Deep Learning for Science and Engineering Teaching Kit

Deep Learning for Scientists and Engineers

Lecture 10: PINN Extensions

Instructors: George Em Karniadakis, Khemraj Shukla, Lu Lu
Teaching Assistants: Vivek Oommen and Aniruddha Bora

Course Roadmap

Module-1 (Basics)

- Lecture 1: Introduction
- Lecture 2: A primer on Python, NumPy, SciPy and *jupyter* notebooks
- Lecture 3: Deep Learning Networks
- Lecture 4: A primer on TensorFlow and PyTorch
- Lecture 5: Training and Optimization
- Lecture 6: Neural Network Architectures

Module-2 (PDEs and Operators)

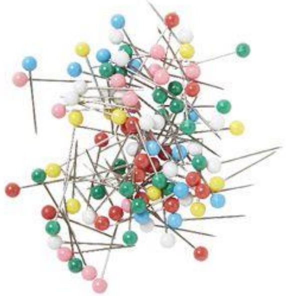
- Lecture 7: Machine Learning using Multi-Fidelity
- Lecture 8: Physics-Informed Neural Networks (PINNs)
- Lecture 9: PINN Extensions
- Lecture 10: Neural Operators

Module-3 (Codes & Scalability)

- Lecture 11: Multi-GPU Scientific Machine Learning

Contents

An Alphabet of PINNs



- vPINNs/hp-vPINNs: variational PINNs (Lecture 9)

- gPINNs: gradient-enhanced PINNs
- cPINNs: conservative PINNs
- xPINNs: eXtended PINNs
- fPINNs: fractional PINNs
- sPINNs: stochastic PINNs

- BPINNs: Bayesian PINNs (Lecture 12)
- …

gPINNS: gradient-enhanced PINNs

$\boxed{\text{Idea: Provide additional information to the network through the gradient}}$

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_d}\right) = \mathbf{0}, \quad \mathbf{x} \in \Omega.$$

New terms in Loss Function:

$$\mathcal{L} = w_f \mathcal{L}_f + w_b \mathcal{L}_b + w_i \mathcal{L}_i + \sum_{i=1}^{d} w_{g_i} \mathcal{L}_{g_i}\left(\boldsymbol{\theta}; \mathcal{T}_{g_i}\right)$$

$$\mathcal{L}_{g_i}\left(\boldsymbol{\theta}; \mathcal{T}_{g_i}\right) = \frac{1}{\left|\mathcal{T}_{g_i}\right|} \sum_{\mathbf{x} \in \mathcal{T}_{g_i}} \left|\frac{\partial f}{\partial x_i}\right|^2$$

- Yu J, Lu L, Meng X, Karniadakis GE. Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems. Computer Methods in Applied Mechanics and Engineering. 2022 Apr 1;393:114823.

gPINNS

Example: Additional loss terms $\Delta u = f$

$$\text{In 1D,} \quad \mathcal{L}_g = w_g \frac{1}{|\mathcal{T}_g|} \sum\nolimits_{\mathbf{x} \in \mathcal{T}_g} \left| \frac{d^3 u}{dx^3} - \frac{df}{dx} \right|^2$$

$$\text{In 2D,} \quad \mathcal{L}_{g_1} = w_{g_1} \frac{1}{|\mathcal{T}_{g_1}|} \sum\nolimits_{\mathbf{x} \in \mathcal{T}_{g_1}} \left| \frac{\partial^3 u}{\partial x^3} + \frac{\partial^3 u}{\partial x \partial y^2} - \frac{\partial f}{\partial x} \right|^2,$$

$$\mathcal{L}_{g_2} = w_{g_2} \frac{1}{|\mathcal{T}_{g_2}|} \sum\nolimits_{\mathbf{x} \in \mathcal{T}_{g_2}} \left| \frac{\partial^3 u}{\partial x^2 \partial y} + \frac{\partial^3 u}{\partial y^3} - \frac{\partial f}{\partial y} \right|^2.$$

CRUNCH TIME

CRUNCH Group

Brown University

DEEP LEARNING INSTITUTE

BROWN

Diffusion Equation

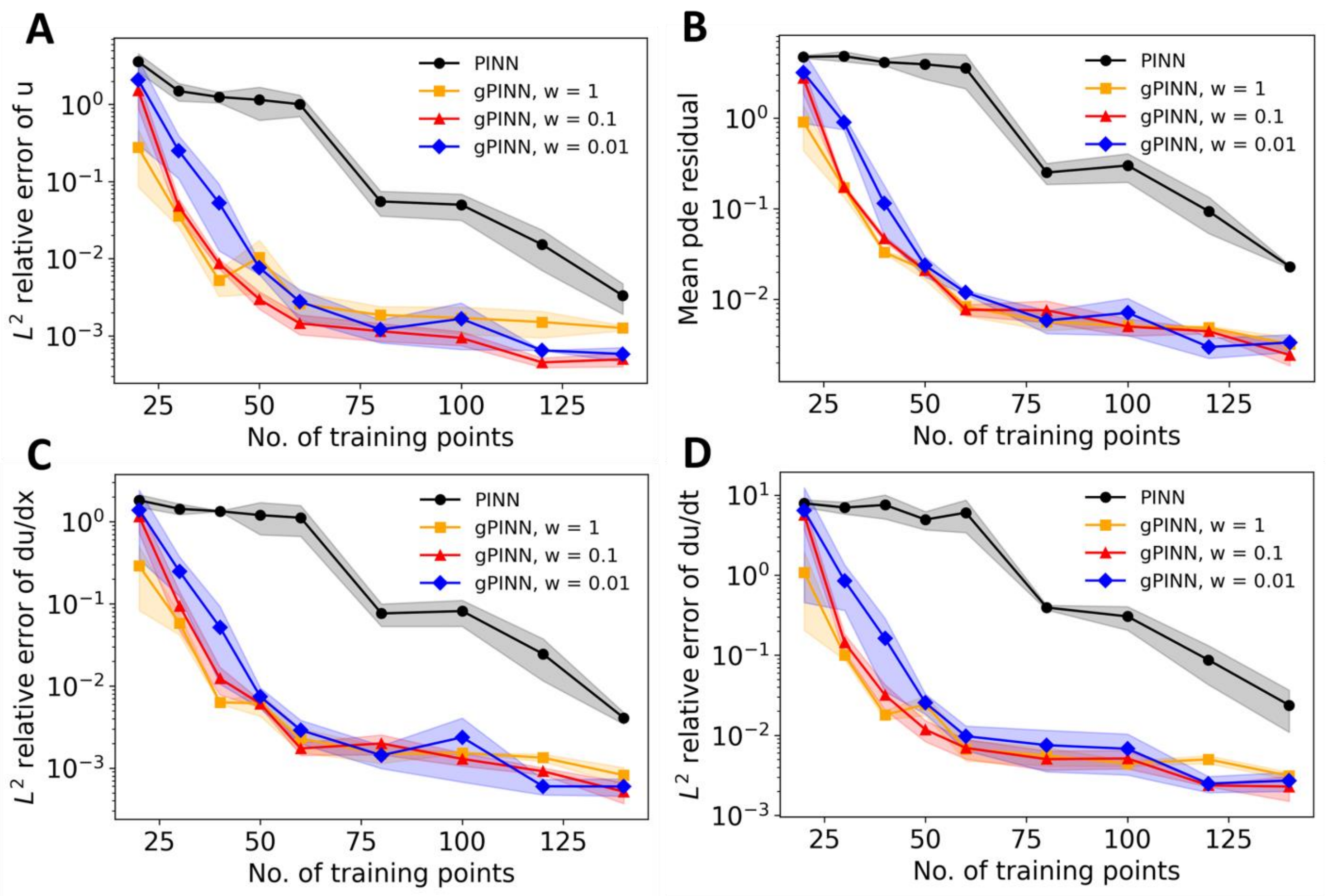
$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + R(x,t), \quad x \in [-\pi, \pi],\ t \in [0,1],$$

$$R(x,t) = e^{-t}\left[\frac{3}{2}\sin(2x) + \frac{8}{3}\sin(3x) + \frac{15}{4}\sin(4x) + \frac{63}{8}\sin(8x)\right].$$

$$u(x,0) = \sum_{i=1}^{4}\frac{\sin(ix)}{i} + \frac{\sin(8x)}{8},$$

$$u(-\pi,t) = u(\pi,t) = 0,$$

analytic solution for $u(x,t) = e^{-t}\left[\sum_{i=1}^{4}\frac{\sin(ix)}{i} + \frac{\sin(8x)}{8}\right].$



Inverse Problem

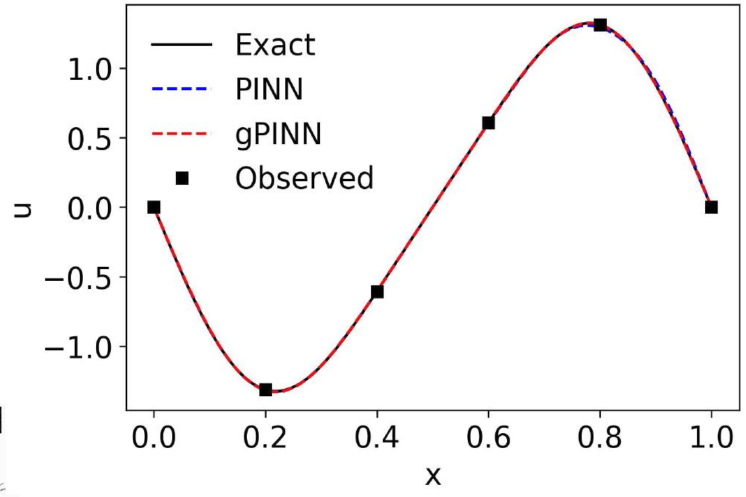
$$f\left(\mathbf{x}; \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 u}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right) = 0, \quad \mathbf{x} \in \Omega$$

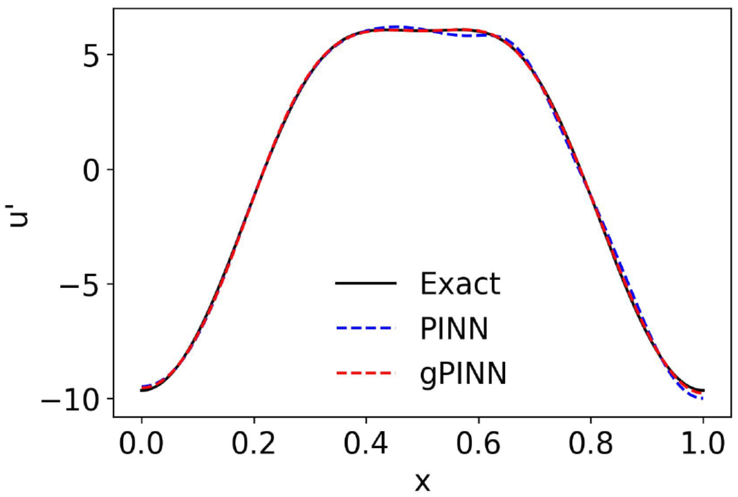
$$\mathcal{B}(u, \mathbf{x}) = 0 \quad \text{on} \quad \partial\Omega$$

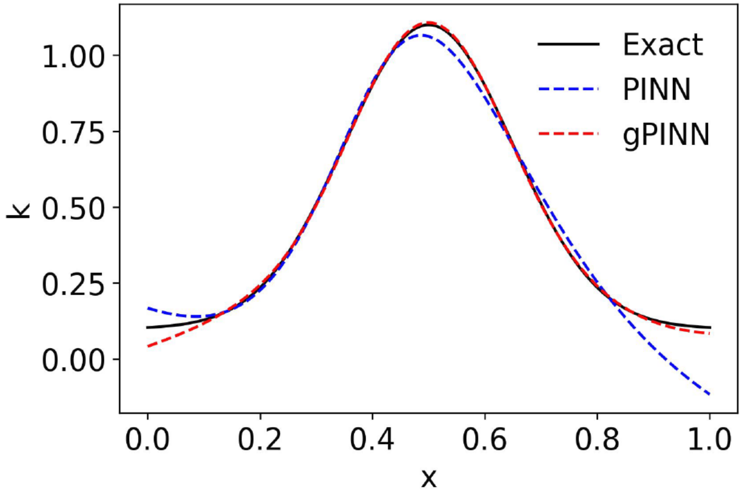
$$\mathcal{L}_i\left(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_i\right) = \frac{1}{|\mathcal{T}_i|} \sum_{\mathbf{x} \in \mathcal{T}_i} |\hat{u}(\mathbf{x}) - u(\mathbf{x})|^2$$

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

Inferring space-dependent reaction rate $\lambda \frac{\partial^2 u}{\partial x^2} - k(x)u = f, \quad x \in [0,1], \quad u(x) = 0$ is imposed at $x = 0$ and 1 $k(x) = 0.1 + \exp\left[-0.5 \frac{(x-0.5)^2}{0.15^2}\right]$ Inferring whole function instead of just a constant







gPINNS with Residual Adaptive Refinement (RAR)

Algorithm 1: gPINN with RAR.

Step 1 Train the neural network using gPINN on the training set $\mathcal{T}$ for a certain number of iterations.

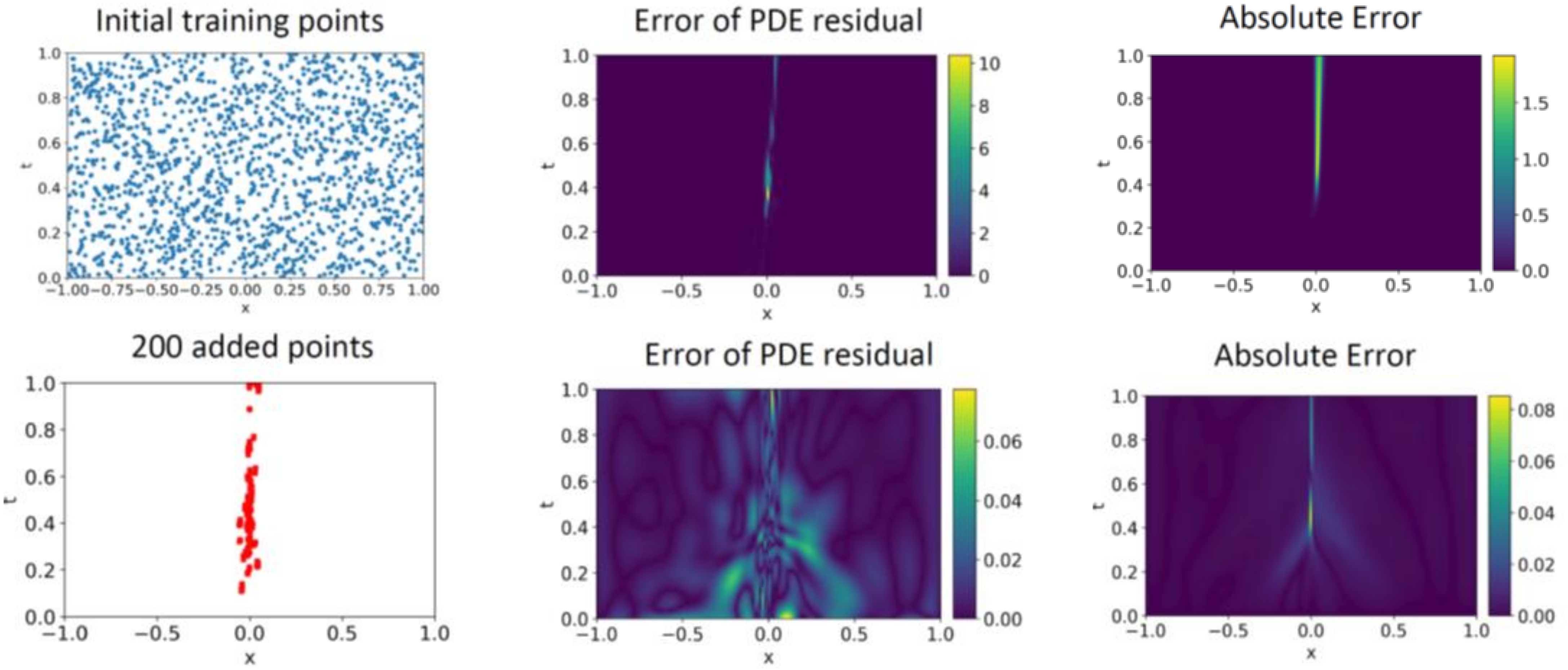
Step 2 Compute the PDE residual $\left| f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right)\right|$ at random points in the domain.

Step 3 Add m new points to the training set $\mathcal{T}$ where the residual is the largest.

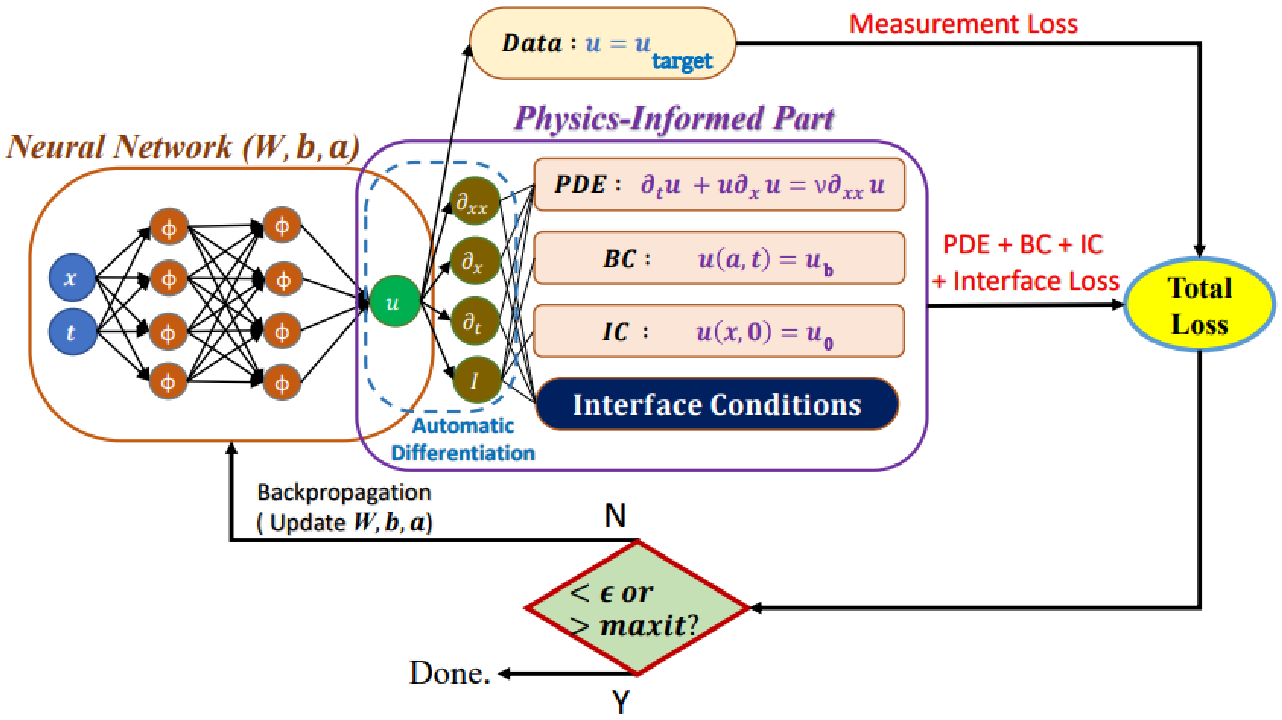
Step 4 Repeat Steps 1, 2, and 3 for n times, or until the mean residual falls below a threshold $\mathcal{E}$.

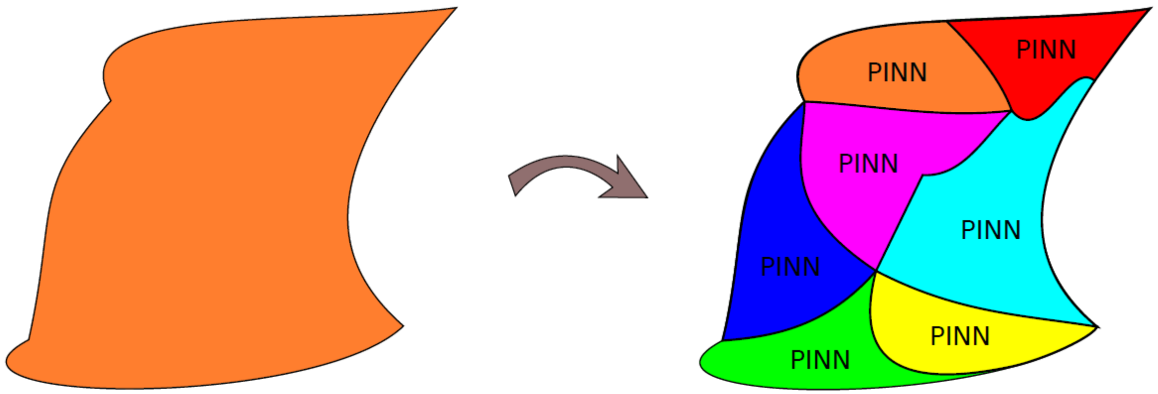
Burgers Equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}, \qquad x \in [-1,1], t \in [0,1]$$
$$u(x,0) = -\sin(\pi x), \quad u(-1,t) = u(1,t) = 0$$



PINNs with Domain Decomposition





$$\mathcal{L}(\tilde{\Theta}) = \frac{1}{N_u}\sum_{i=1}^{N_u}\left|u^i_{\text{target}} - u_{\tilde{\Theta}}\left(x^u_i\right)\right|^2 + \frac{1}{N_f}\sum_{i=1}^{N_f}\left|\mathcal{F}_{\tilde{\Theta}}\left(x^f_i\right)\right|^2 + \text{Interface Loss}$$

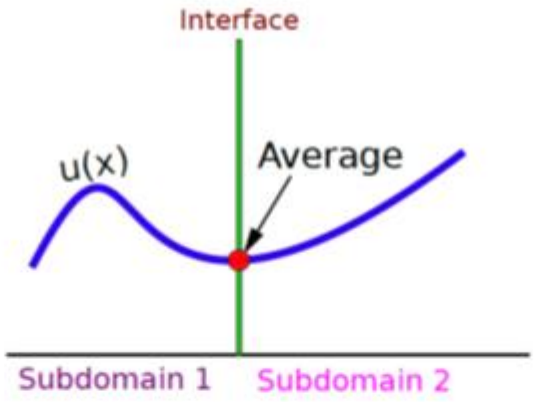
Conservative PINNs (cPINNs): Application to Conservation Laws

cPINNs: Interface conditions in the " q^{th} " subdomain.

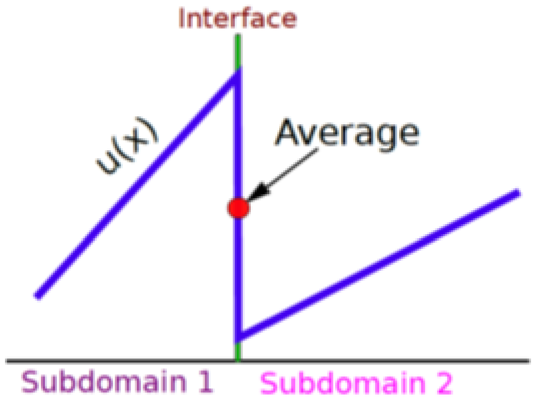
$$\mathrm{MSE}_{\mathrm{flux}} = \sum_{\forall q^+} \left(\frac{1}{N_{l_q}} \sum_{i=1}^{N_{l_q}} \left| f_q\left(u\left(\mathbf{x}_{l_q}^{(i)}\right)\right) \cdot \mathbf{n} - f_{q^+}\left(u\left(\mathbf{x}_{l_q}^{(i)}\right) \cdot \mathbf{n}\right) \right|^2 \right)$$

where f is the flux and $q = 1, 2, \cdots, N_{sd}$.

$$\mathrm{MSE}_{u_{\mathrm{avg}}} = \sum_{\forall q^+} \left(\frac{1}{N_{l_q}} \sum_{i=1}^{N_{l_q}} \left| u_{\tilde{\Theta}_q}\left(\mathbf{x}_{l_q}^{(i)}\right) - \left\{\left\{ u_{\tilde{\Theta}_q}\left(\mathbf{x}_{l_q}^{(i)}\right) \right\}\right\} \right|^2 \right)$$



(a)



(b)

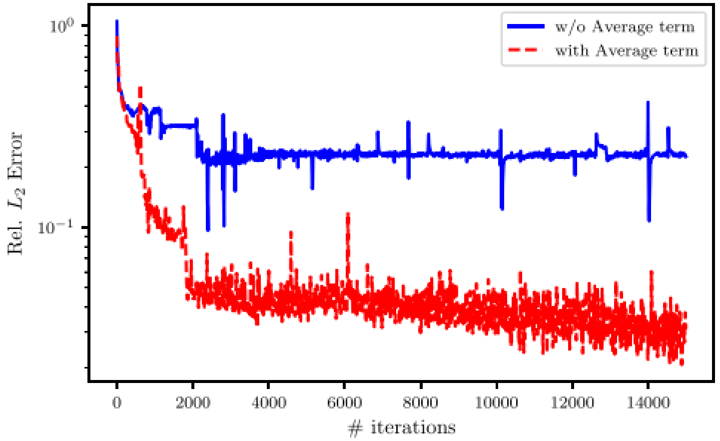
- Jagtap AD, Kharazmi E, Karniadakis GE. Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. Computer Methods in Applied Mechanics and Engineering. 2020 Jun 15;365:113028

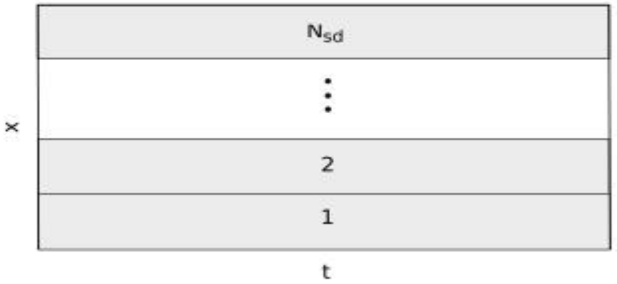
cPINN Results: Burgers Equation

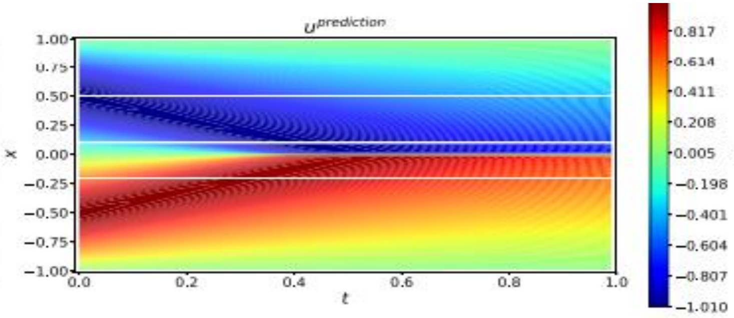
$u_t + uu_x = \nu u_x x, x \in \mathbf{R}, t > 0$ with IC : $u(x,0) = -\sin(\pi x)$ and **BC** :$u(t,1) = u(t,-1) = 0.$

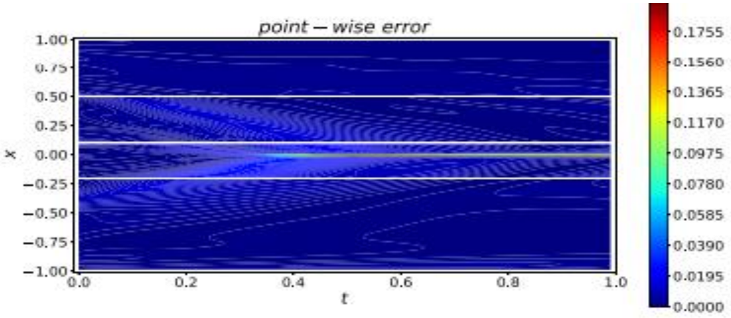
Subdomain number	1	2	3	4
# Layers	2	6	3	2
# Neurons	20	30	25	20
# Residual points	6000	8000	4000	4000
Adaptive activation function	cos	sin	tanh	sin

Learning rate : 8e-4, Optimizer : Adam









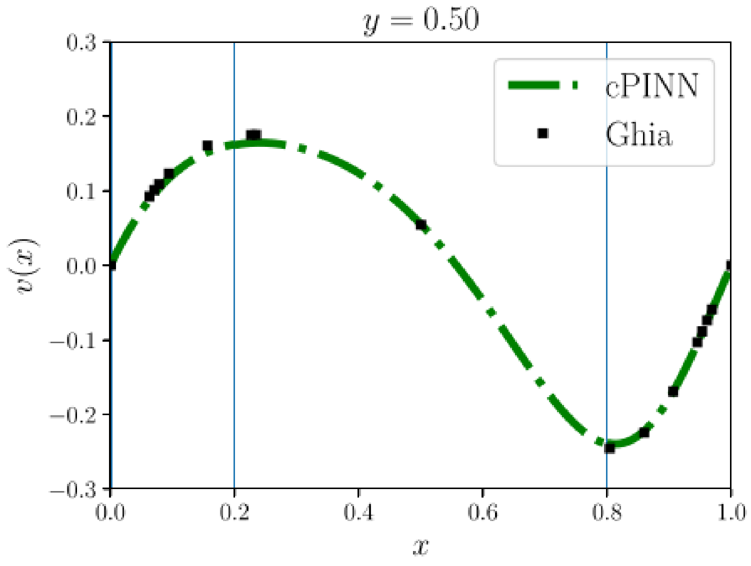
cPINN Results: 2D Incompressible Navier Stokes Equation

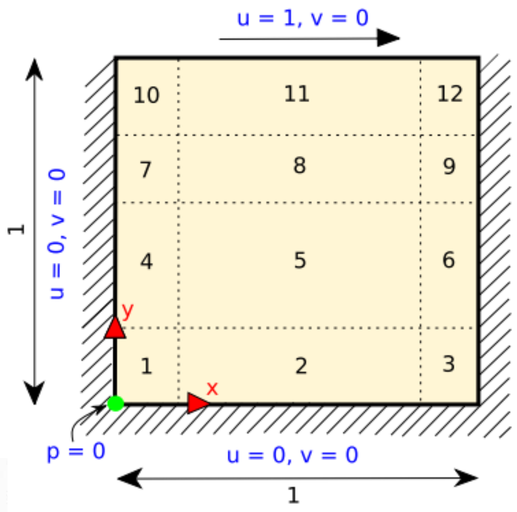
$$\mathbf{u}_t + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla p + \frac{1}{Re}\nabla^2\mathbf{u}, \quad \text{in } \Omega$$

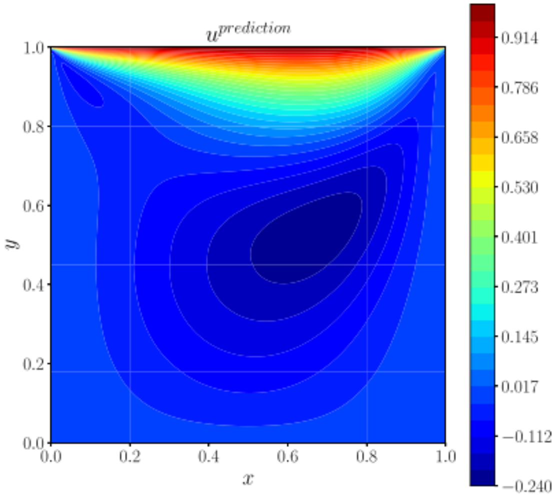
$$\nabla\cdot\mathbf{u} = 0, \quad \text{in } \Omega$$

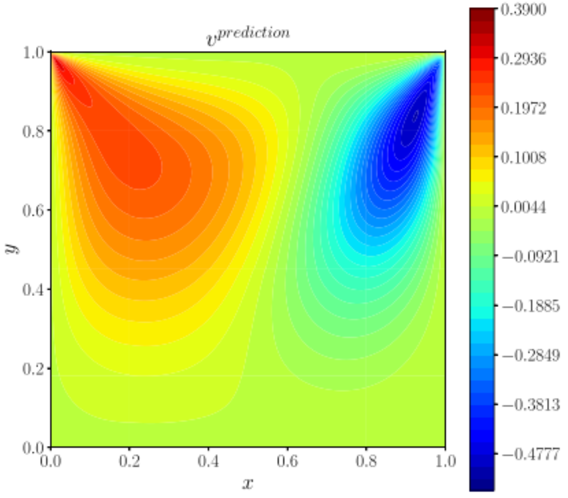
SD No.	1	2	3	4	5	6	7	8	9	10	11	12
# L	6	6	6	6	4	6	6	4	6	6	6	6
# N	20	20	20	20	20	20	20	20	20	20	20	20
#R.pts	2.5k	5k	2.5k	2.5k	5k	2.5k	2.5k	5k	2.5k	2.5k	5k	2.5k

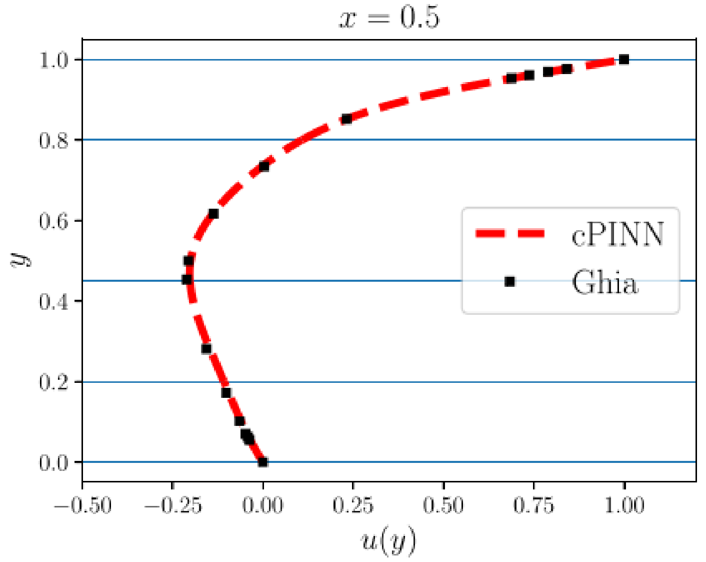
Learning rate : 6e-4, Optimizer : Adam, Adaptive Activation Fun. : tanh









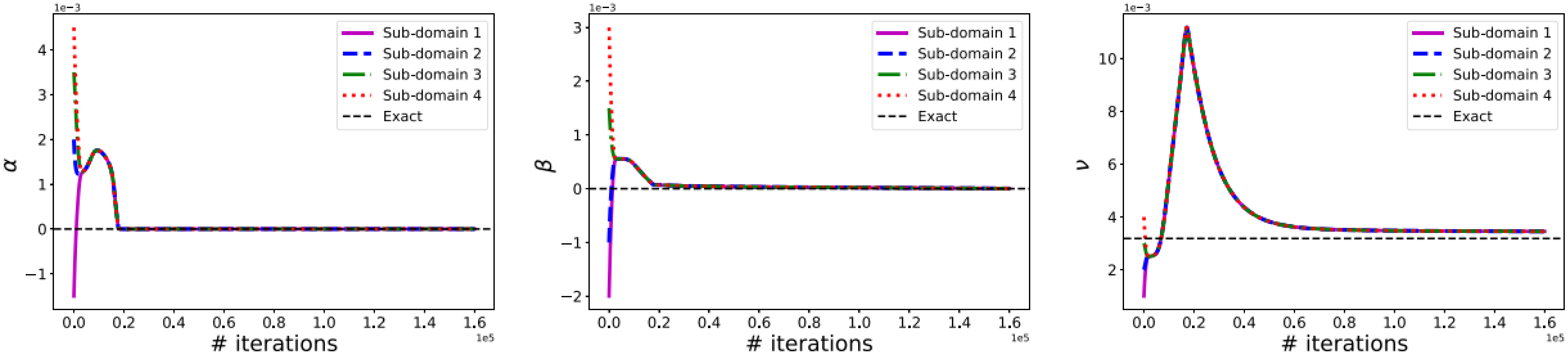


cPINN Results: Inverse 2D Burgers Equation

We have the following equation with some unknown parameters α, β, ν.

$$u_t + uu_x - \beta u_x - \nu u_{xx} + \alpha u_{xxx} = 0, \quad x \in \Omega \subset \mathbb{R}, t > 0,$$

$\boxed{\text{Aim : Identify all the terms } \alpha, \beta, \nu \text{ with data given by the viscous Burgers equation.}}$



Extended PINNs (XPINNs)

XPINNs : Interface conditions in the " q^{th} " subdomain.

$$\text{MSE}_{u_{\text{avg}}} = \sum_{\forall q^{+}} \left(\frac{1}{N_{l_q}} \sum_{i=1}^{N_{l_q}} \left| u_{\tilde{\Theta}_q}\left(\mathbf{x}_{l_q}^{(i)}\right) - \left\{\left\{ u_{\tilde{\Theta}_q}\left(\mathbf{x}_{l_q}^{(i)}\right) \right\}\right\} \right|^2 \right)$$

$\{\} \rightarrow$ average value

$$\text{MSE}_{\mathcal{R}} = \sum_{\forall q^{+}} \left(\frac{1}{N_{l_q}} \sum_{i=1}^{N_{l_q}} \left| \mathcal{F}_{\tilde{\Theta}_q}\left(\mathbf{x}_{l_q}^{(i)}\right) - \mathcal{F}_{\tilde{\Theta}_{q^{+}}}\left(\mathbf{x}_{l_q}^{(i)}\right) \right|^2 \right)$$

$+$

Additional continuity conditions

Additional advantages over cPINN

(1) Extension to any differential equation(s)

(2) Generalized space-time domain decomposition

(3) Simple interface conditions

Comparison of PINN, cPINN and XPINN frameworks

	PINN	cPINN	XPINN
Spatial Domain Decomposition	×	✓	✓
Parallelization capacity	×	✓	✓
Localized representation capacity	×	✓	✓
Efficient hyperparameter adjustment	×	✓	✓
Applicability	Any DEs	Conservation laws	Any DEs
Interface conditions	–	Complex	Simple
Spatio-Temporal Domain Decomposition	×	×	✓

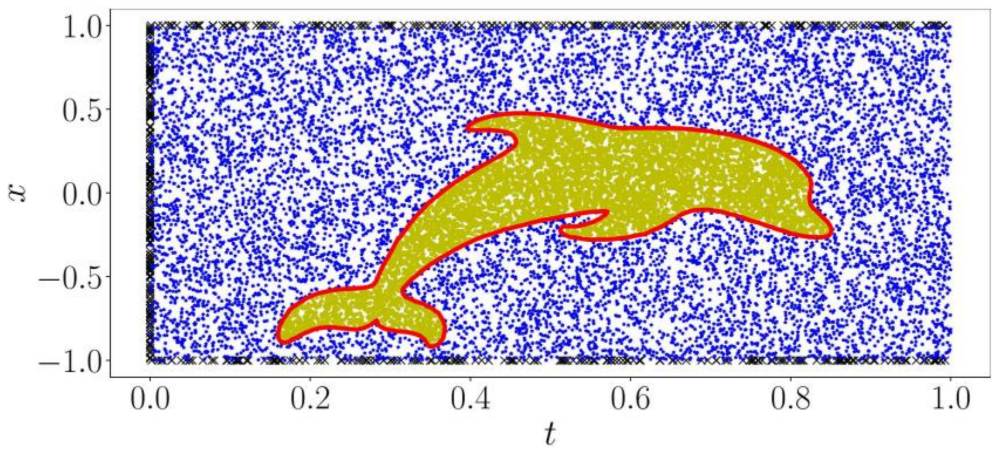
- Jagtap AD, Karniadakis GE. Extended physics-informed neural networks (xPINNs): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. Communications in Computational Physics. 2020 Nov 1;28(5):2002-41.

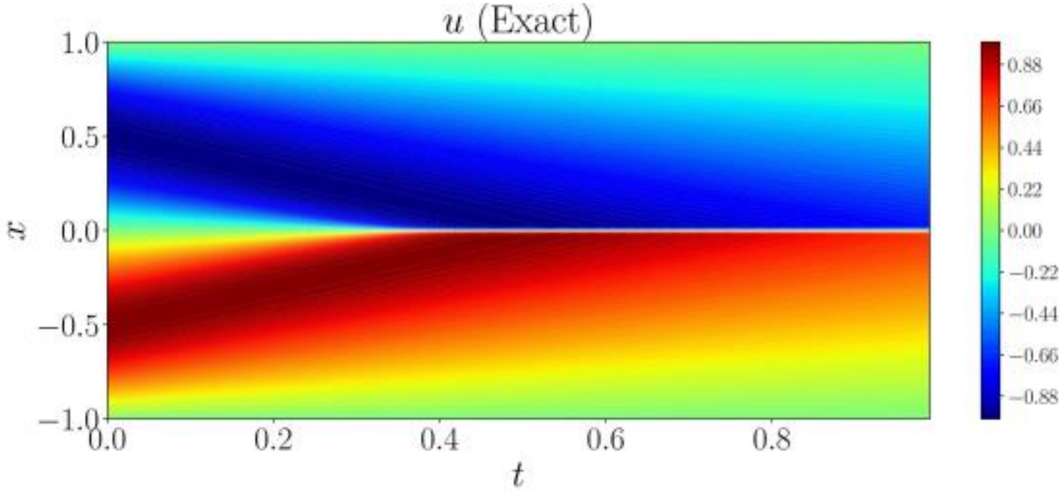
XPINN Results: Burgers Equation

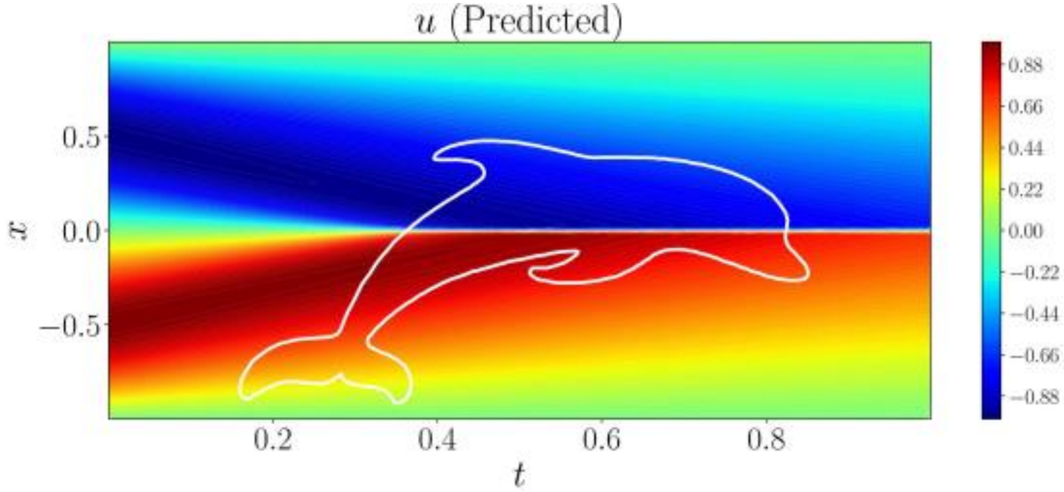
Subdomain number	1	2
# Layers	6	7
# Neurons	20	25
# Residual points	7000	3000
Adaptive Activation function	tanh	sin

Learning rate : 8e − 4, Optimizer : Adam

Relative L_2 error in the whole domain : 8.93265e − 3



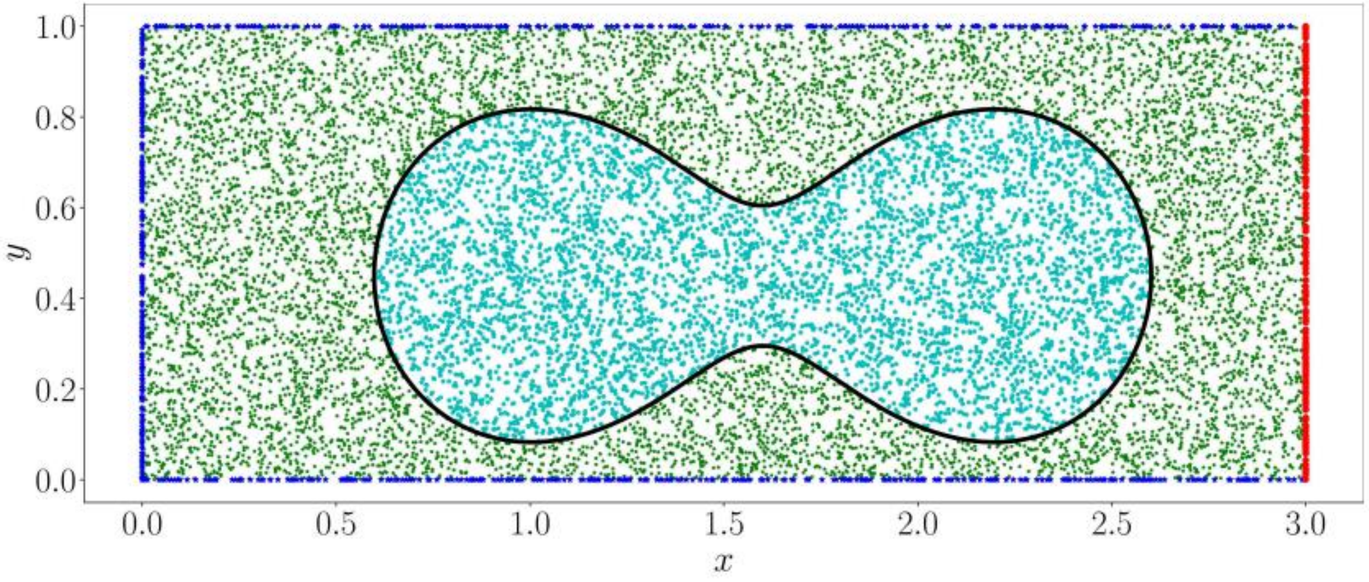




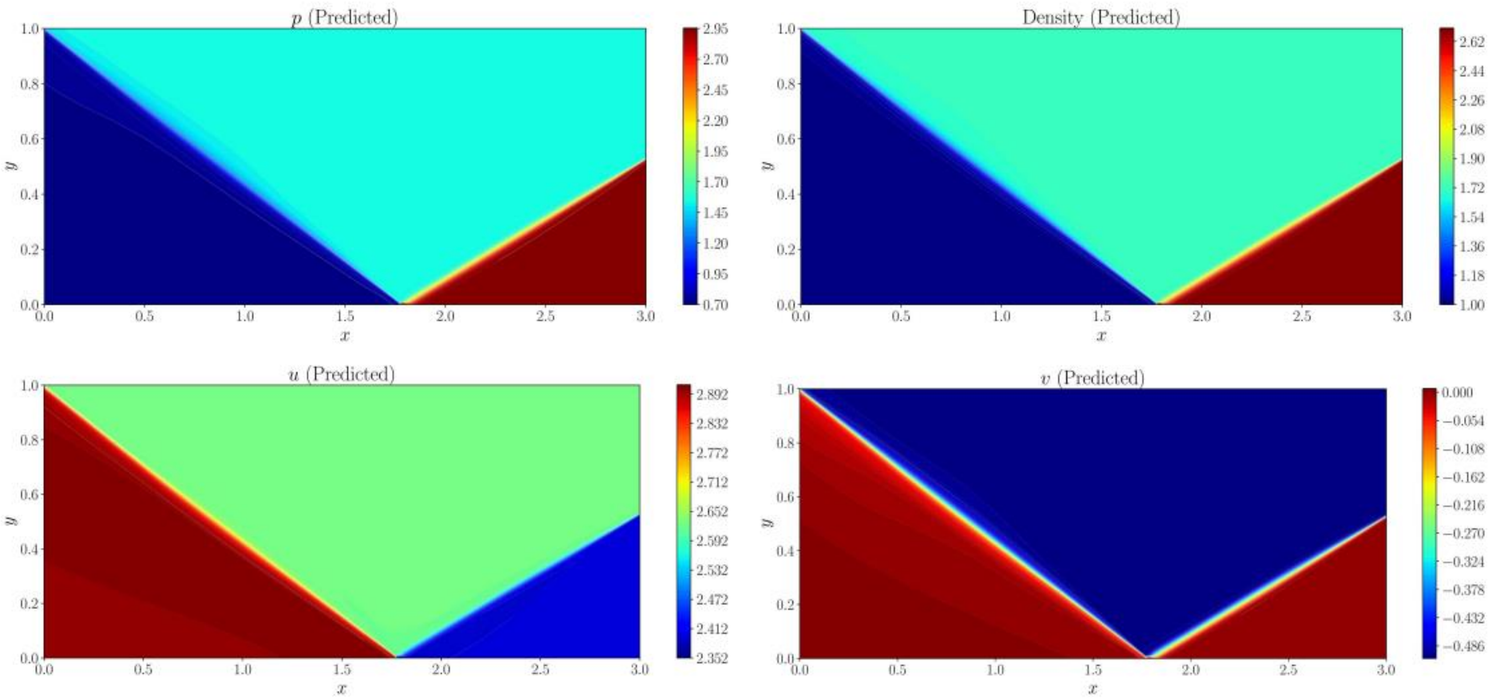
XPINN Results: 2D Compressible Euler Equations

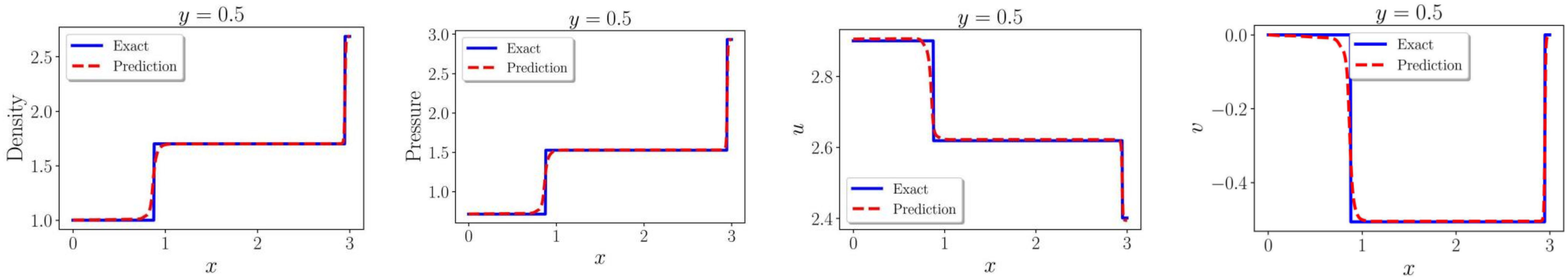
$U_t + F_x + G_y = 0, \quad (\mathbf{x}, t) \in \Omega \times (0, T] \subset \mathbb{R}^2 \times \mathbb{R}_+$, with appropriate initial and boundary conditions.

Subdomain number	1	2
# Layers	6	6
# Neurons	20	25
# Residual points	12000	8000
Adaptive activation function	sin	tanh



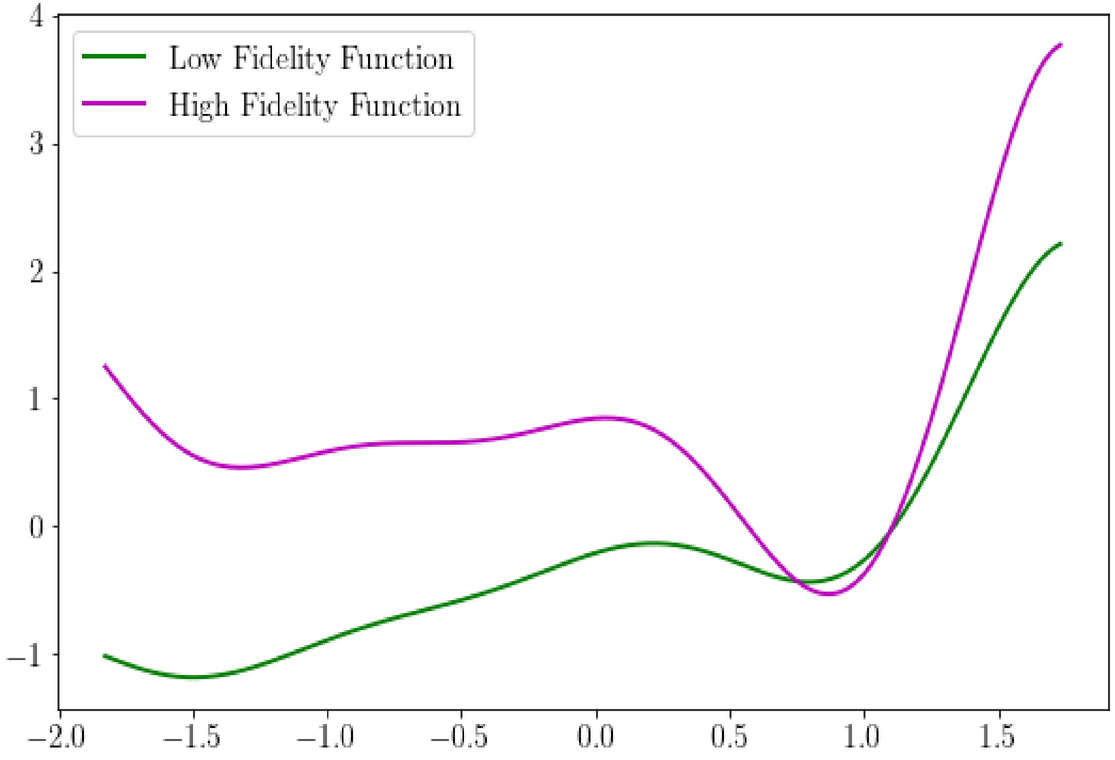
XPINN Results: 2D Compressible Euler Equations



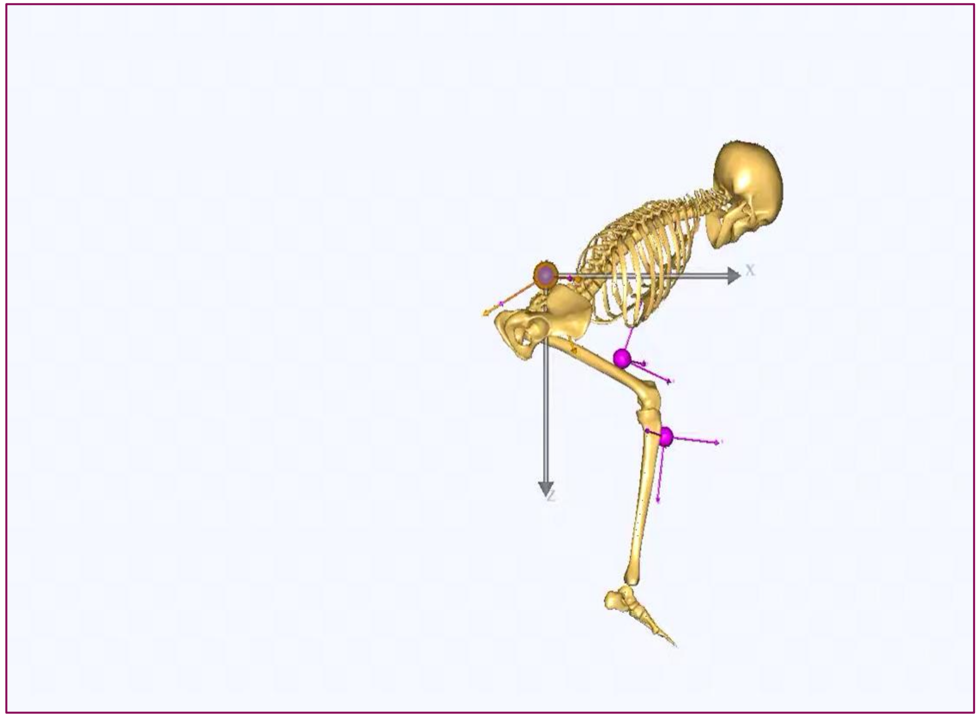


XPINN Results: 2D Compressible Euler Equations

Forrester Function



Breaststroke kinematics simulation



Composite Neural Network for Multi-fidelity Data

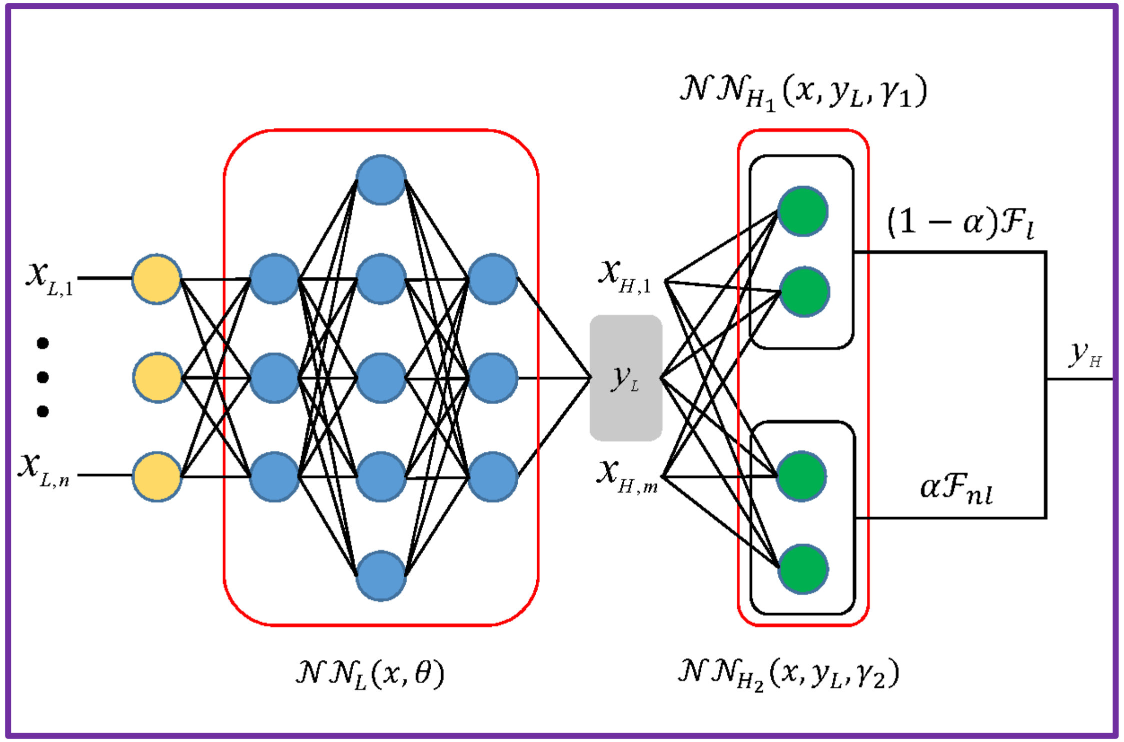
General formula for low- and high-fidelity data:

- General formula for low- and high-fidelity data:

$$y_h = F(x, y_L),$$

where y_h : High-fidelity function and y_L : Low fidelity function

- $y_L \approx NN_L(x_L, \theta), \qquad y_H = F(x, y_L)$
- $MSE = MSE_{y_L} + MSE_{y_H} + \lambda \sum \boldsymbol{\omega}^2$
- $MSE_{y_L} = \frac{1}{N_{y_L}} \sum (y_L - y_L^*)^2$ and $MSE_{y_H} = \frac{1}{N_{y_H}} \sum (y_H - y_H^*)^2$
- **Goal**: Learn unknown function F from multi-fidelity data



- Meng X, Karniadakis GE. A composite neural network that learns from multi-fidelity data: Application to function approximation and inverse PDE problems. Journal of Computational Physics. 2020 Jan 15;401:109020.

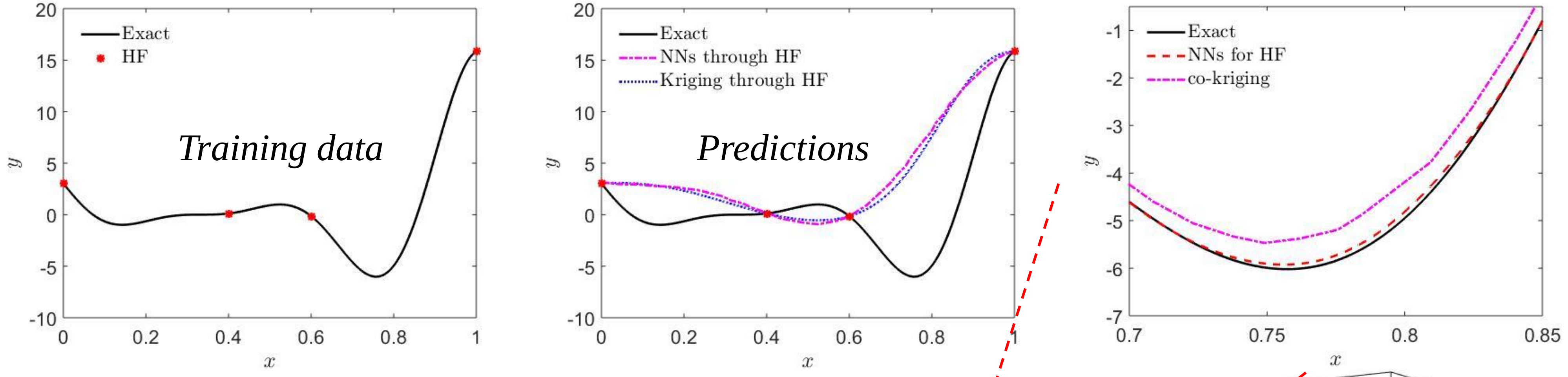
Function Approximation: Continuous Function - Linear Correlation

- Low-fidelity function:
  $$y_L(x) = 0.5(6x-2)^2 \sin(12x-4) + 10(x-0.5) - 5, \qquad x \in [0,1]$$
- High-fidelity function:
  $$y_H(x) = 2y_L(x) - 20(x-0.5) + 10, \qquad x \in [0,1]$$

single-fidelity

$NN_H : 20\times 4$

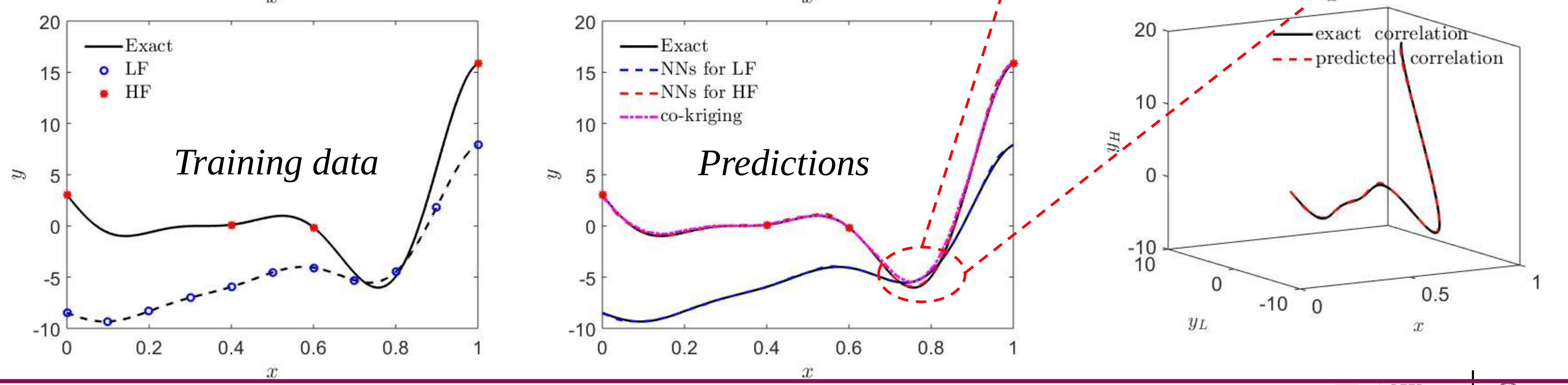
(*width* $\times$ *depth*)



multi-fidelity

$NN_L : 20\times 4$

$NN_H : 20\times 1$



Implementation of Multi-fidelity DNN

1. Import Modules

```
## Code for Multi-fidelity DNN
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(seed=1234)
tf.random.set_seed(1234)
```

2. Define Low and High-fidelity functions

```
# Exact Low Fidelity Fucntion
def fun_lf(x):
    y = 0.5*(6*x - 2)**2*np.sin(12*x - 4) + 10*(x - 0.5) - 5
    return y

#Exact high-fidelity function
def fun_hf(x):
    y = (6*x - 2)**2*np.sin(12*x - 4)
    return y
```

3. Define DNN Class

```
class DNN:
    def __init__(self):
        pass

    def hyper_initial(self, layers):
        L = len(layers)
        W = []
        b = []
        for l in range(1, L):
            in_dim = layers[l-1]
            out_dim = layers[l]
            std = np.sqrt(2/(in_dim + out_dim))
            weight = tf.Variable(tf.random.truncated_normal(shape=[in_dim, out_dim], stddev=std))
            bias = tf.Variable(tf.zeros(shape=[1, out_dim]))
            W.append(weight)
            b.append(bias)

        return W, b

    def fnn(self, W, b, X, Xmin, Xmax):
        A = 2.0*(X - Xmin)/(Xmax - Xmin) - 1.0
        L = len(W)
        for i in range(L-1):
            A = tf.tanh(tf.add(tf.matmul(A, W[i]), b[i]))
        Y = tf.add(tf.matmul(A, W[-1]), b[-1])

        return Y

    def train_vars(self, W, b):
        return W + b
```

Implementation of Multi-fidelity DNN

1. Import Modules

```python
## Code for Multi-fidelity DNN
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(seed=1234)
tf.random.set_seed(1234)
```

2. Define DNN Class

```python
class DNN:
    def __init__(self):
        pass

    def hyper_initial(self, layers):
        L = len(layers)
        W = []
        b = []
        for l in range(1, L):
            in_dim = layers[l-1]
            out_dim = layers[l]
            std = np.sqrt(2/(in_dim + out_dim))
            weight = tf.Variable(tf.random.truncated_normal(shape=[in_dim, out_dim], stddev=std))
            bias = tf.Variable(tf.zeros(shape=[1, out_dim]))
            W.append(weight)
            b.append(bias)

        return W, b

    def fnn(self, W, b, X, Xmin, Xmax):
        A = 2.0*(X - Xmin)/(Xmax - Xmin) - 1.0
        L = len(W)
        for i in range(L-1):
            A = tf.tanh(tf.add(tf.matmul(A, W[i]), b[i]))
        Y = tf.add(tf.matmul(A, W[-1]), b[-1])

        return Y

    def train_vars(self, W, b):
        return W + b
```

2. Define Low and High-fidelity functions

$$y_L = 0.5(6x-2)^2\sin(12x-4) + 10(z-0.5) - 5$$

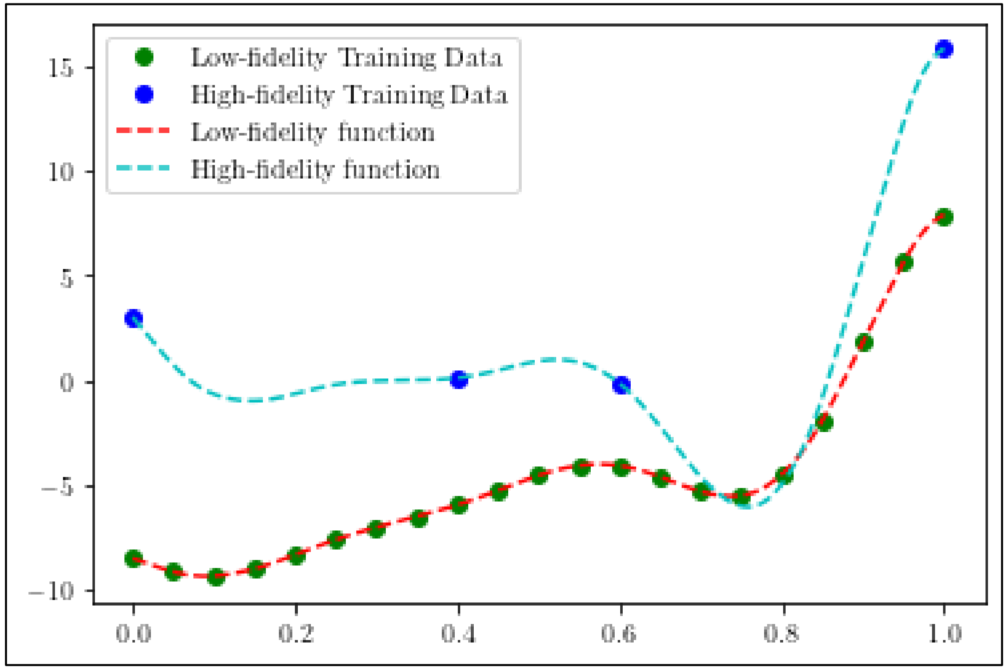
$$y_H = (6x-2)^2\sin(12x-4)$$

```python
# Exact Low Fidelity Fucntion
def fun_lf(x):
    y = 0.5*(6*x - 2)**2*np.sin(12*x - 4) + 10*(x - 0.5) - 5
    return y

#Exact high-fidelity function
def fun_hf(x):
    y = (6*x - 2)**2*np.sin(12*x - 4)
    return y
```

Implementation of Multi-fidelity DNN: Contd.

4. Define Training data and Tensors

```
## Dimension of Function
D = 1
#low-fidelity NN
layers_lf = [D] + 2*[20] + [1]

#nonlinear correlation
layers_hf_nl = [D+1] + 2*[10] + [1]

#linear correlation
layers_hf_l = [D+1] + [1]

#low-fidelity training data
x_lf = np.linspace(0, 1, 21).reshape((-1, 1))
y_lf = fun_lf(x_lf)

#high-fidelity training data
x_hf = np.array([0., 0.4, 0.6, 1.0]).reshape((-1, 1))

#low-fidelity training data at x_H
y_lf_hf = fun_lf(x_hf)
y_hf = fun_hf(x_hf)
X_hf = np.hstack((x_hf, y_lf_hf))

Xmin = x_lf.min(0)
Xmax = x_lf.max(0)
Ymin = y_lf.min(0)
Ymax = y_lf.max(0)

Xhmin = np.hstack((Xmin, Ymin))
Xhmax = np.hstack((Xmax, Ymax))

x_train_lf = tf.convert_to_tensor(x_lf, dtype=tf.float32)
y_train_lf = tf.convert_to_tensor(y_lf, dtype=tf.float32)
x_train_hf = tf.convert_to_tensor(X_hf, dtype=tf.float32)
y_train_hf = tf.convert_to_tensor(y_hf, dtype=tf.float32)
```



Implementation of Multi-fidelity DNN: Contd.

TensorFlow 2.0

4. Define Training data and Tensors

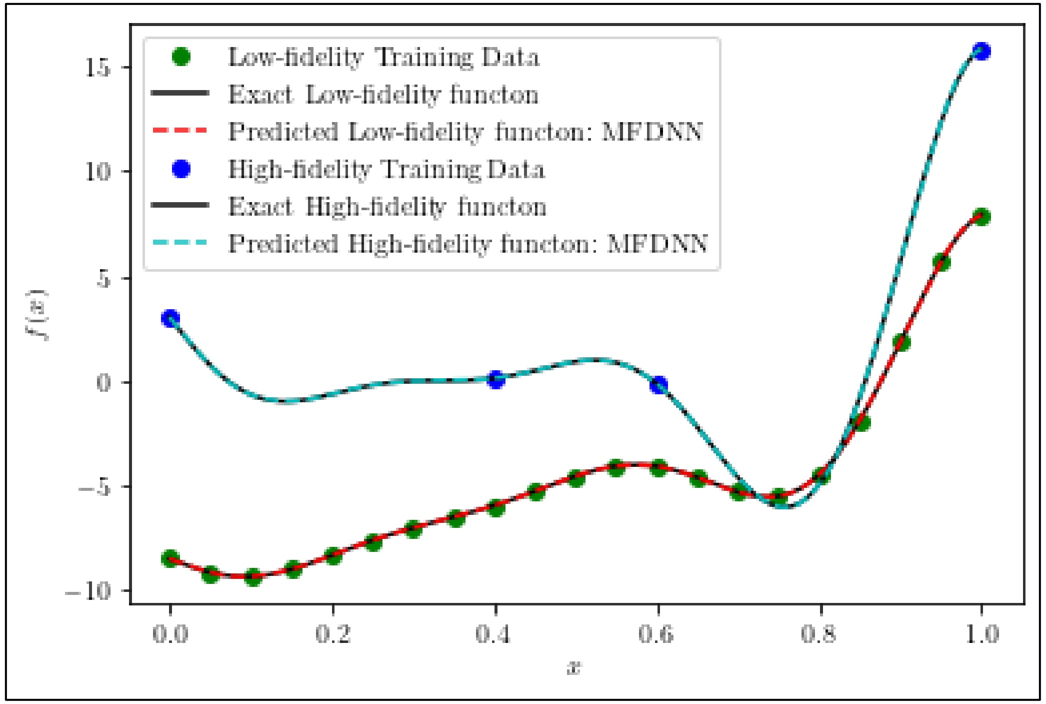
```
model = DNN()
W_lf, b_lf = model.hyper_initial(layers_lf)
W_hf_nl, b_hf_nl = model.hyper_initial(layers_hf_nl)
W_hf_l, b_hf_l = model.hyper_initial(layers_hf_l)

W=[W_lf, W_hf_nl, W_hf_l]
b=[b_lf, b_hf_nl, b_hf_l]

lr =0.001
optimizer = tf.optimizers.Adam(learning_rate=lr)



nmax = 30000
loss_c = 1.0e-3
loss_ = 1.0
n = 0

while n < nmax:
    n += 1
    loss_, loss_lf_, loss_hf_ = train_step(W, b, model, x_train_lf, y_train_lf , \
                                           x_train_hf, y_train_hf,optimizer, train=1)

    if n%1000 == 0:
        print('n: %d, loss: %.3e, loss_lf: %.3e, loss_hf: %.3e'%(n, loss_, loss_lf_, loss_hf_))
```

```
@tf.function
def train_step(W, b, model, x_train_lf, y_train_lf , x_train_hf, y_train_hf,opt, train=1):
    W_lf = W[0]
    W_hf_nl = W[1]
    W_hf_l = W[2]
    b_lf =   b[0]
    b_hf_nl = b[1]
    b_hf_l = b[2]
    if train==1:
        with tf.GradientTape() as tape:
            tape.watch([W_lf, W_hf_nl,W_hf_l, b_lf, b_hf_nl, b_hf_l])
            y_pred_lf = model.fnn(W_lf, b_lf, x_train_lf, Xmin, Xmax)
            y_pred_hf_nl = model.fnn(W_hf_nl, b_hf_nl, x_train_hf, Xhmin, Xhmax)
            y_pred_hf_l = model.fnn(W_hf_l, b_hf_l, x_train_hf, Xhmin, Xhmax)
            y_pred_hf = y_pred_hf_l + y_pred_hf_nl
            loss_l2 = 0.01*tf.add_n([tf.nn.l2_loss(w_) for w_ in W_hf_nl])
            loss_lf = tf.reduce_mean(tf.square(y_pred_lf - y_train_lf))
            loss_hf =  tf.reduce_mean(tf.square(y_pred_hf - y_train_hf))
            loss = loss_lf + loss_hf + loss_l2
        grads = tape.gradient(loss, W_lf + b_lf + W_hf_nl + b_hf_nl +  W_hf_l + b_hf_l)
        opt.apply_gradients(zip(grads, W_lf + b_lf + W_hf_nl + b_hf_nl +  W_hf_l + b_hf_l))
        return loss, loss_lf, loss_hf

    if train == 0:
        y_pred_lf = model.fnn(W_lf, b_lf, x_train_lf, Xmin, Xmax)
        y_pred_hf_nl = model.fnn(W_hf_nl, b_hf_nl, x_train_hf, Xhmin, Xhmax)
        y_pred_hf_l = model.fnn(W_hf_l, b_hf_l, x_train_hf, Xhmin, Xhmax)
        y_pred_hf = y_pred_hf_l + y_pred_hf_nl
        return  y_pred_hf, y_pred_lf
```

Implementation of Multi-fidelity DNN: Contd.

5. Prediction

```python
#prediction
import matplotlib
import matplotlib.pyplot as plt
import mlai
x_test = np.linspace(0, 1, 1000, dtype=float).reshape((-1, 1))
y_lf_ref = fun_lf(x_test)
y_hf_ref = fun_hf(x_test)
x_test = tf.convert_to_tensor(x_test, dtype=tf.float32)
_, y_lf_test = train_step(W, b, model, x_test, y_train_lf , x_train_hf, y_train_hf,optimizer, train=0)
X_test = np.hstack((x_test, y_lf_test))
X_test = tf.convert_to_tensor(X_test, dtype=tf.float32)
y_hf_test, _ = train_step(W, b, model, x_test, y_train_lf , X_test, y_train_hf,optimizer, train=0)
plt.figure(facecolor='w')
plt.rc('text', usetex=True)
plt.rc('font', family='serif', size=10)

plt.plot(x_lf, y_lf, 'go', label="Low-fidelity Training Data")
plt.plot(x_test, y_lf_ref, 'k-', label="Exact Low-fidelity functon")
plt.plot(x_test, y_lf_test, 'r--', label="Predicted Low-fidelity functon: MFDNN")

plt.plot(x_hf, y_hf, 'bo', label="High-fidelity Training Data")
plt.plot(x_test, y_hf_ref, 'k-', label="Exact High-fidelity functon")
plt.plot(x_test, y_hf_test, 'c--', label="Predicted High-fidelity functon: MFDNN")
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
plt.legend()
plt.show()
```



Summary

- There are many extensions of PINNs that can enhance accuracy and reduce training time.
- The gradient enhanced PINN (gPINN) increases accuracy up to two orders of magnitude and requires only a small set of residual points but it involves expensive extra automatic differentiation.
- The conservative PINN (cPINN) is based on domain decomposition and mimics the discontinuous Galerkin method. It can lead to great parallelization in space.
- The extended PINN (XPINN) is also based on domain decomposition but it is not limited to conservation laws and can lead to parallelization in space and time. It can be applied to any partial differential equation, and it ensures that the residual and state are continuous at the interfaces.
- The fractional PINN (fPINN) can be used to solve fractional PDES in space or time. Unlike the vanilla PINN, it requires an auxiliary grid of points and numerical discretization of operators as authomatic differentiation cannot be applied to fractional derivatives.
- The stochastic PINN (sPINN) can be used to solve stochastic PDEs. It is based on the Wasserstein GAN combined with a gradient penalty term for stabilization. It scales to high dimensions and the cost is approximately quadratic with the dimension.

References

- Blum AL, Rivest RL. Training a 3-node neural network is NP-complete. Neural Networks. 1992 Jan 1;5(1):117-27.
- Hu Z, Jagtap AD, Karniadakis GE, Kawaguchi K. When Do Extended Physics-Informed Neural Networks (XPINNs) Improve Generalization?. arXiv preprint arXiv:2109.09444. 2021 Sep 20.
- Jagtap AD, Karniadakis GE. Extended physics-informed neural networks (xPINNs): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. Communications in Computational Physics. 2020 Nov 1;28(5):2002-41.
- L. Lu, Y. Su, and G. E. Karniadakis. Collapse of Deep and Narrow Neural Nets. arXiv preprint arXiv:1808.04947. 2018 Aug 15.
- Lee JD, Simchowitz M, Jordan MI, Recht B. Gradient descent only converges to minimizers. InConference on learning theory 2016 Jun 6 (pp. 1246-1257). PMLR.
- Jagtap AD, Kharazmi E, Karniadakis GE. Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. Computer Methods in Applied Mechanics and Engineering. 2020 Jun 15;365:113028.
- Pang G, Lu L, Karniadakis GE. fPINNs: Fractional physics-informed neural networks. SIAM Journal on Scientific Computing. 2019;41(4):A2603-26.
- Yang L, Zhang D, Karniadakis GE. Physics-informed generative adversarial networks for stochastic differential equations. SIAM Journal on Scientific Computing. 2020;42(1):A292-317.
- Yu J, Lu L, Meng X, Karniadakis GE. Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems. Computer Methods in Applied Mechanics and Engineering. 2022 Apr 1;393:114823.

Deep Learning for Science and Engineering Teaching Kit

Thank You