Deep Learning for Science and Engineering Teaching Kit

Deep Learning for Scientists and Engineers

DAY 1: PINNs and PIKANs

Instructors: George Em Karniadakis, Khemraj (Raj) Shukla

ourse Roadmap

ecture 1: PINNs and PIKANs: 90 Minutes

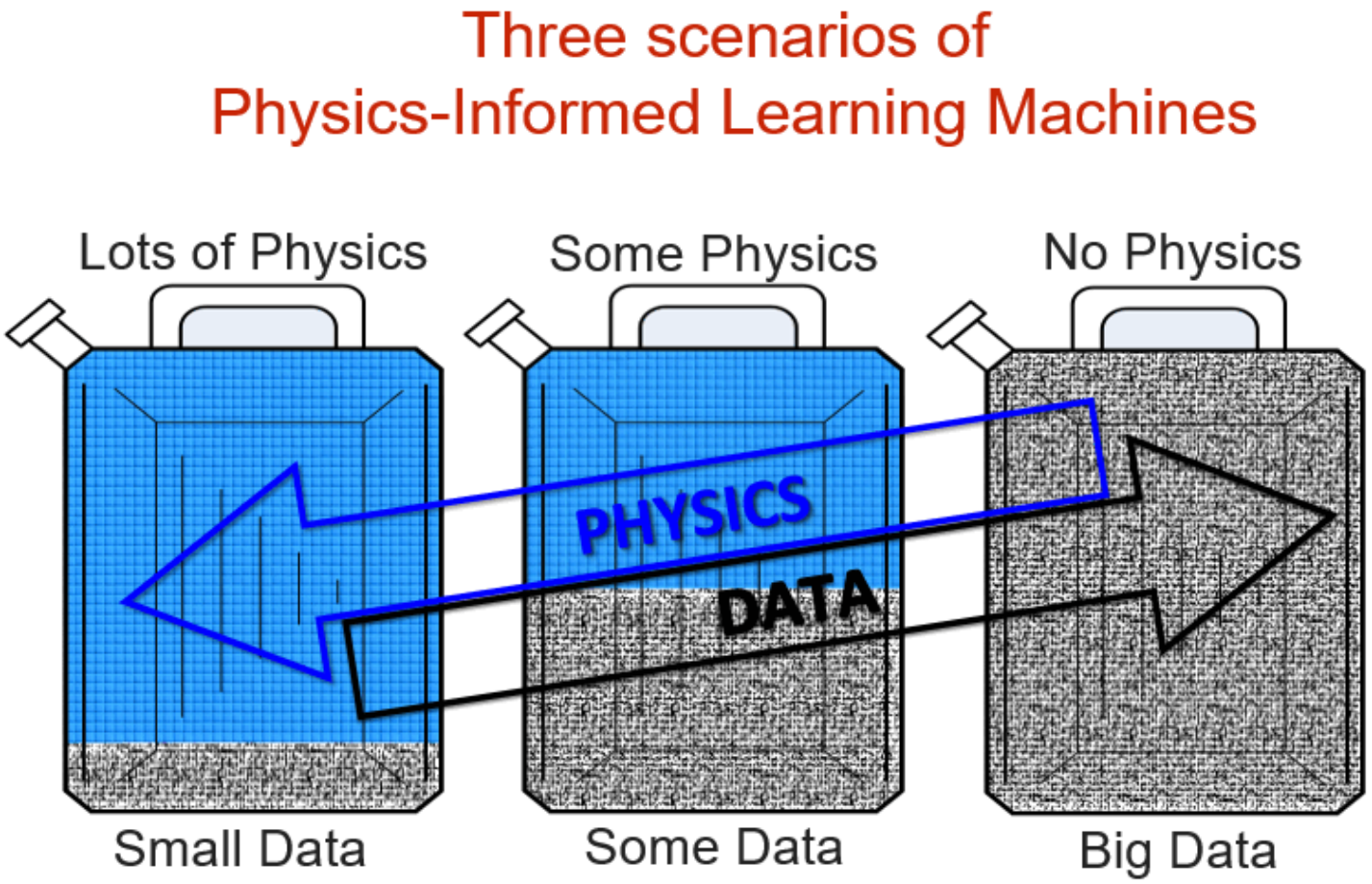
ecture 1: Hands On: 90 Mins

ecture 2: Neural Operators: 105 Minutes

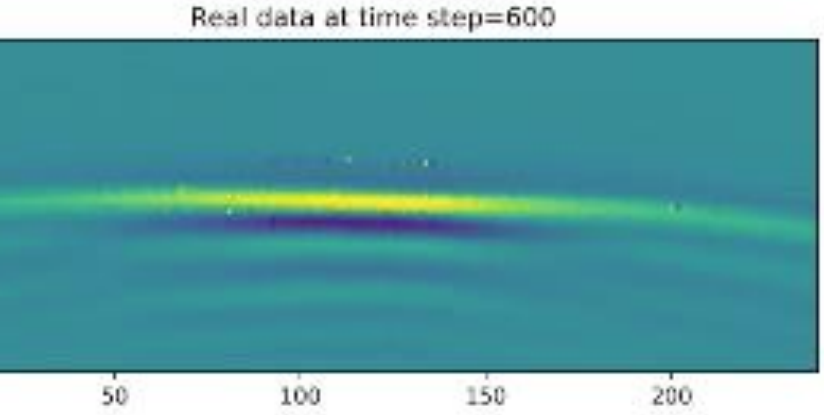
ecture 2: Hands On: 75 Mins

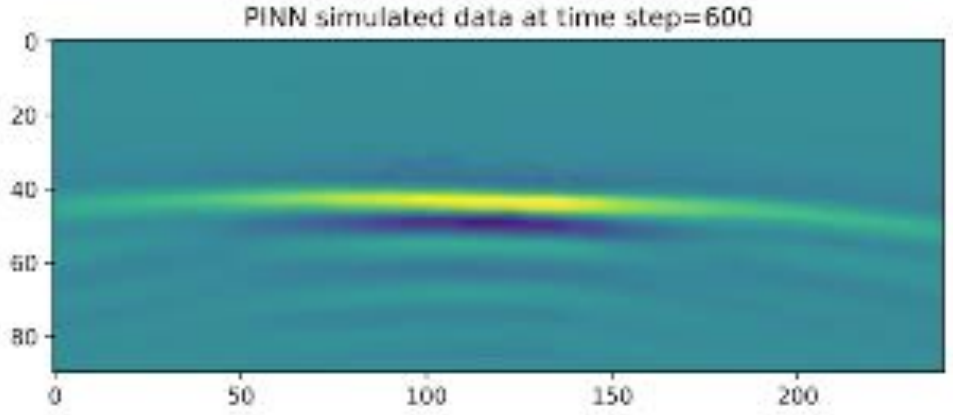
Contents

Data + Physical Laws

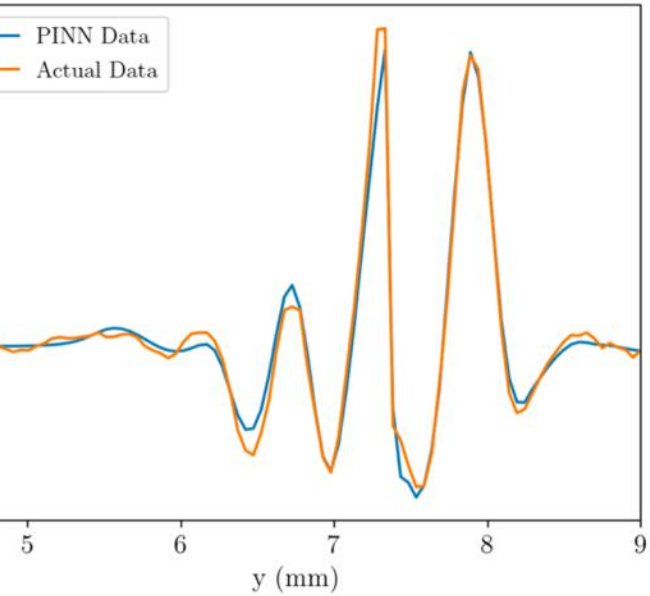


Crack characterization: 0° Pre-crack + Post-crack + $v(x, y)$

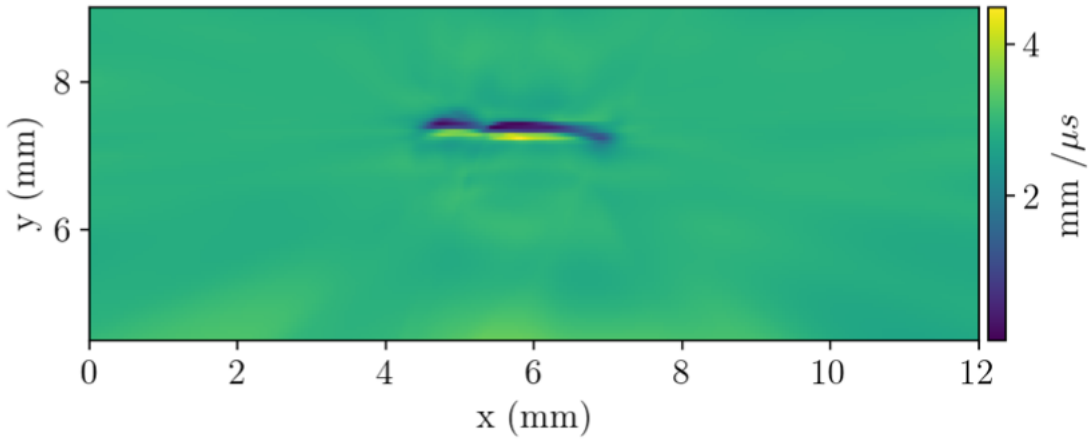




$$f := \frac{\partial^2 \Psi_{\mathrm{NN}}}{\partial t^2} - c^2$$



ces of (a) and (b) at $x = 6$ mm at $t = 12.38$ μs.



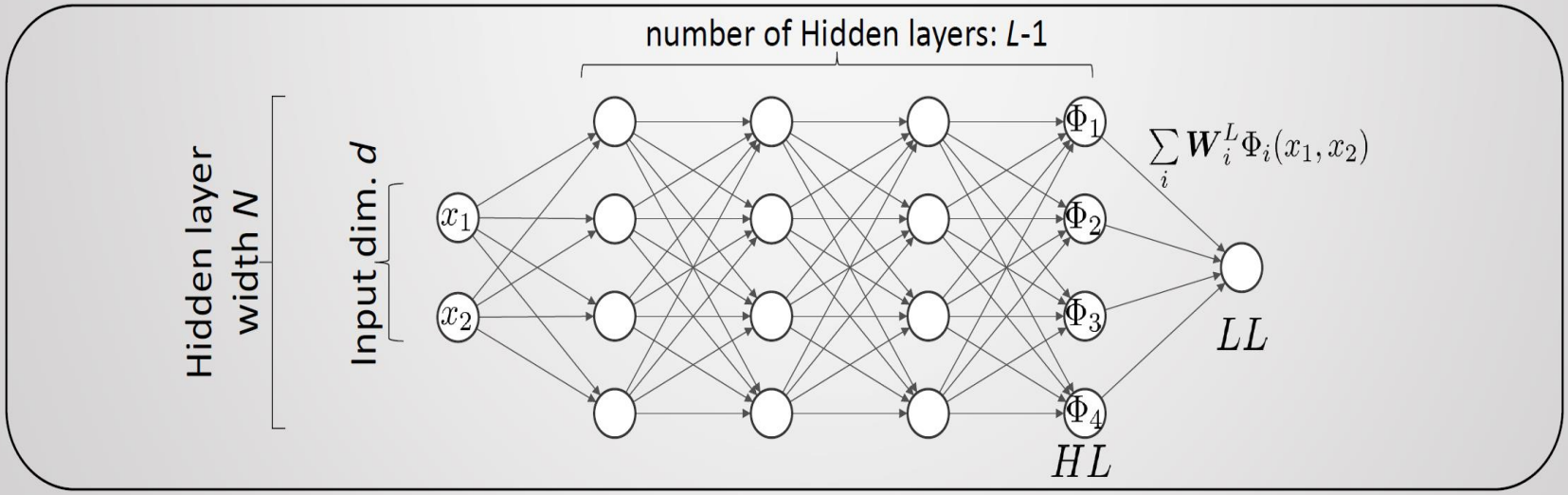
(d) Speed $v(x, y)$ recovered from PINN simulation.

Adaptive Basis Viewpoint: What is a Neural Network?

- We consider a family of neural networks $\mathcal{N}_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}$ consisting of $L-1$ hidden layers of width N composed with a final linear layer, admitting the representation

$$\mathcal{N}_{\boldsymbol{\theta}}(x) = \sum_{i=1}^{N} \boldsymbol{W}_i^L \Phi_i(x; \boldsymbol{\theta}_H)$$

where $\boldsymbol{W}^L$ and $\boldsymbol{\theta}_H$ are the parameter corresponding to the final linear layer and the hidden layers respectively. We interpret $\boldsymbol{\theta}$ as a concatenation of $\boldsymbol{W}^L$ and $\boldsymbol{\theta}_H$.



- This view point makes it clear that $\boldsymbol{\theta}_H$ parameterizes the basis (like FEM mesh & Shape functions), while $\boldsymbol{W}^L$ are just coefficients for these basis functions.

Connecting Neural Networks with Finite Elements (FEM) – 1D

Recall the hat function

$$\varphi(x) = \begin{cases} 2x & x \in \left[0, \frac{1}{2}\right], \\ 2(1-x) & x \in \left[\frac{1}{2}, 1\right], \\ 0, & \text{otherwise.} \end{cases}$$

$$\varphi(x) = 2\,\mathrm{ReLU}(x) - 4\,\mathrm{ReLU}(x - \frac{1}{2}) + 2\,\mathrm{ReLU}(x-1).$$

- Linear finite element basis function φ_i

$$\varphi_i = \varphi\left(\frac{x - x_{i-1}}{2h}\right) = \varphi\left(w_h x + b_i\right), \quad w_h = \frac{1}{2h}, \quad b_i = \frac{-(i-1)}{2}.$$

$$\varphi_i \in \mathrm{Span}\left\{\mathrm{ReLU}(wx+b), w, b \in \mathbb{R}\right\}$$

- The Linear FEM space in 1D is a subspace of shallow neural networks with the ReLU activation function.

Connecting Neural Networks with Finite Elements (FEM) - 2D/3D

$\diamondsuit$ For $d \geq 2$, the linear FEM space is not a subspace of shallow ReLU networks (He et al. 2020)

$$\left\{ \sum_{i=1}^{n} a_i \, \mathrm{ReLU}(w_i^T x + b_i), w_i \in \mathbb{R}^{1\times d}, \, a_i, b_i \in \mathbb{R} \right\}$$

$\diamondsuit$ For $d \geq 2$, the linear FEM space is a subspace of deep ReLU networks (Arora et al. 2016).

- Arora R, Basu A, Mianjy P, Mukherjee A. Understanding deep neural networks with rectified linear units. arXiv preprint arXiv:1611.01491. 2016.
- He J, Li L, Xu J, Zheng C. ReLU deep neural networks and linear finite elements. arXiv preprint arXiv:1807.03973. 2018 Jul 11.

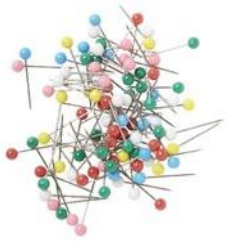
Data + Neural Networks + Physical Laws

PINNs 

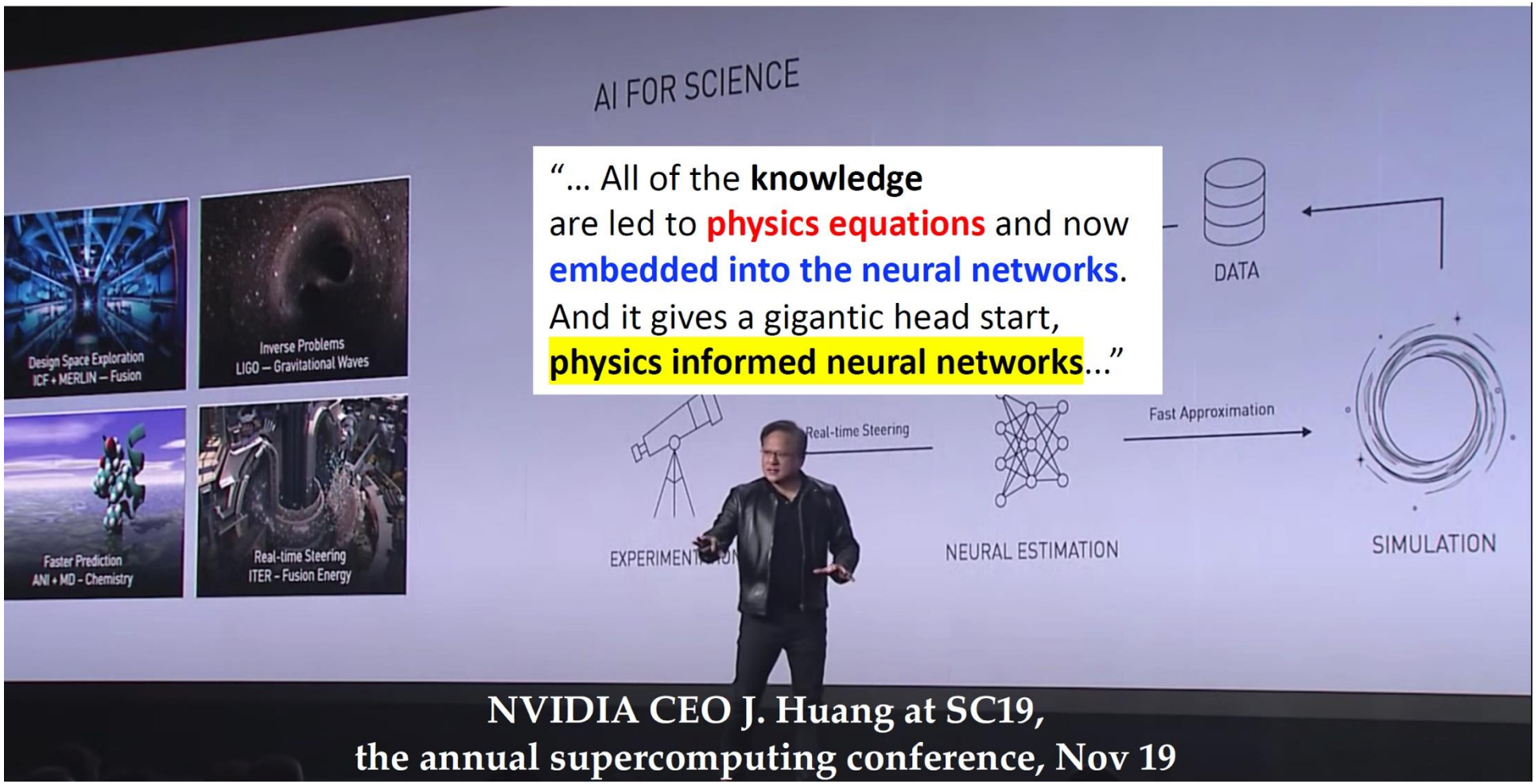
Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations

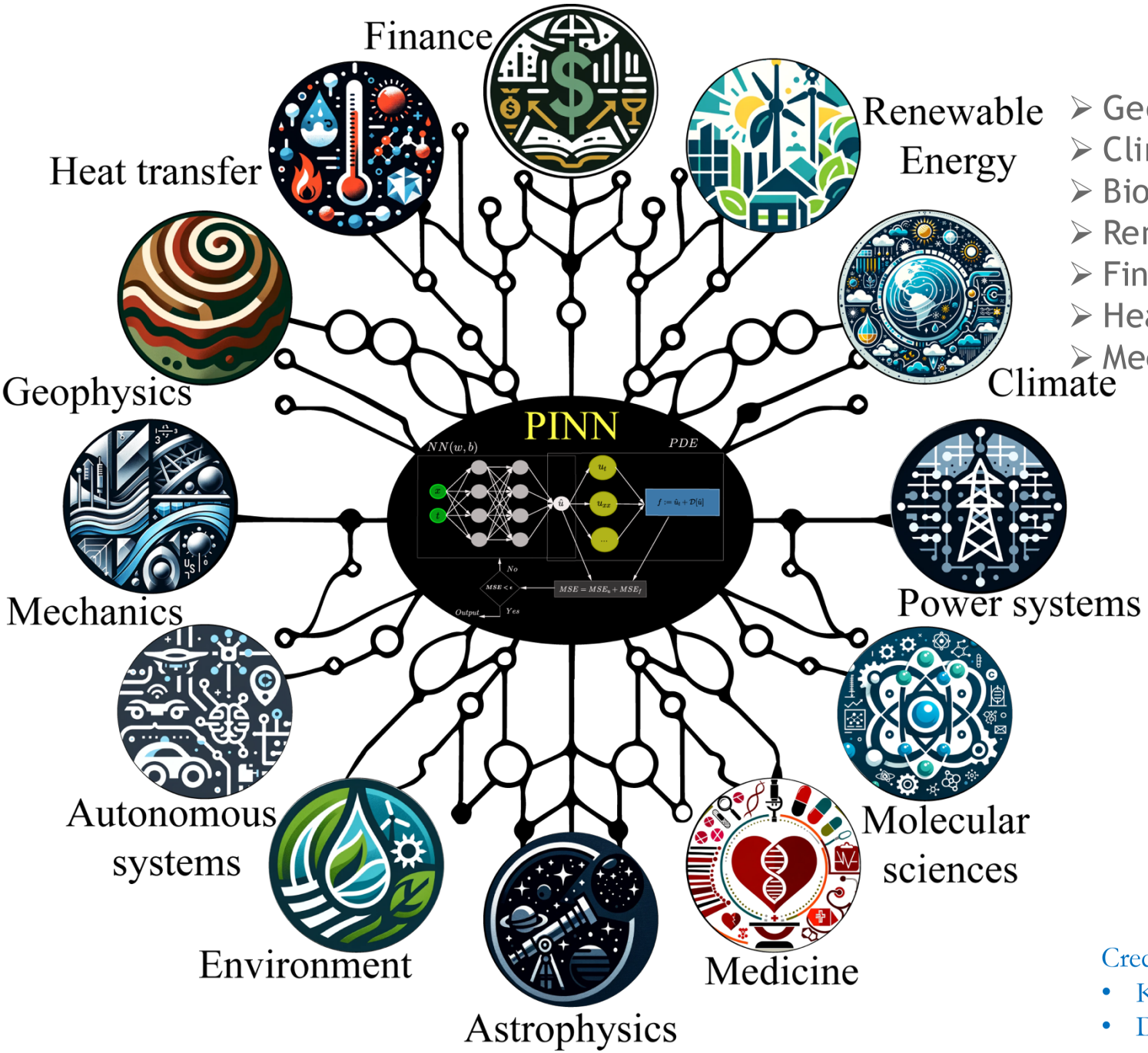
M Raissi, P Perdikaris, GE Karniadakis
Journal of Computational Physics 378, 686-707, 2019

arXiv:1711.10561; arXiv:1711.10566 (2017)



PINNs: Physics-Informed Neural Networks





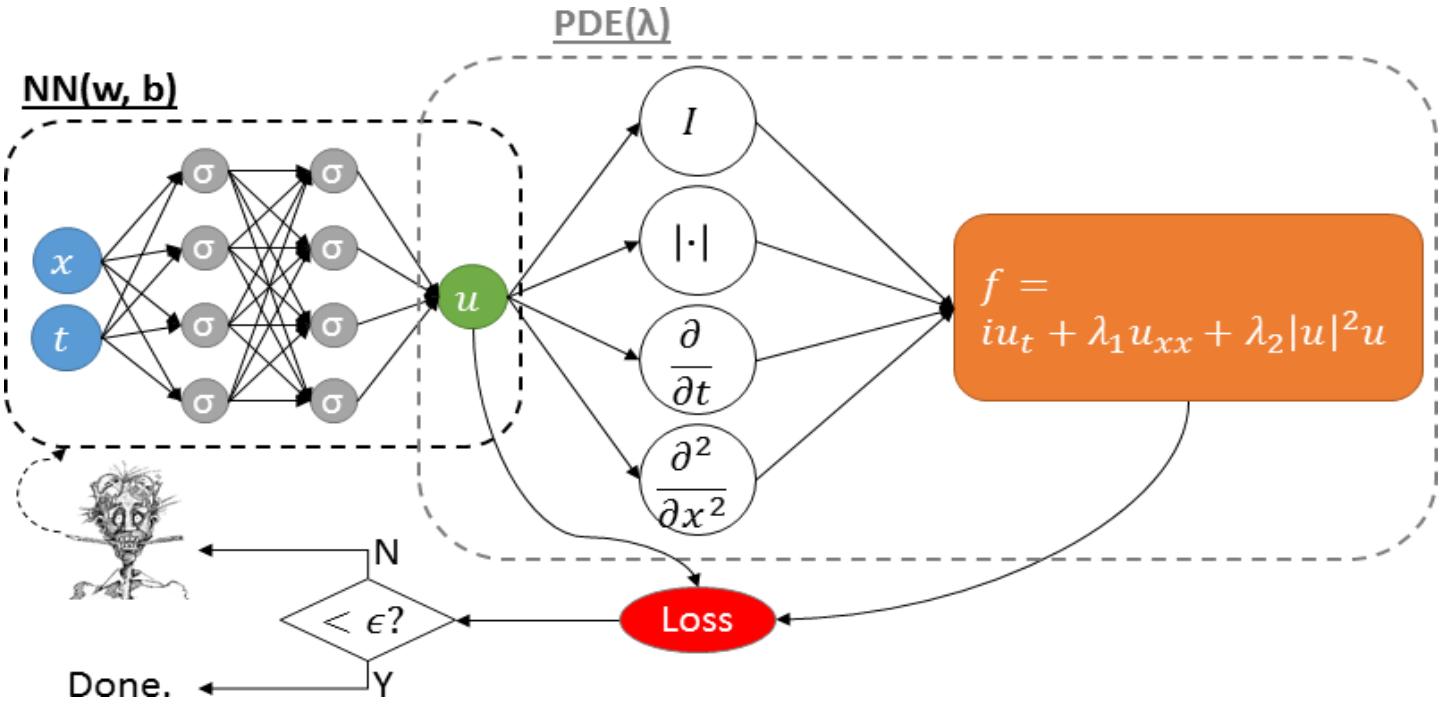
- Geophysics ~ 958 papers
- Climate ~ 1100
- Biophysics ~ 952
- Renewable Energy ~ 440
- Finance ~ 743
- Heat transfer ~ 1430
- Mechanics ~ 4870

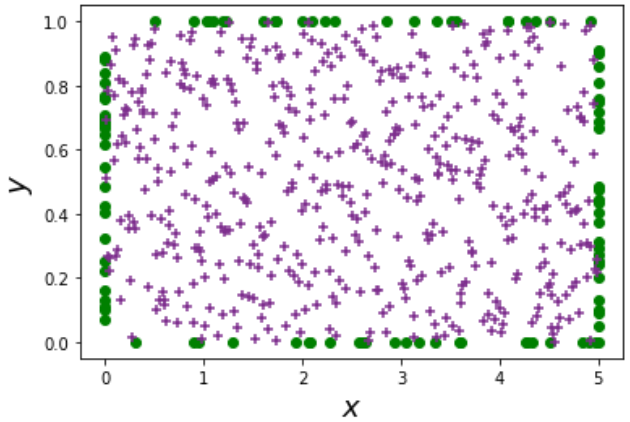
Credits

- Kateryna Morozovska
- DALL-E

What is a PINN? Physics-Informed Neural Network

We employ two (or more) NNs that share the same parameters



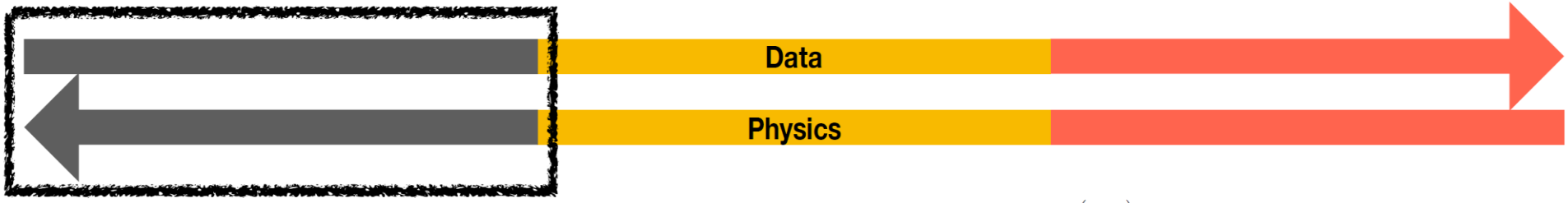


- Minimize:

$$MSE = MSE_u + MSE_f,$$

$$MSE_u = \frac{1}{N_u}\sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2,$$

$$MSE_f = \frac{1}{N_f}\sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2.$$



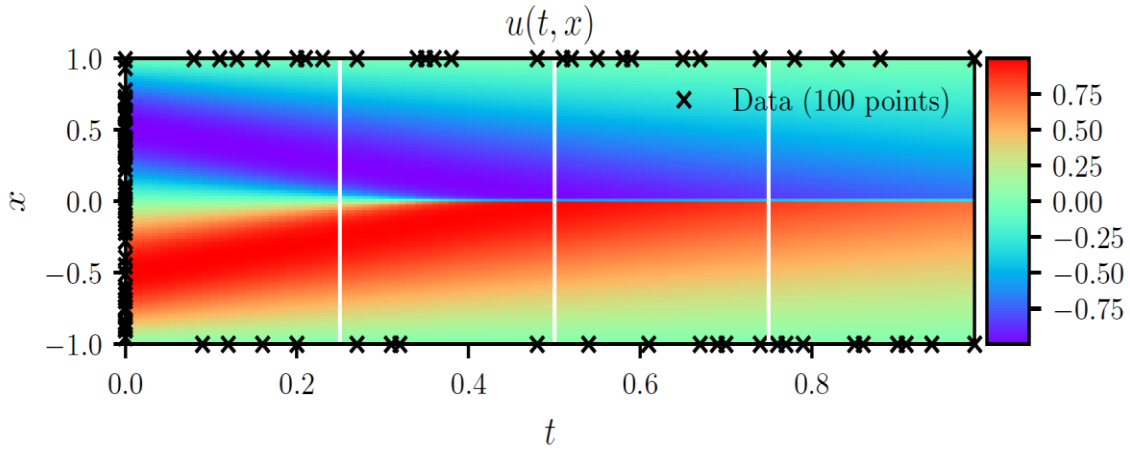
Physics Informed Neural Networks

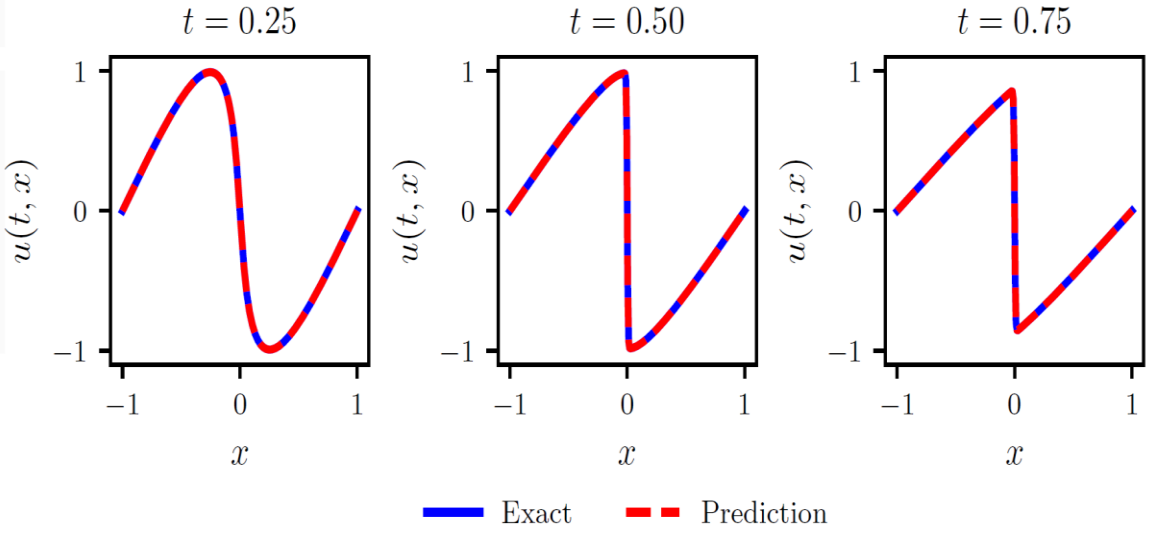
$$u_t + uu_x - (0.01/\pi)u_{xx} = 0$$

```
def u(t, x):
    u = neural_net(tf.concat([t,x],1), weights, biases)
    return u
```

```
def f(t, x):
    u = u(t, x)
    u_t = tf.gradients(u, t)[0]
    u_x = tf.gradients(u, x)[0]
    u_xx = tf.gradients(u_x, x)[0]
    f = u_t + u*u_x - (0.01/tf.pi)*u_xx
    return f
```

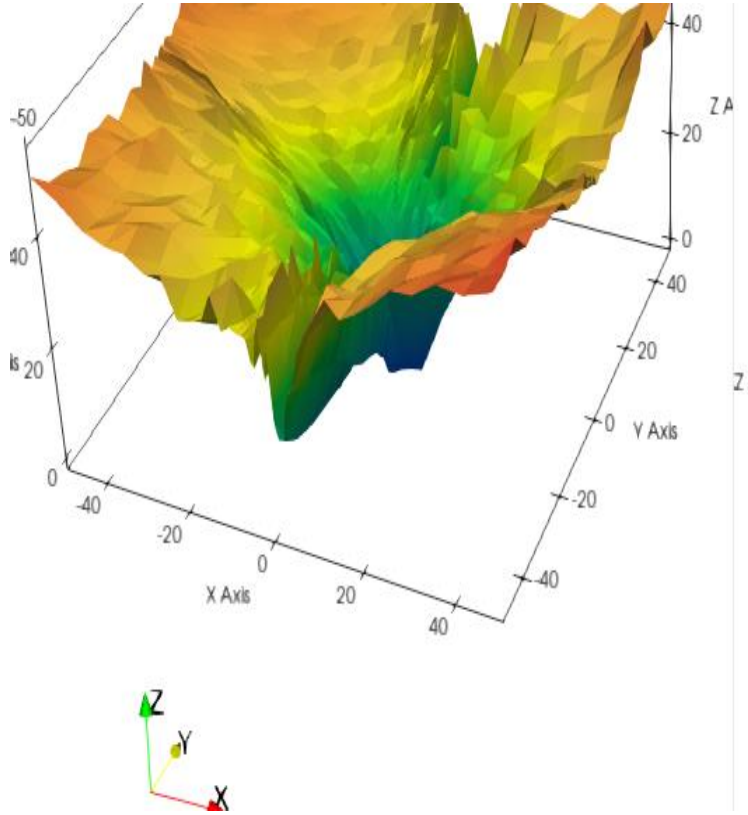
$$\sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2 + \sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2$$



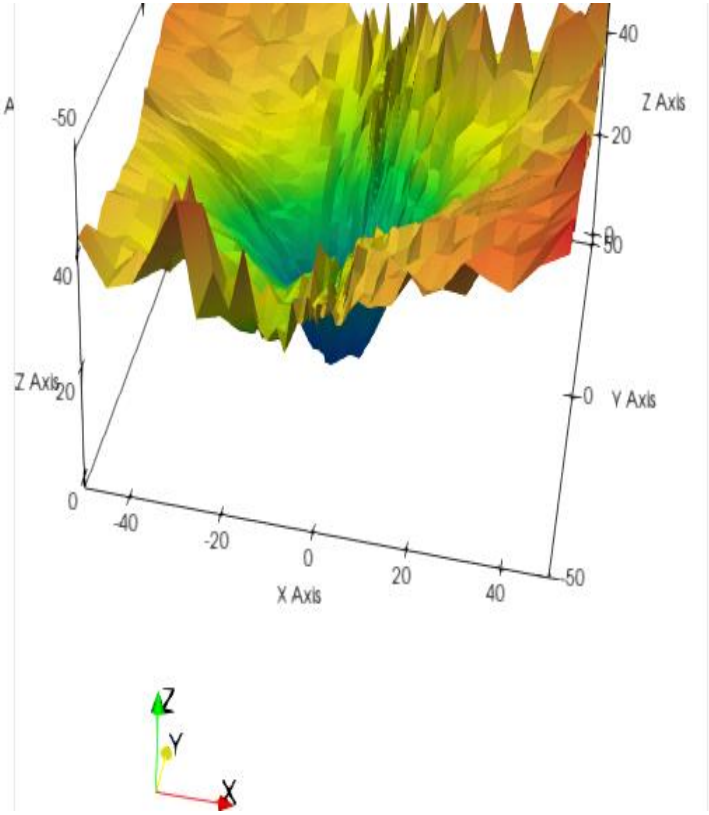


sualizing PINN Landscapes: 1-D viscous Burgers Equation

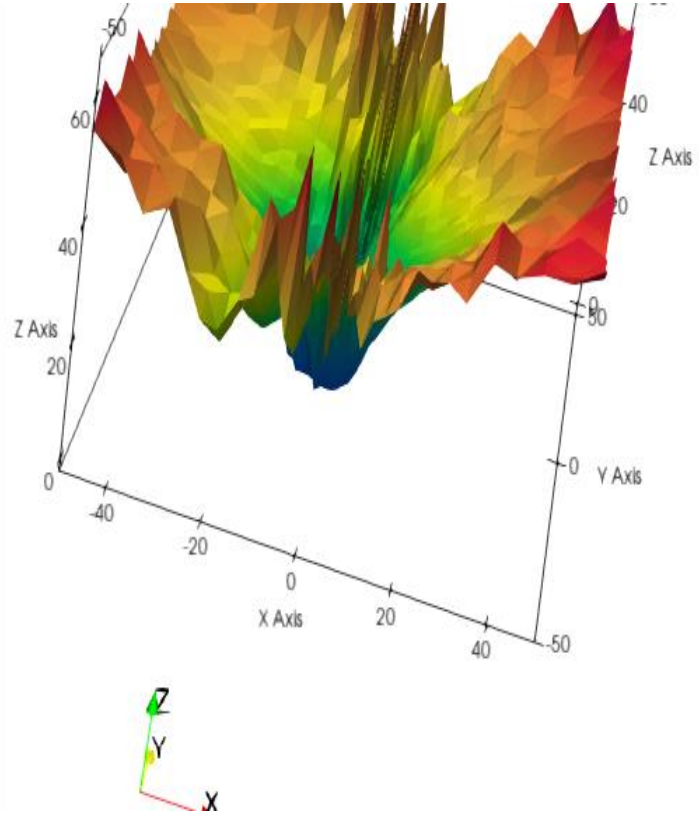
$\nu = 10^{-2}$



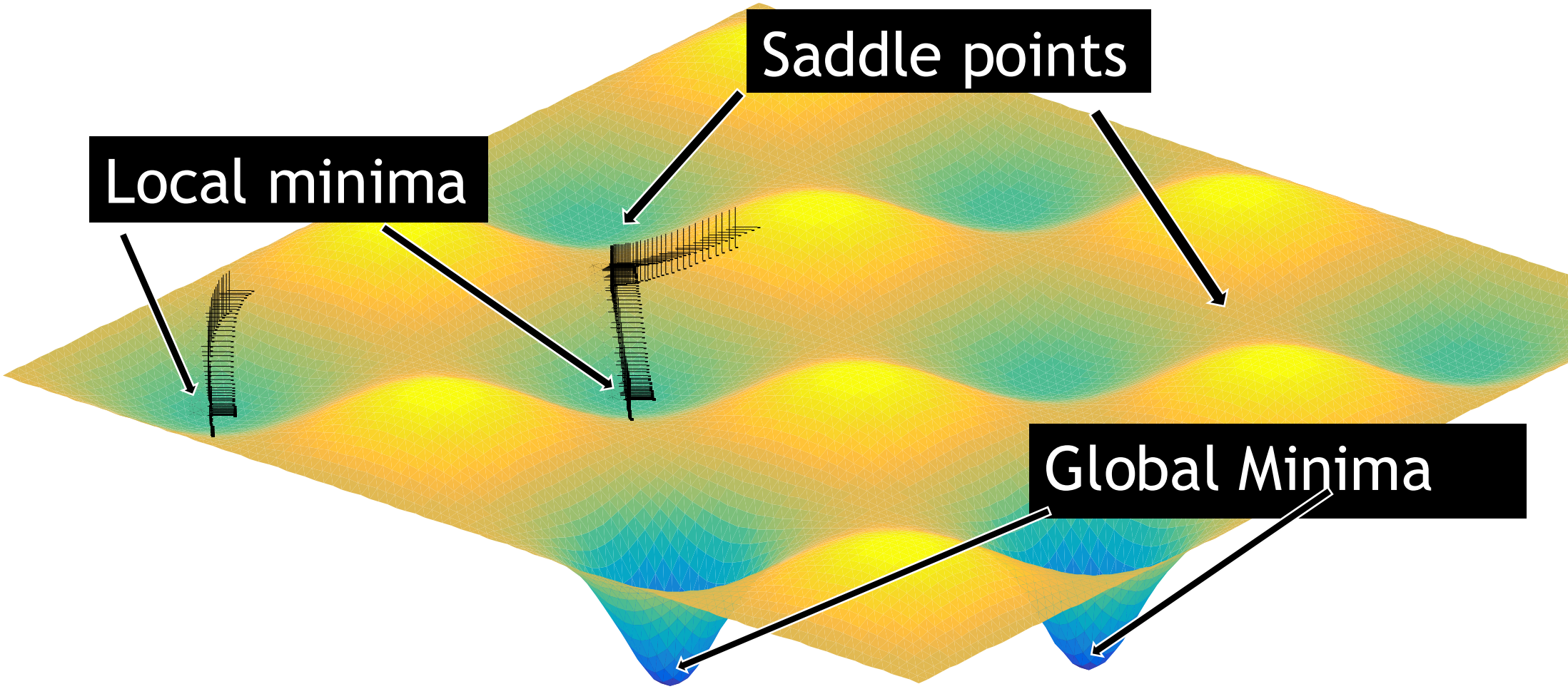
$\nu = 10^{-3}$



$\nu = 10^{-8}$



pes of Stationary Points

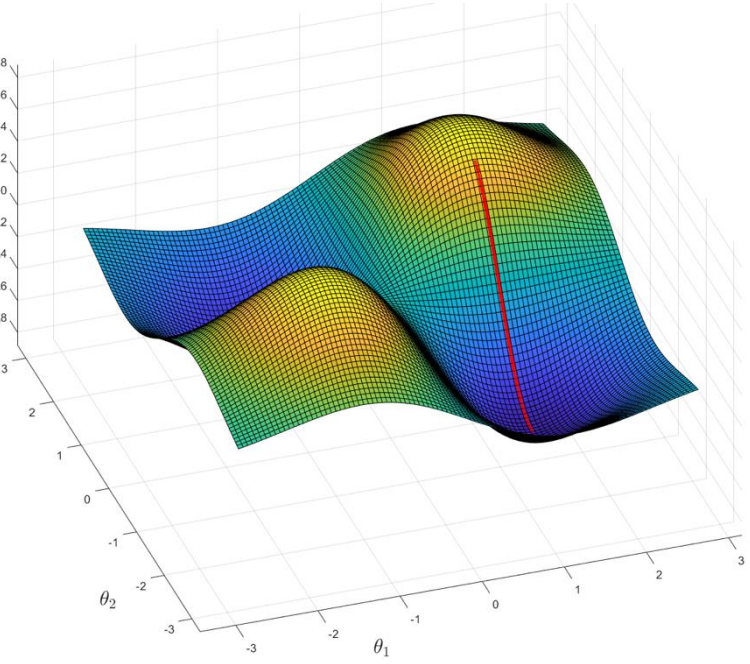


- x-y plane = search space for optimization
- z axis = value of objective function to be minimized

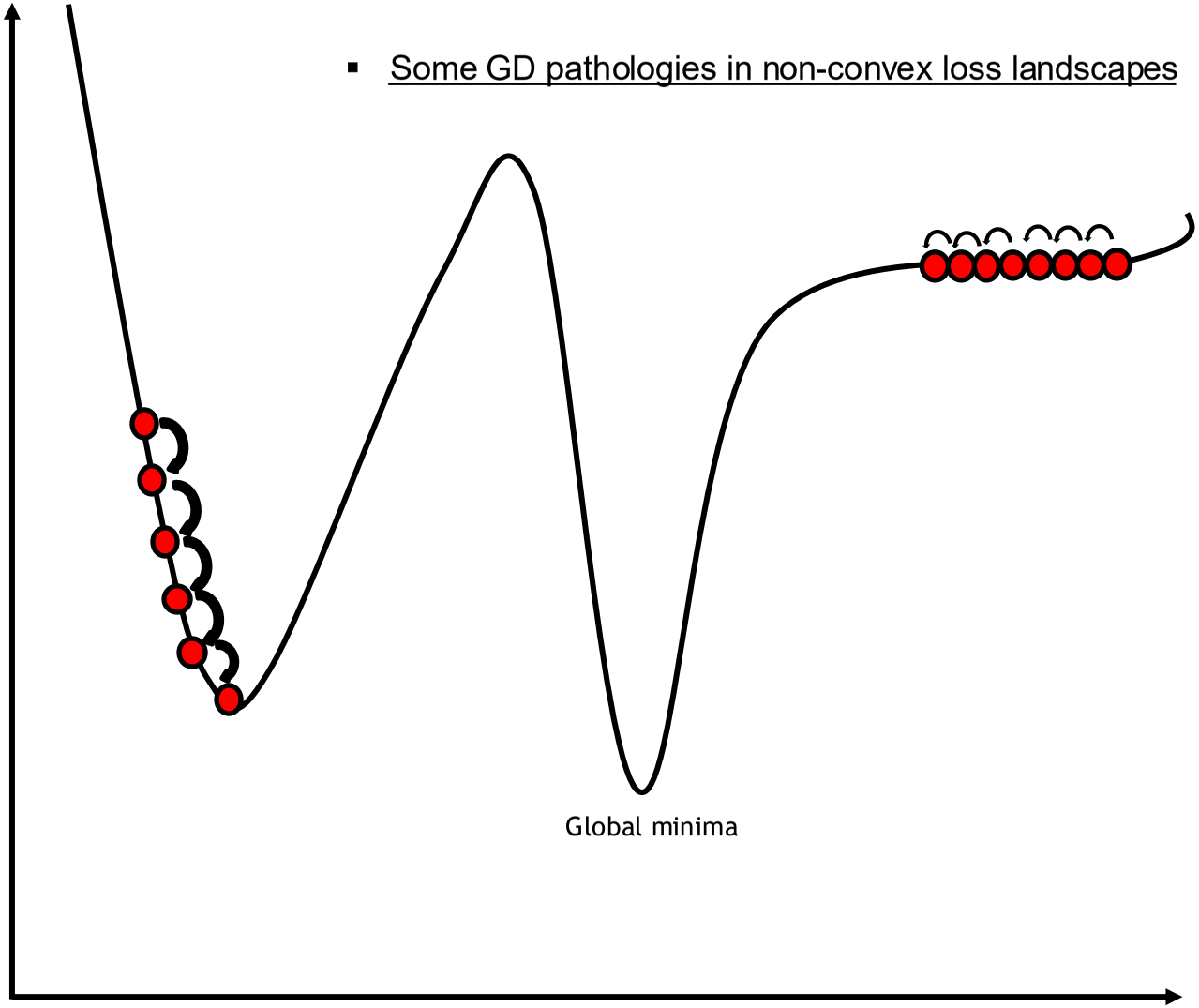
radient Descent (GD)

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

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$



- Some GD pathologies in non-convex loss landscapes

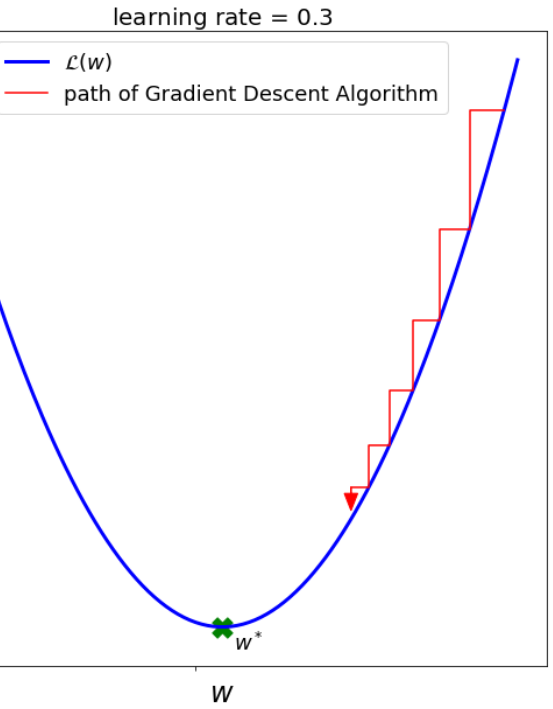


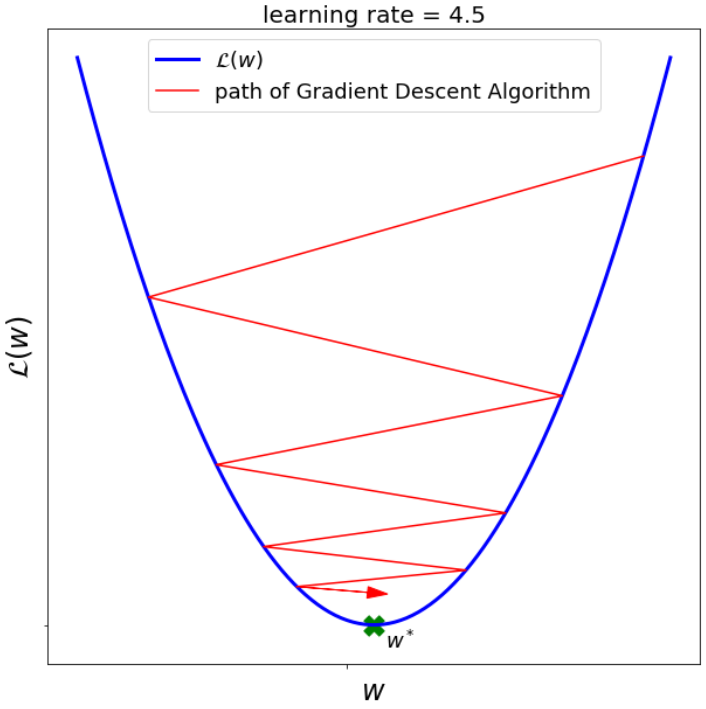

```
tf.keras.optimizers.SGD(learning_rate=0.01
)
```

```
torch.optim.SGD(params, lr=0.01
)
```

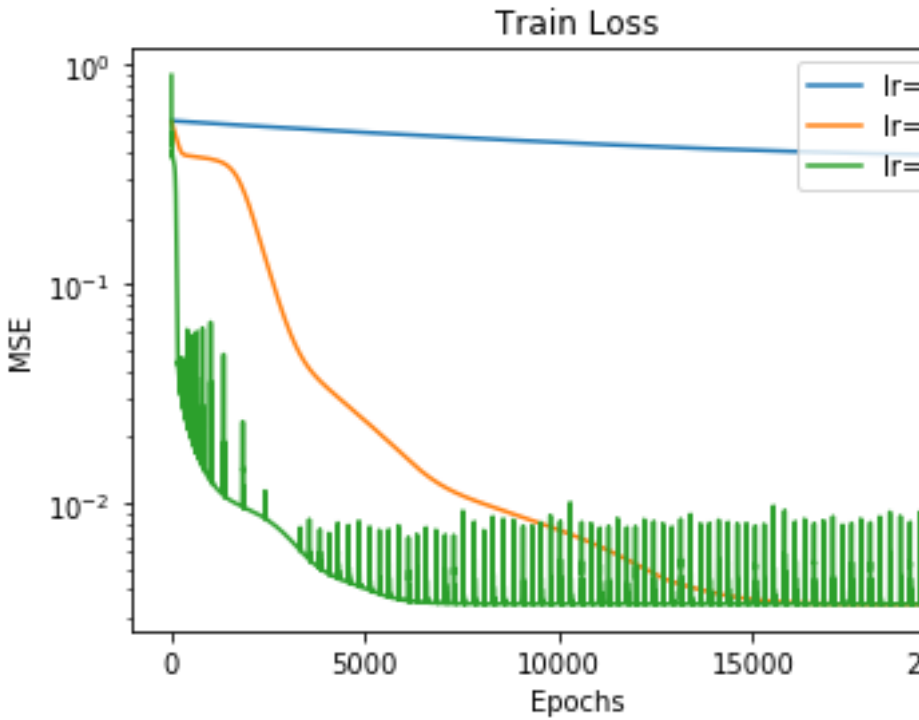
fect of Learning Rate

near regression we have convexity (hence global minimum) but still we should scale all features for faster convergence





Loss plot associated with learning the sine function with noise using a fully connected neural network



earning rate is too small

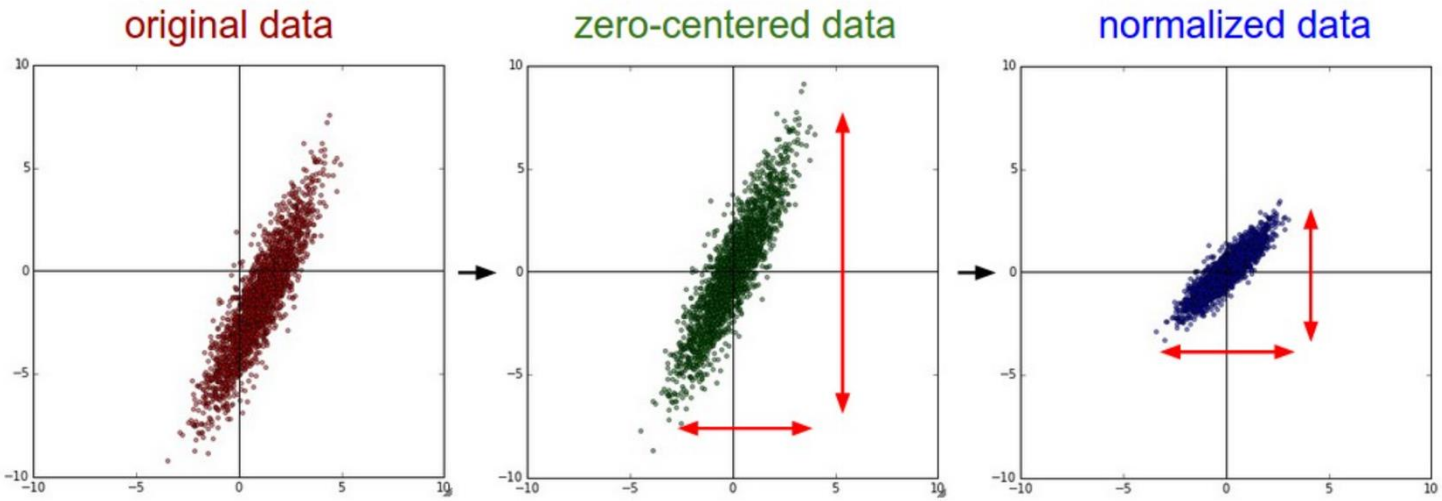
Learning rate is too large

Convergence depends strongly on t

❑ **An effective strategy is to use a variable/decaying learning rate**

a Normalization

Ref: https://zaffnet.github.io/batch-normalization



Common data preprocessing pipeline. **Left**: Original toy, 2-dimensional input data. **Middle**: The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin. **Right**: Each dimension is additionally scaled by its standard deviation. The red lines indicate the extent of the data - they are of unequal length in the middle, but of equal length on the right.

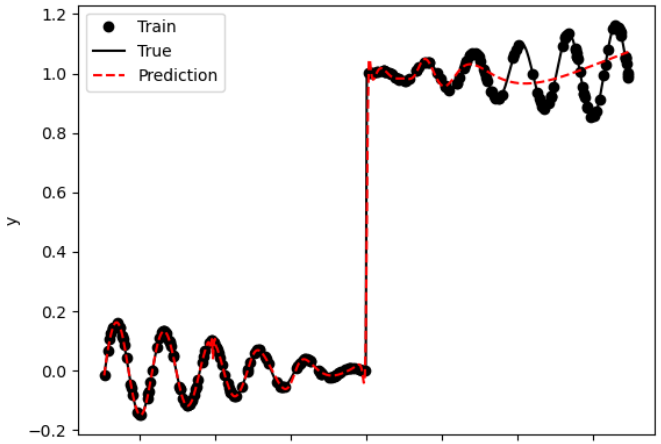
In 1998, Yan LeCun in his paper, Efficient BackProp, highlighted the importance of normalizing the inputs. Preprocessing of the inputs using normalization is a standard machine learning procedure and is known to help in faster convergence. Normalization is done to achieve the following objectives:

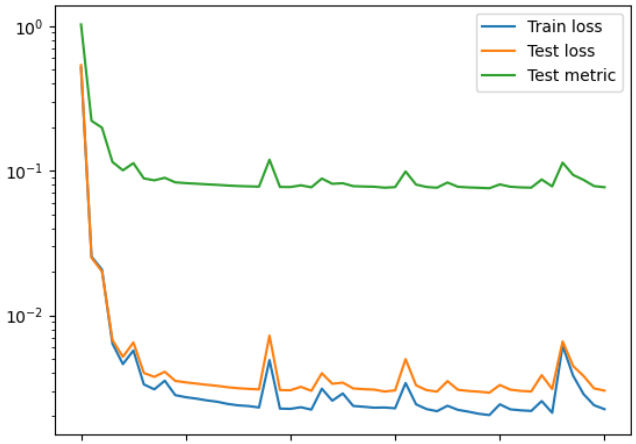
- ❑ The average of each input variable (or feature) over the training set is close to zero (Mean subtraction).
- ❑ Covariances of the features are same (Scaling).
- ❑ Decorrelate the features (Whitening – not required for CNNs).

a Normalization - Example

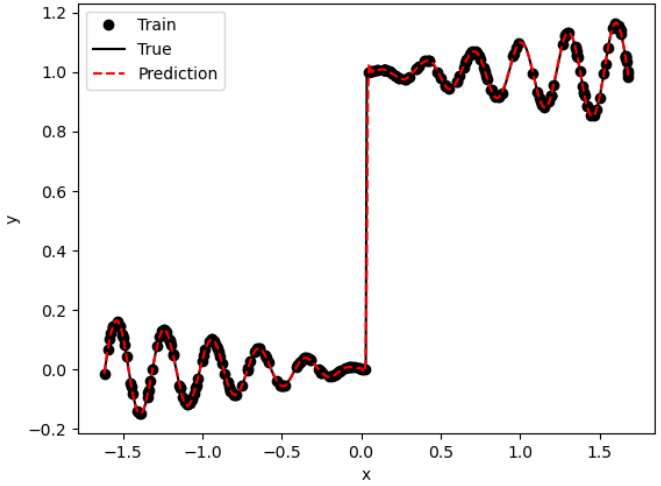
Without data-normalization

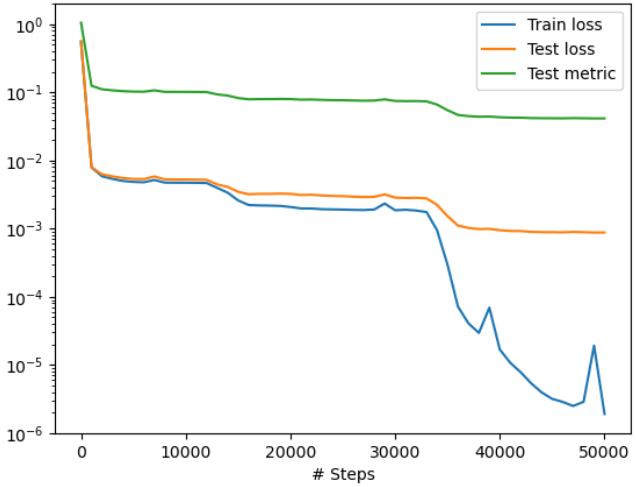
$$y = \begin{cases} \frac{1}{10}xsin(20x) & \text{if } x < 0 \\ 0.5 & \text{if } x = 0.5 \\ 1 + \frac{1}{10}xsin(20x) & \text{if } x > 0 \end{cases}$$





With data-normalization

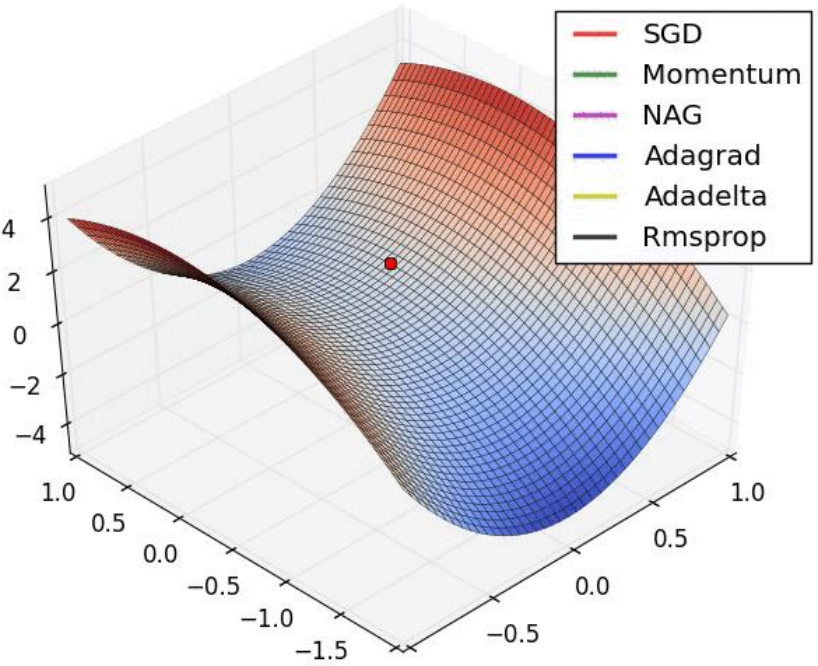


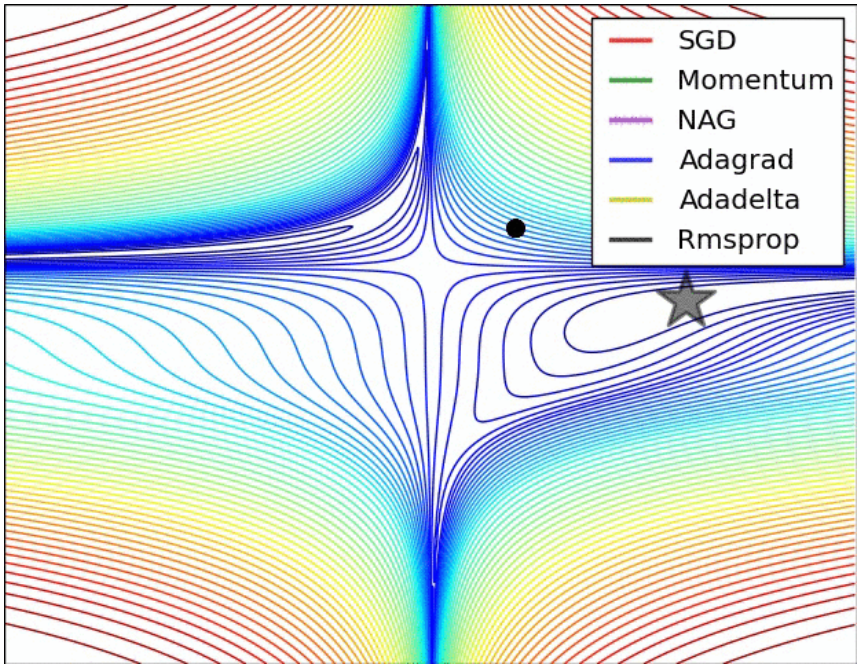


Overview of Gradient Descent Optimization Algorithms

org/abs/1609.04747

post explores how many of the most popular gradient-based optimization algorithms actually work.





movie shows the **behaviour of the algorithms at a saddle point** Notice that omentum, and NAG find it difficult to break symmetry, although the two latter ly manage to escape the saddle point, while Adagrad, RMSprop, and Adadelta head down the negative slope.

B. In this movie, we see their **behavior on the contours of a loss su (the Beale function) over time**. Note that Adagrad, Adadelta, and RM almost immediately head off in the right direction and converge similarl while Momentum and NAG are led off-track, evoking the image of a ball r down the hill. NAG, however, is quickly able to correct its course due increased responsiveness by looking ahead and heads to the minimum.

➢ These two animations (Images credit: Alec Radford) provide some intuitions towards the optimization behavior of most of the presented optimization metho

at Optimizer to Use?

n faster optimizers : **second-order**

$$\nabla^2_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1^2} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2^2} & \cdots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2 \partial \theta_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d \partial \theta_1} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d \partial \theta_2} & \cdots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d^2} \end{bmatrix}$$

sian based optimizers

timization technique discussed so far is based on first-order partial derivatives (Jacobians).

ere are optimizers, which are based on the second-order partial derivatives (the Hessian).

ese algorithms are very expensive as there are n^2 Hessians to be computed for each output (n is number of parameter
nparison to n Jacobians per output.

xt, we will discuss a couple of Hessian-based optimizer and their implementation in TensorFlow and PyTorch.

at Optimizer to Use?

ive algorithm

timization uses iterative algorithms, which consider a sequence of $\boldsymbol{x}_n$ converging to $\boldsymbol{x}^*$. Therefore, $\forall\ f''(\boldsymbol{x}) \neq 0$, urrent estimate being $\boldsymbol{x}_n$, the next estimate $\boldsymbol{x}_{n+1}$ should satisfy

$$f(\boldsymbol{x}_{n+1}) < f(\boldsymbol{x}_n)$$

sing Taylor Approximation

$$f(\boldsymbol{x} + \Delta x) \approx f(\boldsymbol{x}) + \Delta x^\top \nabla f(x) + \frac{1}{2}\Delta x^T(\nabla^2 f(x))\Delta x$$

rative algorithm produces a sequence of such quadratic approximations $\boldsymbol{h}_n$ and considers $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta\boldsymbol{x}$;

$$h_n(\Delta\boldsymbol{x}) = f(\boldsymbol{x}_n) + \Delta x^\top \boldsymbol{g}_n + \tfrac{1}{2}\Delta x^\top \boldsymbol{H}_n \Delta x$$

Where $\boldsymbol{g}_n$ and $\boldsymbol{H}_n$ are gradient and Hessian of f at x_n.

nimality condition $\frac{\partial h_n(\Delta\boldsymbol{x})}{\partial \Delta x} = \boldsymbol{g}_n + \boldsymbol{H}_n \Delta x = 0$

$$\Delta x = -\boldsymbol{H}_n^{-1}\boldsymbol{g}_n$$

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha(\boldsymbol{H}_n^{-1}\boldsymbol{g}_n)(\boldsymbol{x}_n)$$

at Optimizer to Use?

ats: Iterative algorithm

he main issue with iterative algorithm is that it needs to compute the inverse Hessian matrix.

DL applications, the dimensionality of the input corresponds to model parameters. It is quite common to have hundr
millions of parameters.

or these reasons, computing the Hessian or its inverse is often impractical.

herefore, Iterative algorithm in this form is rarely used in practice to optimize functions corresponding to large proble

uckily, the above algorithm can still work even if $\boldsymbol{H}_n^{-1}$ does not correspond to the exact inverse Hessian at $\boldsymbol{x}_n$, but is
stead a good approximation. These are handled by **quasi-Newton methods and are implemented through BFGS a
-BFGS methods.**

at Optimizer to Use?

Newton Methods

eral form of quasi-Newton methods is

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \boldsymbol{H}_n^{-1} \nabla f(\boldsymbol{x}_n)$$

here α is line search parameter and $\boldsymbol{H}_n^{-1}$ is the inverse Hessian or some approximation.

si-Newton methods lose their quadratic convergence of Newton's method, but super linear convergence is often achie

e we will discus two forms of quasi-Newton methods:

- Broyden-Fletcher-Goldfarb-Shannon (BFGS) method
- Limited-memory Broyden-Fletcher-Goldfarb-Shannon (L-BFGS) method

at Optimizer to Use?

GS Method

neral form of quasi-Newton methods is

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \boldsymbol{H}_n^{-1} \nabla f(\boldsymbol{x}_n)$$

where α is line search parameter and is $\boldsymbol{H}_n^{-1}$ Hessian or some approximation to the Hessian.

ssume

$$\boldsymbol{H}_n \boldsymbol{s}_n = -\nabla f(\boldsymbol{x}_n), \qquad \boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \alpha \boldsymbol{s}_n$$
$$\boldsymbol{y}_n = \nabla f(\boldsymbol{x}_{n+1}) - \nabla f(\boldsymbol{x}_n)$$

e updated approximated Hessian should satisfy the following under-determined system

$$\boldsymbol{H}_{n+1} \boldsymbol{s}_n = \boldsymbol{y}_n$$

he objective is to adjust $\boldsymbol{H}_n$ minimally to satisfy the above condition by ensuring $\boldsymbol{H}_n$ remains symmetric-positive-de
.

at Optimizer to Use?

GS Method

GS proposed

$$\boldsymbol{H}_{n+1} = \boldsymbol{H}_n + \beta \boldsymbol{u}\boldsymbol{u}^\top + \gamma \boldsymbol{v}\boldsymbol{v}^\top$$

bstitute $\boldsymbol{u} = \boldsymbol{y}_n$ and $\boldsymbol{v} = \boldsymbol{H}_n \boldsymbol{s}_n$ and $\boldsymbol{H}_n^\top = \boldsymbol{H}_n$ to obtain

$$\boldsymbol{H}_{n+1} = \boldsymbol{H}_n + \beta \boldsymbol{y}_n \boldsymbol{y}_n^\top + \gamma \boldsymbol{H}_n \boldsymbol{s}_n \boldsymbol{s}_n^\top \boldsymbol{H}_n$$

ce

$$\boldsymbol{y}_n = \boldsymbol{H}_{n+1}\boldsymbol{s}_n = \boldsymbol{H}_n \boldsymbol{s}_n + \beta \boldsymbol{y}_n \boldsymbol{y}_n^\top \boldsymbol{s}_n + \gamma \boldsymbol{H}_n \boldsymbol{s}_n \boldsymbol{s}_n^\top \boldsymbol{H}_n \boldsymbol{s}_n$$

$$\beta = \frac{1}{\boldsymbol{y}_n^\top \boldsymbol{s}_n} \qquad \gamma = -\frac{1}{\boldsymbol{s}_n^\top \boldsymbol{H}_n \boldsymbol{s}_n}$$

nce the update approximation to Hessian is

$$\boldsymbol{H}_{n+1} = \boldsymbol{H}_n + \frac{\boldsymbol{y}_n \boldsymbol{y}_n^\top}{\boldsymbol{y}_n^\top \boldsymbol{s}_n} - \frac{\boldsymbol{H}_n \boldsymbol{s}_n \boldsymbol{s}_n^\top \boldsymbol{H}_n}{\boldsymbol{s}_n^\top \boldsymbol{H}_n \boldsymbol{s}_n}$$

at Optimizer to Use?

- Liu DC, Nocedal J. On the limited memory BFGS method for large scale optimizat Mathematical programming. 1989 Aug;45(1):503-28.

FGS optimizer

GS is the most popular of all Quasi-Newton methods but has storage complexity.

FGS (Limited memory BFGS), which does not require to explicitly store H^{-1} but instead stores the previous data $\nabla f(x_i))\}_{i=1}^{k}$ and manages to compute $d = H^{-1}\nabla f(x)$ directly from this data. L-BFGS has storage complexity of $\mathcal{O}(n$

BFGS implementation is not straightforward in PyTorch and TF2. A detailed implementation is discussed in Lecture t here we provide a simple API for both.

m 1: BFGS Algorithm

: initial $\boldsymbol{x}_0 \in \mathbb{R}^n$, functions $f(\boldsymbol{x})$, $\nabla f(\boldsymbol{x})$. tolerance θ
t: x
e $\boldsymbol{H}^{-1} \leftarrow \mathbb{I}$

pute $\boldsymbol{s} = \boldsymbol{H}^{-1}\nabla f(\boldsymbol{x})$
orm a line search $\min_\alpha f(x + \alpha d)$
$\alpha \boldsymbol{d}$
$\nabla f(\boldsymbol{x} + \boldsymbol{s}) - \nabla f(\boldsymbol{x})$
$\boldsymbol{x} + \boldsymbol{s}$
ate $\boldsymbol{H}^{-1} \leftarrow \left(\mathbb{I} - \frac{\boldsymbol{y}\boldsymbol{s}^\top}{\boldsymbol{s}^\top\boldsymbol{y}}\right)\boldsymbol{H}^{-1}\left(\mathbb{I} - \frac{\boldsymbol{y}\boldsymbol{s}^\top}{\boldsymbol{s}^\top\boldsymbol{y}}\right) + \frac{\boldsymbol{s}\boldsymbol{s}^\top}{\boldsymbol{s}^\top\boldsymbol{y}}$
ntil $||s||_\infty < \theta$;

```
torch.optim.LBFGS(params, lr=1, max_iter=
20, max_eval=None, tolerance_grad=1e-
07, tolerance_change=1e-
09, history_size=100, line_search_fn=None
)
```

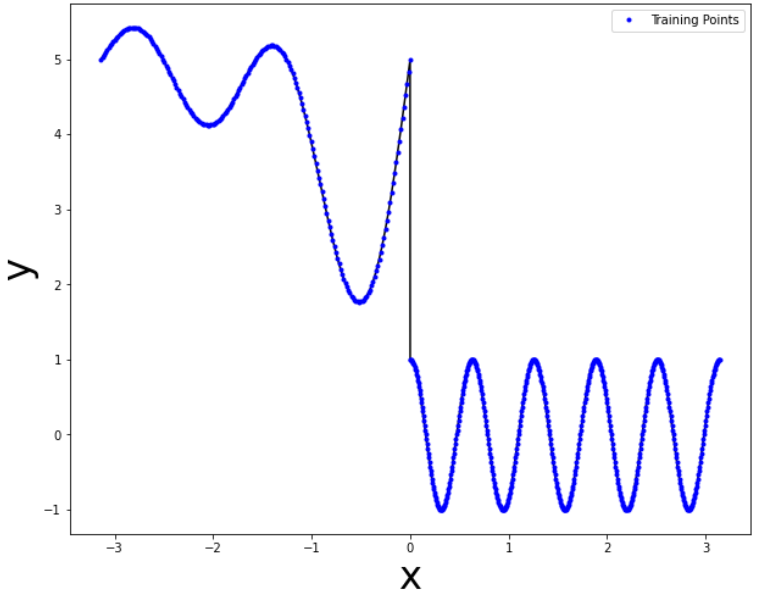
L-BFGS implementation in TensorFlow is prov through TF Probability package

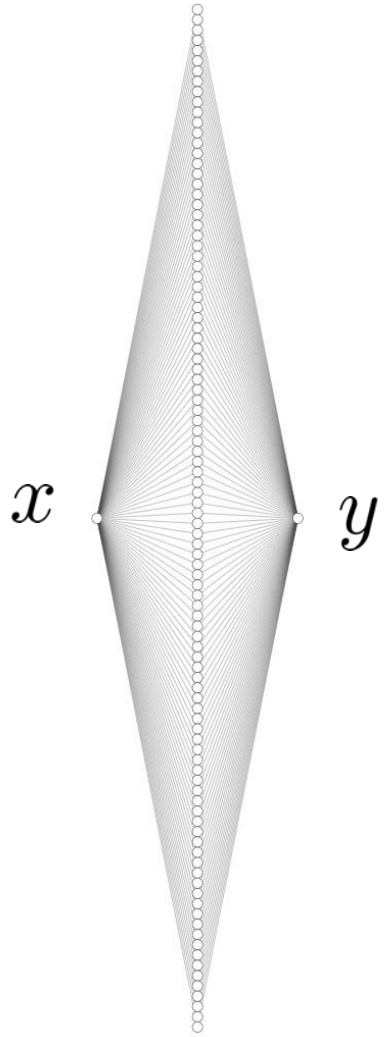
```
tfp.optimizer.lbfgs_minimize(f,
initial_position=self.get_weights(),
num_correction_pairs=50,
max_iterations=2000)
```

+ Function Approximation + Adam + L-BFGS

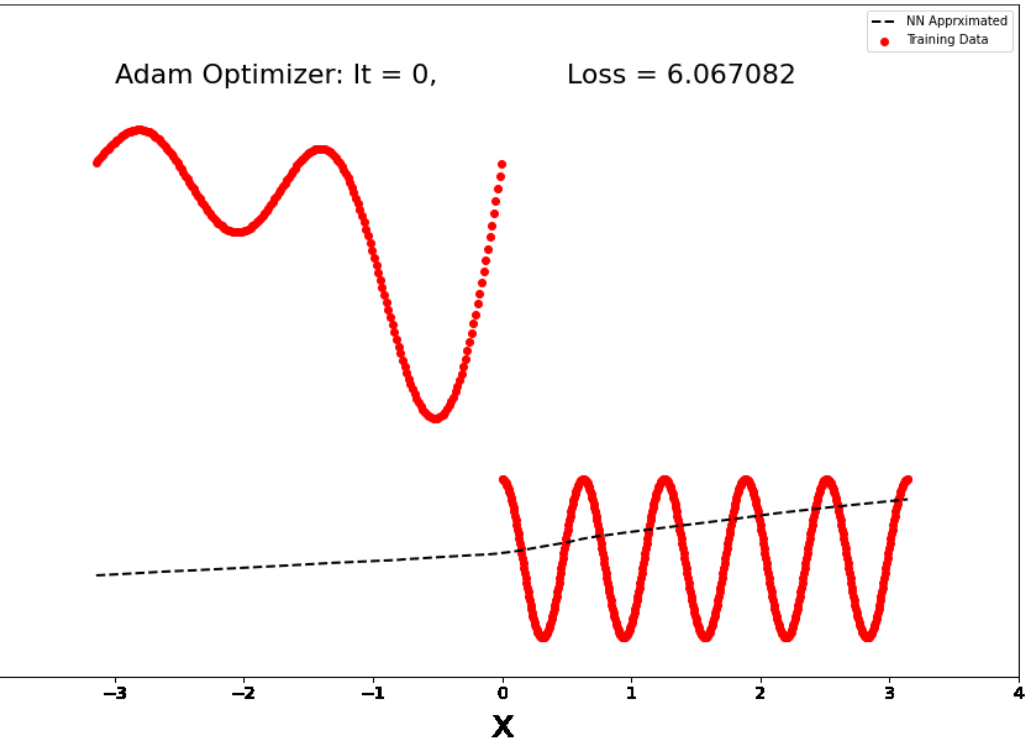
$$y = 5 + \sum_{k=1}^{4} \sin(kx), \ x < 0$$

$$y = \cos(10x), \ x \geq 0$$

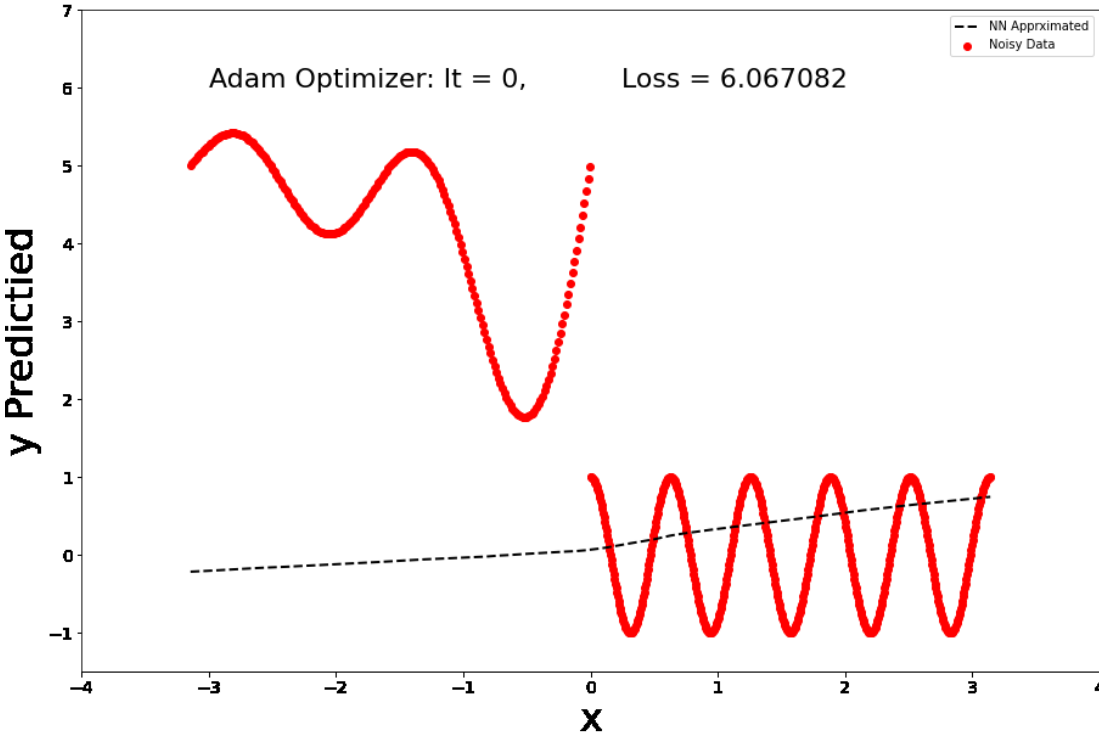




+ Function Approximations + Adam + L-BFGS



Adam (30 Iterations) + 50 L-BFGS (50 Iterations)



FASTER OPTIMIZATION: Self-Scaled Broyden's method

- Instead of computing the inverse Hessian matrix directly, the algorithm reformulates the scaling parameter calculation, avoiding the $O(N^3)$ complexity.

$$H_{k+1} = H_k + \sigma_k \frac{(y_k - H_k s_k)(y_k - H_k s_k)^T}{(y_k - H_k s_k)^T s_k} - \frac{H_k s_k s_k^T H_k}{s_k^T H_k s_k}$$

$$\sigma_k = 1 + a_k \theta_k$$

$$b_k = \frac{\boldsymbol{s}_k \cdot H_k^{-1} \boldsymbol{s}_k}{\boldsymbol{y}_k \cdot \boldsymbol{s}_k} = -\alpha_k \frac{\boldsymbol{s}_k \cdot \nabla \mathcal{J}(\boldsymbol{\Theta}_k)}{\boldsymbol{y}_k \cdot \boldsymbol{s}_k},$$

$$h_k = \frac{\boldsymbol{y}_k \cdot H_k \boldsymbol{y}_k}{\boldsymbol{y}_k \cdot \boldsymbol{s}_k},$$

$$a_k = h_k b_k - 1$$

$$c_k = \sqrt{\frac{a_k}{a_k + 1}},$$

$$\rho_k^- = \min(1, h_k(1 - c_k)),$$

$$\theta_k^- = \frac{\rho_k^- - 1}{a_k},$$

$$\theta_k^+ = \frac{1}{\rho_k^-},$$

$$\theta_k = \max\left(\theta_k^-, \min\left(\theta_k^+, \frac{b_k}{1 - b_k}\right)\right)$$

Urbán, J. F., Stefanou, P., and Pons, J. A. (2024).

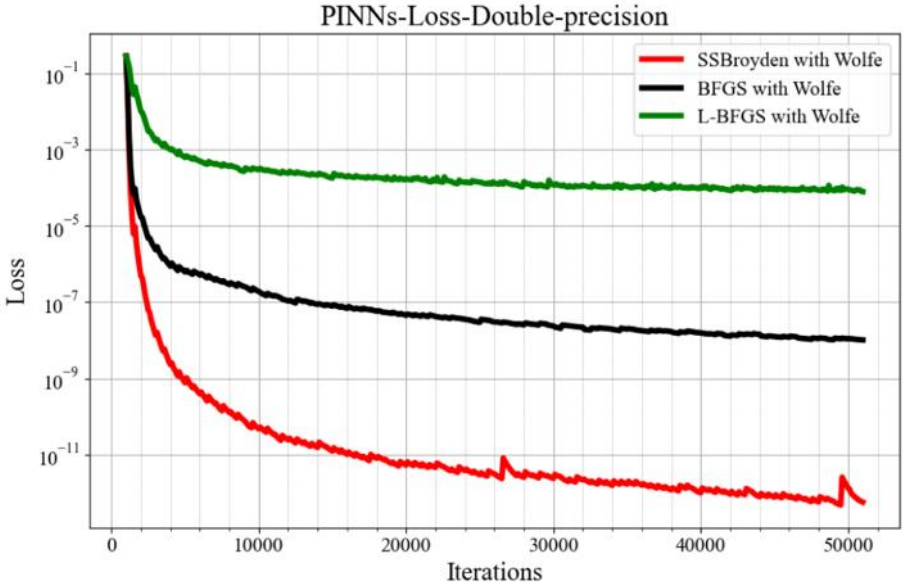
M. Al-Baali, Variational quasi-newton methods for unconstrained optimization, Journal of Optimization Theory and Applications 77 (1) (1993) 127–143. doi:10.1007/BF00940782.

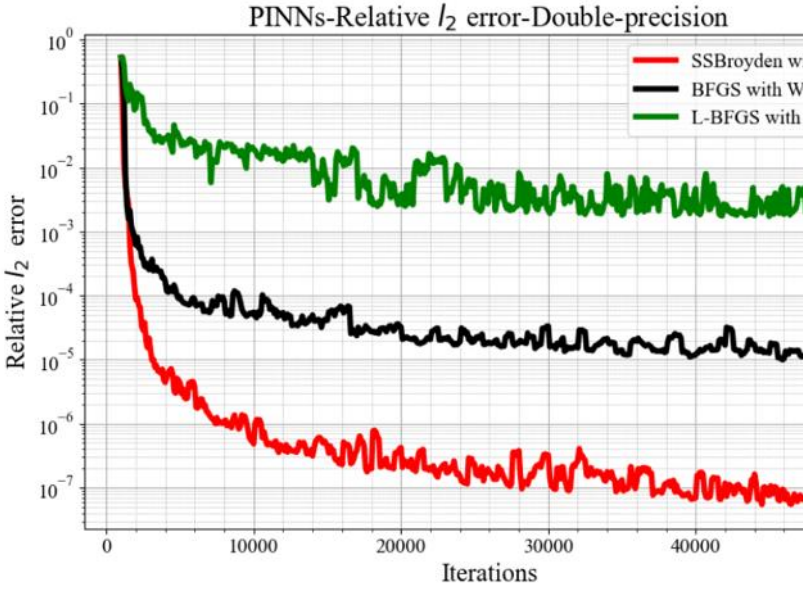
izing PINNs for the Burgers Equation using SSBroyden

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

$$(x, 0) = -\sin(\pi x)$$

$$= \frac{0.01}{\pi} \approx 0.003$$





Case	Optimizer[# Iters.], Line-search [#Iters.]	Relative l_2 error	Training time (s)	Total par
2	Adam [1000] + BFGS with Wolfe [50000]	1.50×10^{-5}	1292	1,341
2	Adam [1000] + SSBroyden with Wolfe [50000]	7.57×10^{-8}	1354	1,341
2	Adam [1000] + L-BFGS with Wolfe [50000]	2.05×10^{-3}	713	1,341
3	Adam [1000] + BFGS with Wolfe [30000]	2.21×10^{-5}	774	1,341
3	Adam [1000] + SSBroyden with Strong Wolfe [30000]	2.12×10^{-7}	819	1,341
4	BFGS with Wolfe [30000]	1.73×10^{-5}	863	1,341
4	SSBroyden with Wolfe [30000]	3.59×10^{-7}	862	1,341
5	Adam [1000] + BFGS with Wolfe [30000]	8.52×10^{-6}	3049	3,021
5	Adam [1000] + SSBroyden with Wolfe [30000]	1.62×10^{-8}	2812	3,021

izing PINNs for the Kuramoto-Sivashinsky Equation using SSBroyden

$$- \alpha u \frac{\partial u}{\partial x} + \beta \frac{\partial^2 u}{\partial x^2} + \gamma \frac{\partial^4 u}{\partial x^4} = 0$$

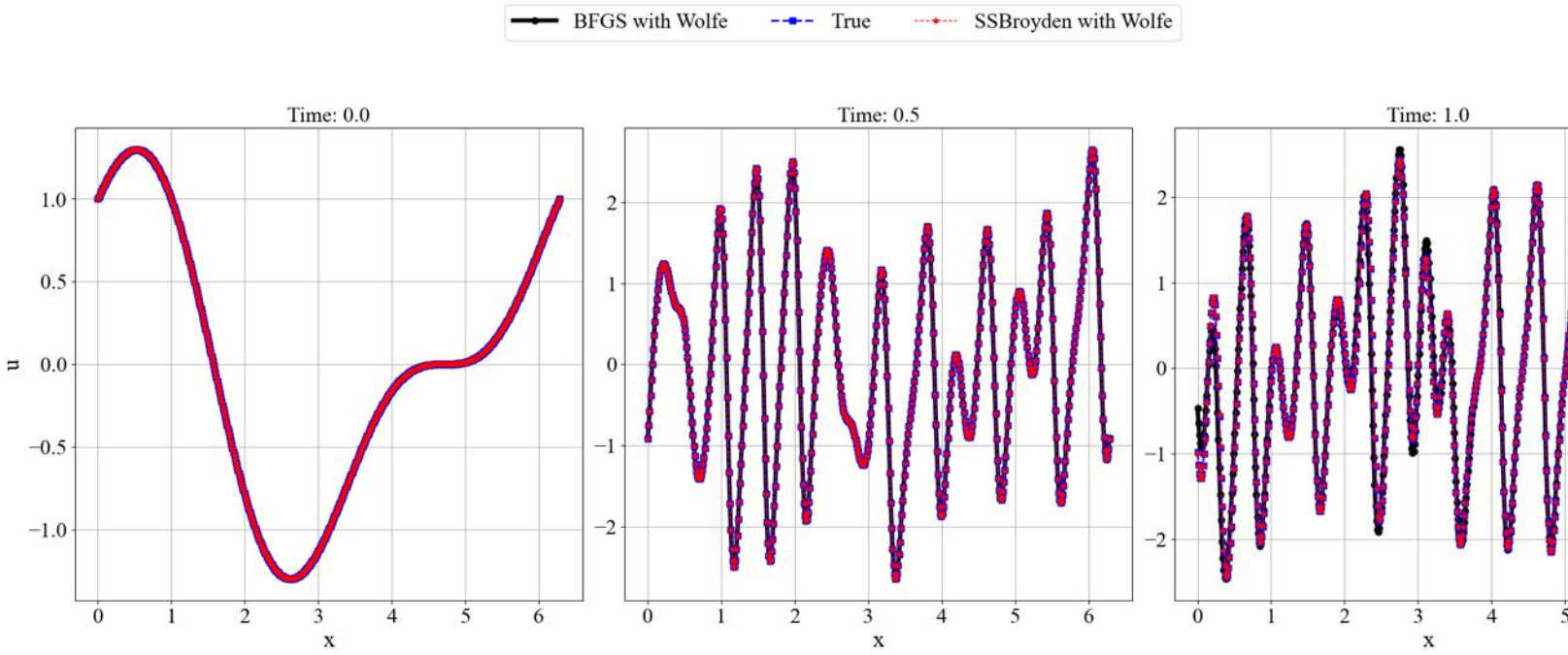
$$(x, 0) = \cos(x)(1 + \sin(x))$$

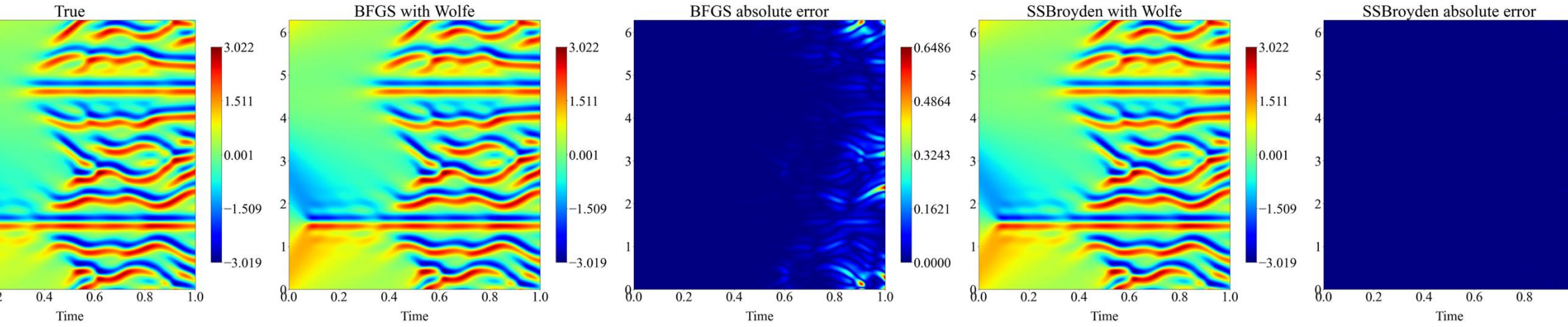
omain: $(t, x) \in [0, 1] \times [0, 2\pi]$

$$= 100/16$$

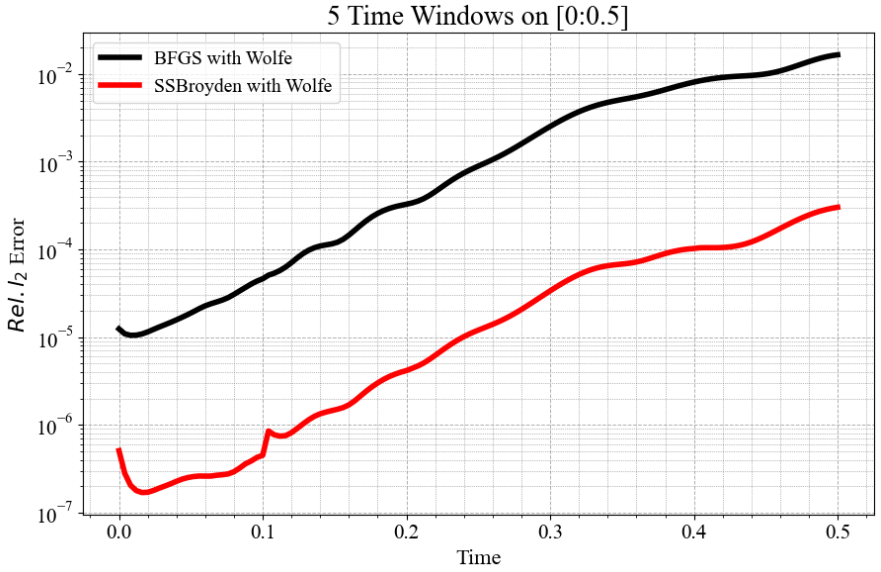
$$= 100/16^2$$

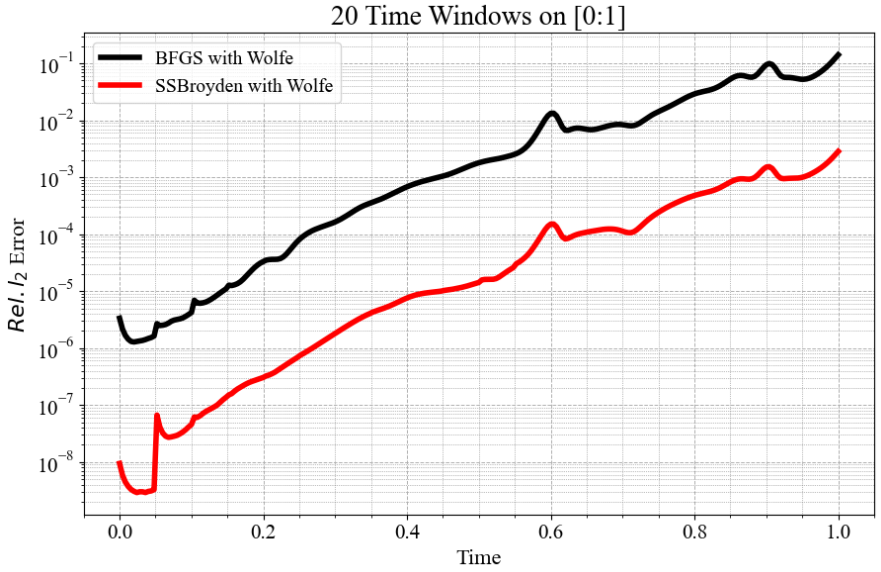
$$= 100/16^4$$





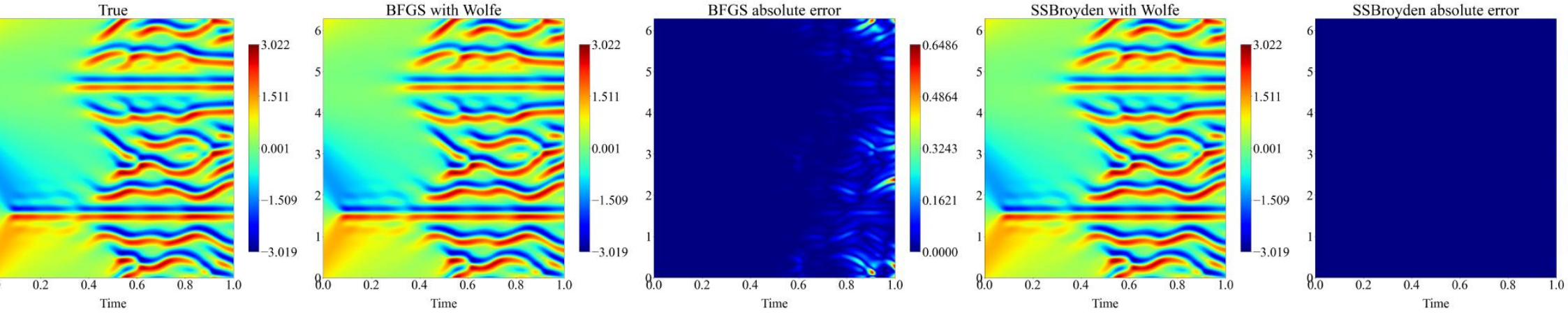
izing PINNs for the Kuramoto-Sivashinsky Equation using SSBroyden





Case	Optimizer[# Iters.], Line-search [#Iters.]	Relative l_2 error	Training time (s)	Total parameters
1	BFGS with Wolfe [20000] (20 windows)	$8.75e \times 10^{-2}$	102177	4,411
1	SSBroyden with Wolfe [20000] (20 windows)	2.13×10^{-3}	113214	4,411
2	BFGS with Wolfe [30000] (20 windows)	3.65×10^{-2}	119286	4,411
2	SSBroyden with Wolfe [30000] (20 windows)	6.51×10^{-4}	130222	4,411
3	Adam [1000] + BFGS with Wolfe [30000] (20 windows)	6.15×10^{-2}	119353	4,411
3	Adam [1000] + SSBroyden with Wolfe [30000] (20 windows)	7.53×10^{-4}	127372	4,411
4	BFGS with Wolfe [20000] (5 windows)	7.54×10^{-3}	17793	4,411
4	SSBroyden with Wolfe [20000] (5 windows)	1.24×10^{-4}	19048	4,411
5	BFGS with Wolfe [30000] (5 windows)	2.39×10^{-3}	26697	4,411
5	SSBroyden with Wolfe [30000] (5 windows)	2.65×10^{-5}	30883	4,411

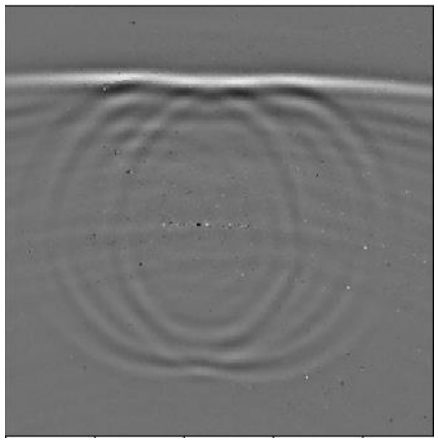
PINNs 2.0 – Better Optimizers



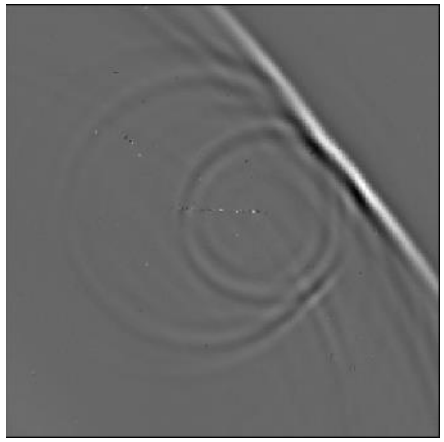
Benchmark	$\mathcal{L}_2$ Error (SOAP [56])	$\mathcal{L}_2$ Error (SSbroyden with Wolfe)
Burgers Equation	8.06×10^{-6}	4.19×10^{-8}
Allen–Cahn Equation	3.48×10^{-6}	9.43×10^{-7}
Kuramoto–Sivashinsky Equation	3.86×10^{-2} on $t \in [0, 0.8]$	2.65×10^{-5} on $t \in [0, 1]$
Ginzburg–Landau Equation	4.78×10^{-3}	2.33×10^{-3}

Crack characterization in an Aluminum Alloy using PINNs

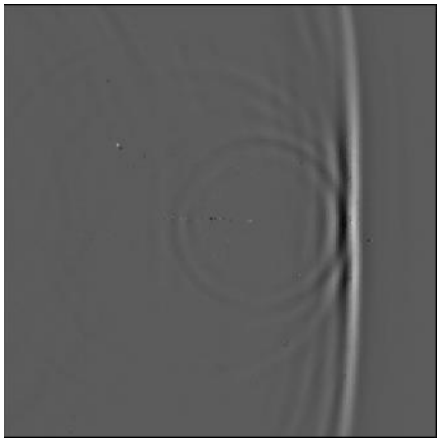
- **Objective:** Identifying and characterizing a surface breaking crack in a 7075-T651 aluminum alloy metal plate.
- **Dataset:** Particle displacement induced by surface acoustic waves of 5 MHz at three incidence angle 0°, 45°, and 90°,
- **Approach:** Reconstruct the wave field and compute the speed of sound using Inverse PINNs



0° incidence



45° incidence

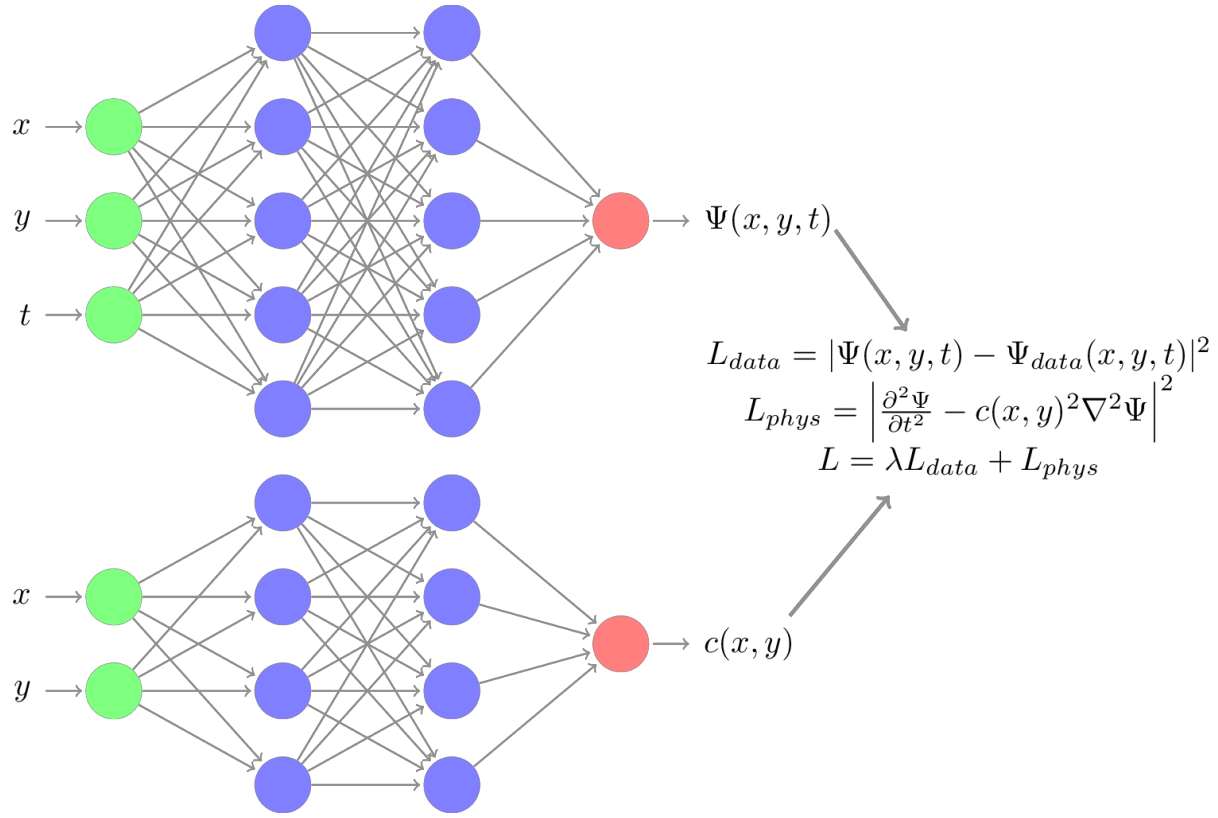


90° incidence

- *K. Shukla, P. Clark Di Leoni, J. Blackshire, D. Sparkman, G. E. Karniadakis," Physics-informed Neural Network for Ultrasound Nondestructive Quantification of Surface Breaking Cracks. J Nondestruct Eval 39, 61 (2020). https://doi.org/10.1007/s10921-020-00705-1*

Problem Setup and Method

Physics-informed Neural Network

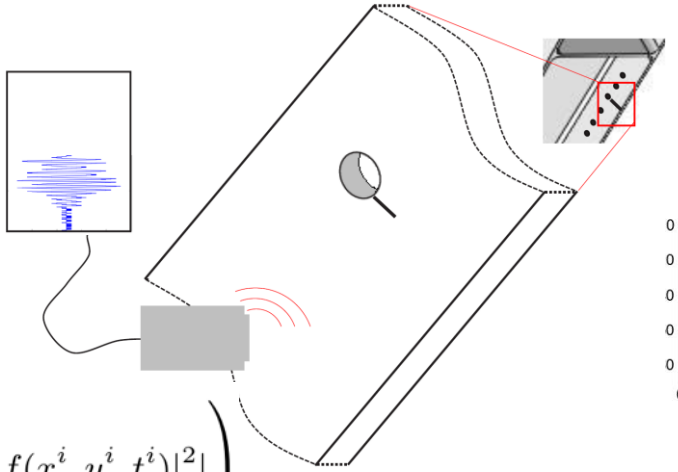


- **Train a Neural Network informed with the acoustic wave equation using sparse data. The speed of sound in the material is approximated by a second neural network**

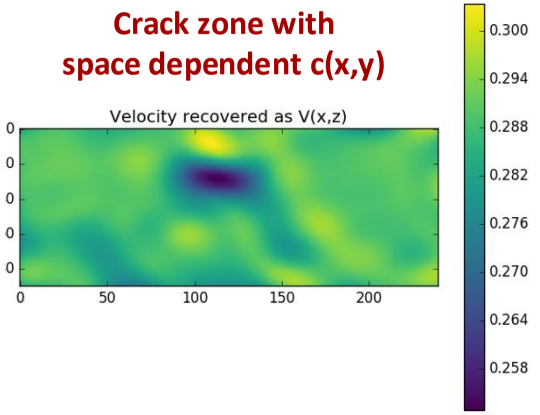
Ultra-Sound Testing of Materials – WPAFB Real Data

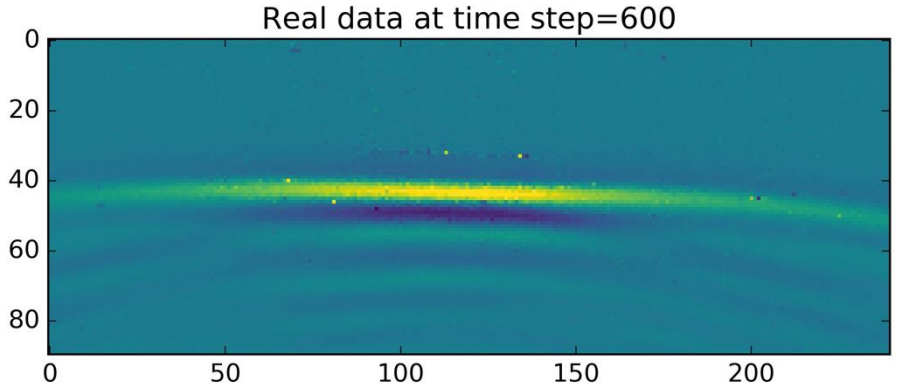
$$f := \frac{\partial^2 \Psi_{\mathrm{NN}}}{\partial t^2} - c^2 \frac{\partial^2 \Psi_{\mathrm{NN}}}{\partial x^2}$$

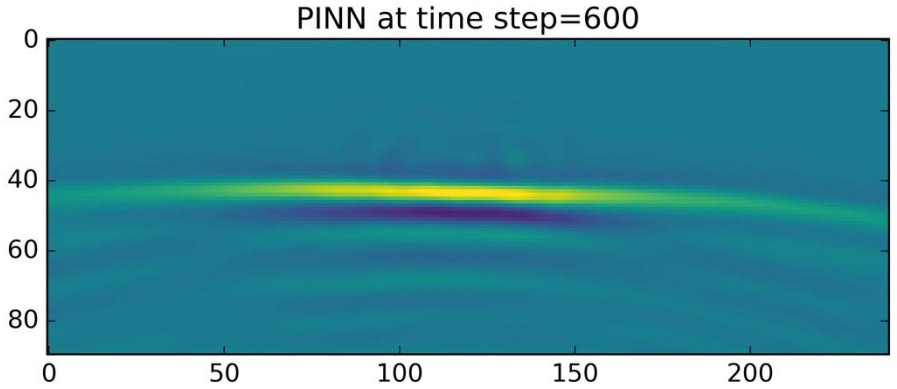
$$L = \sum_{i=1}^{N} \left(\underbrace{|\Psi_{\mathrm{NN}}(x^i, y^i, t^i) - \Psi^i_{\mathrm{data}}|^2}_{\text{data}} + \lambda \underbrace{|f(x^i, y^i, t^i)|^2|}_{\text{eq. enforcing}} \right)$$



Crack zone with space dependent c(x,y)







[Journal of Non-Destructive Evaluation]

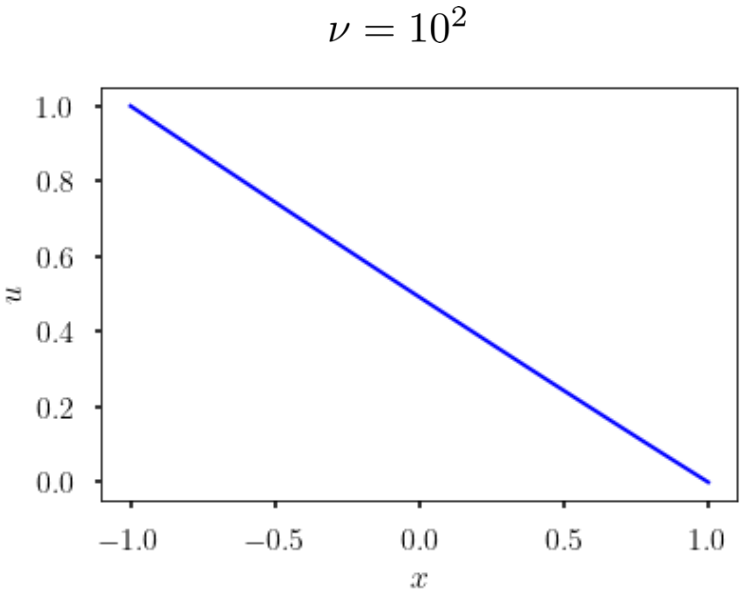
PINNs: Training For Boundary Value Problems

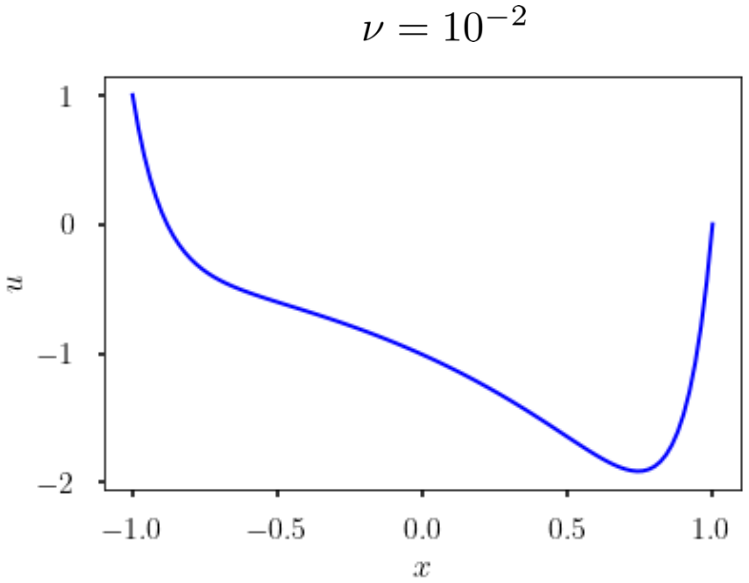
- Define an ODE for given boundary conditions

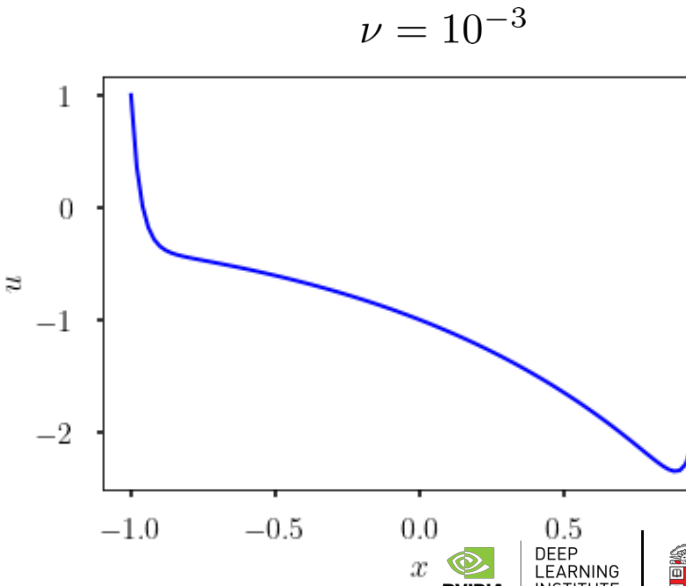
$$\nu \frac{\mathrm{d}^2 u}{\mathrm{d}x^2} - u = e^x,$$

where $x \in [-1, 1],\ u(-1) = 1, u(1) = 0.$

- Boundary Layer vs Viscosity

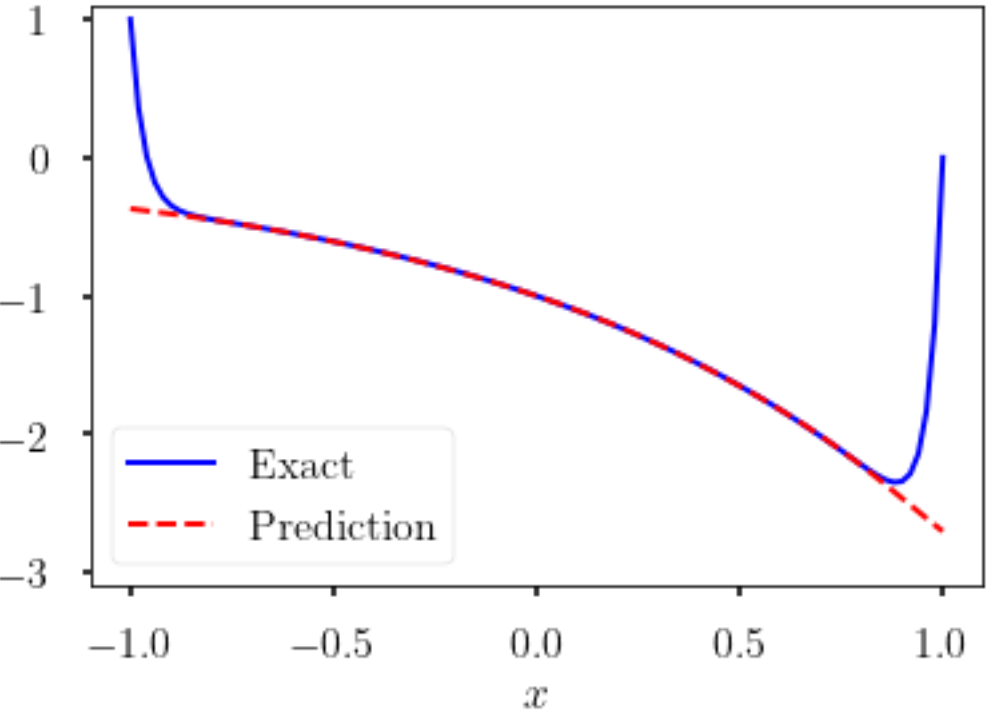


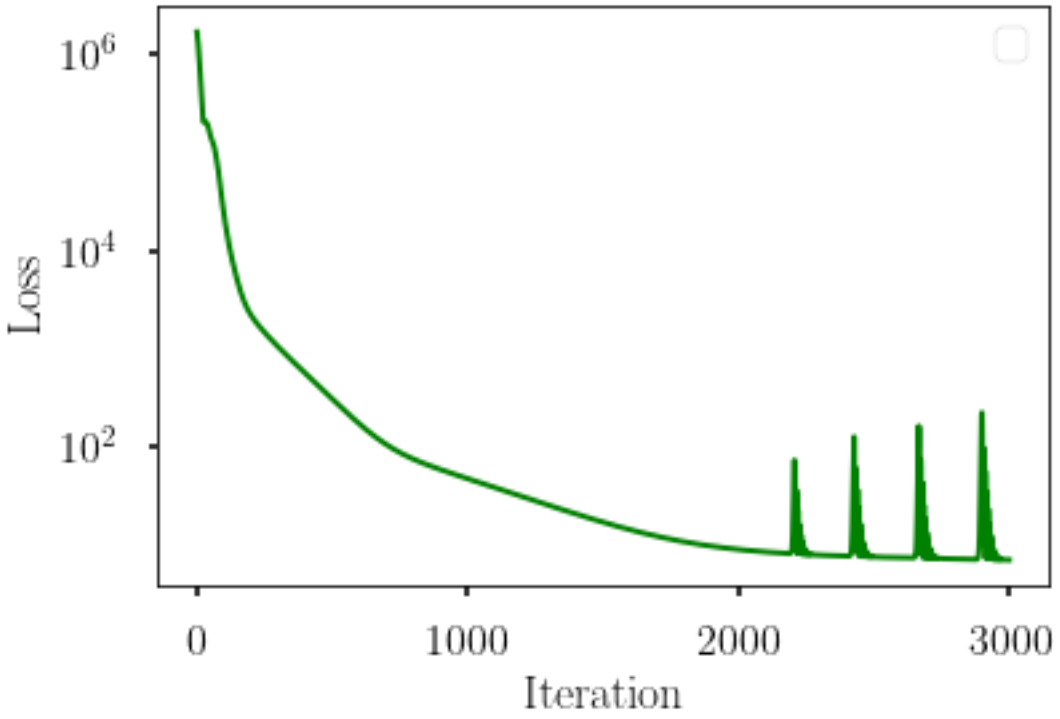




proach 1: PINNs with Soft Constraints

- Results





proach 2: Self-Adaptive PINNs

is Defined as

$$\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x},t)] = 0, \quad \boldsymbol{x} \in \Omega, t \in [0,T]$$
$$u(\boldsymbol{x},t) = g(\boldsymbol{x},t), \quad \boldsymbol{x} \in \partial\Omega, t \in [0,T]$$
$$u(\boldsymbol{x},0) = h(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega$$

function

$$\mathcal{L}(\boldsymbol{w}) = \mathcal{L}_s(\boldsymbol{w}) + \mathcal{L}_r(\boldsymbol{w}) + \mathcal{L}_b(\boldsymbol{w}) + \mathcal{L}_0(\boldsymbol{w})$$

$$\mathcal{L}_s(\boldsymbol{w}) = \frac{1}{N_s}\sum_{i=1}^{N_s}\left|u\left(\boldsymbol{x}_s^i, t_s^i; \boldsymbol{w}\right) - y_s^i\right|^2$$
physics data

$$\mathcal{L}_r(\boldsymbol{w}) = \frac{1}{N_r}\sum_{i=1}^{N_r} r\left(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w}\right)^2$$
physics

$$\mathcal{L}_b(\boldsymbol{w}) = \frac{1}{N_b}\sum_{i=1}^{N_b}\left|u\left(\boldsymbol{x}_b^i, t_b^i; \boldsymbol{w}\right) - g_b^i\right|^2$$
physics data

$$\mathcal{L}_0(\boldsymbol{w}) = \frac{1}{N_0}\sum_{i=1}^{N_0}\left|u\left(\boldsymbol{x}_0^i, 0; \boldsymbol{w}\right) - h_0^i\right|^2$$
physics data

- McClenny L, Braga-Neto U. Self-adaptive physics-informed neural ne using a soft attention mechanism. arXiv preprint arXiv:2009.04544. 20

Self-adaptive PINN utilizes the following loss function

$$\mathcal{L}(w, \lambda_r, \lambda_b, \lambda_0) = \mathcal{L}_s(w) + \mathcal{L}_r(w, \lambda_r) + \mathcal{L}_b(w, \lambda_b) + \mathcal{L}_0(w, \lambda_0)$$

where $\lambda_r = \left(\lambda_r^1, \ldots, \lambda_r^{N_r}\right), \lambda_b = \left(\lambda_b^1, \ldots, \lambda_b^{N_b}\right)$, and $\lambda_n = \left(\lambda_0^1, \ldots, \lambda_0^{N_0}\right)$ are trainable selfadap weights for the initial, boundary, and collocation points, respectively, and

$$\mathcal{L}_r(w, \lambda_r) = \frac{1}{N_r}\sum_{i=1}^{N_r}\left[\lambda_r^i r\left(x_r^i t_r^1 : w\right)\right]^2$$

$$\mathcal{L}_b(w, \lambda_b) - \frac{1}{N_b}\sum_{i=1}^{N_b}\left[\lambda_b^i \left(u\left(x_b^i, t_{b,}^i; w\right) - g_b'\right)\right]^2$$

$$\mathcal{L}_0(w, \lambda_0) = \frac{1}{N_0}\sum_{i=1}^{N_0}\left[\lambda_0^i \left(u\left(x_0^i, 0; w\right) - h_0^i\right)\right]^2$$

The key feature of self-adaptive PINNs is that the loss $\mathcal{L}(w, \lambda, \lambda, \lambda_b, \lambda_0)$ is minimized with resp the network weights w, as usual, but is maximized with respect to the self-adaptation weights λ_r, in other words, training seeks a saddle point

$$\min_w \max_{r,\lambda_b,\lambda_0} \mathcal{L}(w, \lambda_r, \lambda_b, \lambda_0).$$

This can be accomplished by a gradient descentascent procedure, with updates given by:

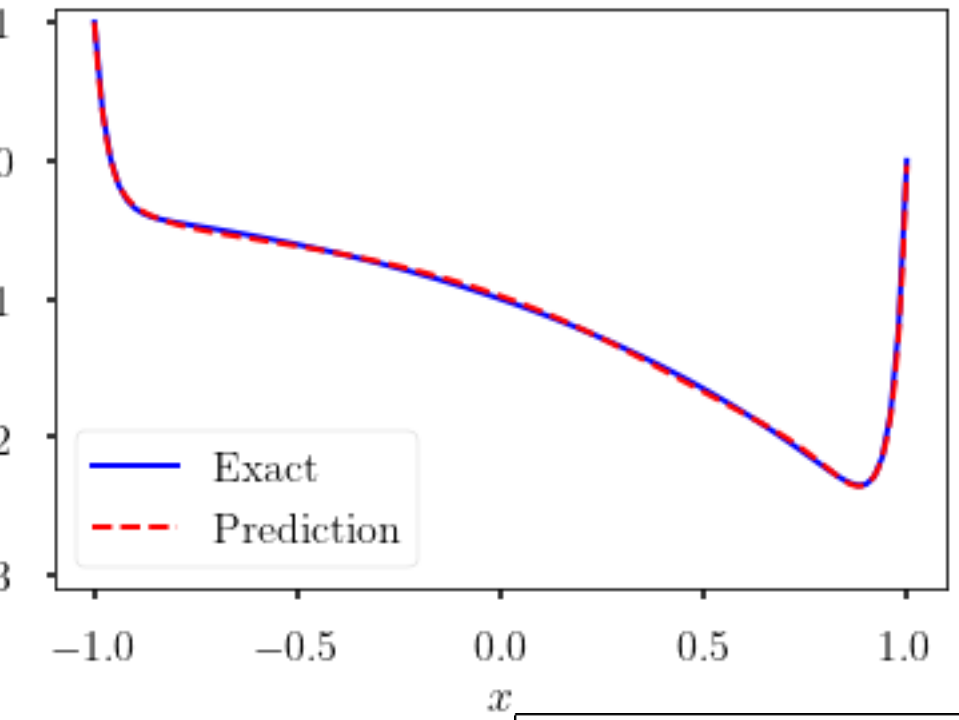
gradient descent

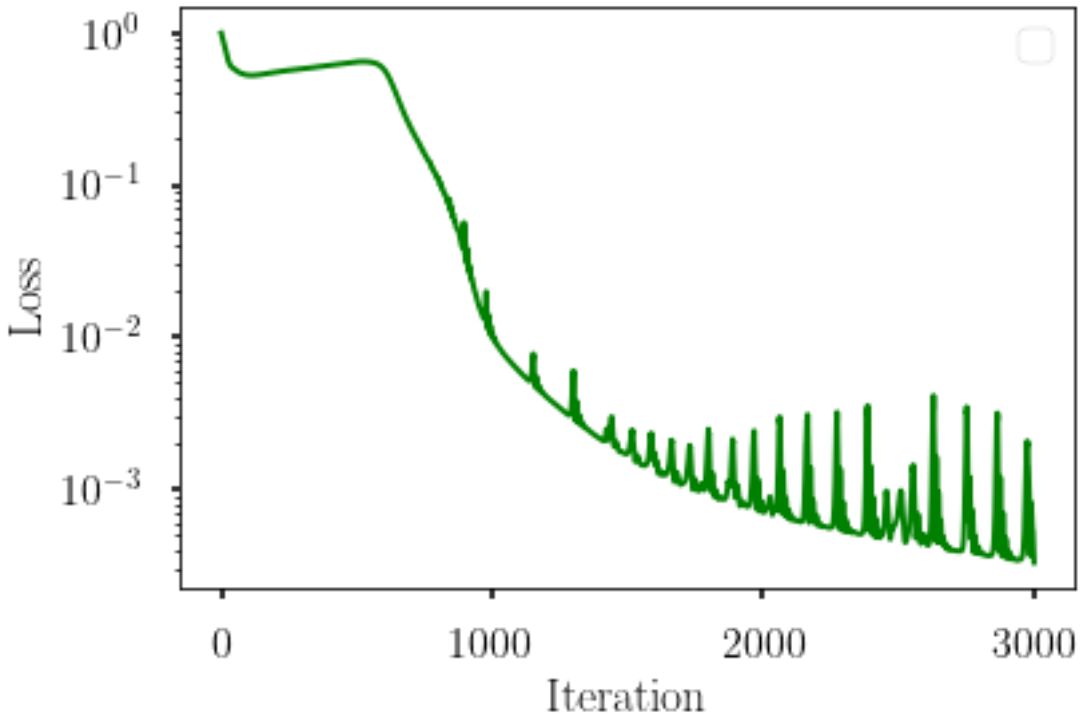
$$w^{k-1} = w^k - \eta_k \nabla_w \mathcal{L}\left(w^k, \lambda_r^k, \lambda_i^k, \lambda_0^k\right)$$

gradient ascent

$$\lambda_r^{k+1} = \lambda_r^k + \eta_k \nabla_{\lambda_r} \mathcal{L}\left(w^k, \lambda_r^k, \lambda_b^k, \lambda_0^k\right)$$
$$\lambda_b^{k+1} = \lambda_b^k + \eta_k \nabla_{\lambda_b} \mathcal{L}\left(w^k, \lambda_r^k, \lambda_b^k, \lambda_0^k\right)$$
$$\lambda_0^{k+1} = \lambda_0^k + \eta_k \nabla_{\lambda_0} \mathcal{L}\left(w^k, \lambda_r^k, \lambda_b^k, \lambda_0^k\right)$$

proach 2: Results





Self-Adaptive PINN Architecture	
Hyper-parameters	**Value**
No. of Layers	8
Neurons per layer	8
No. of Iterations	1500
Learning rate	1×10^{-3}
No. of Residuals Points	500

Residual Based Attention (RBA) Weights

(Anagnostopoulos et al., CMAME 2024)

Weighting strategies:

Modify loss terms contribution during training

Fixed global multipliers (per loss term):

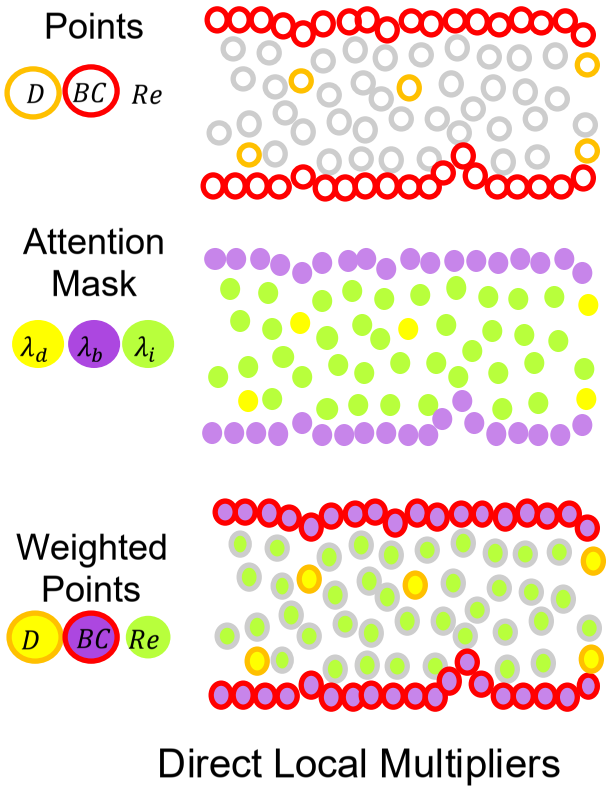
$$\mathcal{L} = \lambda_D \mathcal{L}_{Data} + \lambda_B \mathcal{L}_{BCs} + \lambda_{PDE} \mathcal{L}_{PDE}$$

Adaptive local multipliers (per training point):

$$\mathcal{L} = \langle (\lambda_d \cdot e_d)^2 \rangle + \langle (\lambda_b \cdot e_b)^2 \rangle + \langle (\lambda_i \cdot r_i)^2 \rangle$$

Requirements for λ_i^k:

- Identify high error regions ⇨ $f(r_i)$
- Efficient and easy to compute ⇨ $|r_i|$
- Stable ⇨ $\eta |r_i| / |r_{max}|$
- Remember the previous iterations ⇨ $f(\lambda_i^{k-1})$
- Relate to the current training state ⇨ $\gamma f(\lambda_i^{k-1})$



Residual Based Attention (RBA) Weights

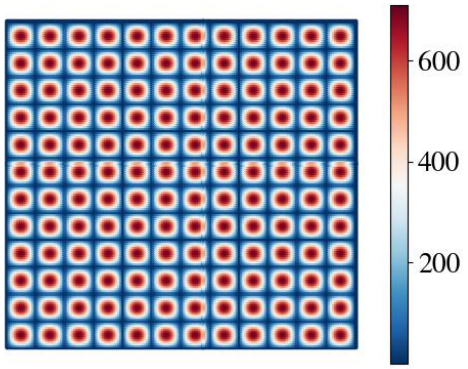
(Anagnostopoulos et al., CMAME 2024)

Residual-based attention weights (Cumulative residuals)

$$\lambda_i^k = \gamma \lambda_i^{k-1} + \frac{|r_i|}{\max\limits_{0<j<N} |r_j|}$$

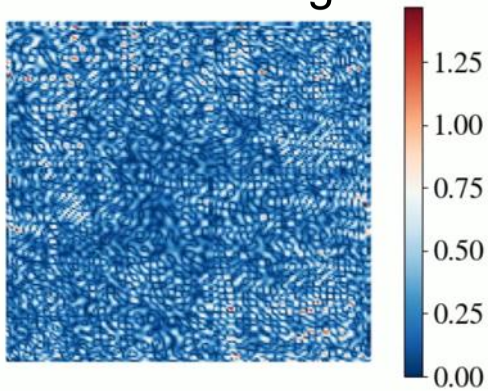
Helmholtz Equation

Ordered initial states



Residuals transition

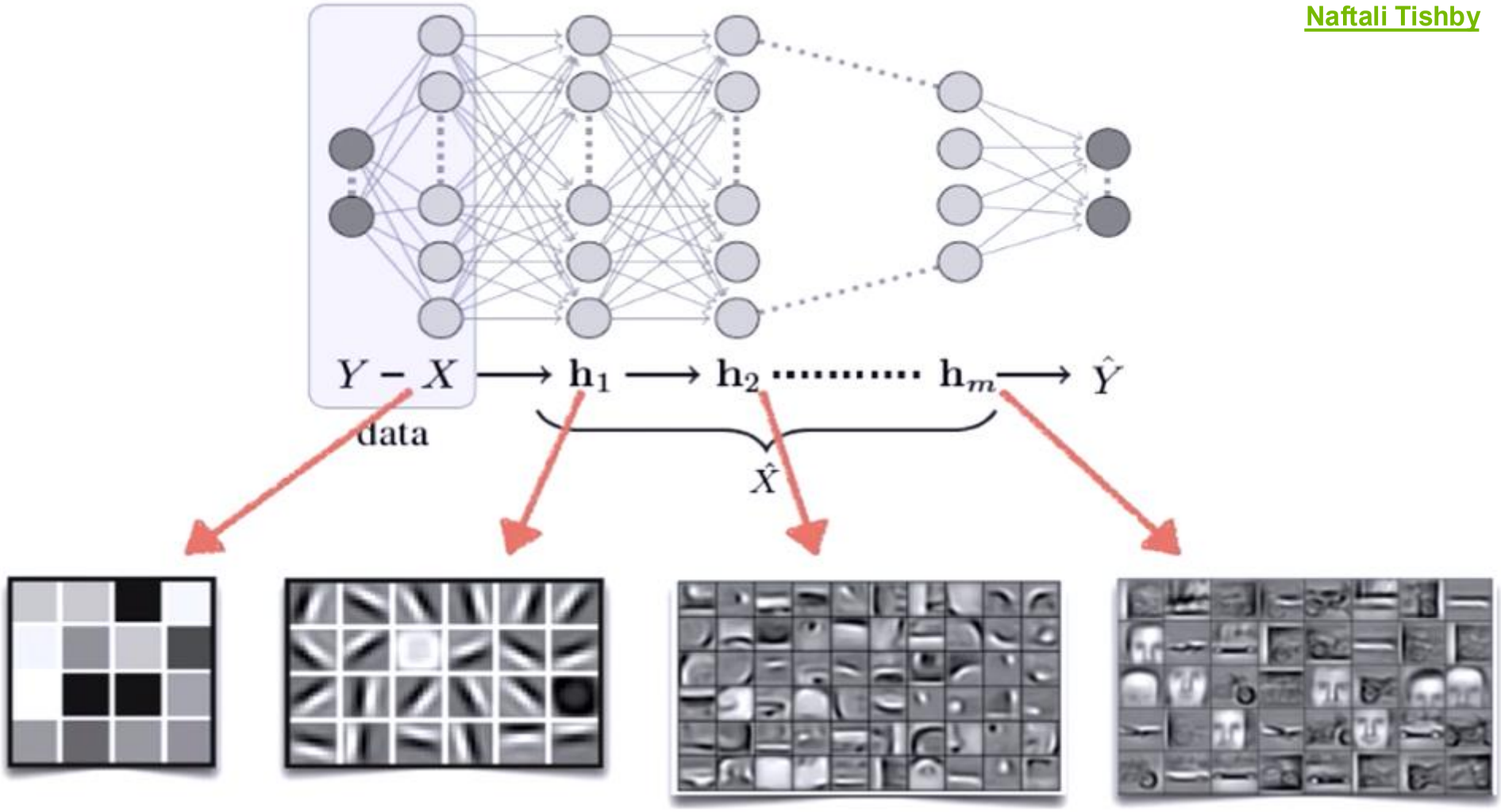
Chaotic final stages



Weight and residual evolution reveal two distinct phases akin to “fitting” and “diffusion.”

formation Bottleneck Theory for Deep Learning

Naftali Tishby



formation Bottleneck Theory for Deep Learning

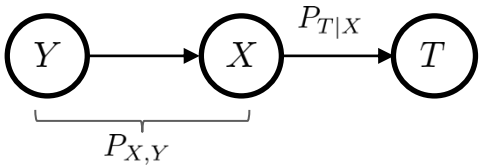
- Entropy: $H(p) = -\sum_i p_i log_2(p_i)$
- Cross-Entropy: $H(p,q) = -\sum_i p_i log_2(q_i)$
- KL Divergence: $D(p||q) = H(p,q) - H(p)$
- KL-distribution divergence: for any two distributions $p(x), q(x)$ over X:

$$D[p(x)||q(x)] = \sum_x p(x) log \frac{p(x)}{q(x)} \geq 0$$

ormation Bottleneck Theory for Deep Learning

for any two random variables X, Y

- The mutual Information: $I(X;Y) = D[p(x,y)||p(x)p(y)] = D[p(x|y)||p(y)] = D[p(y|x)||p(x)]$



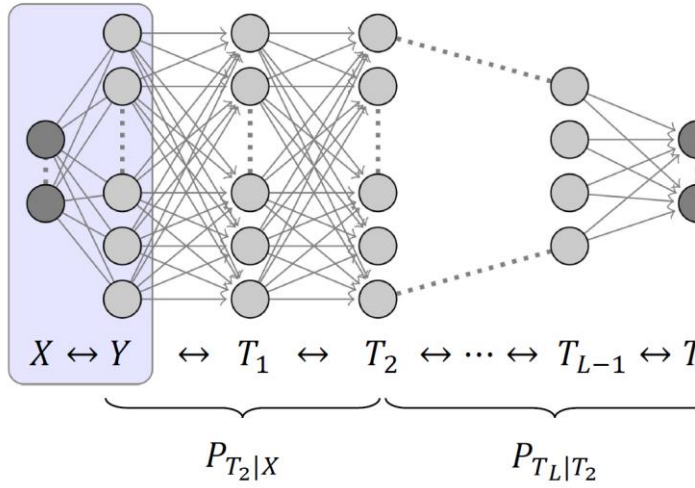
Graphical representation of probabilistic relations in framework. The triple (X, Y, T) is jointly distributed accord $P_{X,Y}P_{T|X}$, thus forming a Markov chain $Y \leftrightarrow X \leftrightarrow T$. Th is to find a compressed representation T of X (via the tra kernel $P_{T|X}$), i.e., minimize $I(X;T)$, while preserving at l bits of the information about Y, i.e., $I(Y;T) \geq \alpha$

- Data Processing Inequality (DPI) & Invariance:

  for any Markov chain $X \leftrightarrow Y \leftrightarrow Z$: $I(X;Y) \geq I(X;Z)$

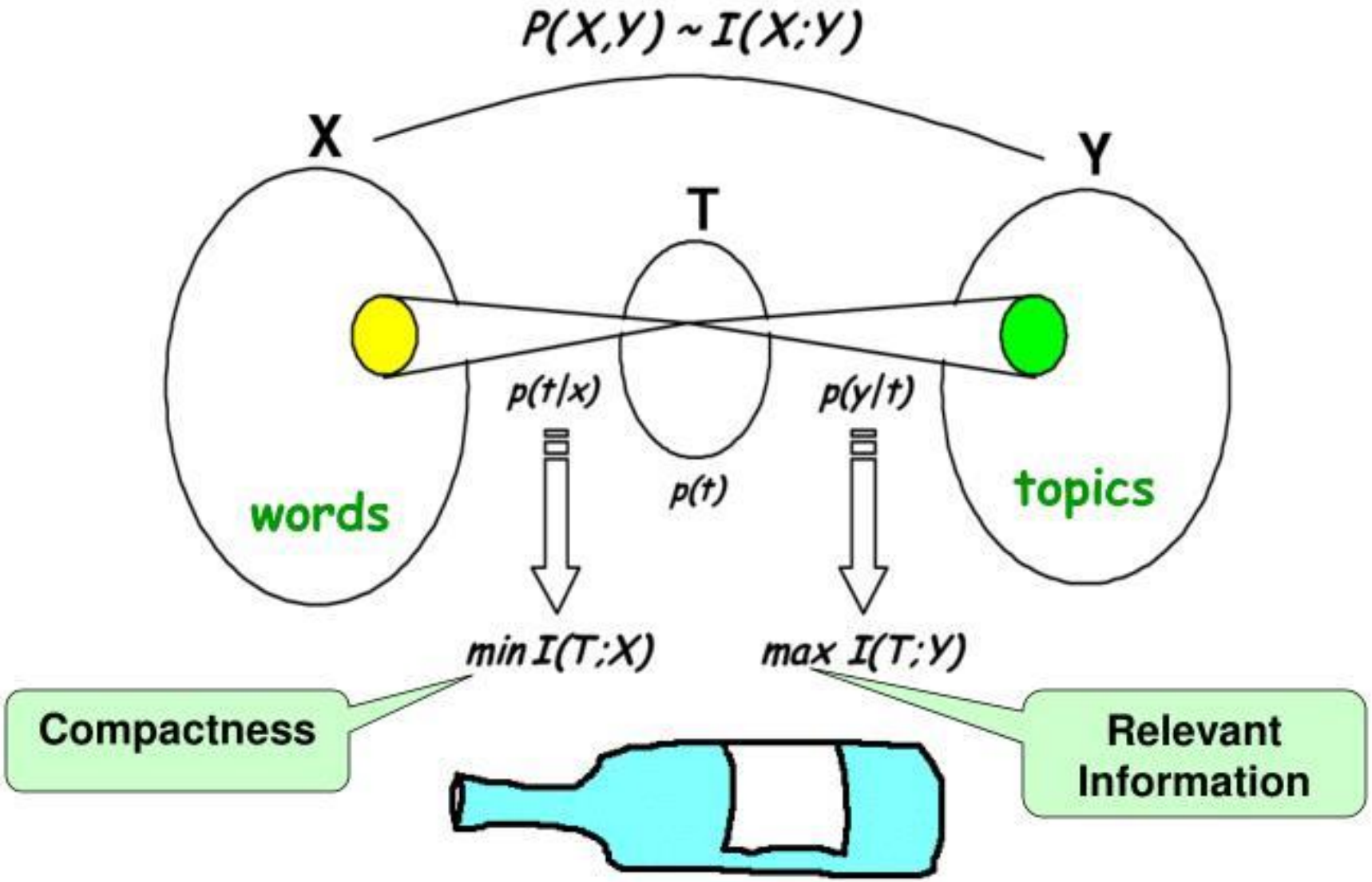
  Parameterization Invariance for invertible ϕ, ψ: $I(X;Y) = I(\phi(x);\psi(x))$

- Information Plane: For large typical X, the sample complexity of a DNN is completely determined by the encoder mutual information $I(X;T)$, of the last hidden layer. The accuracy (generalization error) is determined by the decoder information $I(T;Y)$ of the last hidden layer



1. Tishby, Naftali, and Noga Zaslavsky. "*Deep learning and the information bottleneck principle.*" *Information Theory Workshop (ITW), 2015 IEEE*. IEEE, 2015.
2. Shwartz-Ziv, Ravid, and Naftali Tishby. "*Opening the black box of deep neural networks via information.*" *arXiv preprint arXiv:1703.00810* (2017).

Information Bottleneck



grange Dual Form and Phase Transition

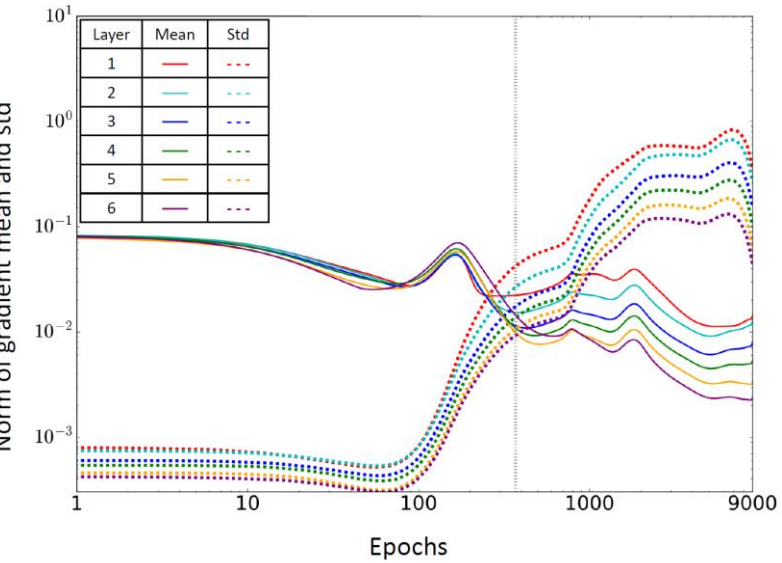
$$\mathcal{L}_\beta(P_{T|X}) := I(X;T) - \beta I(T;Y)$$

- With this definition, the IB problem can be recast as minimizing $\mathcal{L}_\beta(P_{T|X})$ over all possible $P_{T|X}$
- β controls the amount of compression in the representation T of X:
  - Small β implies more compression (sacrificing informativeness)
  - Larger β pushes towards finer representation granularity that favors informativeness.
  - Varying β regulates the tradeoff between informativeness and compression
- The working assumption of IB theory is that by optimizing the DNN's layers $T_l\{l = 1, ..., L\}$ for that task via standard SGD, the encoder-decoder pair for each hidden layer converges to optimal IB solution.
- The compression phase was further claimed to be responsible for the outstanding generalization performance of deep networks although no rigorous connection between $I(X;T_l)$ and the generalization error is currently known

A. M. Saxe, Y. Bansal, J. Dapello, M. Advani, A. Kolchinsky, B. D. Tracey, and D. D. Cox, "On the information bottleneck theory of deep learning," in Proceedings of the International Conference on Lea Representations (ICLR), 2018.

B. Z. Goldfeld, E. van den Berg, K. Greenewald, I. Melnyk, N. Nguyen, B. Kingsbury, and Y. Polyanskiy, "Estimating information flow in neural networks," in Proceedings of the International Conference Machine Learning (ICML-2019), vol. Long Beach, California, USA, Jun. 2019.

agrange Dual Form and Phase Transition: Summary



Norms of gradient mean (solid lines) and standard deviation (dashed lines) for the different layers. For each layer, the values are normalized by the L2 norm of the corresponding weight matrix. The grey vertical line marks the epoch when a phase transition between a high-SNR to a low-SNR occurs.

- First, they observed that the training process comprises two consecutive phases, termed 'fitting' and 'compression': after a quick initial fitting phase, almost the entire training process is dedicated to compressing X, while staying informative about Y .
- Second, it was observed that the compression phase starts when the mean and variance of the stochastic gradient undergoes a phase transition from high to low signal-to-noise (SNR).
- Third, it was observed that the two training phases accelerate when more layers are added to the network, which led to an argument that the benefit of deeper architectures is computational.

NEW: More recent results identified that for deterministic DNNs, IB does not apply. They als disputed the compression phase and argued that the geometric clustering of representations i the fundamental phenomenon of interest, while elucidated some of the machinery DNN employ for learning. Leveraging the clustering perspective, evidence shows that compressio and generalization may not be causally related.

A. M. Saxe, Y. Bansal, J. Dapello, M. Advani, A. Kolchinsky, B. D. Tracey, and D. D. Cox, "On the information bottleneck theory of deep learning," in Proceedings of the International Conference on Learning Representations (ICLR), 2018.
B. Z. Goldfeld, E. van den Berg, K. Greenewald, I. Melnyk, N. Nguyen, B. Kingsbury, and Y. Polyanskiy, "Estimating information flow in neural networks," in Proceedings of the International Conference on Machine Learning (ICML-2019), vol. Long Beach, California, USA, Jun. 2019.

Information Bottleneck Theory

Equation:

- Helmholtz

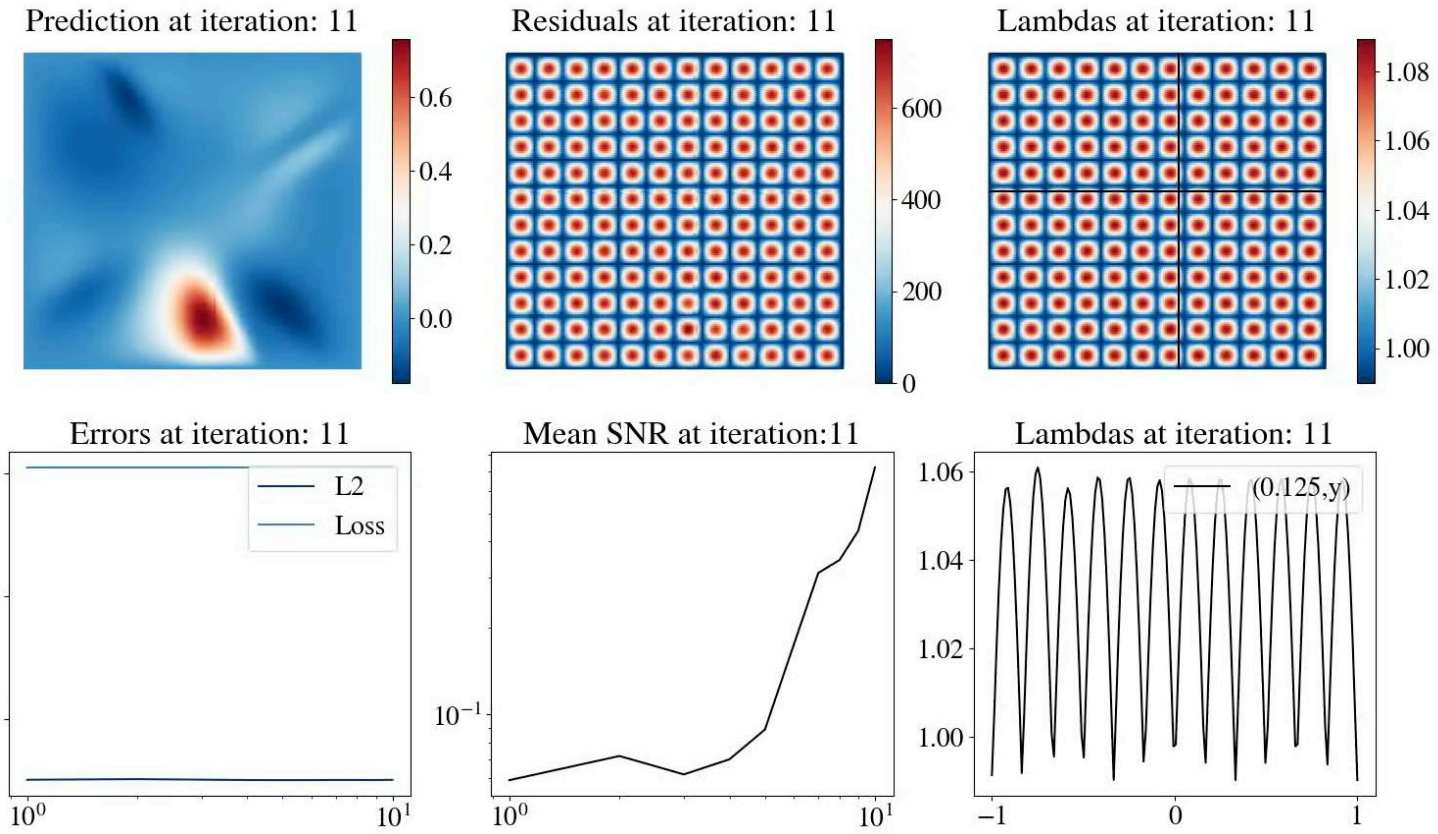
Features:

- Exact Boundary Imposition
- RBA

$$\|\mu_l\| = \frac{\|\langle \nabla_w \mathcal{L}_i \rangle\|_F}{\|w_i\|_F}$$

$$\|\sigma_l\| = \frac{\|std(\nabla_w \mathcal{L}_i)\|_F}{\|w_i\|_F}$$

$$SNR = \frac{\|\mu_l\|}{\|\sigma_l\|},$$



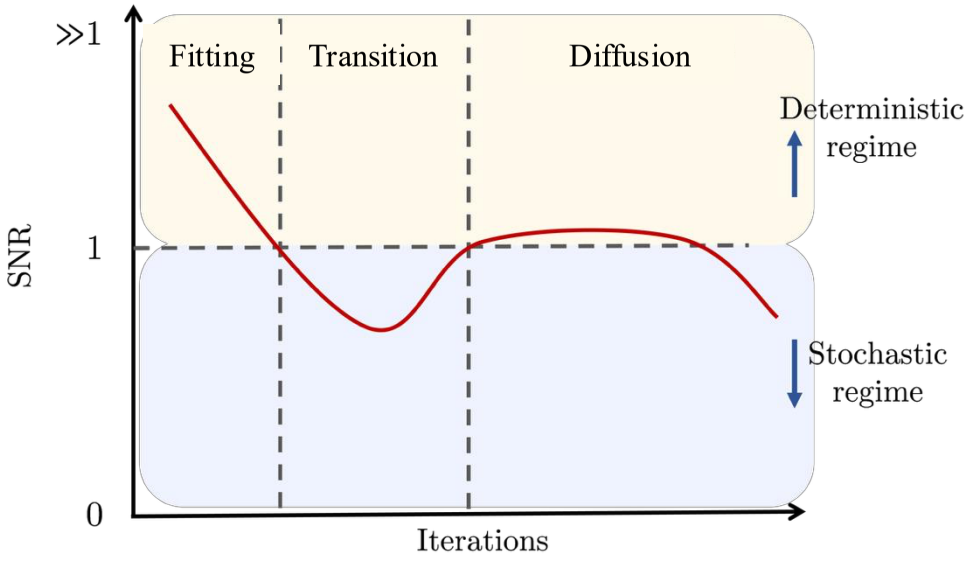
IB Theory: Learning Regimes

- **Fitting**: Neural gradients undergo a phase transition from high to low SNR
- **Transition**: The gradient fluctuations dominate the signal.
- **Diffusion**: Sudden jump on SNR due to gradient homogeneity (i.e., inner batches agree with each other)

$$\|\mu_l\| = \frac{\|\langle \nabla_w \mathcal{L}_i \rangle\|_F}{\|w_i\|_F}$$

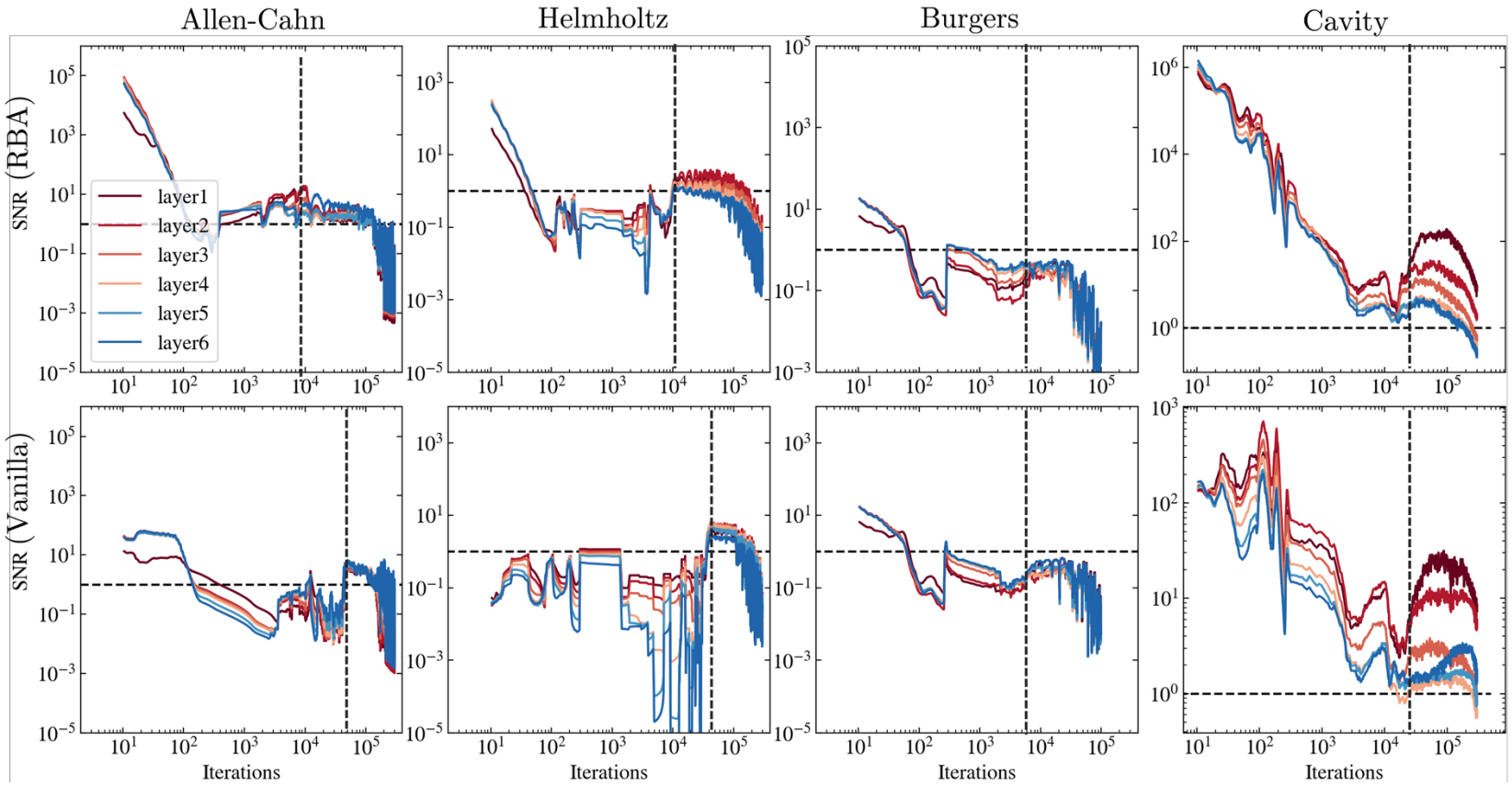
$$\|\sigma_l\| = \frac{\|std(\nabla_w \mathcal{L}_i)\|_F}{\|w_i\|_F}$$

$$SNR = \frac{\|\mu_l\|}{\|\sigma_l\|},$$



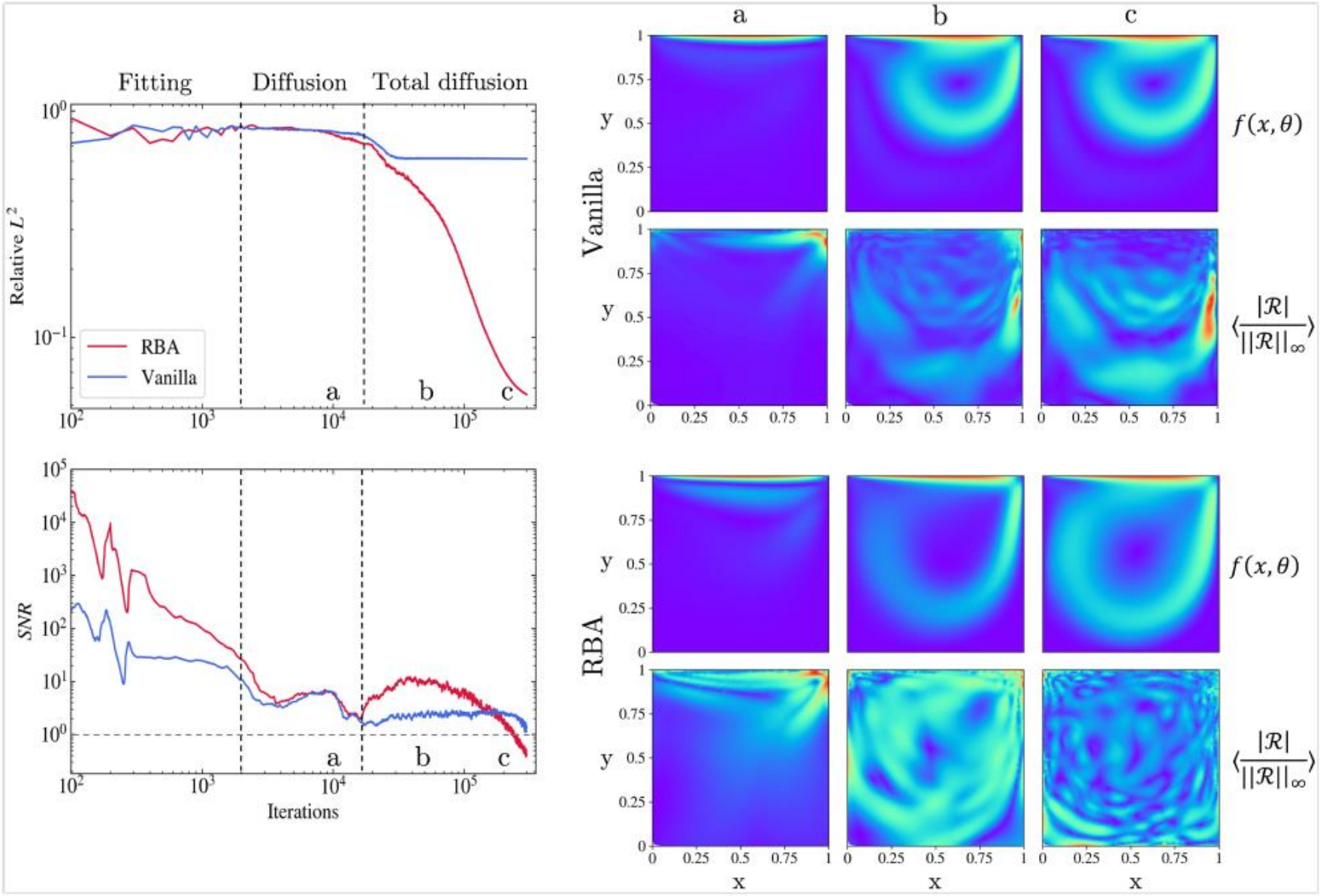
Gradient-based optimization regimes

IB Theory: Learning Regimes



Layer-wise SNR -tanh activation function. (Anagnostopoulos et al., 2024)

IB Theory: Cavity Flow, Re=1000



eighted Residual Methods

Problem Definition:

$$
\begin{aligned}
\mathcal{L}^q u(\boldsymbol{x}, t) &= f(\boldsymbol{x}, t), \quad && (\boldsymbol{x}, t) \in \Omega \times (0, T] && \text{:PDE} \\
u(\boldsymbol{x}, t) &= h(\boldsymbol{x}, t), && (\boldsymbol{x}, t) \in \partial\Omega \times [0, T] && \text{:B.C.} \\
u(\boldsymbol{x}, 0) &= g(\boldsymbol{x}), && \boldsymbol{x} \in \Omega && \text{:I.C.}
\end{aligned}
$$

$\Omega \subset \mathbb{R}^d$ – Bounded domain with boundaries $\partial\Omega$

$\mathcal{L}^q$ – Differential/Integral/Identity operator with parameters q

$u(\boldsymbol{x}, t)$ – Solution to be obtained

f, h, g – Known source, boundary and initial conditions

Given the PDE and boundary/initial conditions, what is the solution $u(\boldsymbol{x}, t)$?

Ns Family: Weighted Residual Methods

pproximation: $u \approx \tilde{u} = u_{NN}$

trong-Form Residuals:

$r(\tilde{u}) = \mathcal{L}^q \tilde{u} - f$: PDE residual

$_b(\tilde{u}) = \tilde{u} - h$: boundary residual

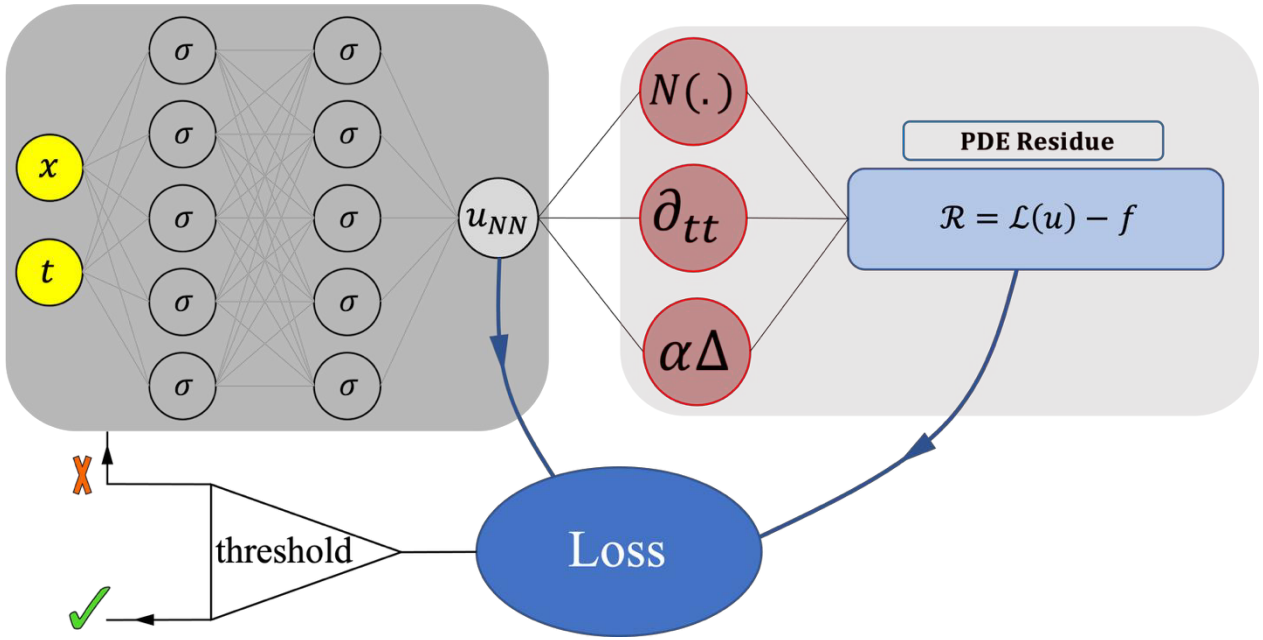
$_0(\tilde{u}) = \tilde{u} - g$: initial residual

Loss Function:

$$L^s(\tilde{u}) = \frac{1}{N_r}\sum_{i=1}^{N_r} \left|r(\boldsymbol{x}_r^i, t_r^i)\right|^2 + \tau_b \frac{1}{N_b}\sum_{i=1}^{N_b} \left|r_b(\boldsymbol{x}_b^i, t_b^i)\right|^2 + \tau_0 \frac{1}{N_0}\sum_{i=1}^{N_0} \left|r_0(\right.$$

physics data physics data physics

minimization problem: $\min L^s(\tilde{u})$



Ns Family: Weighted Residual Methods

Veighted/Varialtional Residuals:
rojections onto space of test function V

$$\mathcal{R}_j(\tilde{u}) = \int_{\Omega\times(0,T]} r(\tilde{u})v_j dxdt$$

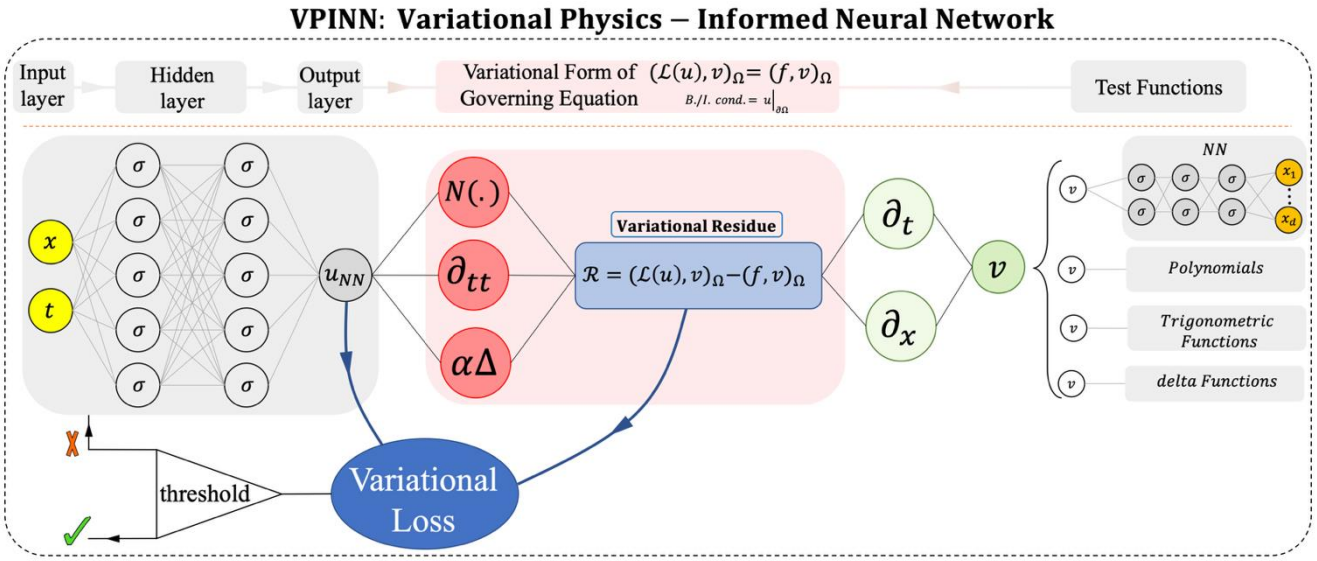
$$\mathcal{R}_{b,j}(\tilde{u}) = \int_{\partial\Omega\times(0,T]} r_b(\tilde{u})v_j dxdt$$

$$\mathcal{R}_{0,j}(\tilde{u}) = \int_{\Omega} r_0(\tilde{u})v_j dx$$

Loss Function:

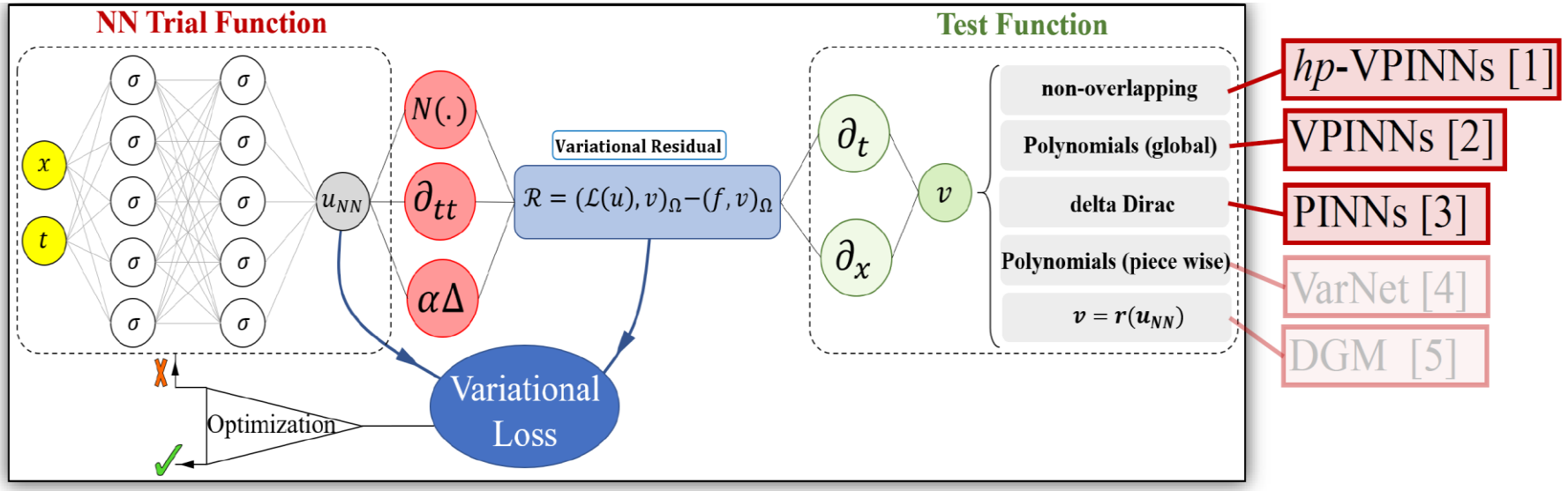
$$L^{\mathfrak{v}}(\tilde{u}, v) = w\sum_{j=1}^{N_r}\mathcal{R}_j^2(\tilde{u}) + w_b\sum_{j=1}^{N_b}\mathcal{R}_{b,j}^2(\tilde{u}) + w_0\sum_{j=1}^{N_0}\mathcal{R}_{0,j}^2(\tilde{u})$$

minimization problem: $\min L^{\mathfrak{v}}(\tilde{u}, v)$



Kharazmi E, Zhang Z, Karniadakis GE. Variational physics-informed neural networks for solving partial differential equations. arXiv preprint arXiv:1912.00873. 2019 Nov 27.

VPINNs: A General Framework for Solving PDEs



	Local NN	Global NN
Local v	Conservative VPINNs [6]	*hp*-VPINNs [5]
Global v	-	VPINNs[3], VarNet[4], D3M [7]

Global:= single domain

Local:= domain decomposition

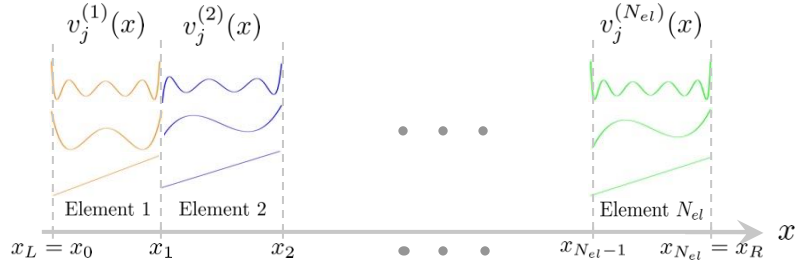
[1] Kharazmi et al., CMAME (2020).
[2] Kharazmi et al., *arXiv* (2019).
[3] Raissi et al., JCP (2019)
[4] Khodayi-Mehr et al., arXiv (2019).
[5] Sirignano et al., JCP (2018)
[6] Jagtap et al., CMAME (2020)
[7] Li et al., arXiv (2019)

VPINNs: A General Framework for Solving PDEs

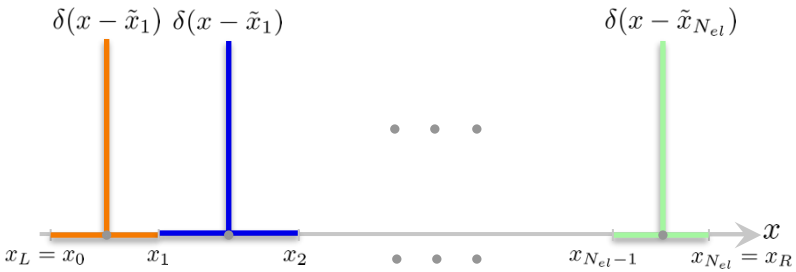
$$\Omega = \bigcup_e \Omega^{(e)}$$

$$v_j^{(e)} = \begin{cases} \bar{v}_j \neq 0, & x \in \Omega^{(e)} \\ 0, & \text{otherwise} \end{cases}$$

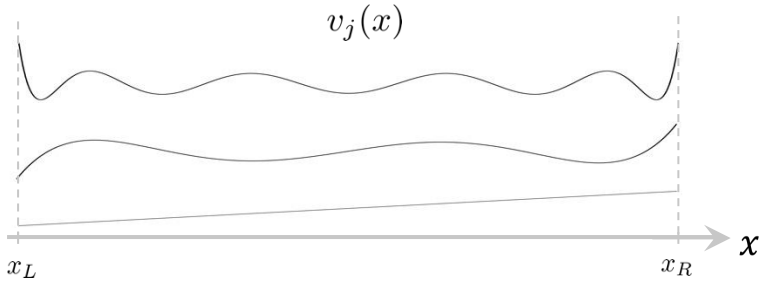
h-refinement & p-refinement



PINNs — ***hp*-VPINNs** — **VPINNs**



Delta Dirac test functions



p-refinement

- nonlinear approximation via neural networks
- h-refinement via domain decomposition
- p-refinement via projection onto high-order polynomial space

VPINNs: A General Framework for Solving PDEs

ss Function

elemental/local variational loss

$$\text{VPINNs:} \quad L^{\mathfrak{v}} = \boxed{\sum_{e=1}^{N_{el}} \frac{1}{N_{el} K^{(e)}} \sum_{k=1}^{K^{(e)}} \left| \mathcal{R}_k^{(e)} \right|^2} + \tau_b \frac{1}{N_b} \sum_{i=1}^{N_b} \left| r_b(\boldsymbol{x}_b^i, t_b^i) \right|^2 + \tau_0 \frac{1}{N_0} \sum_{i=1}^{N_0} \left| r_0(\boldsymbol{x}_0^i) \right|^2$$

physics data physics data

global variational loss

$$\text{VPINNs:} \quad L^{\mathfrak{v}} = \boxed{\frac{1}{K} \sum_{j=1}^{K} \left| \mathcal{R}_j \right|^2} + \tau_b \frac{1}{N_b} \sum_{i=1}^{N_b} \left| r_b(\boldsymbol{x}_b^i, t_b^i) \right|^2 + \tau_0 \frac{1}{N_0} \sum_{i=1}^{N_0} \left| r_0(\boldsymbol{x}_0^i) \right|^2$$

physics data physics data

strong-form loss

$$\text{PINNs:} \quad L^{\mathfrak{s}} = \boxed{\frac{1}{N_r} \sum_{i=1}^{N_r} \left| r(\boldsymbol{x}_r^i, t_r^i) \right|^2} + \tau_b \frac{1}{N_b} \sum_{i=1}^{N_b} \left| r_b(\boldsymbol{x}_b^i, t_b^i) \right|^2 + \tau_0 \frac{1}{N_0} \sum_{i=1}^{N_0} \left| r_0(\boldsymbol{x}_0^i) \right|^2$$

physics data physics data physics data

ational Neural Networks (VNNs): Function Approximation

Variational residual:
$$\mathcal{R}_k^{(e)} = \int_{\Omega_e} (u_{NN}(\boldsymbol{x}) - u(\boldsymbol{x}))v_k^{(e)}(\boldsymbol{x})d\Omega_e$$

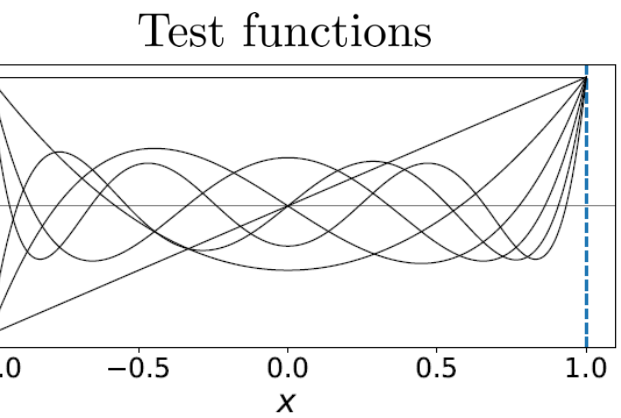
Variational loss:
$$L^{\mathfrak{v}} = \sum_{e=1}^{N_{el}} \frac{1}{K^{(e)}} \sum_{k=1}^{K^{(e)}} \left|\mathcal{R}_k^{(e)}\right|^2$$

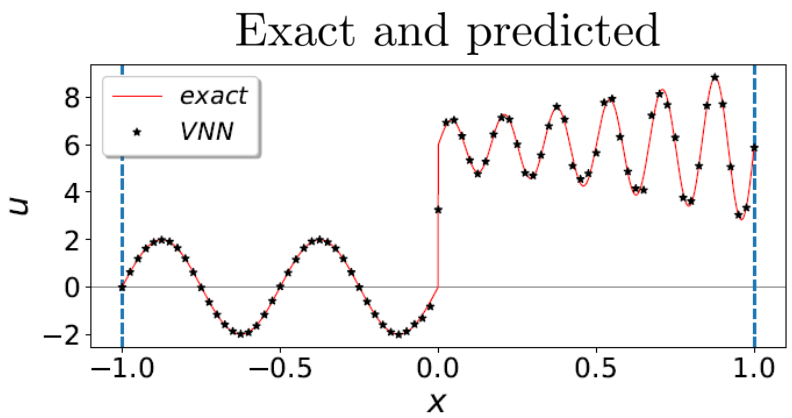
Example (discontinuous function approximation): the target function is

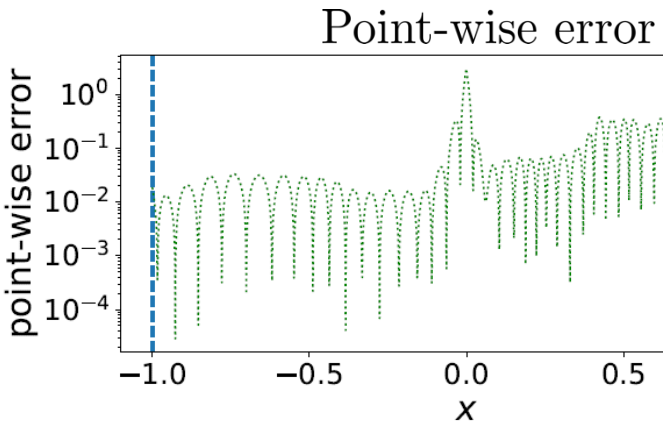
$$u^{exact} = \begin{cases} 2\sin(4\pi x) & x \in [-1, 0], \\ 6 + \exp^{1.2x}\sin(12\pi x) & x \in (0, 1] \end{cases}$$

ational Neural Networks (VNNs): Function Approximation

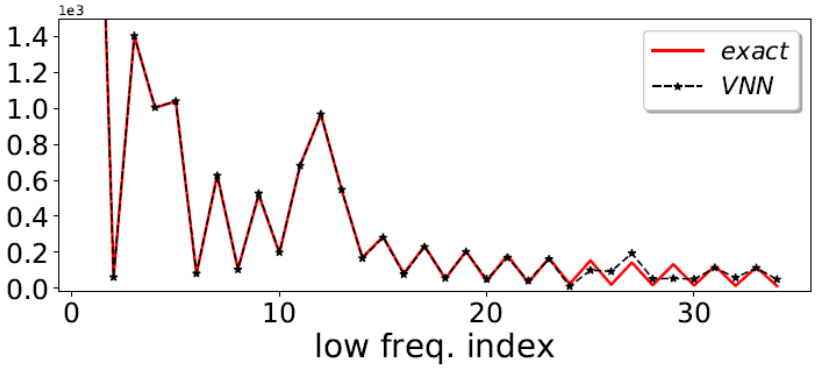
lobal test functions (single element):

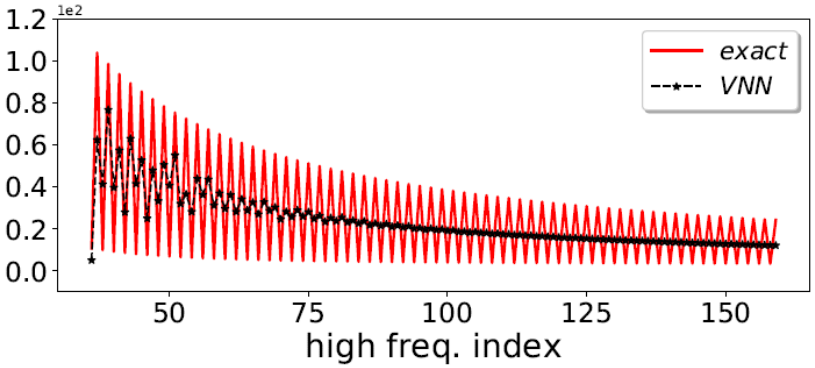






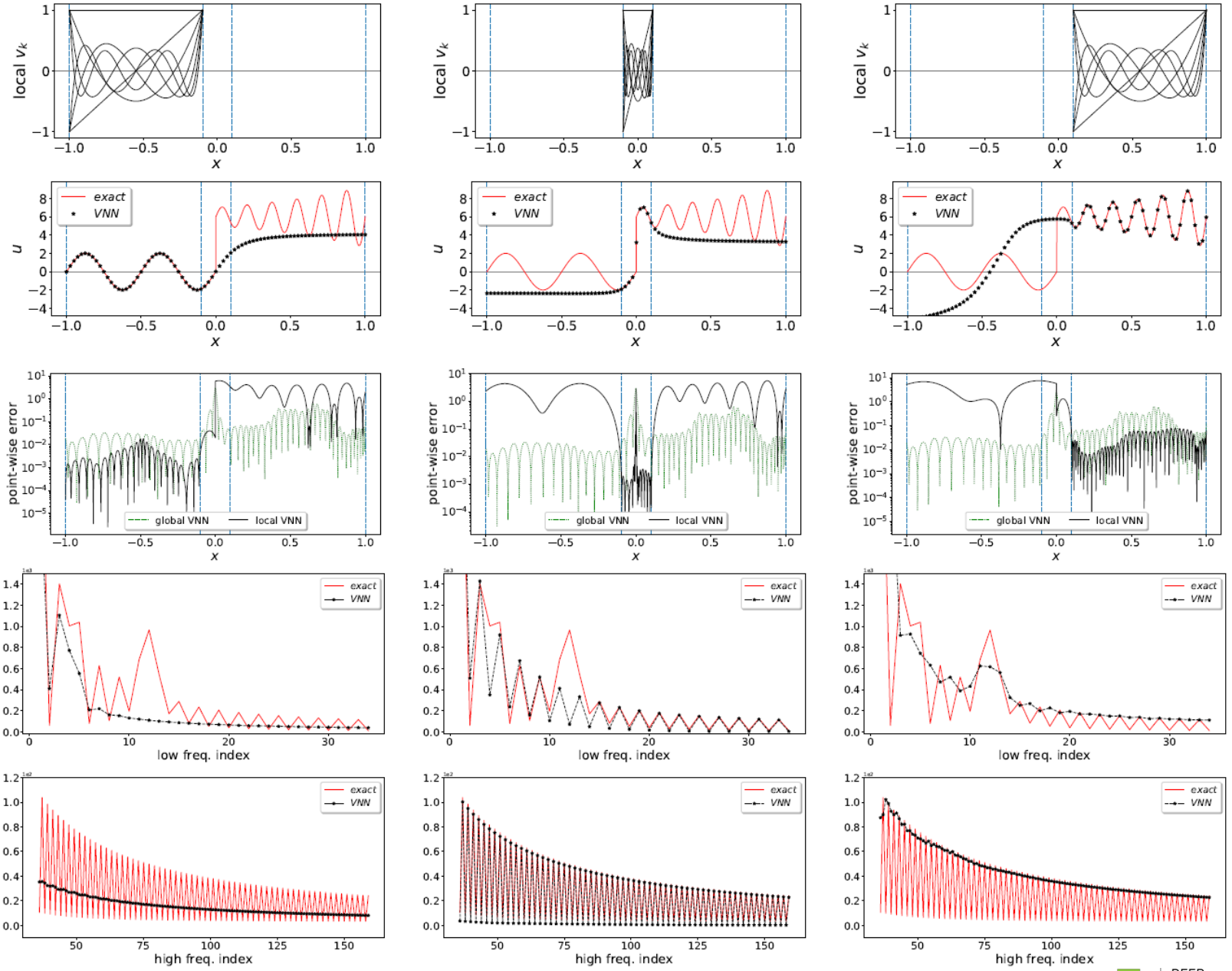
ectrum





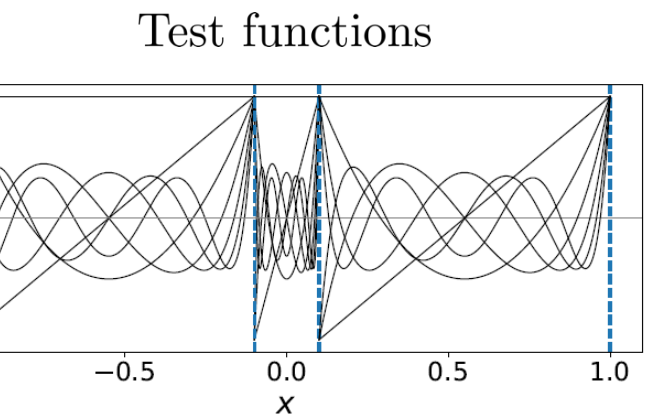
ational Neural Networks (VNNs): Function Approximation

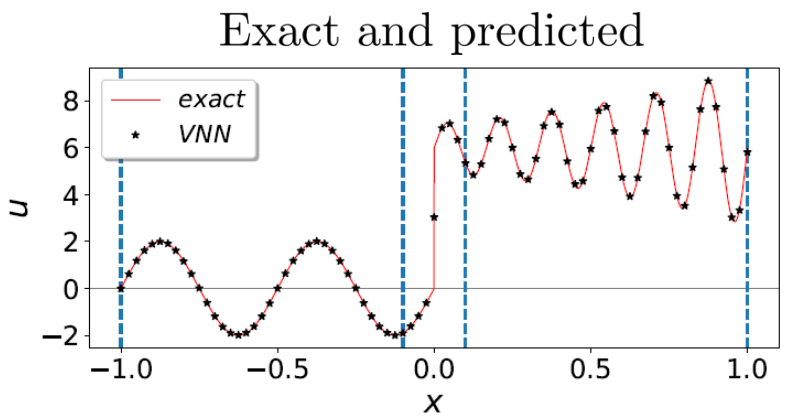
ocal (elemental)
st functions:

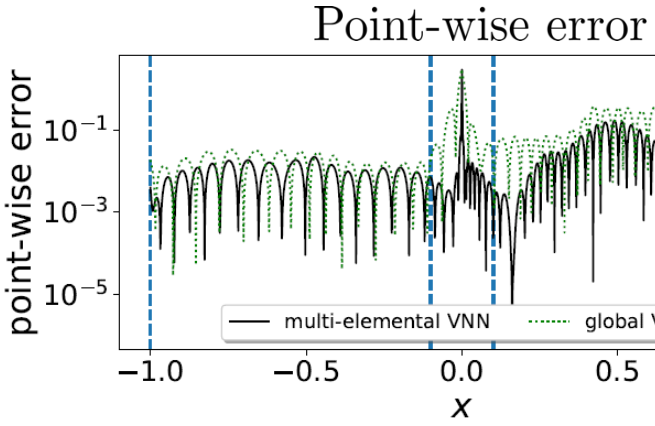


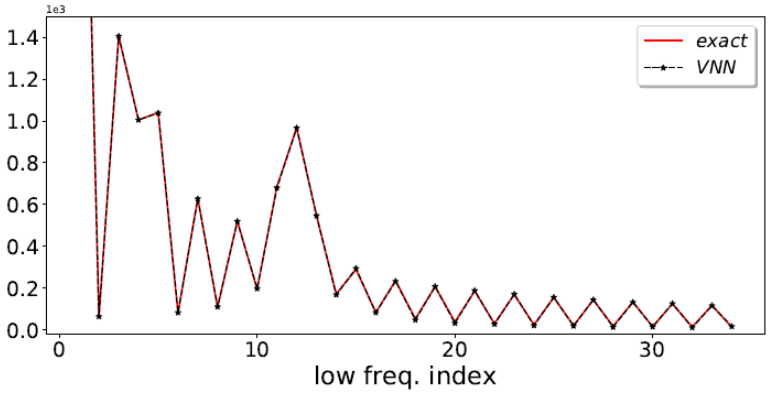
ational Neural Networks (VNNs): Function Approximation

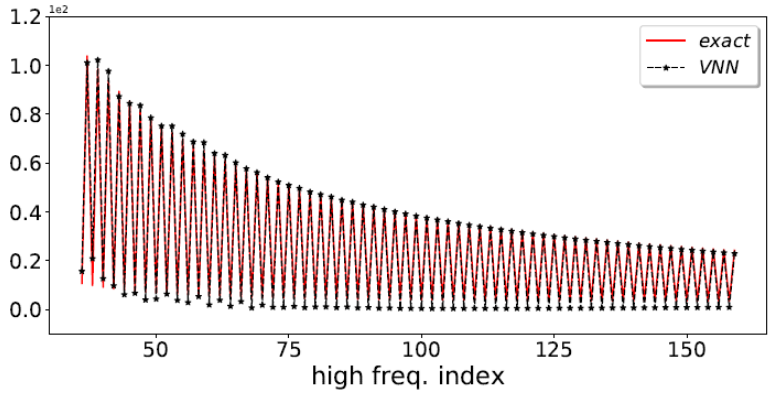
ulti-elemental test functions:





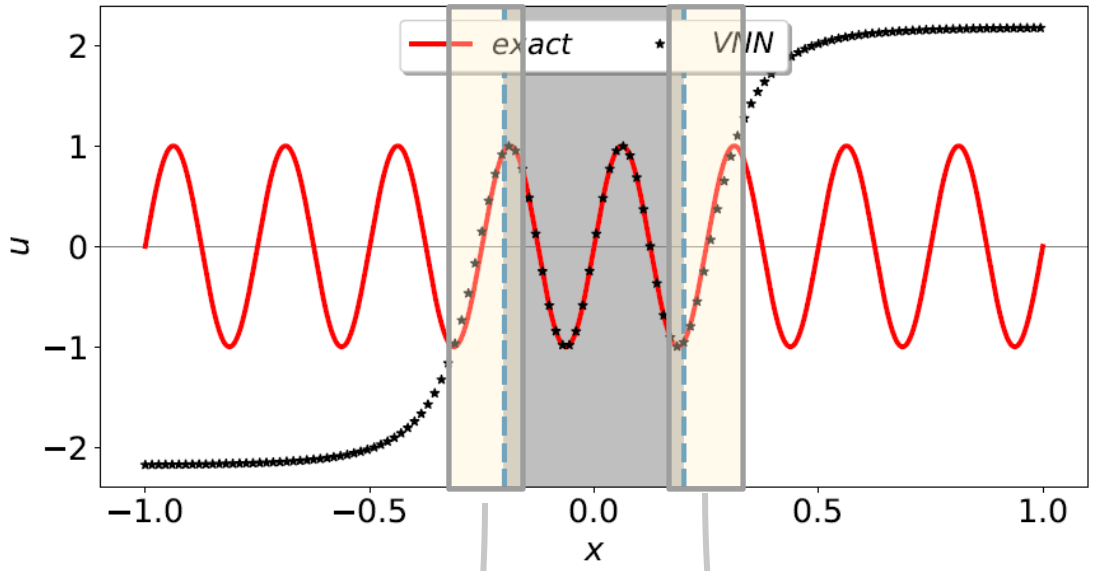


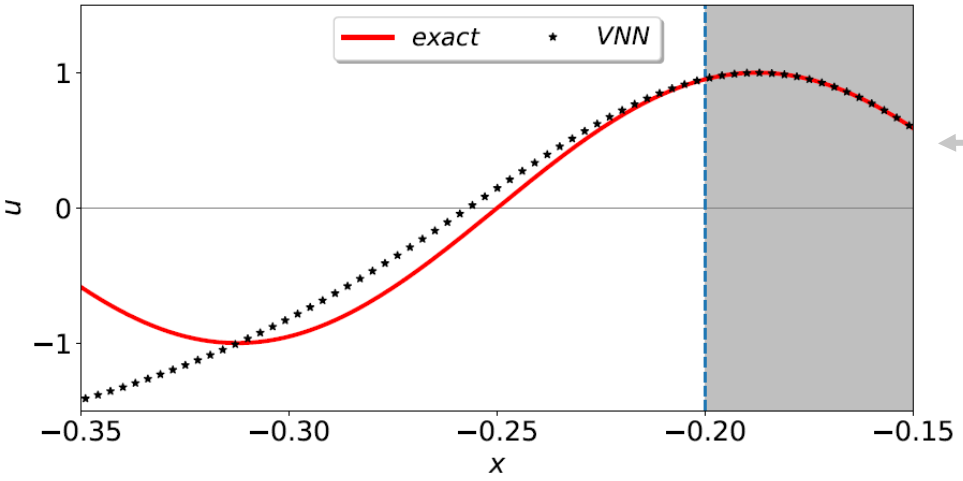


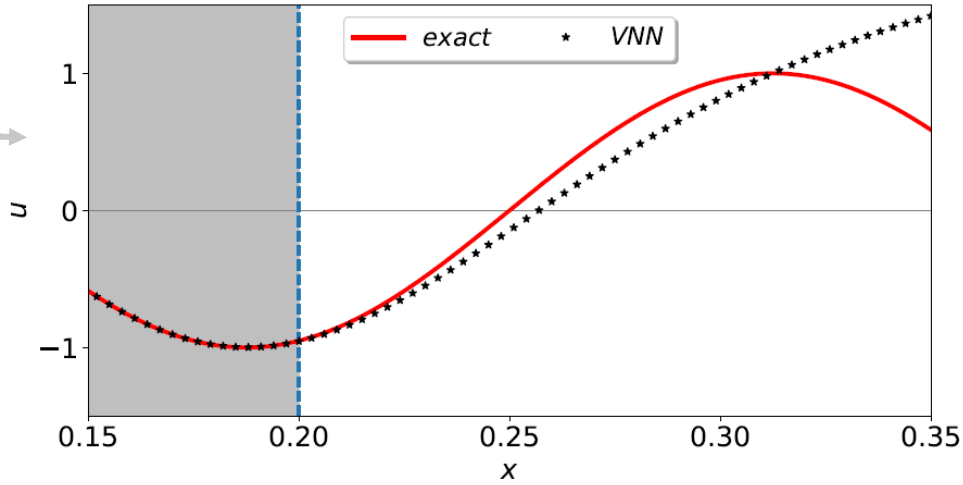


ational Neural Networks (VNNs): Function Approximation

ocalized learning & learning out-of-the-box





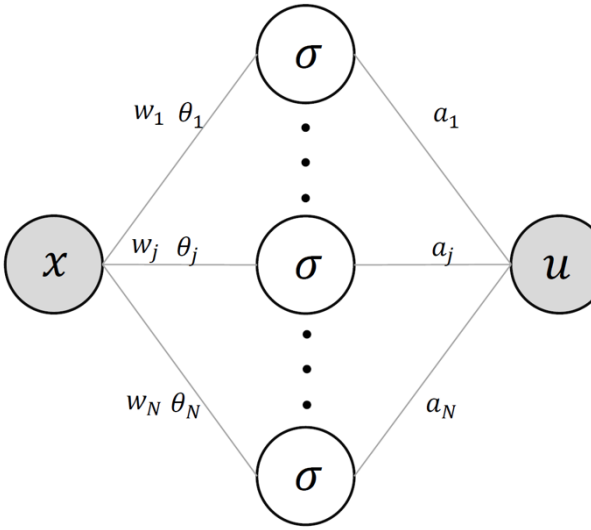


- The test functions have **non-zer** values only in the gray box (-0.2<x<0.2).
- The test functions have **zero** val the white region.
- It is expected that VNN only lear function in the grey box.
- But VNN learns the function sligh side of the box, a notion of generalization to unseen domain is indicated as the yellow region

allow VPINNs: Analytical Approach

Variational formulation requires computing numerical integration.

Q: Can we avoid error due to numerical integration?
Starting with a shalow network to compute the integrals analytically.



$$u_{NN}(x) = \sum_{j=1}^{N} u_{NN_j}(x) = \sum_{j=1}^{N} a_j \sigma(w_j x + \theta_j)$$

Steady-state Burger's equation:

$$\begin{cases} u\dfrac{du}{dx} - \dfrac{d^2u}{dx^2} = f(x) \\ u(-1) = g, \quad u(1) = h \end{cases}$$

$$Residual_k^{\mathfrak{v}} = \mathcal{R}_k + \mathcal{R}_k^{NL} - F_k - r^b$$

$$\mathcal{R}_k = -\left(\frac{d^2 u_{NN}(x)}{dx^2}, v_k(x)\right)_\Omega \qquad \mathcal{R}_k^{NL} = \left(u_{NN}\frac{du_{NN}(x)}{dx}, v_k(x)\right)_\Omega \qquad F_k = (f(x), v_k(x))_\Omega$$

allow VPINNs: Analytical Approach

- Consider **Sine** activation functions with **Sine** test function $v_k(x) = sin(k\pi x)$, $k = 1, 2, ..., K$
- **Analytical** expression of **variational** residuals

$$\mathcal{R}_k^{(1)} = \mathcal{R}_k^{(2)} = 2(-1)^k k\pi \sum_{j=1}^{N} \frac{a_j w_j^2 \cos(\theta_j)\sin(w_j)}{w_j^2 - k^2\pi^2},$$

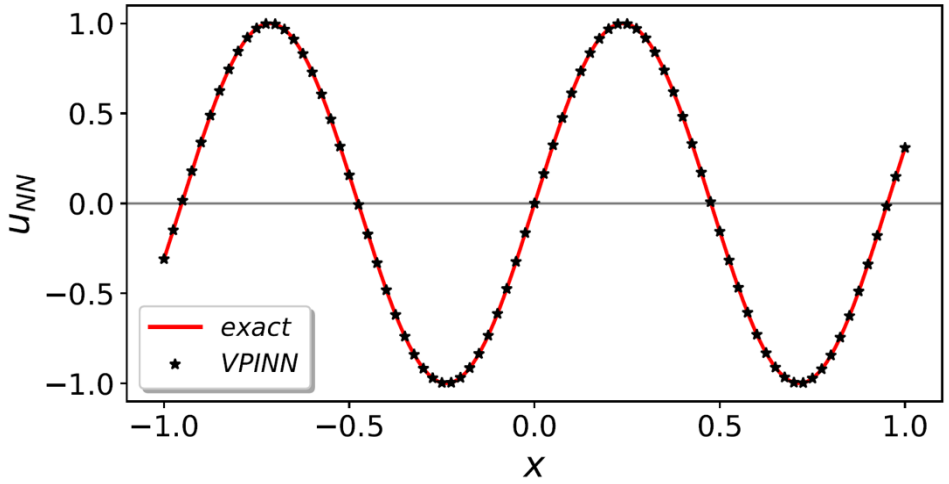
$$\mathcal{R}_k^{(3)} = 2(-1)^k k\pi \sum_{j=1}^{N} \frac{a_j k^2\pi^2 \cos(\theta_j)\sin(w_j)}{w_j^2 - k^2\pi^2} + (-1)^k k\pi(h-g),$$

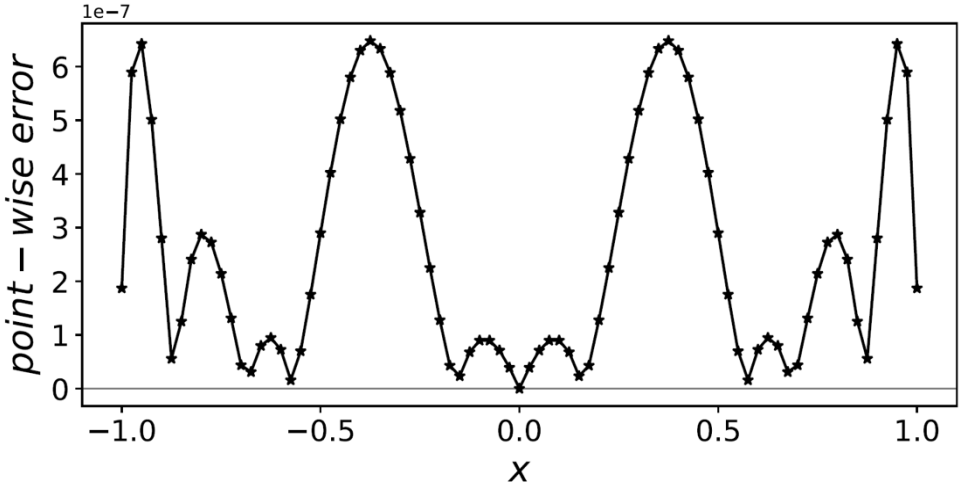
$$\mathcal{R}_k^{NL} = (-1)^k k\pi \sum_{i=1}^{N}\sum_{j=1}^{N} a_i a_j w_i \left[\frac{\sin(w_j + w_i)\cos(\theta_j + \theta_i)}{(w_j + w_i)^2 - k^2\pi^2} + \frac{\sin(w_j - w_i)\cos(\theta_j - \theta_i)}{(w_j - w_i)^2 - k^2\pi^2}\right]$$

- We can also use **sine** activation functions with **Legendre polynomial** as test functions $v_k(x) = P_{k+1}(x) - P_{k-1}(x)$, $k = 1, 2, ..., K$.
- We then use the **recursion** formula to obtain the **variational** residuals.

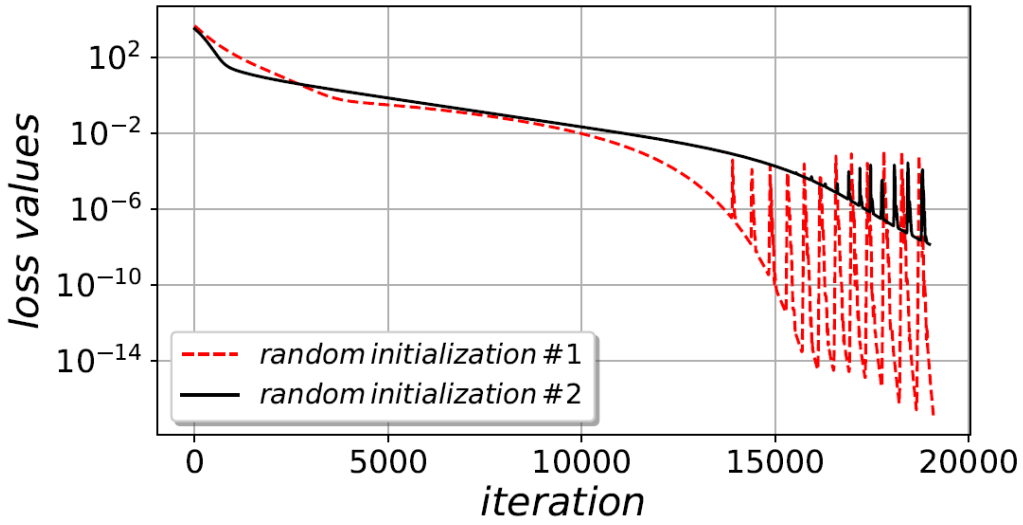
allow VPINNs: Analytical Approach

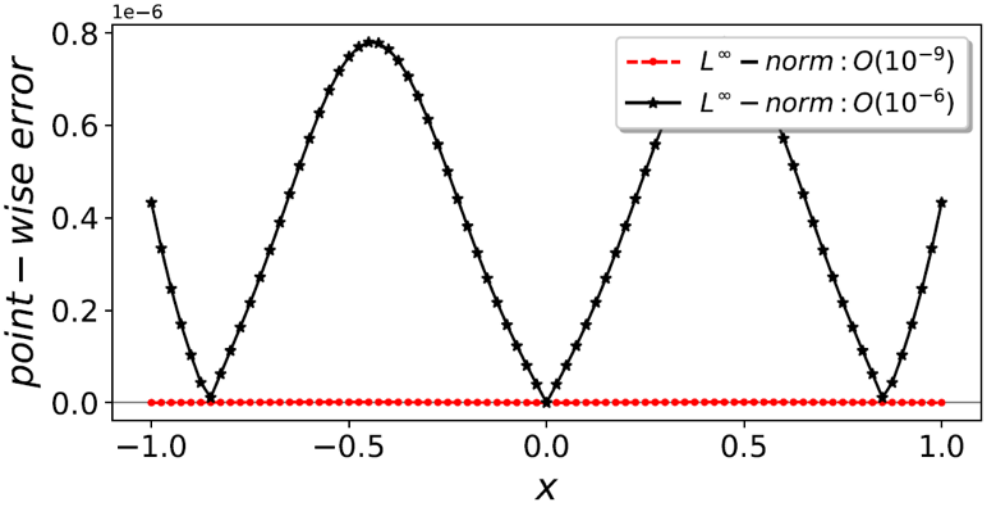
ample: Steady-state Burger's equation with **smooth solution**. $u^{exact} = A\sin(\omega x)$





work initializations matters:





VPINNs: Poisson's Equation (1D)

- Governing Equation:

$$-\frac{d^2u(x)}{dx^2} = f(x)$$

- Boundary conditions:

$$u(-1) = g$$
$$u(1) = h$$

- Variational residual:

$$\mathcal{R}_k = \sum_{e=1}^{N_{el}} \mathcal{R}_k^{(e)} = \sum_{e=1}^{N_{el}} \int_{x_{e-1}}^{x_e} \left(-\frac{d^2u_{NN}(x)}{dx^2} - f(x) \right) v_k^{(e)}(x) dx,$$

VPINNs: Poisson's Equation (1D)

- Reducing the regularity and simplifying the network by integration-by-parts:

$$
{}^{(1)}\mathcal{R}_k^{(e)} = -\int_{x_{e-1}}^{x_e} \frac{d^2 u_{NN}(x)}{dx^2} v_k^{(e)}(x) dx - F_k^{(e)},
$$

$$
{}^{(2)}\mathcal{R}_k^{(e)} = \int_{x_{e-1}}^{x_e} \frac{du_{NN}(x)}{dx} \frac{dv_k^{(e)}(x)}{dx} dx - \left. \frac{du_{NN}(x)}{dx} v_k^{(e)}(x) \right|_{x_{e-1}}^{x_e} - F_k^{(e)},
$$

$$
{}^{(3)}\mathcal{R}_k^{(e)} = -\int_{x_{e-1}}^{x_e} u_{NN}(x) \frac{d^2 v_k^{(e)}(x)}{dx^2} dx - \left. \frac{du_{NN}(x)}{dx} v_k^{(e)}(x) \right|_{x_{e-1}}^{x_e} + \left. u_{NN}(x) \frac{dv_k^{(e)}(x)}{dx} \right|_{x_{e-1}}^{x_e} - F_k^{(e)}
$$

- Variational loss:

$$
L^{\mathfrak{v}(i)} = \sum_{e=1}^{N_{el}} \frac{1}{K^{(e)}} \sum_{k=1}^{K^{(e)}} \left| {}^{(i)}\mathcal{R}_k^{(e)} \right|^2 + \frac{\tau_b}{2} \left(|u_{NN}(-1) - g|^2 + |u_{NN}(1) - h|^2 \right), \quad i = 1, 2, 3
$$