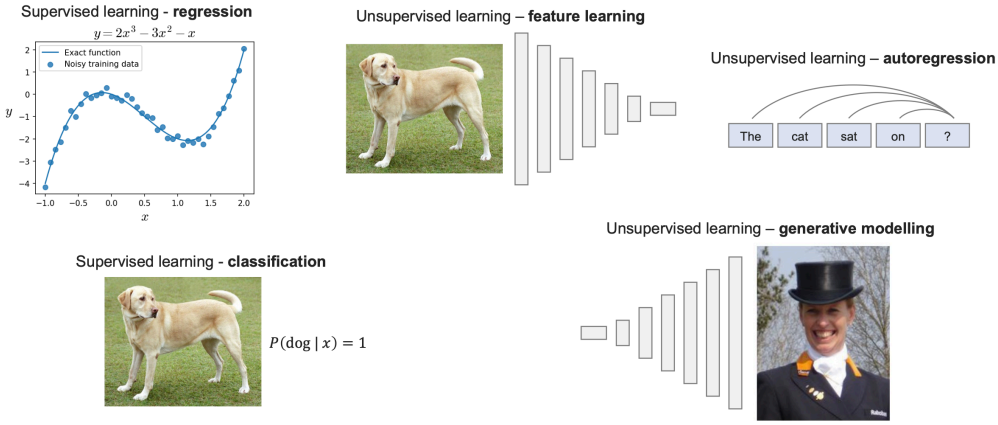
It is a way of efficiently computing exact gradients

◦ ◦ Autodiff是一种高效计算计算图在某点出的gradients的方法，不是基于符号微分（相当于任意点处的微分）。

3 Introduction to Deep Learning

Recap

Popular deep learning tasks



Automatic differentiation

Chain rule can be efficiently evaluated using vector-Jacobian products (**reverse-mode differentiation**)

Lecture Overview

- Challenges of function fitting
  - Overfitting / underfitting
  - Bias / variance
- Regularising deep neural networks
  - Architecture
  - Training data
  - Loss function
- Optimising deep neural networks
  - Stochastic gradient descent
  - Adam / higher-order
- State-of-the-art models
  - Transformers, ChatGPT

Error Sources

Training loss:

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

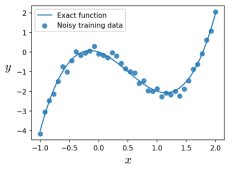
This is also known as the **empirical loss**, and can be more generally written as:

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

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

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

But this is impossible in practice!



This means there are two sources of error:

1) **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

2) **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

Regularization

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

4 PINN

curse of dimensionality:

- Conventional numerical methods (FD): error: $\varepsilon = O(\delta x^2)$ $\#(\text{pts}) = (\frac{1}{\delta x})^{d+2} \Rightarrow \delta x = (\#\text{point})^{d+2}$
  - which means that if we increase the dimension of the problem. If we don't change the sampling points, the error of the problem will increase exponentially. Conservely, if we want to ensure the resolution/accuracy, the increased work will grow exponentially. curse of dimensionality
- PINN
  - PINN with Depth 4, Width 20, Interior training points 2^{16}, Boundary points 2^{15}

Dimension	Training Error	Generalization error
1	2.8×10^{-5}	0.0035%
5	0.0002	0.016%
10	0.0003	0.03%
20	0.006	0.79%
50	0.006	1.5%
100	0.004	2.6%

    - sublinearly growth.

5 PINN - applications

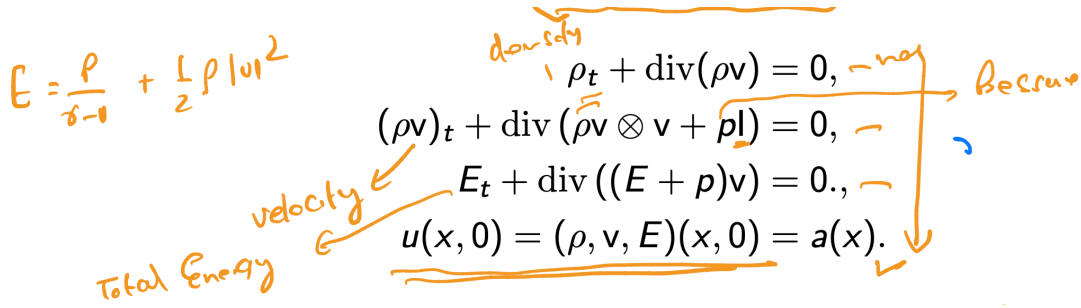
6 PINN

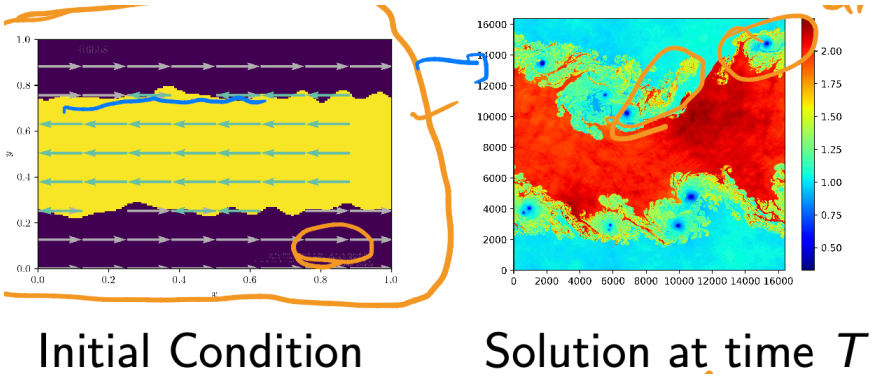
7 Operator Learning

7.1 Introduction

What does solving a PDE mean? -> find the solution operator

Two example:

- Darcy PDEs : $-\text{div}(\mathbf{a}\nabla u) = f$ solution operator $\mathcal{G} : \mathbf{a} \to \mathcal{G}\mathbf{a} = u$
  - prototypical elliptic PDE (time-independent)
- Compressible Euler equations
  -


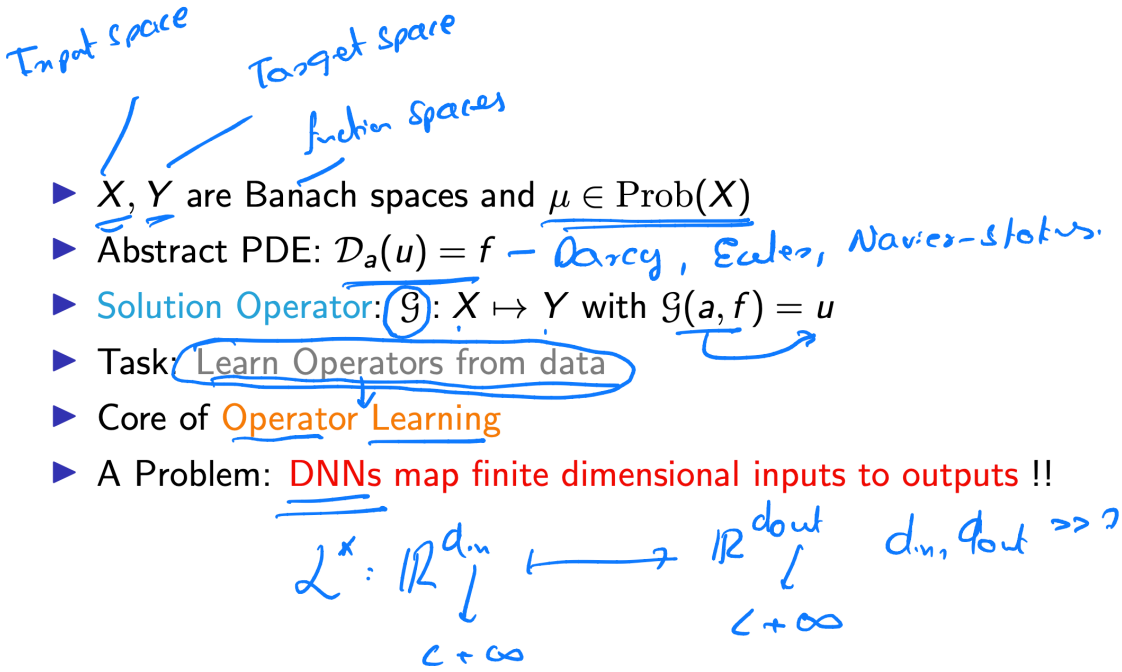
  - Solution operator $\mathcal{G} : \mathbf{a} \to \mathcal{G}\mathbf{a} = u(T)$.
  -


  - Learn the evolution operator, input $u(T) \to \mathcal{G}u(T) = u(2T)$

在time-evolution PDE中，建立的还是figure到figure的一个映射，而不是figure到video的一个mapping。

Setup

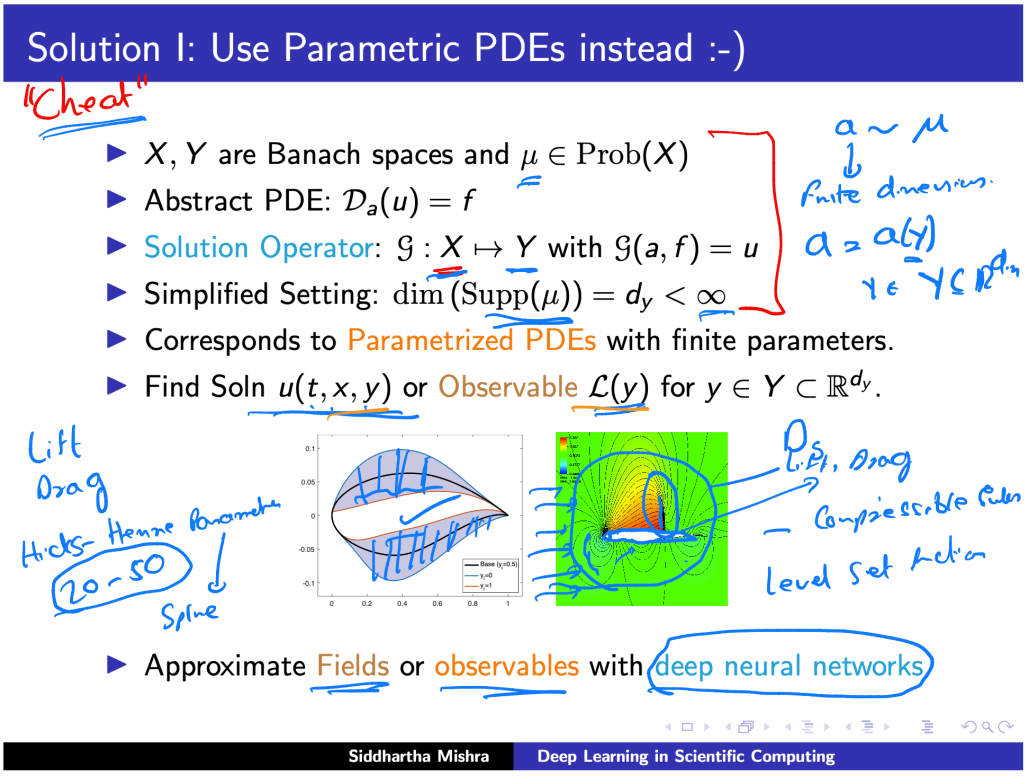
这里考虑的一类问题是，当我们有很多的数据时，我们怎么样求解我们的PDE？这里给出的求解PDE的本质是find a solution operator。因此，我们的Setup如下：



但是，我们想用NN学习这个无穷维到无穷维的映射很困难，因此，我们可以考虑如下解决方案：

Solution I: Use Parametric PDEs instead

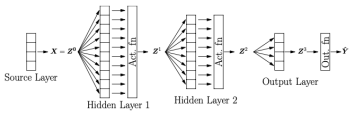
也就是将无穷维的function space参数化（基于一些basis），然后无穷维的function space reduced into a finite space.。接着，学习这些参数到solution的映射（有限维到有限维）。这种方法我们称为“cheat”：



- 这种方法可行的理论背景是函数的fourier basis展开。此外，这个思路和有限元很像，有限元也是Galerkin Discretization之后，将解reduced到一个finite space上去。
- 但是，即使这种parametric (reduced space)的方法也会存在问题，这面我们会介绍。

当我们使用传统的supervised learning的时候，我们通常有以下setup：

Supervised learning of target $\mathcal{L}$ with Deep Neural networks



- $\mathcal{L}^*(z) = \sigma_o \odot C_K \odot \sigma \odot C_{K-1} \ldots\ldots\ldots \odot \sigma \odot C_2 \odot \sigma \odot C_1(z)$.
- At the k-th Hidden layer: $z^{k+1} := \sigma(C_k z^k) = \sigma(W_k z^k + B_k)$
- Tuning Parameters: $\theta = \{W_k, B_k\} \in \Theta$,
- σ: scalar Activation function: ReLU, Tanh
- Random Training set: $\mathcal{S} = \{z_i\}_{i=1}^N \in Z$, with i.i.d z_i
- Use SGD (ADAM) to find $\mathcal{L} \approx \mathcal{L}^* = \mathcal{L}^*_{\theta^*}$

$$\theta^* := \arg\min_{\theta \in \Theta} \sum_{i=1}^{N} |\mathcal{L}(z_i) - \mathcal{L}^*_\theta(z_i)|^p,$$

Siddhartha Mishra | Deep Learning in Scientific Computing

- For high dimensional input, we will often employ random training set.

但是在理论上，这会引起一些问题：

Supervised learning for high-d Parametric PDEs

- Can we find DNN such that $\|\mathcal{L}^* - \mathcal{L}\| \sim \mathcal{O}(\epsilon)$?
- YES: Universal Approximation Property of DNNs $\Rightarrow$:
- Given any Continuous (measurable) $\mathcal{L}$, exists a $\hat{\mathcal{L}}$:

$$\|\mathcal{L} - \hat{\mathcal{L}}\| < \epsilon$$

- If $\mathcal{L} \in W^{s,p}$, $\exists$ DNN $\hat{\mathcal{L}}$ with M parameters (Yarotsky):

$$\|\mathcal{L} - \hat{\mathcal{L}}\|_p \sim \mathcal{O}\left(M^{-\frac{s}{\bar{d}}}\right).$$

- But in Scientific Computing (often):
- $\mathcal{L}$ is not be very Regular and $\bar{d} >> 1$
- If $\mathcal{L} \in W^{1,\infty}$, $\bar{d} = 6$: 1% error, need network of size 10^{12} !!
- Curse of dimensionality: DNN Size $M \sim \epsilon^{-\frac{\bar{d}}{s}}$

Siddhartha Mishra | Deep Learning in Scientific Computing

- 收敛速度由target function的smoothness和integrability来决定。首先就是你如果想控制你的error，你的神经网络得非常非常大。当你的输入维度比较多的时候，所需要的parameters will expontentially increase.

下面，我们考虑的是我们能否refined our error estimates，然后在一些情况下，误差的某些部分能够被控制，我们则专注于提升另一部分。这就是下面这一部分：

Refined Error Estimates

- Error $\mathcal{E} := \|\mathcal{L} - \mathcal{L}^*\|_p$ Decomposition: $\mathcal{E} \leq \mathcal{E}_{app} + \mathcal{E}_{gen} + \mathcal{E}_{opt}$.
- Approximation error $\mathcal{E}_{app} = \|\mathcal{L} - \hat{\mathcal{L}}\|_p$,
  - $\hat{\mathcal{L}}$ is best approximation of $\mathcal{L}$ in $\mathcal{NN}(M)$.
  - One can prove that $\mathcal{E}_{app} \sim \mathcal{O}\left(d^\sigma M^{-\eta}\right)$ for,
  - Linear Elliptic PDEs: (Schwab, Kutyniok et al).
  - Semi-linear Parabolic PDEs: (E, Jentzen et al).
  - Nonlinear Hyperbolic PDEs: (DeRyck, SM, 2021).
- Optimization Error $\mathcal{E}_{opt} \sim$ Computable Training error:
- Generalization Error

$$\mathcal{E}_{gen}(\theta) := \|\mathcal{L} - \mathcal{L}^*_\theta\|_p^p - \frac{1}{N}\sum_i |\mathcal{L}^*_\theta(y_i) - \mathcal{L}(y_i)|^p$$

- Using Concentration inequalities + Covering number bounds:

$$\mathcal{E}_{gen} \sim \frac{C\left(M, \log(\|W\|)\right)\log(\sqrt{N})}{\sqrt{N}}$$

Siddhartha Mishra | Deep Learning in Scientific Computing

- 我们将error归结为approximation error，generalization error和optimization error的和。
- 这个地方不是很明白三种error所表达的意思

好像我们的重点在generalizatoin error上面：

Well-trained Networks

- Assume we can find DNN such that $\mathcal{E}_{app}, \mathcal{E}_T << 1$
- Still Overall Error behaves as

$$\mathcal{E} \sim \frac{C(M)}{N^{\alpha}}, \quad \alpha \leq \frac{1}{2}.$$

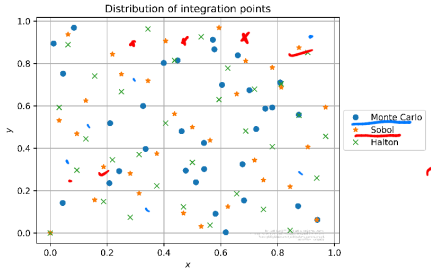
- If $C(M) \sim \mathcal{O}(1)$, error of 1% requires 10^4 training samples !!
- Challenge: learn maps of low regularity in a data poor regime
- Contrast with Big Data successes of machine learning.

- 而generalization error我们也会遇到一个问题，就是分母是$\sqrt{N}$。意味着我们想要提升1%的error需要多10^4个数据。而在scientific computation中，获得一个数据是非常耗时的。
- 因此，有没有什么trick。这里老师提了一种方法，也就是我们知道我们要学习的是pde，而这些pde不是完全随机的，我们可以有一些先验知识在里面，基于这些先验知识，我们可以借助数值分析里面的一些手段，让我们更好地学习模型。
  - 这里采用的是在采样上做变化。这里提到一种方法，就是使用low discrepancy sequences来作为我们的traing dataset，我们可以理论上将我们的error随N的变化控制的更小：
  -

A Trick: Lye, SM, Ray, 2020

Tricks from Numerical Analysis.

- Use Low discrepancy sequences $\{y_i\}_{i=1}^N \in Y$ as Training Set



$\int f(y)\,dy \approx \frac{1}{m}\sum f(y_i)$

$y_i \to$ LDS

- These sequences are Equidistributed (better spread out).
- Examples: Sobol, Halton, Owen, Niederreiter ++
- Basis of Quasi-Monte Carlo (QMC) integration.

quasi-random number generator

Training on Low-Discrepancy Sequences

$\int_0 \left|\frac{\partial^d \varphi}{\partial y_1 \dots \partial y_d}\right| dy \quad c < \infty$

- For $\mathcal{L}$ with Bounded Hardy-Krause variation and smooth σ.
- Generalization Error for Sobol sequences: (SM, Rusch, 2020),

$$\mathcal{E} \leq \mathcal{E}_T + C\left(V_{HK}(\mathcal{L}), V_{HK}(\mathcal{L}^*)\right)\frac{(\log N)^d}{N}, \qquad \frac{1}{N}$$

$\mathcal{E}_T$: Training Error

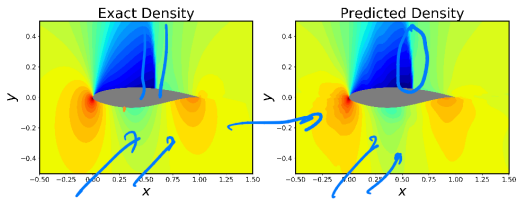
Random: $\mathcal{E} \leq \mathcal{E}_T + \frac{C(\mathcal{L})}{\sqrt{N}}$ $\frac{1}{N}$

– 通过下面这个例子我们也可以看到相比于random sampling的优势：

  -

- Given Hicks-Henne parameter: Predict Drag, Lift, Flow
- DNN with 10^3 – 10^4 parameters and 128 training samples :

	Run time (1 sample)	Training	Evaluation	Error
Lift	2400 s	700 s	10^{-5} s	0.78%
Drag	2400 s	840 s	10^{-5} s	1.87%
Field	2400 s	1 hr	0.2 s	1.9%



- Errors with Random Training pts: Lift 8.2%, Drag: 23.4%

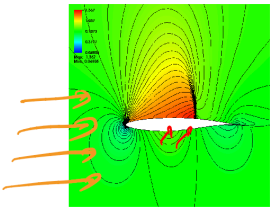
然后，当我们训练好一个surrogate model之后，我们可以做downstream task，这里列举了两个：

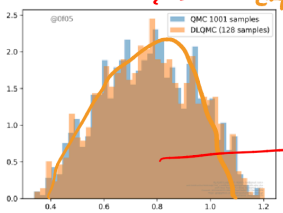
- UQ：

Forward UQ

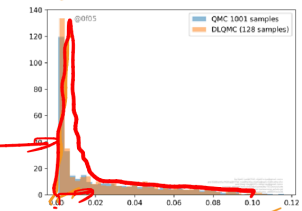
- DL-UQ algorithm of Lye,SM,Ray, 2020 is

$$\mathcal{L}\#\mu \approx \frac{1}{M}\sum_{i=1}^{M}\delta_{\mathcal{L}(y_i)} \approx \frac{1}{M}\sum_{i=1}^{M}\delta_{\mathcal{L}^*(y_i)}$$


Sample



Lift PDF



Drag PDF

Observable	Speedup (MC)	Speedup (QMC)
Lift	246.02	6.64
Drag	179.54	8.56

- Inverse design （见tutorial）

7.2 Operator Networks and DeepONet

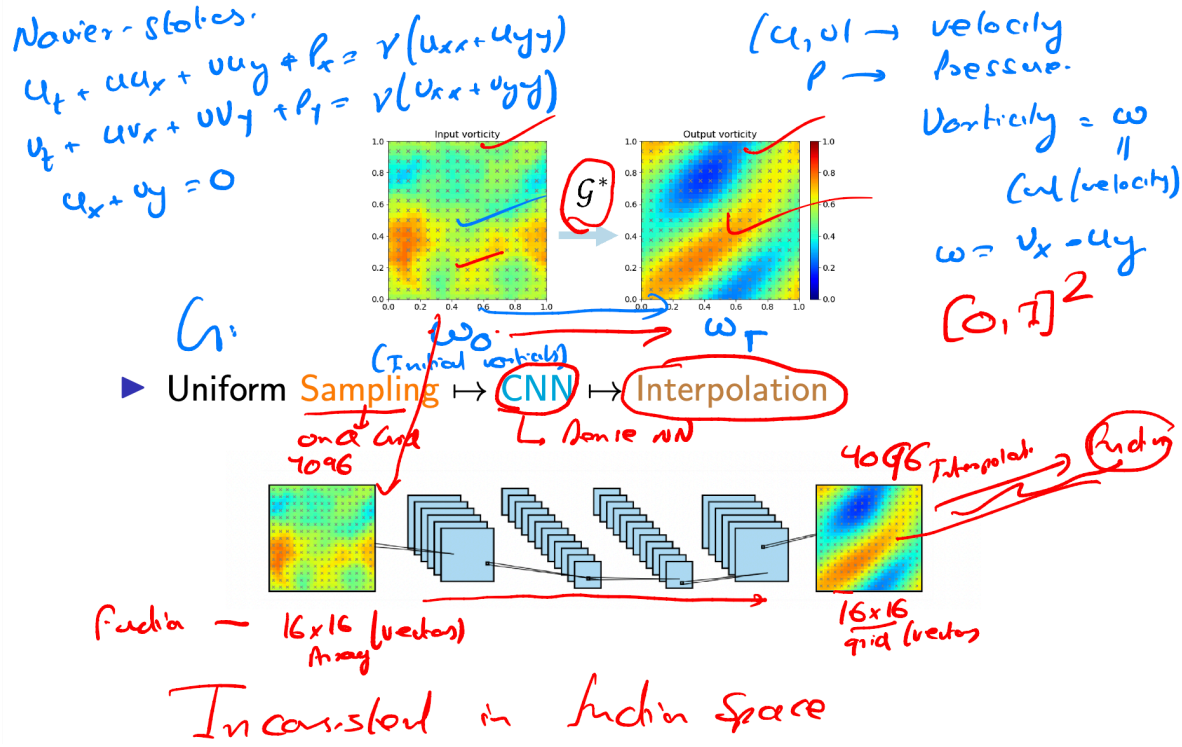
Issue with Parametrization

$\mathcal{G} : x \to y$ (PDE solution operator) $\dim(\text{supp}(\mu)) = d_y < \infty, \mu \in \text{Proab}(x)$

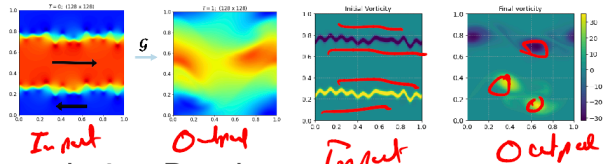
- Difficult to come up with a suitable one → expert domain knowledge
- Not unique
- Does not Generalize well

Solution I: Just use DNN + Interpolation

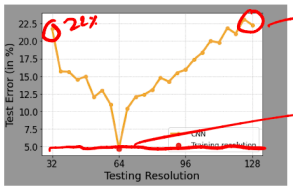
Uniform Sampling → CNN → Interpolation



- Shear flow with Navier-Stokes with $Re >> 1$



- CNN + Interpolation Results:

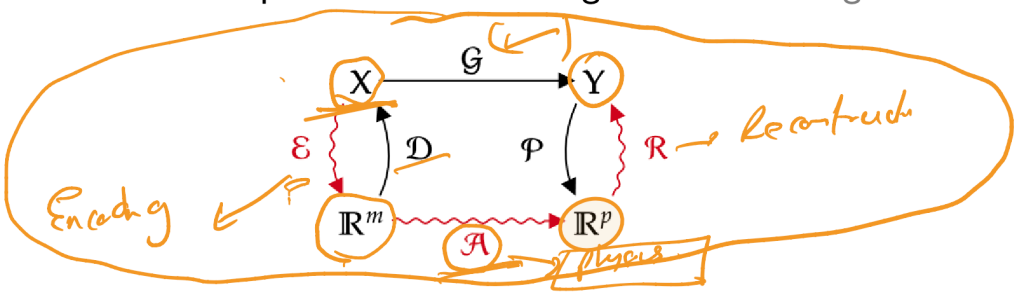


- Consistent with Zhu,Zabaras, 2019.
- Desiderata for Operator Learning:
  - Input + Output are functions.
  - Some notion of Continuous-Discrete Equivalence

- This method didn't work, just learning the mapping from 64×64 → 16×16, not a operator learning, don't have the resolution invariance.
- Practice - discrete representation. Therefore, we have to satisfy the continuous-discrete equivalence.

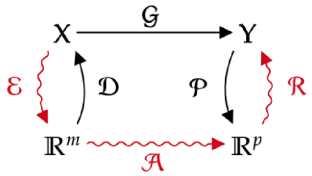
Revisiting Numerical Methods

- Can be reinterpreted in the following abstract Paradigm:



Scheme	Encoder	Approximator	Reconstructor
Finite Difference	Point values	Scheme	Poly. Interpolant
Finite Element	Node Values	Scheme	Galerkin Basis
Finite Volume	Cell Averages	Scheme	Poly. Interpolant
Spectral	Fourier Coeffs.	Scheme	Fourier Interpolant

Other OpLearn Architectures

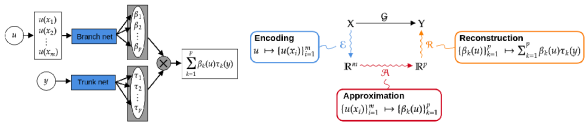


Architecture	Encoder	Approximator	Reconstructor
DeepOnet	Sensor Evals.	DNNs	DNNs
PCA-Net[1]	Input PCA	DNNs	Output PCA
SNO[2]	Coeffs	DNNs	Chebychev basis

DeepONets

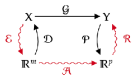
- DeepONets: Chen,Chen 1995, Lu et al, 2020:

$$\mathcal{N}(a)(y) = \tau_0(y) + \sum_{k=1} \beta_k(a)\tau_k(y) \approx \mathcal{G}(a)(y)$$



Why do ONet work?

ONets will face the issue of **curse of dimensionality**.



- Universal Approximation Thm: For $\mu \in Prob(L^2(D))$ and any measurable $\mathcal{G} : H^r \mapsto H^s$ and $\epsilon > 0$, $\exists \mathcal{N}$ (Onet): $\hat{\mathcal{E}} < \epsilon$
- Upper (and lower) bounds via $\mathcal{E}_{\mathcal{R}} \leq \mathcal{E} \leq C(\mathcal{E}_{\mathcal{E}} + \mathcal{E}_{\mathcal{A}} + \mathcal{E}_{\mathcal{R}})$.
- $\mathcal{E}_{\mathcal{E},\mathcal{R}}$ decay as spectrum of Covariance operator of μ and $\mathcal{G}\#_{\mu}$.
- For $G = \mathcal{P} \circ \mathcal{G} \circ \mathcal{D} \in C^k(\mathbb{R}^m, \mathbb{R}^p) \Rightarrow$ size $(\mathcal{N}) \sim \mathcal{O}\left(\epsilon^{-\frac{m(\epsilon)}{k}}\right)$.
- Curse of Dimensionality (CoD) for Onets !!!

- covariance operator可以简单理解为函数更容易通过几个domaint basis来进行展开

However, when the operator is not random and comes from the PDE, DeepONet breaks the **curse of dimensionality**.

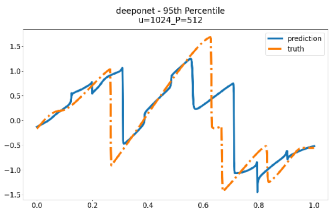
- For operators $\mathcal{G}$ corresponding to many PDEs:

PDE	Operator	Complexity
$-u'' = \sin(u) + f$	$\mathcal{G} : f \to u(T)$	$M \sim \mathcal{O}(\epsilon^{-\eta})$ $\eta \approx 0$
$-\mathrm{div}(a\nabla u) = f$	$\mathcal{G} : a \to u$	$M \sim \mathcal{O}(\epsilon^{-\eta})$ $\eta \approx 0$
$u_t = \Delta u + f(u)$	$\mathcal{G} : u_0 \to u(T)$	$M \sim \mathcal{O}(\epsilon^{-2(d+1)})$
$u_t + u \cdot \nabla u + \nabla p = \nu \Delta u$	$\mathcal{G} : u_0 \to u(T)$	$M \sim \mathcal{O}(\epsilon^{-(d+1)})$
$u_t + \mathrm{div}(f(u)) = 0$	$\mathcal{G} : u_0 \to u(T)$	$M \sim \mathcal{O}(\epsilon^{-\alpha(d+1)})$

- Case by Case basis: Operator Holomorphy, Emulation of Numerical Schemes etc...
- Unified framework in DeRyck, SM, 2022.

Issues

- Affine Reconstructors $\Rightarrow \sqrt{\sum_{\ell>p} \lambda_\ell} \leq \mathcal{E}_{\mathcal{R}} \leq \mathcal{E}$
- Large error for Slow decay of eigenvalues of Covariance operator of $\mathcal{G}\#_\mu$.
- Holds for Transport dominated problems
- Error of 30% for Burgers' equation with GRF initial data !!



- Upper bound的问题是issue。

⇓ Therefore, alternative is neural operator

CNO

Sampling theorems: tell you that if your function has a certain range of frequencies then you need so much of resolution in physical space you need so many great points. So if it has 10 frequencies you need roughly 20 grid points. Anythings more than 23 points can resolve it if not then you will get a aliasing error.

7.3 Lecture 10

Three important points in using DeepOnet for approximation:

- Covariance Operator
- Curse of dimensionality
- The curse of dimensionality can be alleviated by leveraging the structures in PDE

Covariance Operator

The performance of the operator network depends on how fast or how slow the eigenvalues of this object(covariance operator) decay.

After knowing the issues of these Operator Networks, we want to find some alternatives, One possible solution is neural operators.

Because we need to calculate the integral, therefore the computational cost is super large. In order to reduce the complexity, we introduce the fourier operator.