Deep Learning for Science and Engineering Teaching Kit

Deep Learning for Scientists and Engineers

Lecture 9: Physics-Informed Neural Networks (PINNs)

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)

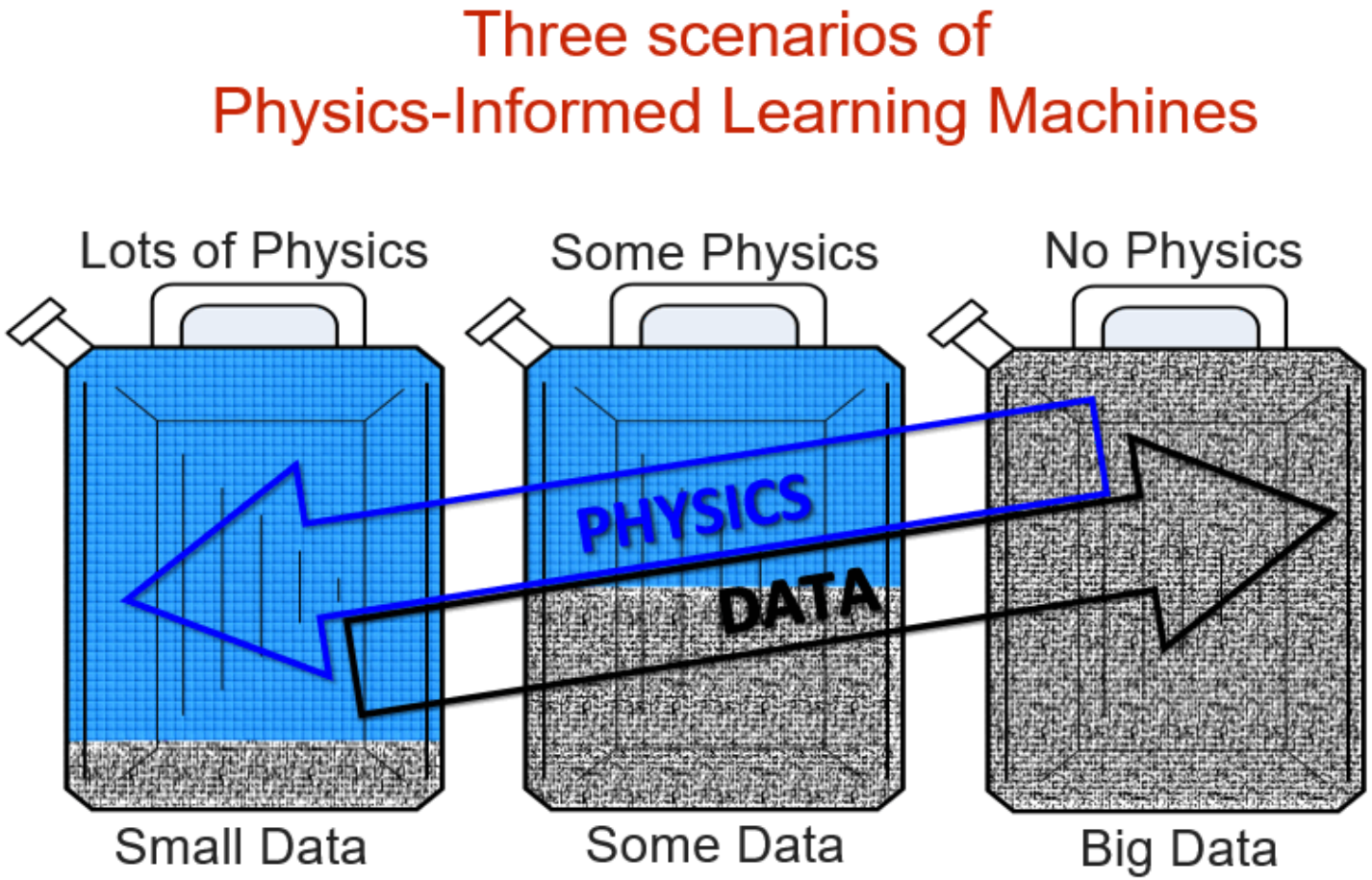
- Lecture 7: Physics-Informed Neural Networks (PINNs)
- Lecture 8: PINN Extensions
- Lecture 9: Neural Operators

Module-3 (Codes & Scalability)

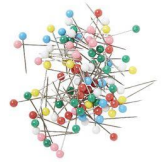
- Lecture 10: Multi-GPU Scientific Machine Learning

Contents

Data + Physical Laws



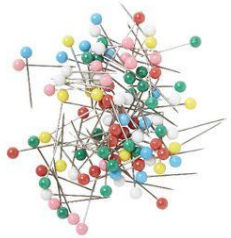
Data + Neural Networks + Physical Laws

PINNs 

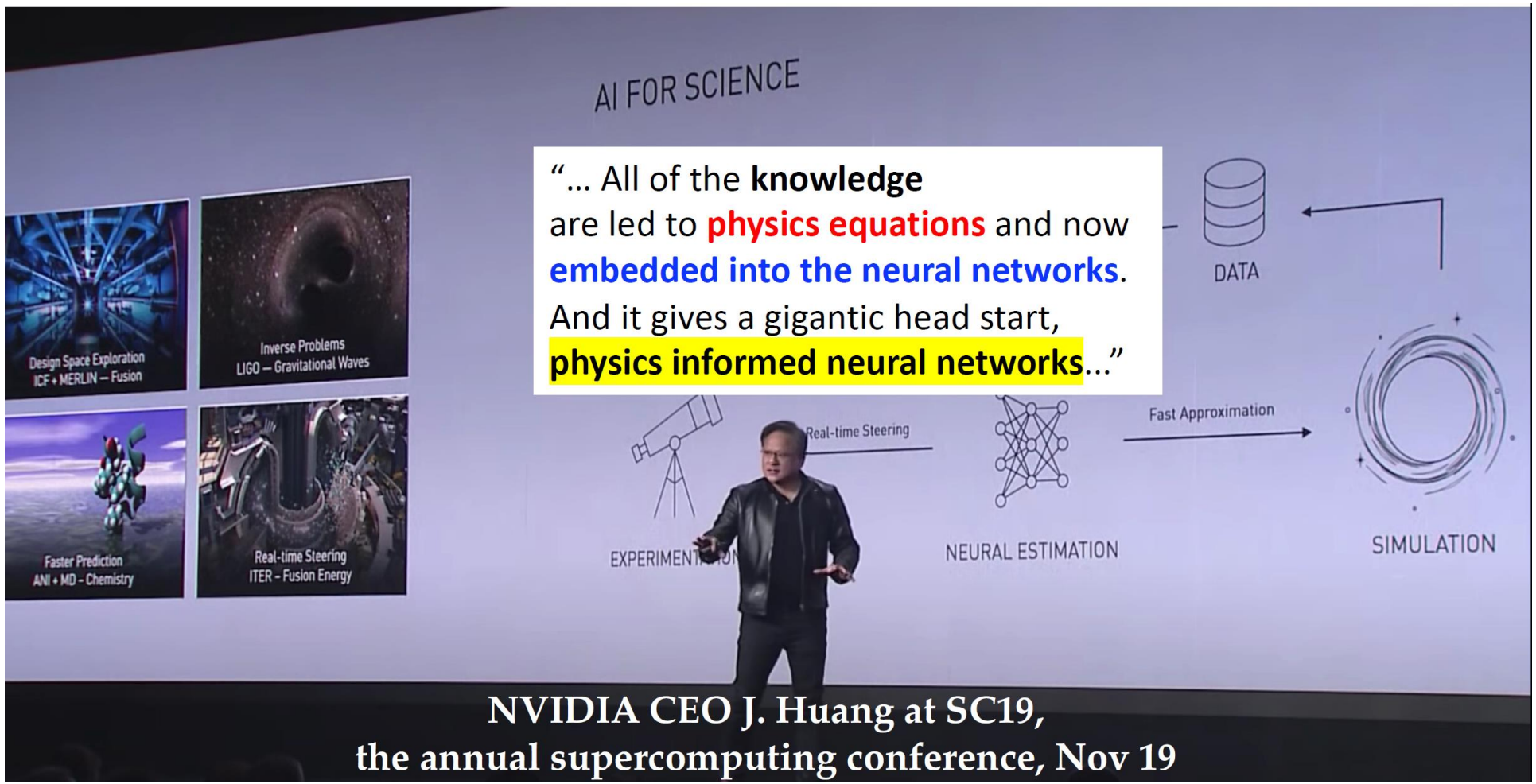
Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations

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

arXiv:1711.10561; arXiv:1711.10566 (2017)

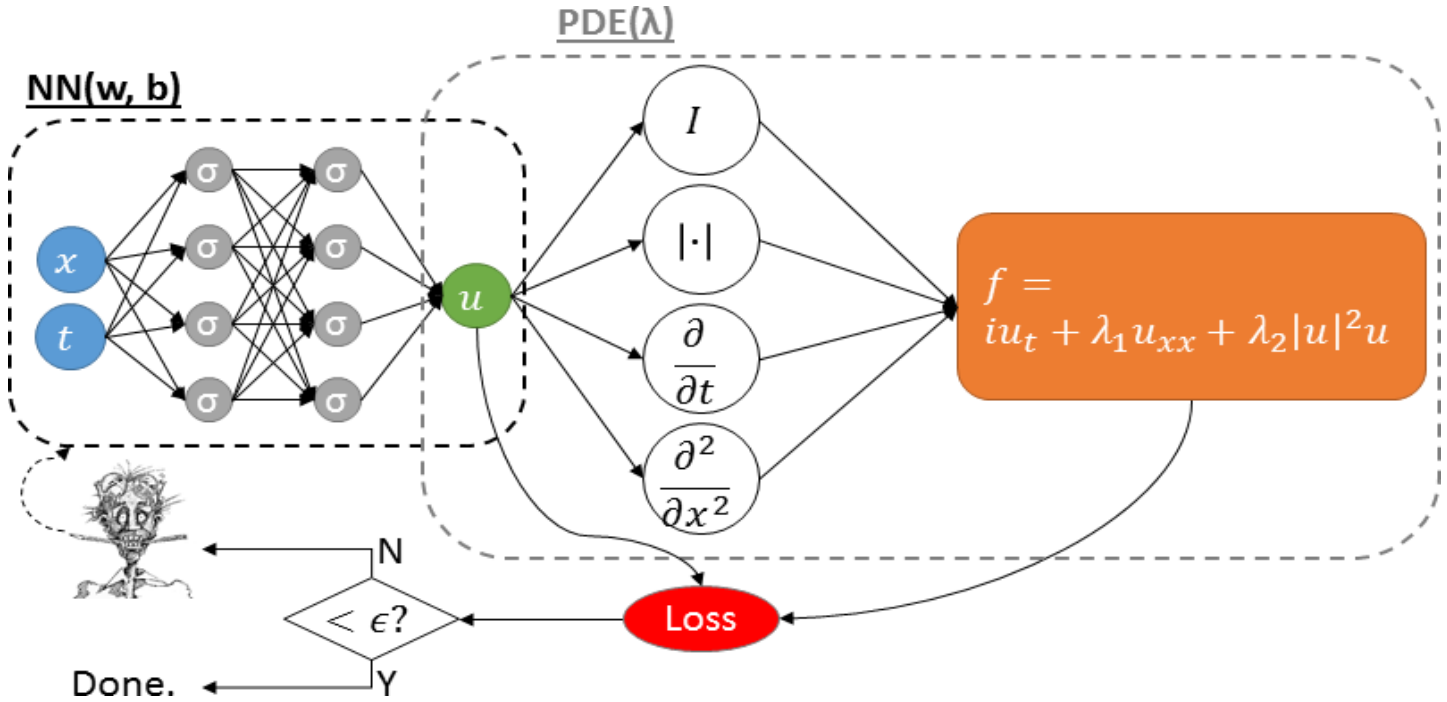


PINNs: Physics-Informed Neural Networks



What is a PINN? Physics-Informed Neural Network

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



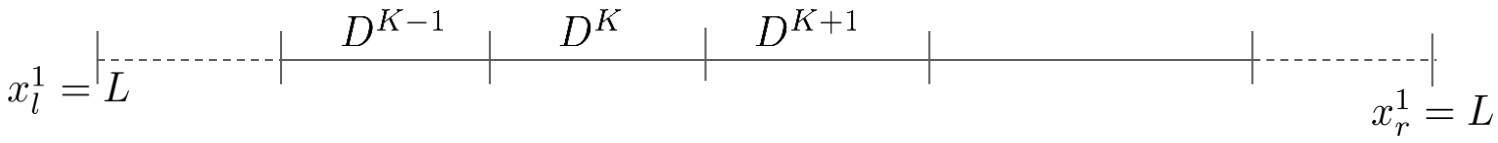
- Minimize:

$$MSE = MSE_u + MSE_f,$$

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

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

Why PINNs: PINN vs (DG) FEM



- Simple Linear Wave Equation with $a = 30$

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0, \qquad x \in [-1, 1]$$

with initial condition $u(x, 0) = -\sin(\pi x)$ and periodic boundary conditions.

- The exact solution reads

$$u(x, t) = -\sin(\pi x - at)$$

- The local solution is well approximated as

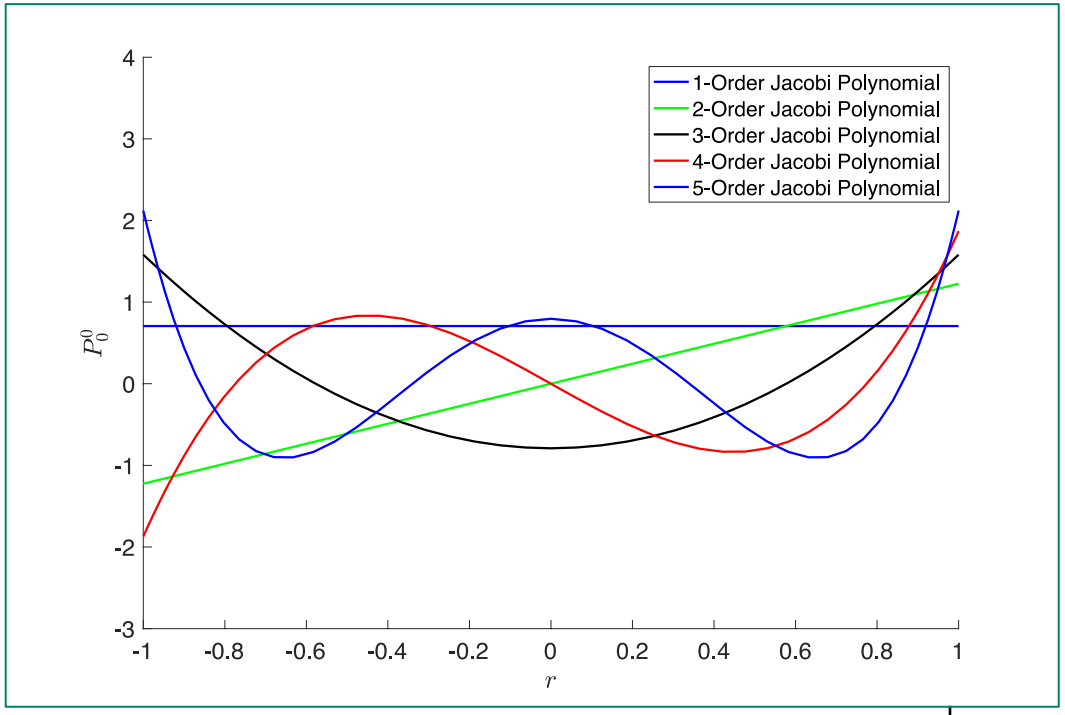
$$u_h^k(x, t) = \sum_{i=1}^{N_p} u_h^k(x_i^k, t) l_i^k(x)$$

The local semi-discrete schemes look like

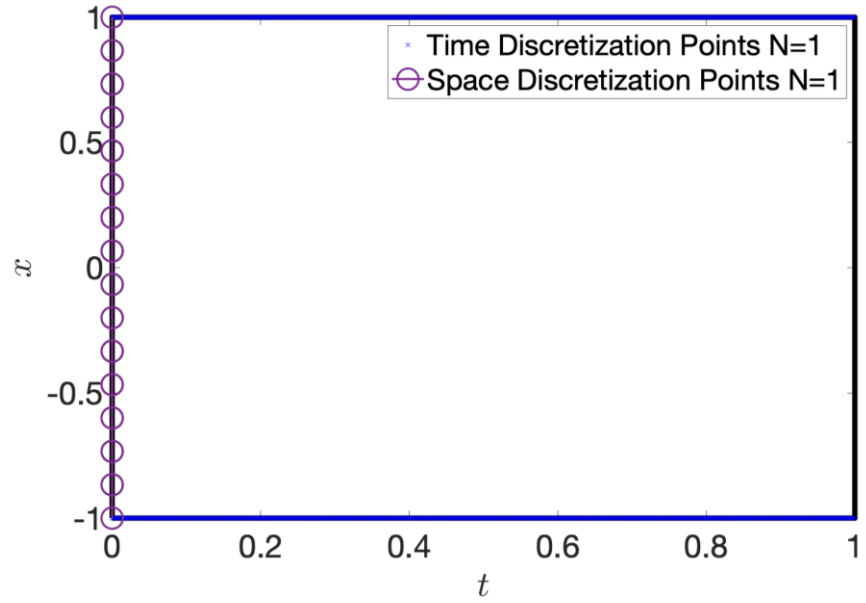
$$\mathcal{M}^k \frac{\mathrm{d}\boldsymbol{u}_h^k}{dt} + 2\pi\mathcal{S}\boldsymbol{u}_h^k = \left[l^k(x)(2\pi u_h^k - (2\pi u_h)^*)\right],$$

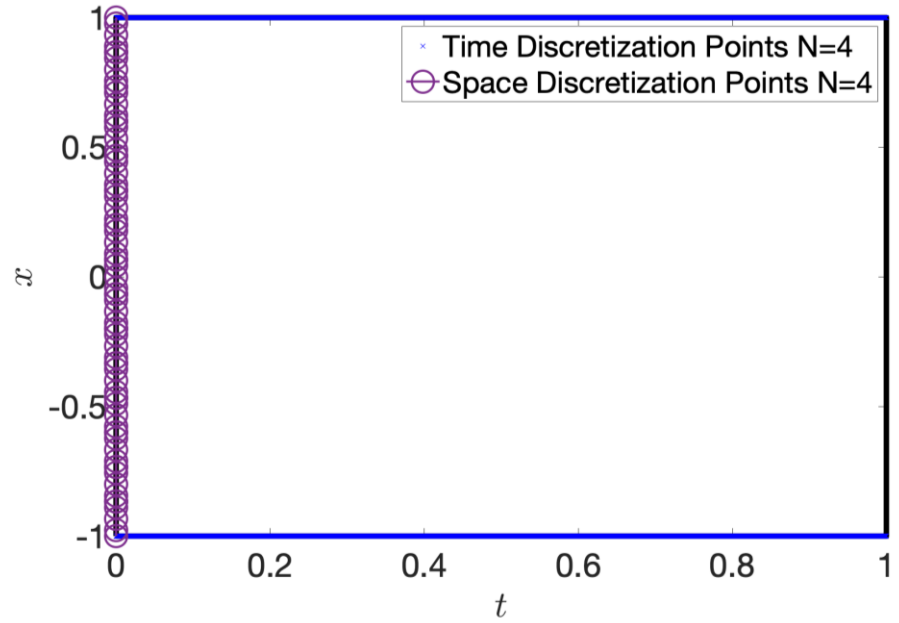
where $(2\pi u_h)^*$ is numerical flux.

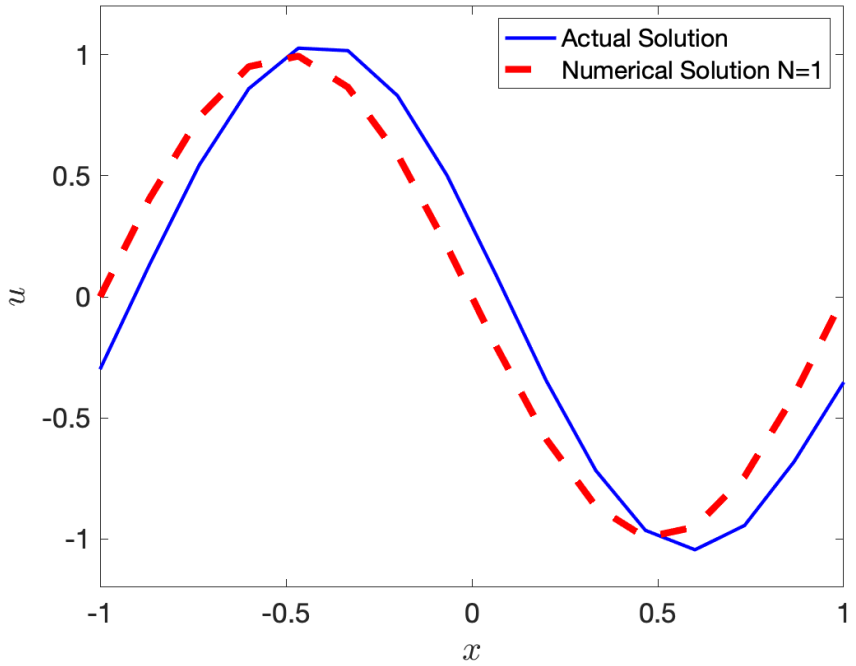
$l_i^k(x)$

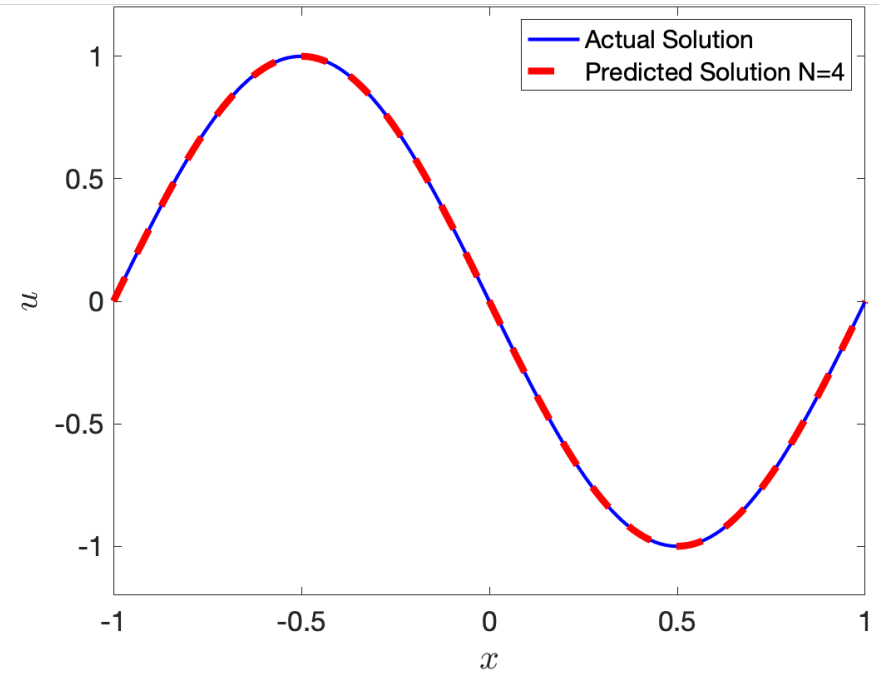


Why PINNs: PINN vs (DG) FEM



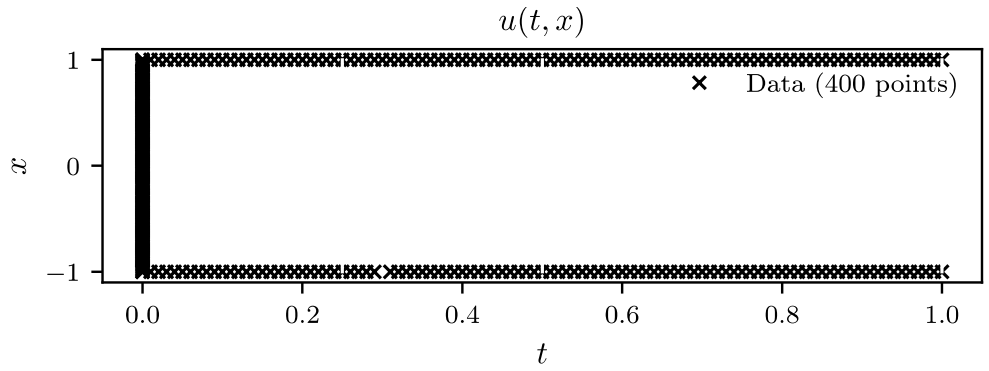




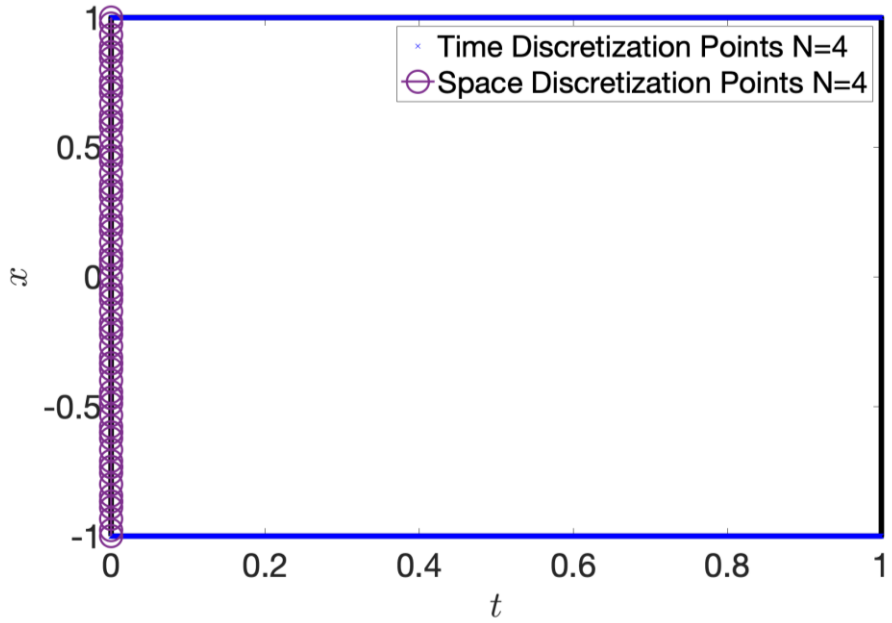


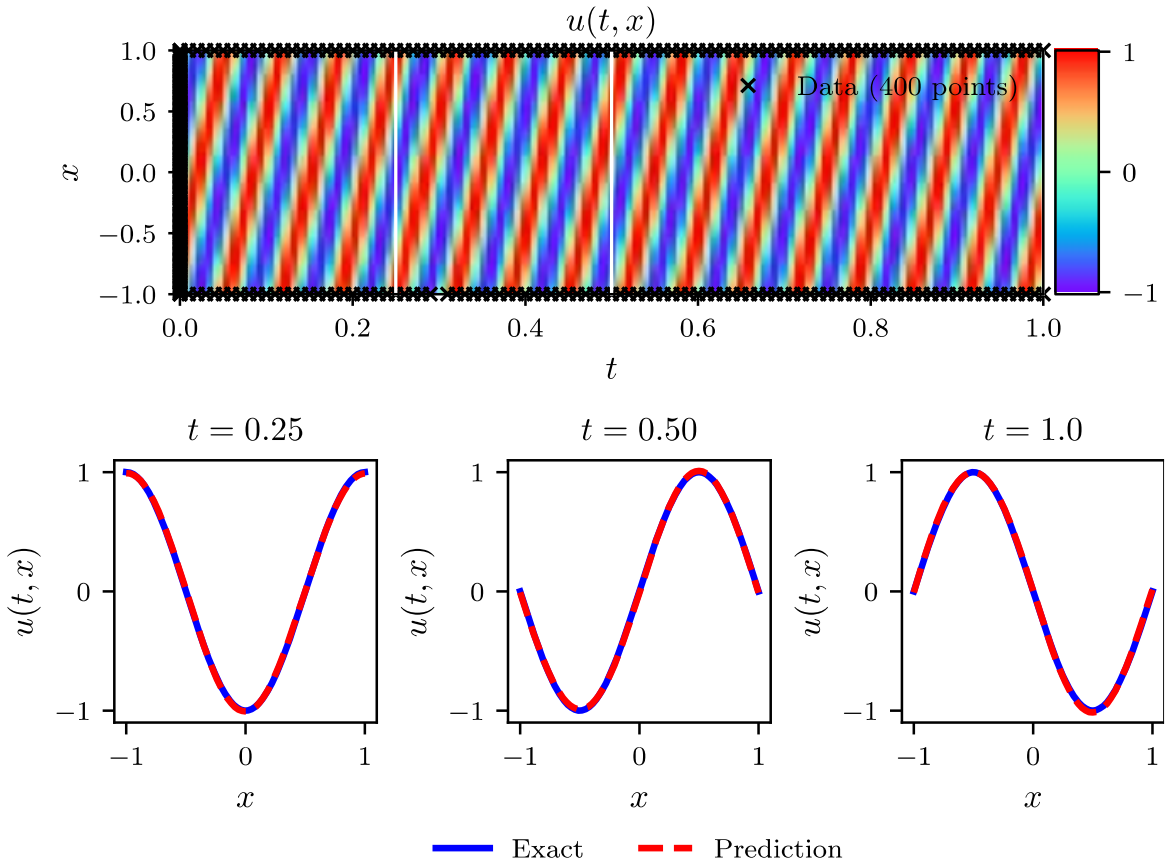
Why PINNs: PINN vs (DG) FEM

Data Distribution PINN



Data Distribution PINN





PINN for Burger's Equation: Implementation

Burger's Equation is defined as

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

1. Import Modules and Deep Learning Framework

```
import sys
sys.path.insert(0, 'Utilities/')
import os

from scipy.interpolate import griddata
from pyDOE import lhs
from plotting import newfig, savefig
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.gridspec as gridspec
from mpl_toolkits.axes_grid1 import make_axes_locatable
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import time
import scipy.io
```

2a. Initialize Layers and NN Parameters

```
layers = [2, 20, 20, 20, 20, 20, 20, 20, 20, 1]
L = len(layers)
W = [hyper_initial([layers[l-1], layers[l]]) for l in range(1, L)]
b = [tf.Variable(tf.zeros([1, layers[l]])) for l in range(1, L)]
```

2b. Glorot Normal Initialization

```
def hyper_initial(size):
    in_dim = size[0]
    out_dim = size[1]
    std = np.sqrt(2.0/(in_dim + out_dim))
    return tf.Variable(tf.random.truncated_normal(shape=size, stddev = std))
```

PINN for Burger's Equation: Implementation

3a. Forward Pass

```
def net_u(x, t, w, b):
    u = DNN(tf.concat([x,t],1), w, b)
    return u
```

3b. DNN Function:

```
# Neural Network
def DNN(X, W, b):
    A = X
    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
```

4. Residual Computation

```
def net_f(x,t,W, b, nu):
    with tf.GradientTape(persistent=True) as tape1:
        tape1.watch([x, t])
        with tf.GradientTape(persistent=True) as tape2:
            tape2.watch([x, t])
            u=net_u(x,t, W, b)
        u_t = tape2.gradient(u, t)
        u_x = tape2.gradient(u, x)
    u_xx = tape1.gradient(u_x, x)
    del tape1
    f = u_t + u*u_x - nu*u_xx
    return f
```

PINN for Burger's Equation: Implementation

5. Backward Propagation

```
def train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, opt, nu):
    x_u = X_u_train_tf[:,0:1]
    t_u = X_u_train_tf[:,1:2]
    x_f = X_f_train_tf[:,0:1]
    t_f = X_f_train_tf[:,1:2]
    with tf.GradientTape() as tape:
        tape.watch([W,b])
        u_nn = net_u(x_u, t_u, W, b)
        f_nn = net_f(x_f,t_f, W, b, nu)
        loss =  tf.reduce_mean(tf.square(u_nn - u_train_tf)) + tf.reduce_mean(tf.square(f_nn))
    grads = tape.gradient(loss, train_vars(W,b))
    opt.apply_gradients(zip(grads, train_vars(W,b)))
    return loss
```

6. Predict

```
def predict(X_star_tf, w, b):
    x_star = X_star_tf[:,0:1]
    t_star = X_star_tf[:,1:2]
    u_pred = net_u(x, t, w, b)
    return u_pred
```

PINN for Burger's Equation: Implementation

7. Data Preparation and Training

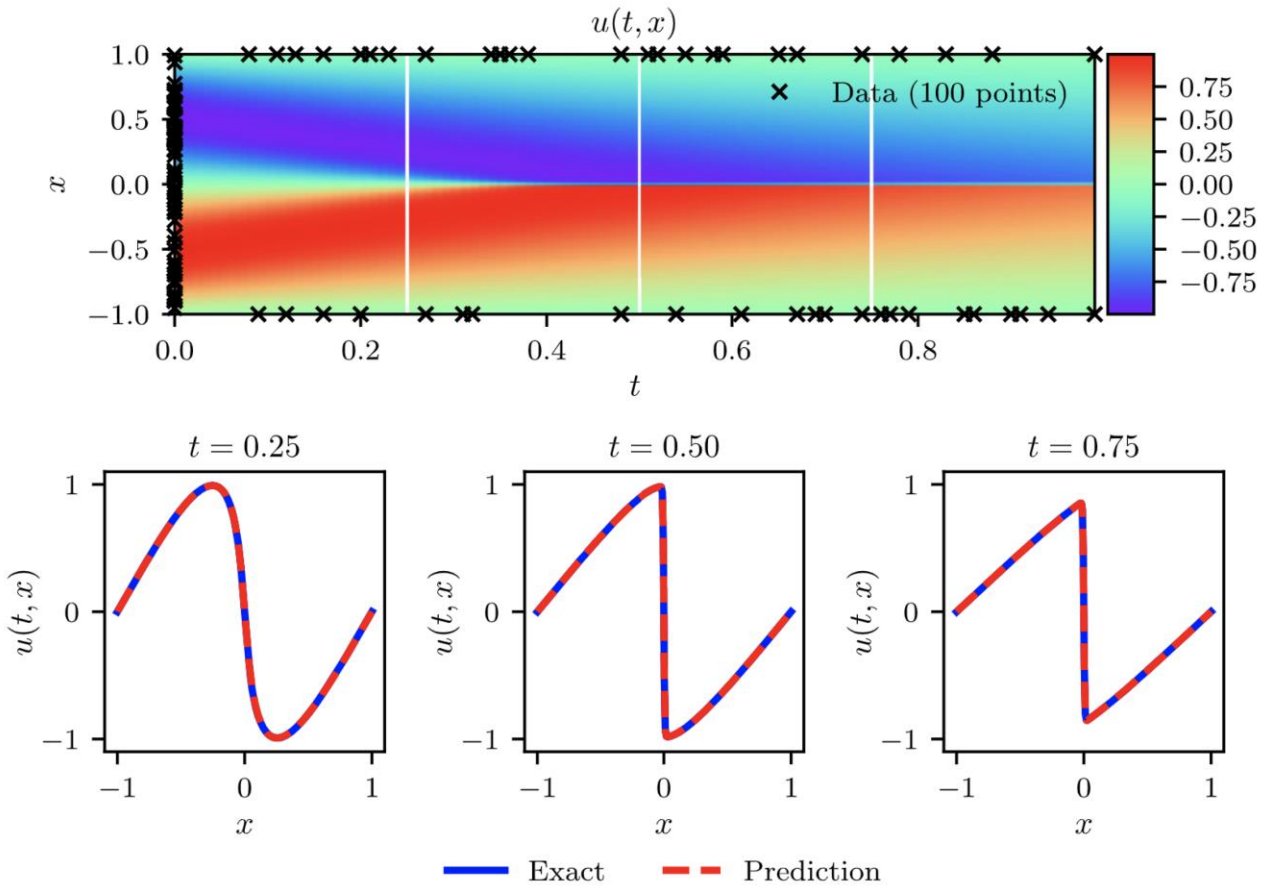
```
nu = 0.01/np.pi
noise = 0.0
N_u = 100
N_f = 10000
Nmax=10000

layers = [2, 20, 20, 20, 20, 20, 20, 20, 20, 1]
L = len(layers)
W = [hyper_initial([layers[l-1], layers[l]]) for l in range(1, L)]
b = [tf.Variable(tf.zeros([1, layers[l]])) for l in range(1, L)]

data = scipy.io.loadmat('./Data/burgers_shock.mat')

t = data['t'].flatten()[:,None]
x = data['x'].flatten()[:,None]
Exact = np.real(data['usol']).T
X, T = np.meshgrid(x,t)
X_star = np.hstack((X.flatten()[:,None], T.flatten()[:,None]))
u_star = Exact.flatten()[:,None]
# Doman bounds
lb = X_star.min(0)
ub = X_star.max(0)
xx1 = np.hstack((X[0:1,:].T, T[0:1,:].T))
uu1 = Exact[0:1,:].T
xx2 = np.hstack((X[:,0:1], T[:,0:1]))
uu2 = Exact[:,0:1]
xx3 = np.hstack((X[:,-1:], T[:,-1:]))
uu3 = Exact[:,-1:]

X_u_train = np.vstack([xx1, xx2, xx3])
X_f_train = lb + (ub-lb)*lhs(2, N_f)
X_f_train = np.vstack((X_f_train, X_u_train))
u_train = np.vstack([uu1, uu2, uu3])

idx = np.random.choice(X_u_train.shape[0], N_u, replace=False)

X_u_train = X_u_train[idx, :]
u_train = u_train[idx,:]

X_u_train_tf = tf.convert_to_tensor(X_u_train, dtype=tf.float32)
u_train_tf =   tf.convert_to_tensor(u_train, dtype=tf.float32)
X_f_train_tf = tf.convert_to_tensor(X_f_train, dtype=tf.float32)
```

8. Optimizer and PINN Training

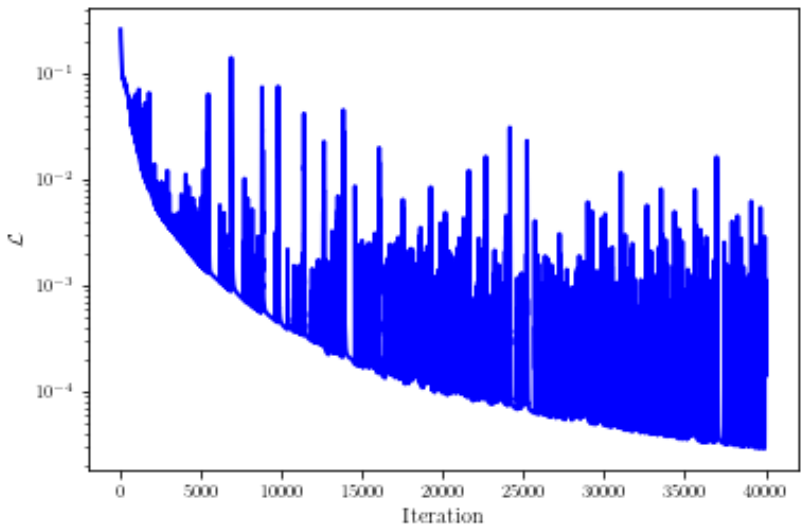
```
lr = 1e-3
optimizer = tf.optimizers.Adam(learning_rate=lr)

start_time = time.time()
n=0
loss = []
while n <= Nmax:
    loss_= train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, optimizer, nu)
```

PINN for Burger's Equation: Results

8. Actual vs Predicted Solutions at Different Times



9. Loss Function



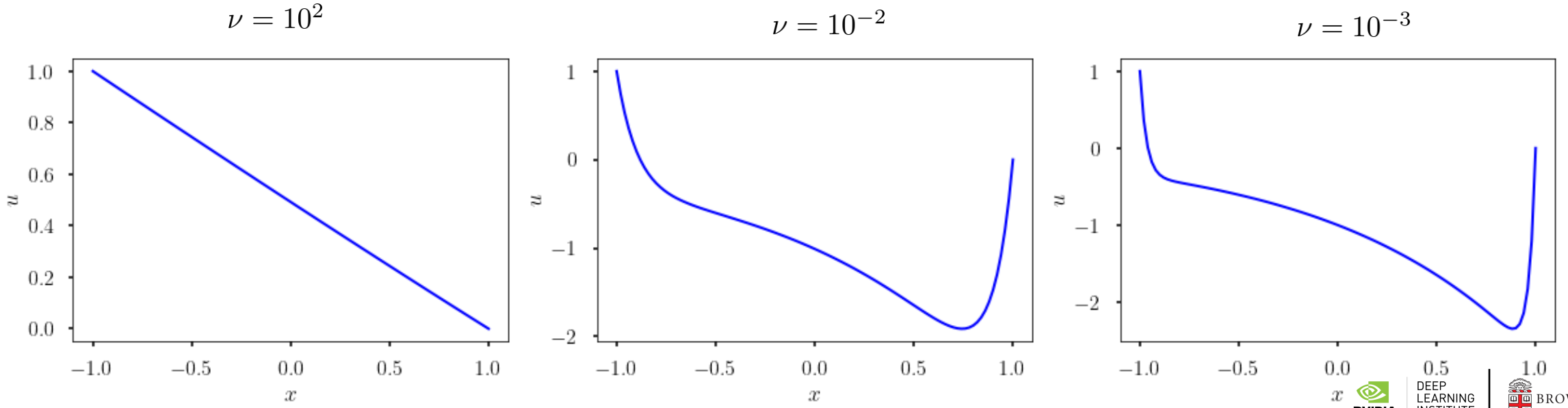
PINNs: Training For Boundary Value Problems

- Define an ODE for given boundary conditions

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

where $x \in [-1, 1],\ u[-1] = 1, u[1] = 0, \nu = 10^{-3}$.

- Boundary Layer vs Viscosity



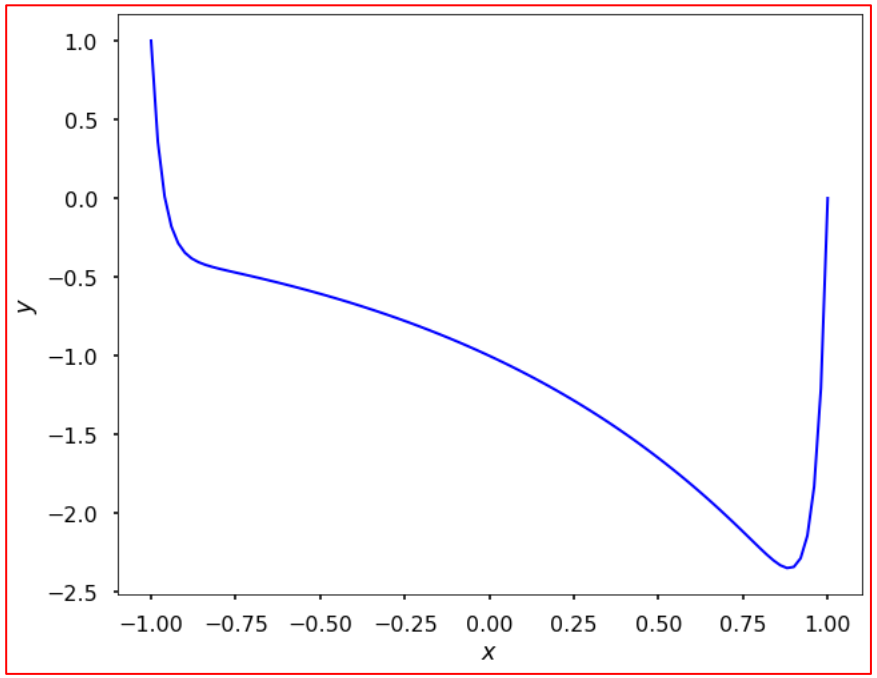
Solution using Central Finite Difference Method

$$u_0 = 1$$

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} - u_i = \exp(x_i)$$

$$u_m = 0, \qquad \nu = 10^{-3}$$

Solution computed using Central Finite-Difference Scheme




```python
### Solution of Equation using Central Finite Difference Equation
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-poster')
%matplotlib inline
### Number of Gridpoints
n = 100
h = (1+1) / n
x = np.linspace(-1,1,n+1)
# Difference Operator
A = np.zeros((n+1, n+1))

## Coefficient For Boundary Condition
A[0, 0] = 1
A[n, n] = 1

### Maric for Interior Point
for i in range(1, n):
    A[i, i-1] = 1
    A[i, i] = -(2 + (10**3)*h**2)
    A[i, i+1] = 1

# Get b
b = np.zeros(n+1)
b = (np.exp(x))*h*h*(10**3)

#### Imposition Boundary Condition
b[0] = 1
b[-1] = 0
# solve the linear equations
y = np.linalg.solve(A, b)
### Plot of Equation
plt.figure(figsize=(10,8))
plt.plot(x, y, "-", lw=2.0, color="b")
plt.xlabel('$x$')
plt.ylabel("$y$")
plt.show()
```

Approach 1: PINNs with Soft Constraints

- Loss = Boundary Losses + Residual Loses

$$\mathcal{L} = \lambda_{B_1}\mathcal{L}_{B_1} + \lambda_{B_2}\mathcal{L}_{B_2} + \lambda_f\mathcal{L}_f$$

PINN Architecture	
Hyper-parameters	**Value**
No. of Layers	6
Neurons per layer	4
No. of Iterations	3000
Learning rate	5×10^{-3}
No. of Residuals Points	200
$(\lambda_{B_1}, \lambda_{B_2}, \lambda_f)$	$(1, 1, 1)$

Approach 1: PINNs with Soft Constraints

- Code

1. Import Modules and Initial Setup

```
import sys
sys.path.insert(0, 'Utilities/')
import os

from scipy.interpolate import griddata
from pyDOE import lhs
from plotting import newfig, savefig
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.gridspec as gridspec
from mpl_toolkits.axes_grid1 import make_axes_locatable
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import time
import scipy.io

np.random.seed(seed=1234)
tf.random.set_seed(1234)
tf.config.experimental.enable_tensor_float_32_execution(False)
lb = -1
ub = 1
```

2. Routines for Initialization, Forward Pass and Residual computation

```
# Initalization of Network
def hyper_initial(size):
    in_dim = size[0]
    out_dim = size[1]
    std = np.sqrt(2.0/(in_dim + out_dim))
    return tf.Variable(tf.random.truncated_normal(shape=size, s

# Neural Network
def DNN(X, W, b):
    A = 2.0*(X - lb)/(ub - lb) - 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(W, b):
    return W + b

def net_u(x,w, b):
    u = DNN(x, w, b)
    return u


#@tf.function(jit_compile=True)
@tf.function
def net_f(x,W, b, nu):
    with tf.GradientTape(persistent=True) as tape1:
        tape1.watch([x])
        with tf.GradientTape(persistent=True) as tape2:
            tape2.watch([x])
            u=net_u(x, W, b)
        u_x = tape2.gradient(u, x)
        del tape2
    u_xx = tape1.gradient(u_x, x)
    del tape1
    f = u_xx - (1/nu)*u-(1/nu)*tf.exp(x)
    return f
```

Approach 1: PINNs with Soft Constraints

- Code

3. Routines for Training and Predict

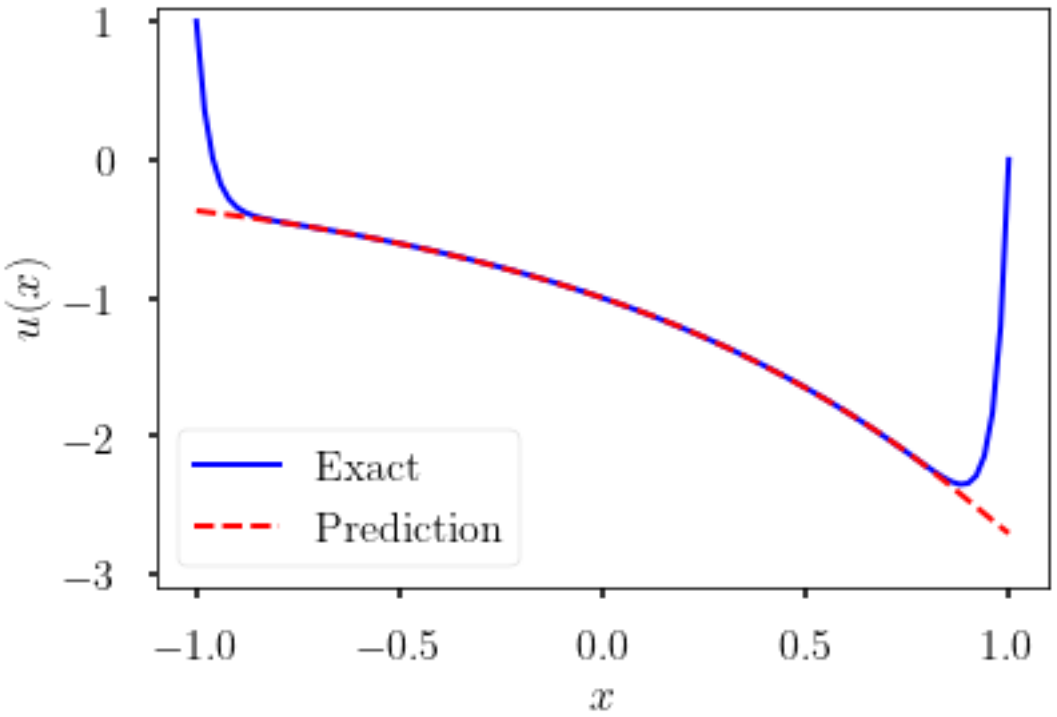
```
#@tf.function(jit_compile=True)
@tf.function
def train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, opt, nu):
    x_u = X_u_train_tf
    x_f = X_f_train_tf
    with tf.GradientTape() as tape:
        tape.watch([W,b])
        u_nn = net_u(x_u, W, b)
        f_nn = net_f(x_f,W, b, nu)
        loss =  tf.reduce_mean(tf.square(u_nn - u_train_tf)) + tf.reduce_mean(tf.square(f_nn))
    grads = tape.gradient(loss, train_vars(W,b))
    opt.apply_gradients(zip(grads, train_vars(W,b)))
    return loss

def predict(X_star_tf, w, b):
    u_pred = net_u(X_star_tf, w, b)
    return u_pred
```

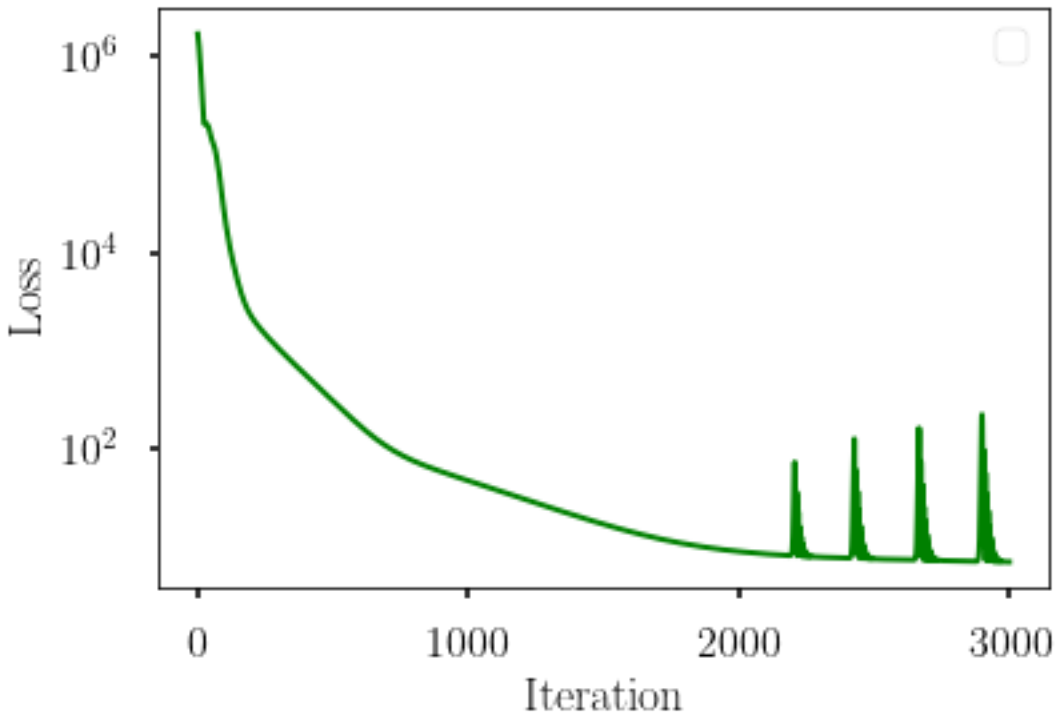
4. Driver

```
nu = 10**-3
noise = 0.0
N_f = 300
Nmax=3000

layers = [1, 4,4,4,4,4,4, 1]
L = len(layers)
W = [hyper_initial([layers[l-1], layers[l]]) for l in range(1, L)]
b = [tf.Variable(tf.zeros([1, layers[l]])) for l in range(1, L)]

x_0 = -1
x_1 = 1
u_0 = 1
u_1 = 0

X_u_train = np.vstack([x_0, x_1])
u_train = np.vstack([u_0, u_1])

X_f_train = lb + (ub-lb)*lhs(1, N_f)
X_f_star =  np.linspace(-1,1,200)
X_f_star = X_f_star.reshape((-1,1))


X_u_train_tf = tf.convert_to_tensor(X_u_train, dtype=tf.float32)
u_train_tf =   tf.convert_to_tensor(u_train, dtype=tf.float32)
X_f_train_tf = tf.convert_to_tensor(X_f_train, dtype=tf.float32)
X_star_tf = tf.convert_to_tensor(X_f_star, dtype=tf.float32)


lr = 5e-3
optimizer = tf.optimizers.Adam(learning_rate=lr)

start_time = time.time()
n=0
loss = []
while n <= Nmax:
    loss_ = train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, optimizer, nu)
    loss.append(loss_)
    print(f"Iteration is: {n} and loss is: {loss_}")
    n+=1

elapsed = time.time() - start_time
print('Training time: %.4f' % (elapsed))
```

Approach 1: PINNs with Soft Constraints

- Results





Approach 2: Self-Adaptive PINNs

- McClenny L, Braga-Neto U. Self-adaptive physics-informed neural networks using a soft attention mechanism. arXiv preprint arXiv:2009.04544. 2020.

PDE is Defined as

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

Loss function

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

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

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

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

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

Self-adaptive PINN utilizes the following loss function

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

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 selfadaptation weights for the initial, boundary, and collocation points, respectively, and

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

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

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

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

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

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

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

Approach 2: Implementation

- Code

1. Import Modules and Initial Setup

```
import sys
sys.path.insert(0, 'Utilities/')
import os

from scipy.interpolate import griddata
from pyDOE import lhs
from plotting import newfig, savefig
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.gridspec as gridspec
from mpl_toolkits.axes_grid1 import make_axes_locatable
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import time
import scipy.io

np.random.seed(seed=1234)
tf.random.set_seed(1234)
tf.config.experimental.enable_tensor_float_32_execution(False)
lb = -1
ub = 1
```

2. Routines for Initialization, Forward Pass and Residual computation

```
# Initalization of Network
def hyper_initial(size):
    in_dim = size[0]
    out_dim = size[1]
    std = np.sqrt(2.0/(in_dim + out_dim))
    return tf.Variable(tf.random.truncated_normal(shape=size, stddev = std))

# Neural Network
def DNN(X, W, b):
    A = 2.0*(X - lb)/(ub - lb) - 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_nn(W, b):
    return W + b

def train_vars_total(W, b, lambda_r, lambda_b):
    return W + b + lambda_r + lambda_b

def train_vars_sa(lambda_r, lambda_b):
    return lambda_r + lambda_b

def net_u(x,w, b):
    u = DNN(x, w, b)
    return u
```

Self Adaptive Weight

```
def loss_weight(N_r, N_b):
    alpha_b = tf.Variable(tf.reshape(tf.repeat(1000.0, N_b), (N_b, -1)))
    alpha_r = tf.Variable(tf.ones(shape=[N_r, 1]), dtype=tf.float32)
    return alpha_r, alpha_b
```

McClenny L, Braga-Neto U. Self-adaptive physics-informed neural networks using a soft attention mechanism. arXiv preprint arXiv:2009.04544. 2020 Sep 7.

Approach 2: Implementation

3. Min-Max Training

```
#@tf.function(jit_compile=True)
@tf.function
def train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, opt, nu, lambda_r,
    x_u = X_u_train_tf
    x_f = X_f_train_tf
    with tf.GradientTape(persistent=True) as tape:
        tape.watch([W,b,lambda_r,lambda_b])
        u_nn = net_u(x_u, W, b)
        f_nn = net_f(x_f,W, b, nu)
        loss_r = tf.square(lambda_r*f_nn)
        loss_b = tf.square(lambda_b*(u_nn-u_train_tf))
        loss =    tf.reduce_mean(loss_b) + tf.reduce_mean(loss_r)
    grads = tape.gradient(loss, train_vars_nn(W, b))
    grads_u = tape.gradient(loss, lambda_r)
    grads_b = tape.gradient(loss, lambda_b)
    opt.apply_gradients(zip(grads, train_vars_nn(W,b)))
    opt.apply_gradients(zip([-grads_u], [lambda_r]))
    opt.apply_gradients(zip([-grads_b], [lambda_b]))
    return loss

def predict(X_star_tf, w, b):
    u_pred = net_u(X_star_tf, w, b)
    return u_pred
```

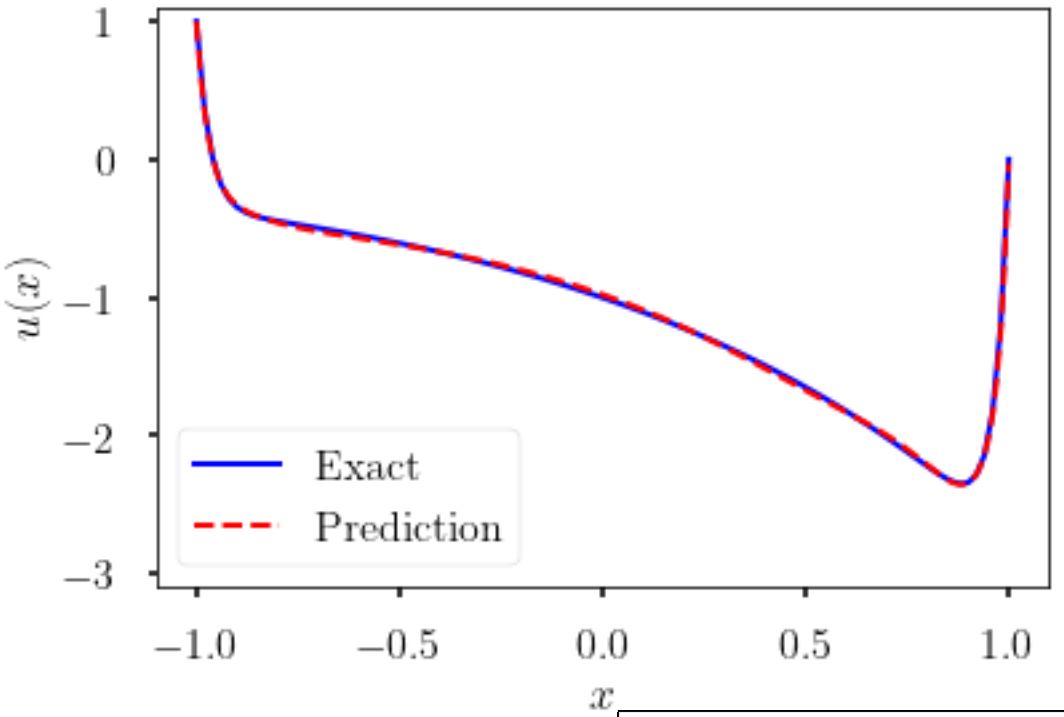
Maximize Loss with Self Adaptive Weight

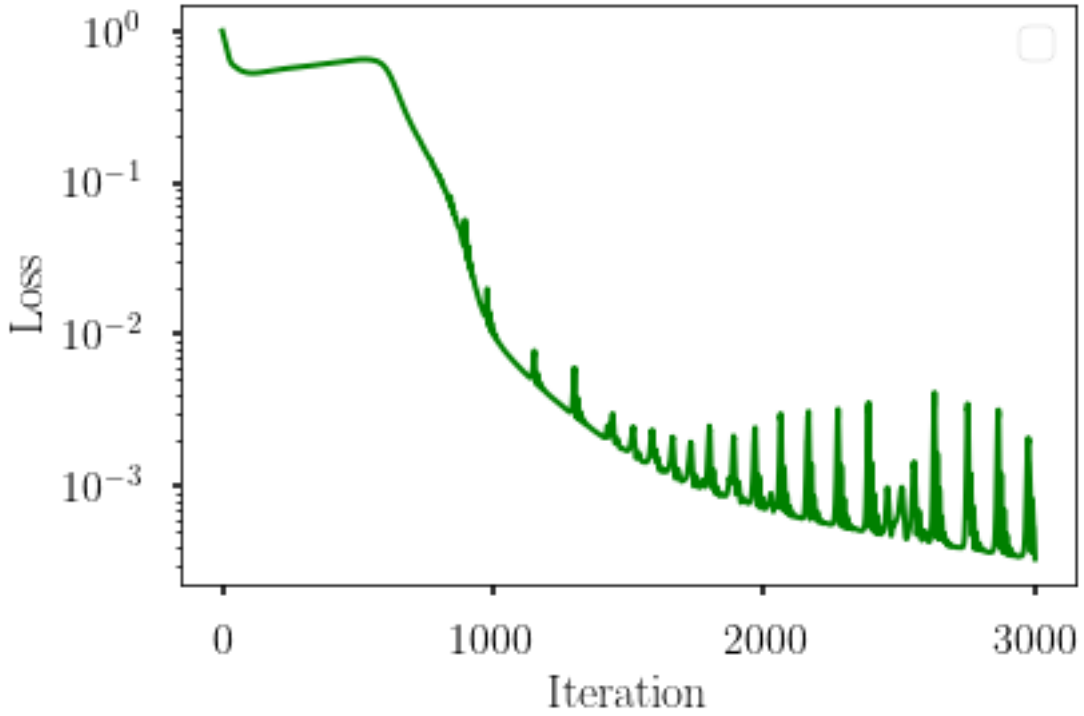
4. Driver

```
nu = 10**-3
Nmax= 1500
N_f = 500
N_b = 2

layers = [1, 8,8,8,8,8,8, 1]
L = len(layers)
W = [hyper_initial([layers[l-1], layers[l]]) for l in range(1, L)]
b = [tf.Variable(tf.zeros([1, layers[l]])) for l in range(1, L)]

alpha_r, alpha_b = loss_weight(N_f, N_b)

x_0 = -1
x_1 = 1
u_0 = 1
u_1 = 0

X_u_train = np.vstack([x_0, x_1])
u_train = np.vstack([u_0, u_1])

X_f_train = lb + (ub-lb)*lhs(1, N_f)
X_f_star =  np.linspace(-1,1,200)
X_f_star = X_f_star.reshape((-1,1))


X_u_train_tf = tf.convert_to_tensor(X_u_train, dtype=tf.float32)
u_train_tf =   tf.convert_to_tensor(u_train, dtype=tf.float32)
X_f_train_tf = tf.convert_to_tensor(X_f_train, dtype=tf.float32)
X_star_tf = tf.convert_to_tensor(X_f_star, dtype=tf.float32)


lr = 1e-3
optimizer = tf.optimizers.Adam(learning_rate=lr)

start_time = time.time()
n=0
loss = []
while n <= Nmax:
    loss_= train_step(W, b, X_u_train_tf, u_train_tf, X_f_train_tf, optimizer, nu, alpha_r, alpha_b)
    loss.append(loss_)
    print(f"Iteration is: {n} and loss is: {loss_}")
    n+=1

elapsed = time.time() - start_time
print('Training time: %.4f' % (elapsed))
```

McClenny L, Braga-Neto U. Self-adaptive physics-informed neural networks using a soft attention mechanism. arXiv preprint arXiv:2009.04544. 2020 Sep 7.

Approach 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

Approach 3: Dynamic Weights for PINNs

- Wang S, Yu X, Perdikaris P. When and why PINNs fail to train: A neural tangent kernel perspective. Journal of Computational Physics. 2022 Jan 15;449:110768.

- Loss = Boundary Losses + Residual Loses

$$\mathcal{L} = \lambda_B \mathcal{L}_B + \lambda_f \mathcal{L}_f$$

- Parameters update

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla_\theta \mathcal{L}_f - \eta \alpha \nabla_\theta \mathcal{L}_B,$$

where $\alpha^{(k+1)} = (1-\lambda)\alpha^{(k)} + \lambda \hat{\alpha}^{(k+1)}$, with $\hat{\alpha}^{(k+1)} = \dfrac{\overline{|\nabla_\theta \mathcal{L}_f|}}{\overline{|\nabla_\theta \mathcal{L}_B|}}$,

- $\overline{|\nabla_\theta \mathcal{L}_B|}$ denote the means of $|\nabla_\theta \mathcal{L}_B|$.
- $0 \leq \lambda \leq 1$.

Jin X, Cai S, Li H, Karniadakis GE. NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations. Journal of Computational Physics. 2021 Feb 1;426:109951.

Approach 3: Implementation

- Code

1. Import Modules and Initial Setup

```
import sys
sys.path.insert(0, 'Utilities/')
import os

from scipy.interpolate import griddata
from pyDOE import lhs
from plotting import newfig, savefig
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.gridspec as gridspec
from mpl_toolkits.axes_grid1 import make_axes_locatable
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import time
import scipy.io
```

2. Routines for Initialization, Forward Pass and Residual computation

```
np.random.seed(seed=1234)
tf.random.set_seed(1234)
tf.config.experimental.enable_tensor_float_32_execution(False)
#os.environ['TF_ENABLE_AUTO_MIXED_PRECISION'] = '1'

# Initalization of Network
def hyper_initial(size):
    in_dim = size[0]
    out_dim = size[1]
    std = np.sqrt(2.0/(in_dim + out_dim))
    return tf.Variable(tf.random.truncated_normal(shape=size, stddev = std))

# Neural Network
def DNN(X, W, b):
    A = X
    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(W, b):
    return W + b

def net_u(x, t, w, b):
    u = DNN(tf.concat([x,t],1), w, b)
    return u


#@tf.function(jit_compile=True)
@tf.function
def net_f(x,t,W, b, nu):
    with tf.GradientTape(persistent=True) as tape1:
        tape1.watch([x, t])
        with tf.GradientTape(persistent=True) as tape2:
            tape2.watch([x, t])
            u=net_u(x,t, W, b)
        u_t = tape2.gradient(u, t)
        u_x = tape2.gradient(u, x)
    u_xx = tape1.gradient(u_x, x)
    del tape1
    f = u_t + u*u_x - nu*u_xx
    return f

```

Approach 3: Implementation

3. Training and Dynamic Weights Implementation

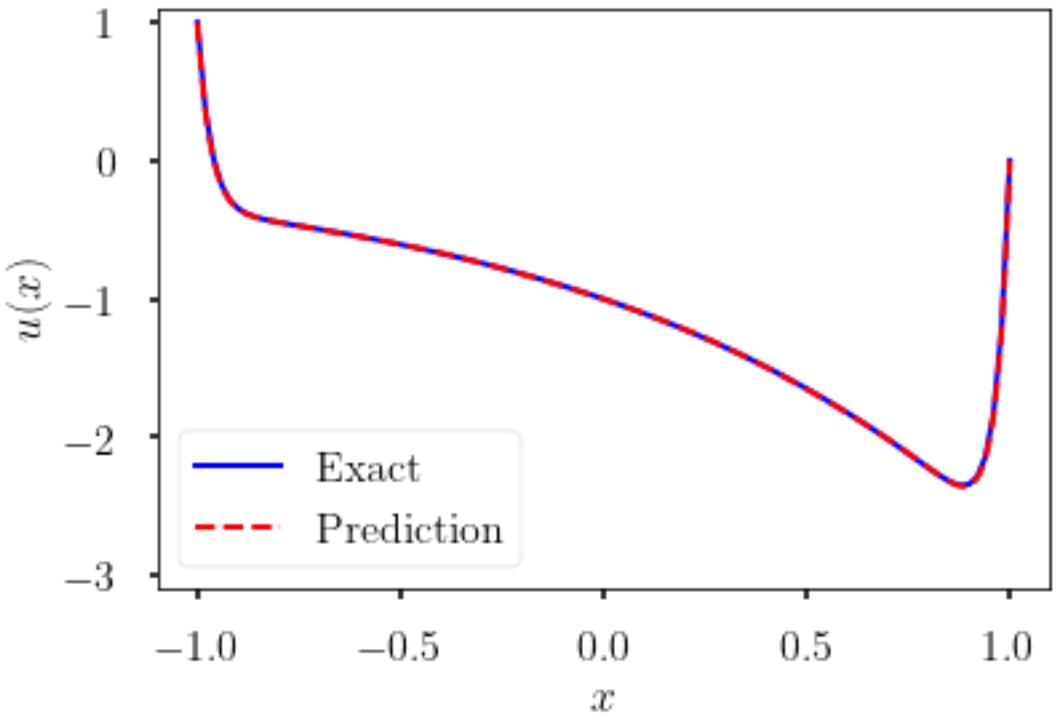
```
@tf.function
def train_step(layers, W, b, X_u_train_tf, u_train_tf, X_f_train_tf, opt, nu, lambda_b, beta):
    x_u = X_u_train_tf[:,0:1]
    x_f = X_f_train_tf[:,0:1]
    adpative_constant_bcs_list = []
    lambda_b_list = []
    with tf.GradientTape(persistent=True) as tape:
        tape.watch([W,b])
        u_nn = net_u(x_u,W, b)
        f_nn = net_f(x_f,W, b, nu)
        bc_loss =  tf.reduce_mean(tf.square(u_nn - u_train_tf))
        phys_loss = tf.reduce_mean(tf.square(f_nn))
        loss = lambda_b[-1]*bc_loss  + phys_loss
        loss = loss / (1 + lambda_b[-1])
        p_loss = phys_loss/(1 + lambda_b[-1])
        b_loss = lambda_b[-1]*bc_loss/(1 + lambda_b[-1])
    grad_loss = tape.gradient(loss, train_vars(W,b))
    opt.apply_gradients(zip(grad_loss, train_vars(W,b)))
    grads_bc = tape.gradient(bc_loss, train_vars(W,b))
    grads_phys = tape.gradient(phys_loss, train_vars(W,b))
    for i in range(len(layers) - 1):
        adpative_constant_bcs_list.append(
                tf.reduce_mean(tf.abs(grads_phys[i])) / tf.reduce_mean(tf.abs(grads_bc[i])))

    lambda_b_new = tf.reduce_mean(tf.stack(adpative_constant_bcs_list))
    lambda_b_new = (1-beta)*lambda_b[-1] + beta*lambda_b_new
    lambda_b_list.append(lambda_b_new)

    return loss, lambda_b_list, p_loss, b_loss
```

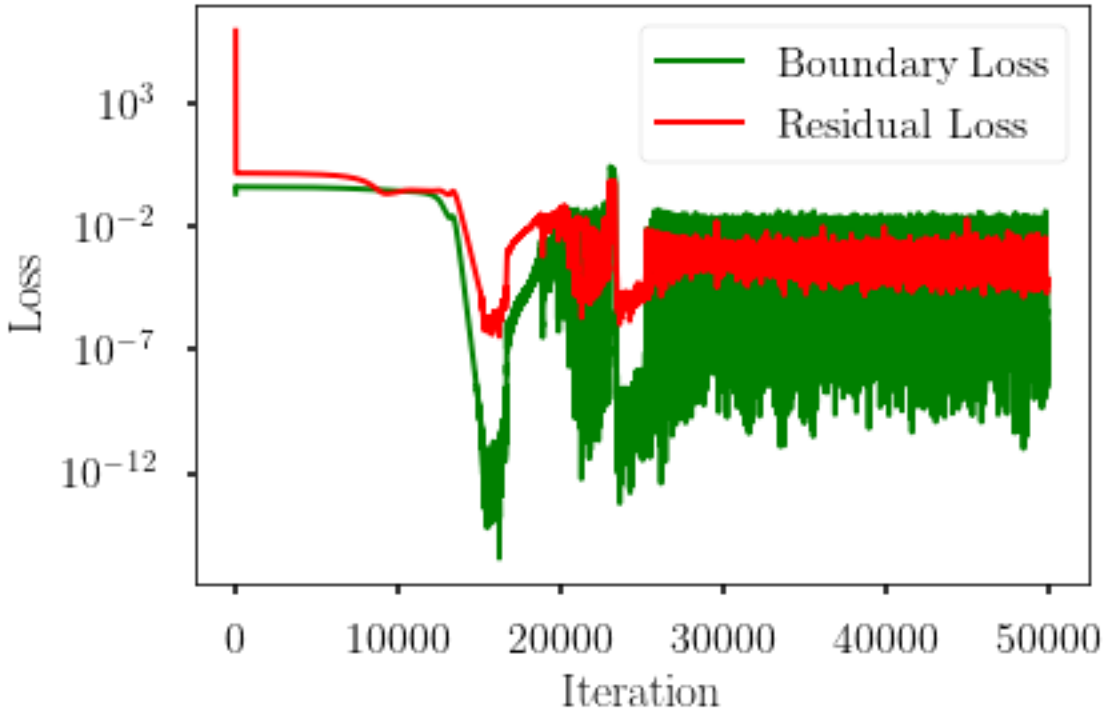
Dynamic weights

4. Driver

```
nu = 10**-3
noise = 0.0
N_f = 5000
Nmax=15000

layers = [1, 8,8,8,8,8,8, 1]
L = len(layers)
W = [hyper_initial([layers[l-1], layers[l]]) for l in range(1, L)]
b = [tf.Variable(tf.zeros([1, layers[l]])) for l in range(1, L)]

x_0 = -1
x_1 = 1
u_0 = 1
u_1 = 0

X_u_train = np.vstack([x_0, x_1])
u_train = np.vstack([u_0, u_1])
X_f_train = lb + (ub-lb)*lhs(1, N_f)
X_f_star =  np.linspace(-1,1,200)
X_f_star = X_f_star.reshape((-1,1))
X_u_train_tf = tf.convert_to_tensor(X_u_train, dtype=tf.float32)
u_train_tf =   tf.convert_to_tensor(u_train, dtype=tf.float32)
X_f_train_tf = tf.convert_to_tensor(X_f_train, dtype=tf.float32)
X_star_tf = tf.convert_to_tensor(X_f_star, dtype=tf.float32)
lam_b = np.array([1.0])
lam_b_tf= tf.convert_to_tensor(lam_b, dtype=tf.float32)
lambda_b_list = [lam_b_tf]
lr = 1e-3
optimizer = tf.optimizers.Adam(learning_rate=lr)
start_time = time.time()
n=0
loss = []
bc_loss = []
phys_loss = []
beta = 0.1

while n <= Nmax:
    loss_, lambda_b_out, phys_loss_, bc_loss_ = train_step(layers, W, b, X_u_train_tf,\
                                                           u_train_tf, X_f_train_tf, optimizer, nu,lambda_b_list, beta)
    lambda_b_list = lambda_b_out
    loss.append(loss_)
    bc_loss.append(bc_loss_)
    phys_loss.append(phys_loss_)
```

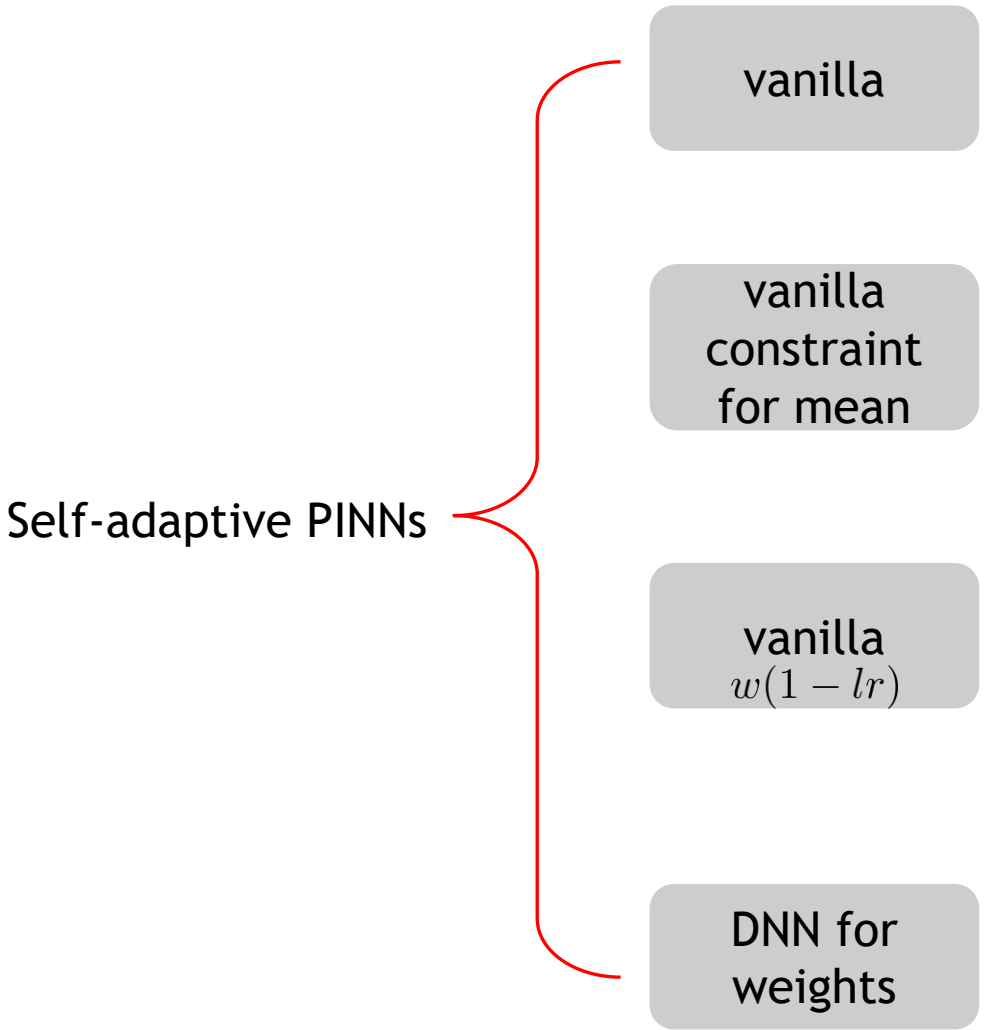
Approach 3: Results





Adaptive weights PINN Architecture	
Hyper-parameters	**Value**
No. of Layers	6
Neurons per layer	8
No. of Iterations	5000
Learning rate	1×10^{-3}
No. of Residuals Points	500

Different Versions of Self-Adaptive Weights in PINNs



- weights increase at every step
- prevent weights from increasing at every step
- weights reach the equilibrium after a certain number of iterations
- reduce memory usage

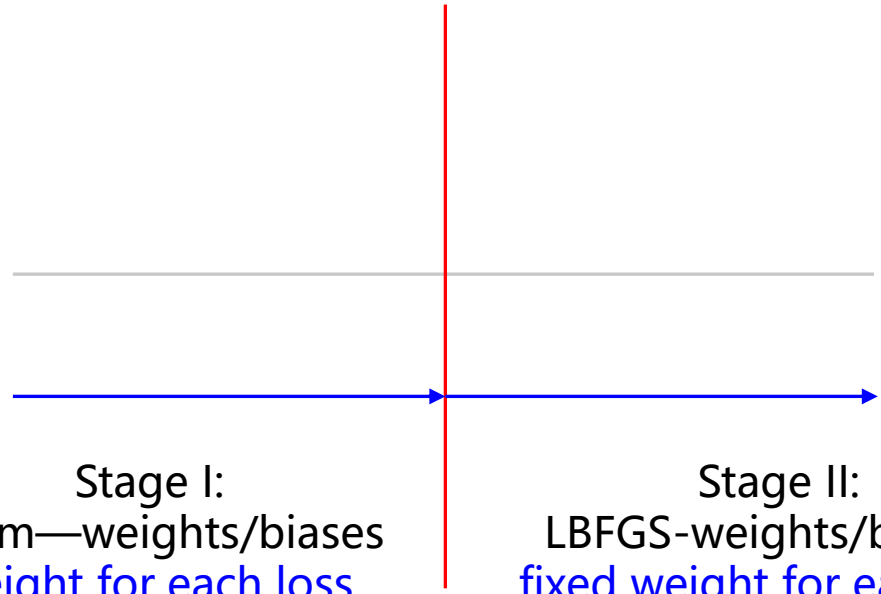
Training

Self-adaptive PINNs

- vanilla
- vanilla constraint for mean
- vanilla $w(1-lr)$
- DNN for weights

I

II



Self-Adaptive Weights: Alan-Cahn Equation

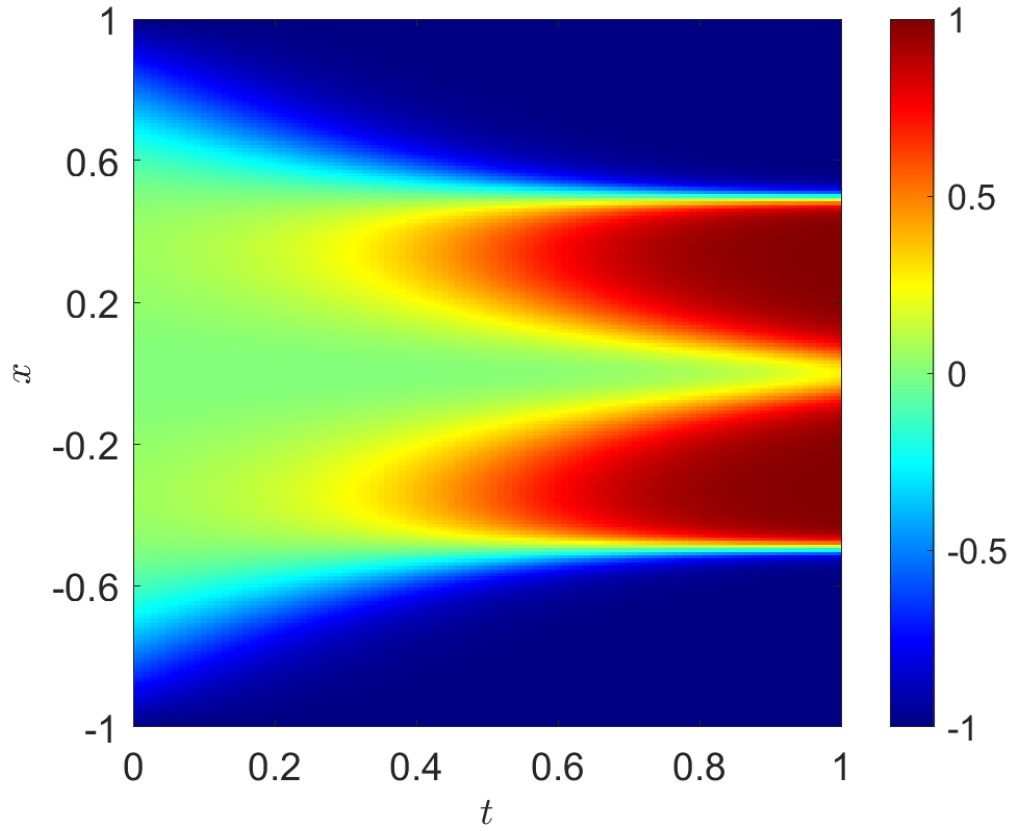
- Alan-Cahn Equation

$$\partial_t u - D\partial_x^2 u + 5\left(u^3 - u\right) = 0, D = 0.0001$$
$$t \in [0,1], x \in [-1,1]$$

- Initial and Boundary condition

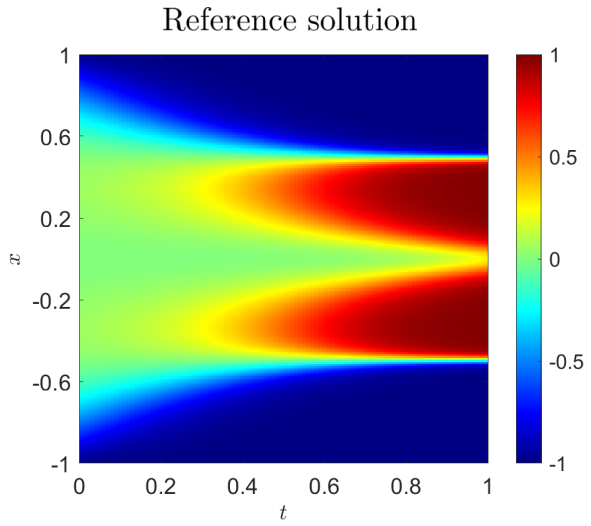
$$u(x,0) = x^2\cos(\pi x)(IC)$$
$$u(-1,t) = u(1,t) = -1(BCs)$$

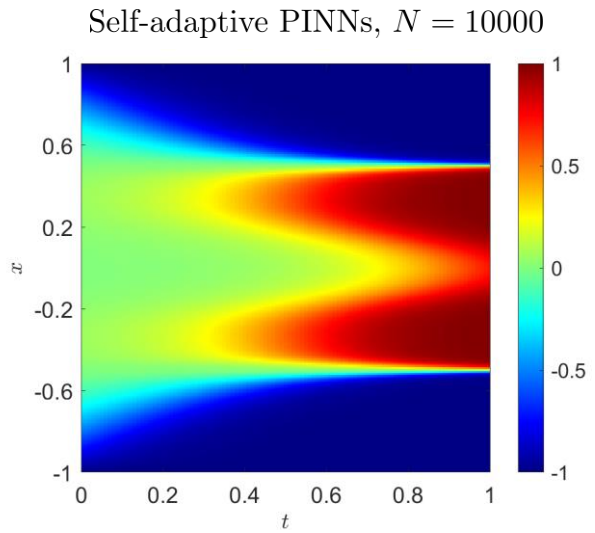
- DNN: 4 hidden layers with 128 neurons per layer
- 20,000 random points for residual
- 200 uniform points for initial conditions
- 100 uniform points for boundary conditions

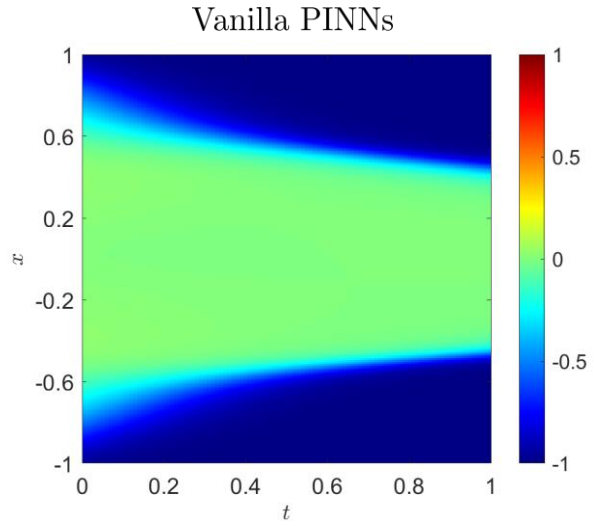


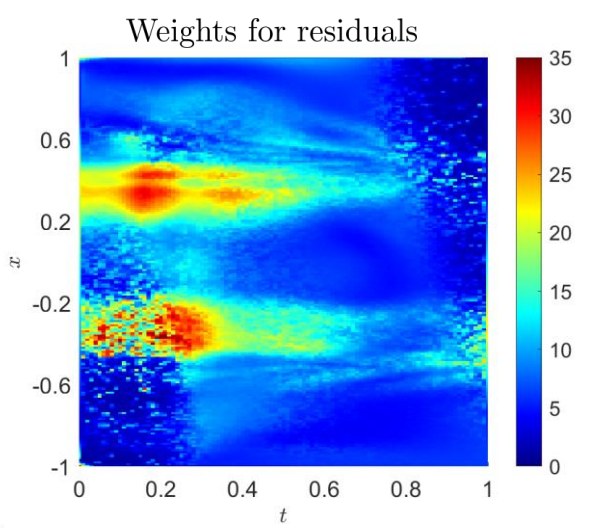
Reference Solution for Alan-Cahn Equation

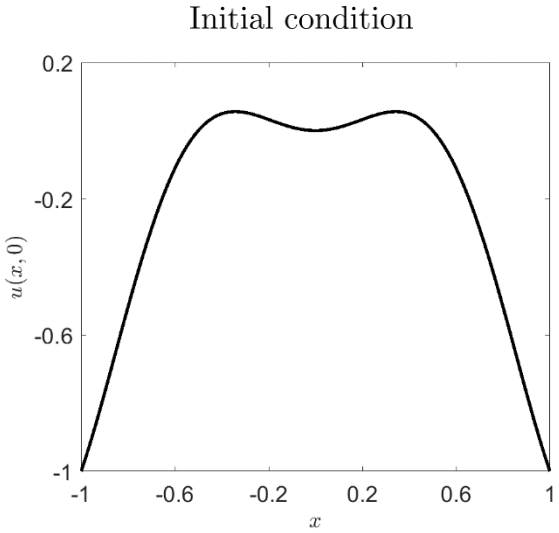
Self-Adaptive Weights: Alan-Cahn Equation (vanilla)

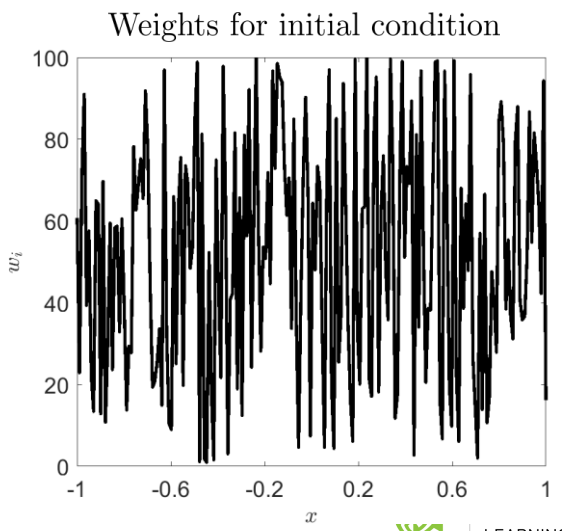




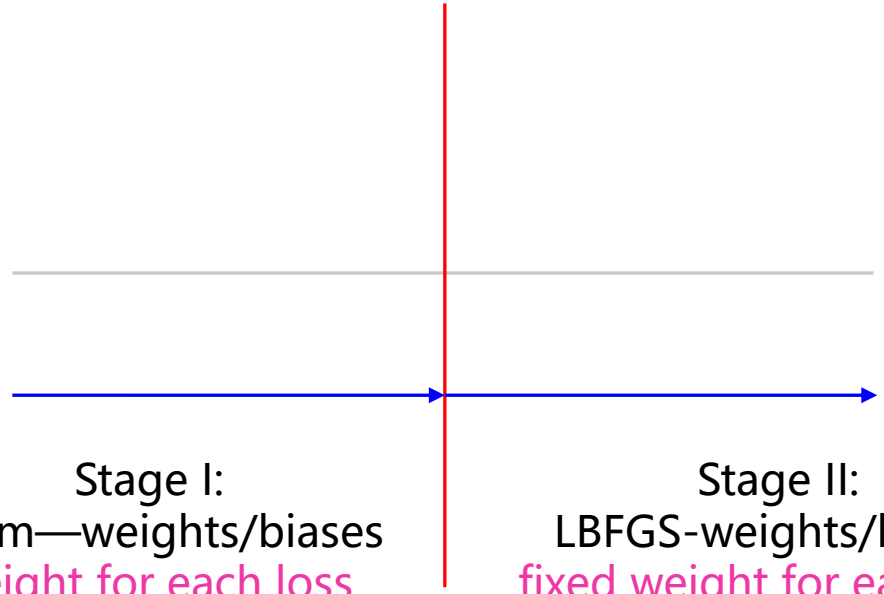








I. Variance for Losses

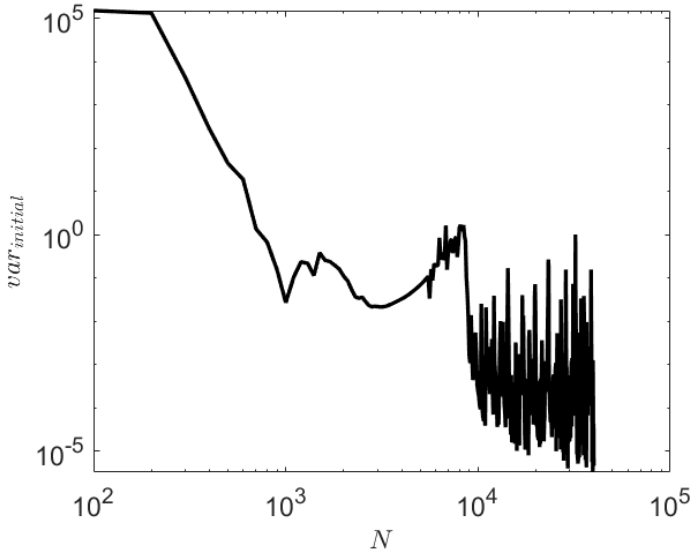


Stage I:
Adam—weights/biases
weight for each loss

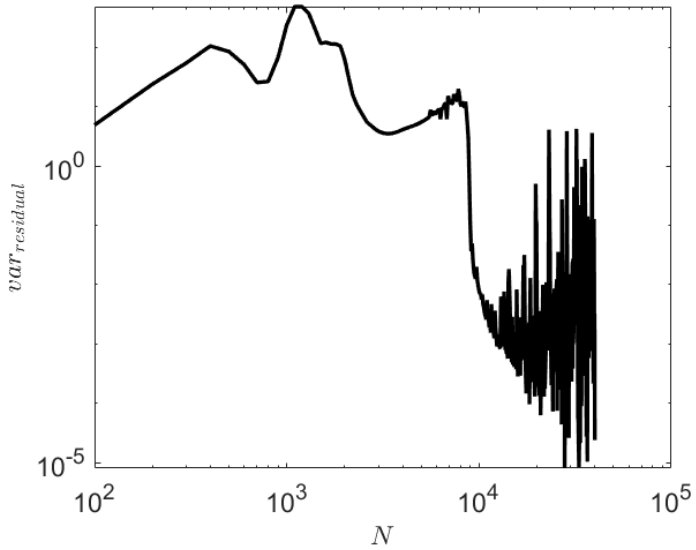
Stage II:
LBFGS-weights/biases
fixed weight for each loss

N

Variance of loss in initial condition

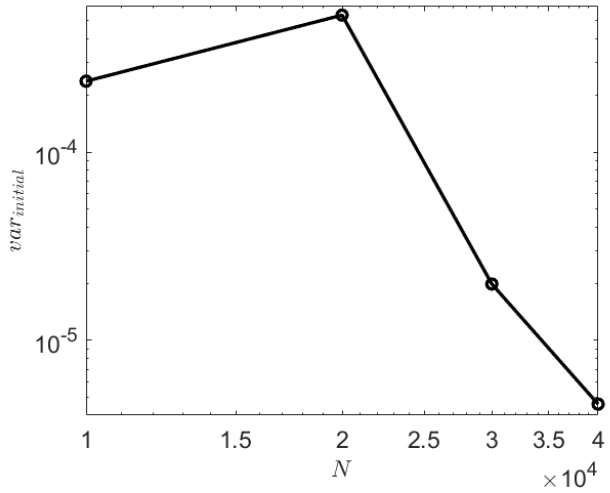


Variance of loss in residuals

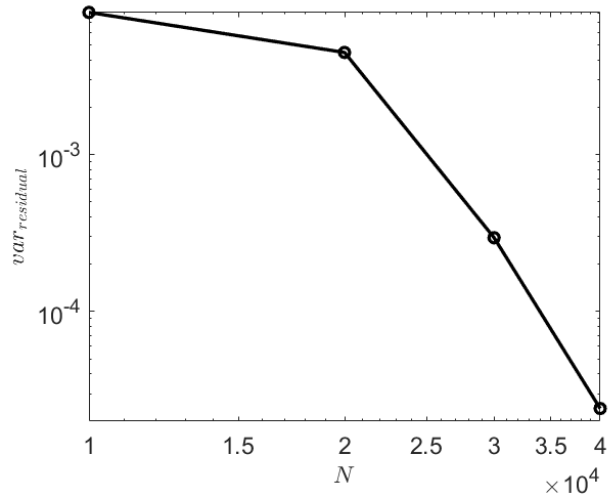


I. Variance for Losses

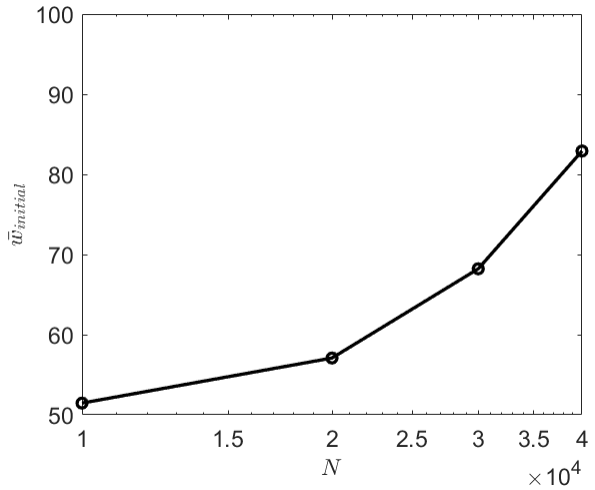
Variance of loss in initial condition



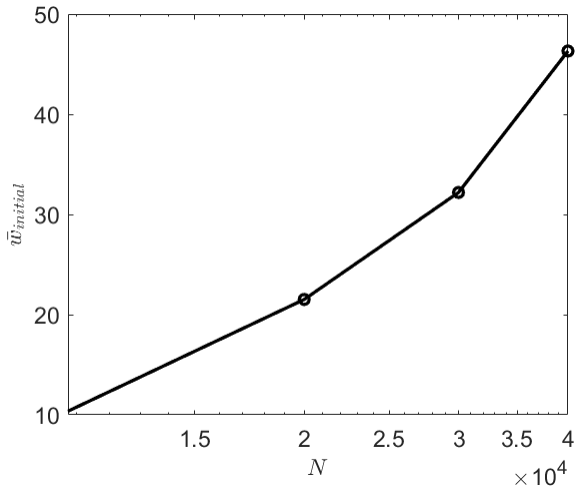
Variance of loss in residuals



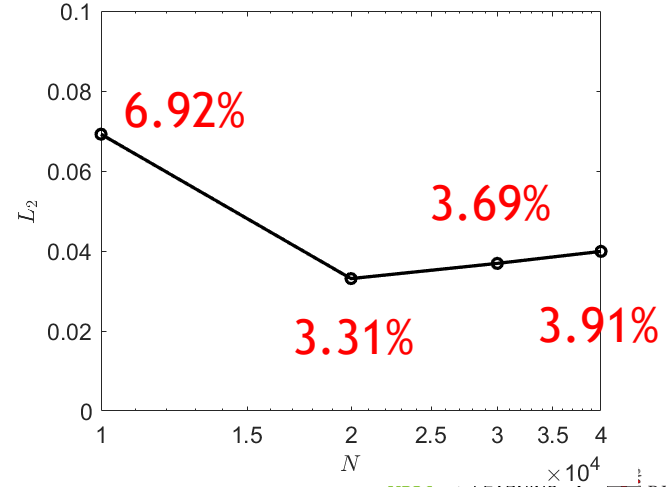
Average of self adaptive weight for initial conditions



Average of self adaptive weight for residual



L_2 Error between actual and predicted solution



II. Variance for Losses: $W' = W(1 - lr)$

Learning rates are important!

The key feature of self-adaptive PINNs is that the loss $\mathcal{L}(w, \boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0)$ is minimized with respect to the network weights w, as usual, but is maximized with respect to the self-adaptation weights $\boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0$; in other words, training seeks a saddle point

gradient ascent

$$\min_{\boldsymbol{w}} \max_{\boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0} \mathcal{L}(\boldsymbol{w}, \boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0)$$

This can be accomplished by a gradient descent/ascent procedure, with updates given by:

$$\begin{aligned}
\boldsymbol{w}^{k+1} &= \boldsymbol{w}^k - \eta_k \nabla_w \mathcal{L}\left(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k\right) \\
\boldsymbol{\lambda}_r^{k+1} &= \boldsymbol{\lambda}_r^k + \eta_k \nabla_{\boldsymbol{\lambda}_r} \mathcal{L}\left(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k\right) \\
\boldsymbol{\lambda}_b^{k+1} &= \boldsymbol{\lambda}_b^k + \eta_k \nabla_{\boldsymbol{\lambda}_b} \mathcal{L}\left(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k\right) \\
\boldsymbol{\lambda}_0^{k+1} &= \boldsymbol{\lambda}_0^k + \eta_k \nabla_{\boldsymbol{\lambda}_0} \mathcal{L}\left(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k\right)
\end{aligned}$$

$$\begin{aligned}
&> \lambda^{k+1} = \lambda^k + \eta_k \nabla_{\lambda_k} L, \qquad \nabla_{\lambda_k} L > 0 \\
&> \lambda^{k+1} = \lambda^{k+1}(1 - \beta), \qquad \beta = lr
\end{aligned}$$

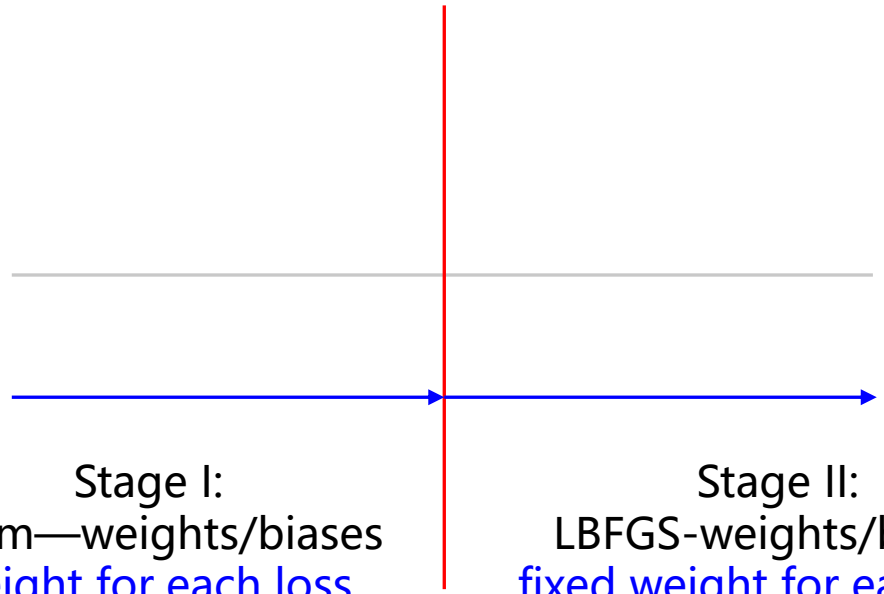
1. First increase λ, then decrease
2. Increment and decrement in λ is defined as

$$\Delta\lambda_+^{k+1} = \eta_k \nabla_{\lambda_k} L, \qquad \Delta\lambda_-^{k+1} = -\beta\lambda^{k+1}$$

3. Combine λ

$$\Delta\lambda^{k+1} = \Delta\lambda_+^{k+1} + \Delta\lambda_-^{k+1} = \eta_k \nabla_{\lambda_k} L - \beta\lambda^{k+1} \quad \begin{cases} \eta_k \nabla_{\lambda_k} L \uparrow, \lambda^{k+1} \uparrow \\ \eta_k \nabla_{\lambda_k} L \downarrow, \lambda^{k+1} \downarrow \end{cases}$$

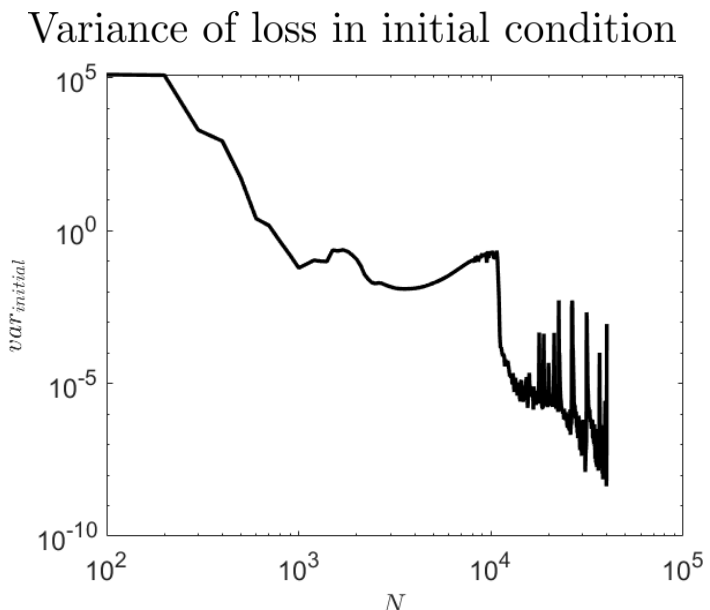
II. Variance for Losses: $\boldsymbol{W'} = \boldsymbol{W}(1 - lr)$

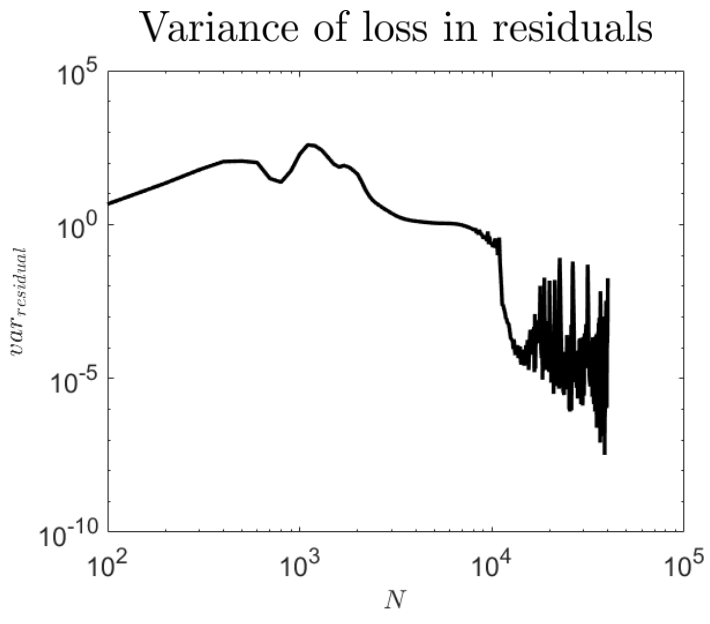


Stage I:
Adam—weights/biases
weight for each loss

Stage II:
LBFGS-weights/biases
fixed weight for each loss

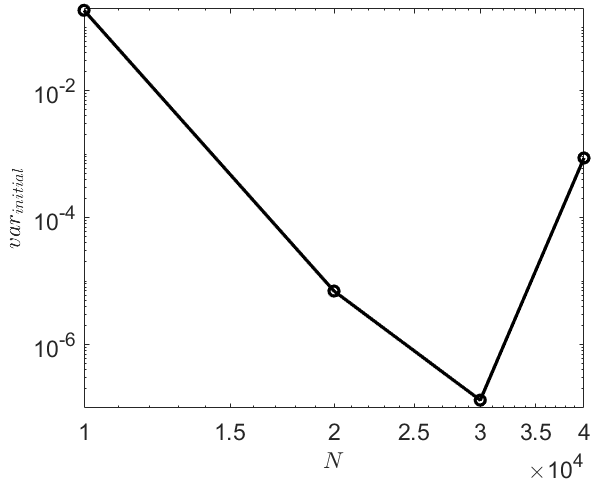
$$> \lambda^{k+1} = \lambda^k + \eta_k \nabla_{\lambda_k} L, \qquad \nabla_{\lambda_k} L > 0$$
$$> \lambda^{k+1} = \lambda^{k+1}(1 - \beta), \qquad \beta = 10^{-4}$$



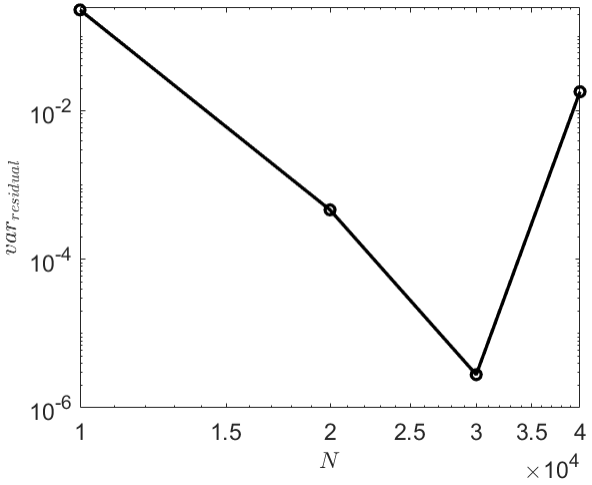


II. Variance for Losses: $W' = W(1 - lr)$

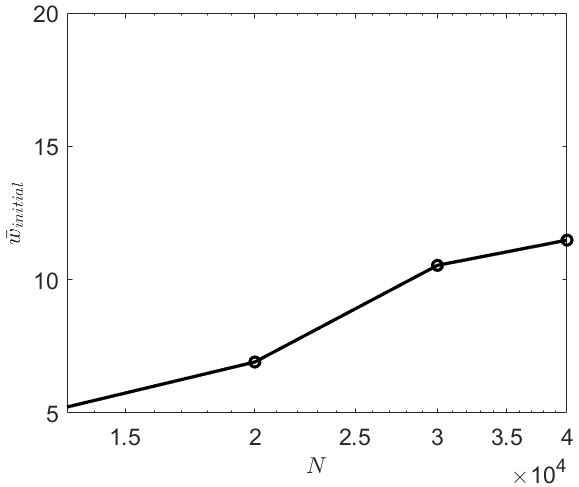
Variance of loss in initial condition



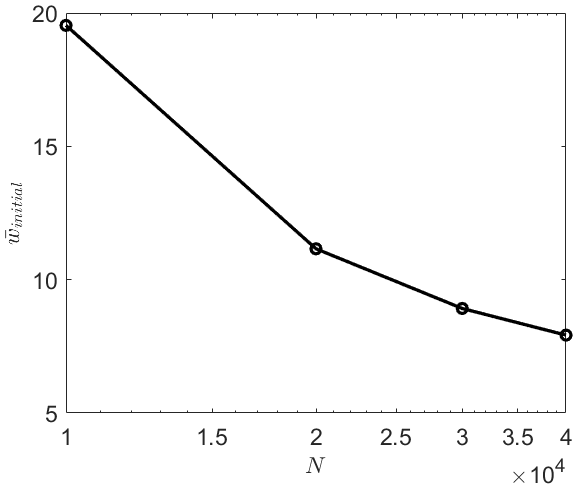
Variance of loss in residuals



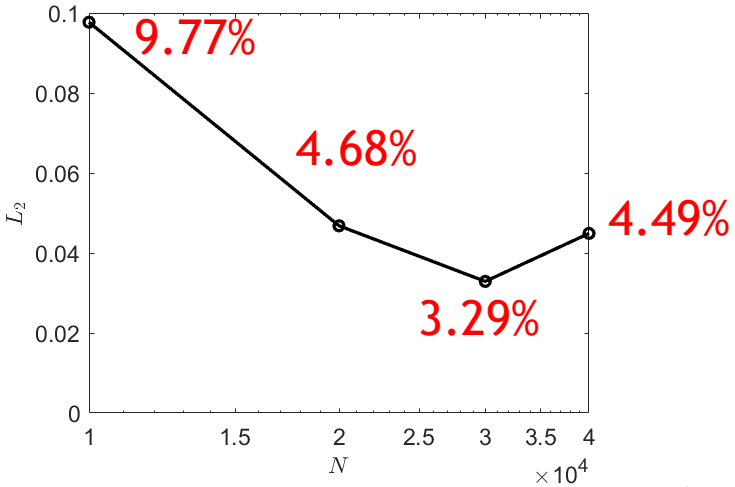
Average of self adaptive weight for initial conditions



Average of self adaptive weight for residual



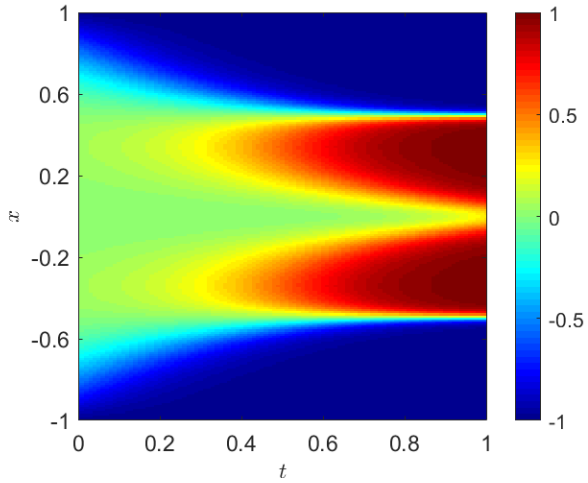
L_2 Error between actual and predicted solution



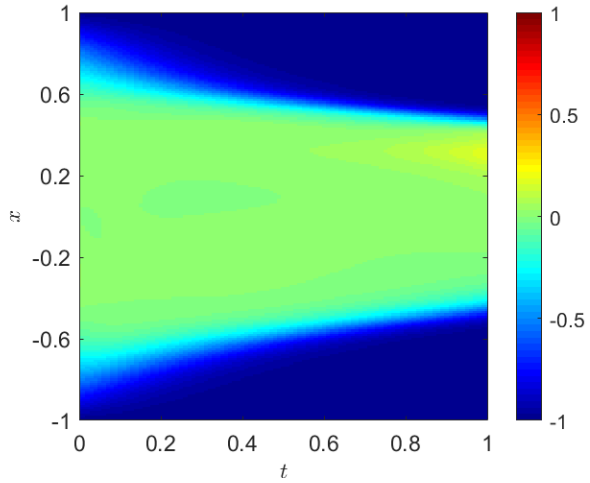
II. Variance for Losses: $W' = W(1 - lr)$ Effect of Learning Rate

$$> \lambda^{k+1} = \lambda^k + \eta_k \nabla_{\lambda_k} L, \qquad \nabla_{\lambda_k} L > 0$$
$$> \lambda^{k+1} = \lambda^{k+1}(1 - \beta), \qquad \beta = 10^{-3}$$

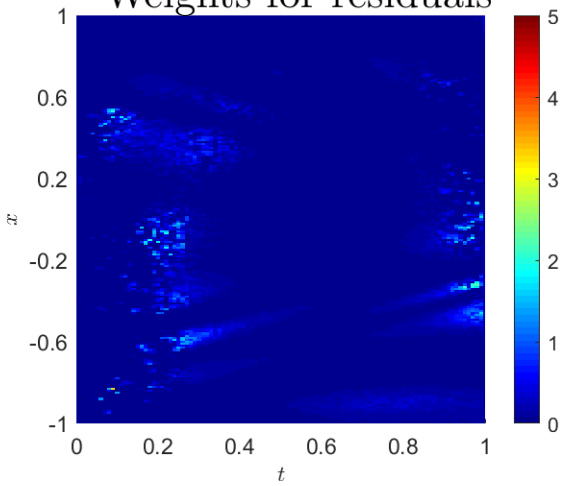
Reference solution



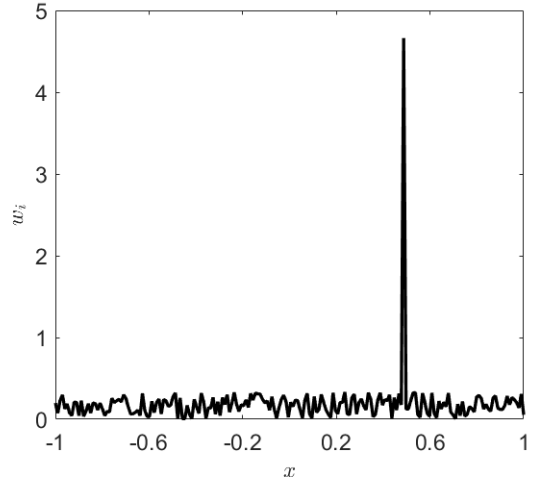
Self-adaptive PINNs (N=10000)



Weights for residuals



Weights for initial condition



Hard Constraints: Dirichlet BC and IC

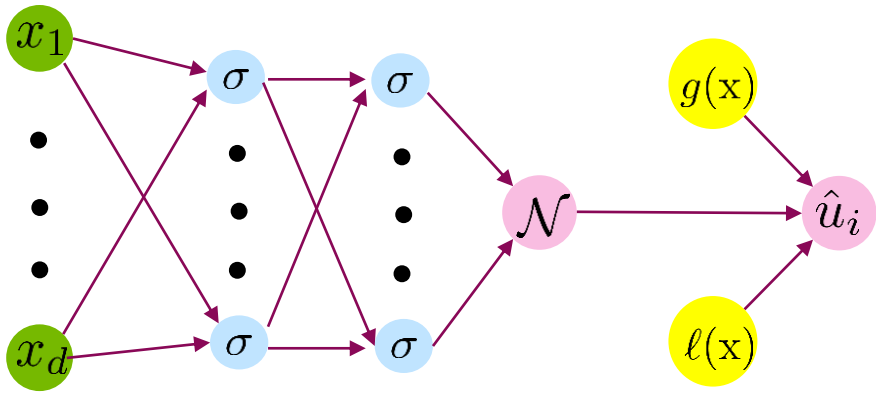
Dirichlet BC or IC : $u_i(\mathbf{x}) = g_0(\mathbf{x}), \quad \mathbf{x} \in \Gamma_D,$

Trial solution: $\hat{u}_i(\mathbf{x}; \boldsymbol{\theta}_u) = g(\mathbf{x}) + \ell(\mathbf{x})\mathcal{N}(\mathbf{x}; \boldsymbol{\theta}_u),$

$g(x)$ is a continuous extension of $g_0(x)$ from Γ_D to Ω where $\mathcal{N}(\mathbf{x}; \boldsymbol{\theta}_u)$ is the network output, and ℓ is a function satisfying the following two conditions:

$$\begin{cases} \ell(\mathbf{x}) = 0, & \mathbf{x} \in \Gamma_D \\ \ell(\mathbf{x}) > 0, & \mathbf{x} \in \Omega - \Gamma_D \end{cases}$$

For example: $u(0) = 0, u(1) = 1 : g(x) = x, \ell(x) = x(1-x)$



[1] P. L. Lagaris, L. H. Tsoukalas, S. Safarkhani, and I. E. Lagaris, Systematic construction ofneural forms for solving partial differential equations inside rectangular domains, subject to initial, boundary and interface conditions, Int. J. Artif. Intell., 29 (2020), 2050009.
[2] L. Lu, R. Pestourie, W. Yao, Z. Wang, F. Verdugo, & S. G. Johnson. Physics-informed neural networks with hard constraints for inverse design. SIAM Journal on Scientific Computing, 43(6), B1105–B1132, 2021.

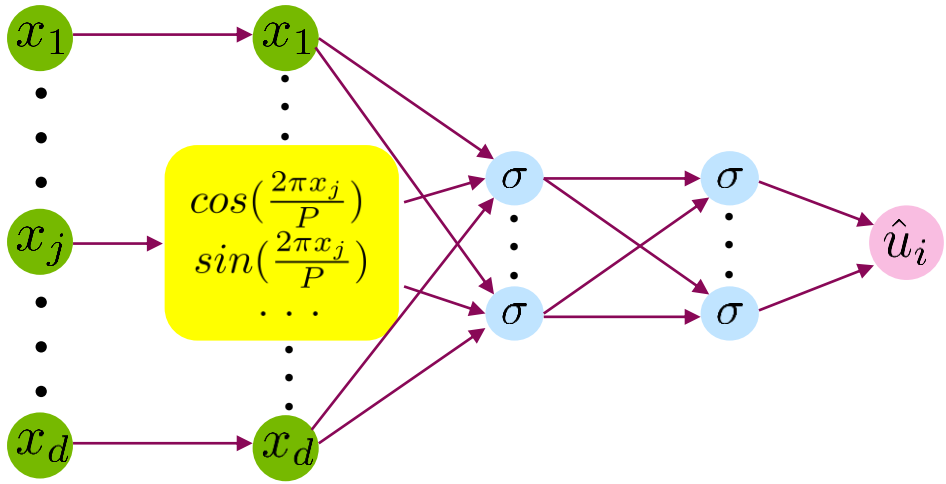
Hard Constraints: Periodic BC and IC

$u_i(x)$ is a periodic function with respect to x_j of the period P.
$\Rightarrow u_i(x)$ can be decomposed into weighted summation of the Fourier series:

$$\{1, \cos(\frac{2\pi x_j}{P}), \sin(\frac{2\pi x_j}{P}), \cos(\frac{4\pi x_j}{P}), \sin(\frac{4\pi x_j}{P}) \cdots\}$$

$\Rightarrow$ replace x_j with Fourier basis functions to impose periodicity.

$$u_i(x) = \mathcal{N}\left(x_1, \cdots, x_{j-1}\left[1, \cos(\frac{2\pi x_j}{P}), \sin(\frac{2\pi x_j}{P}), \cos(\frac{4\pi x_j}{P}), \sin(\frac{4\pi x_j}{P}), \cdots\right], x_{j+1}, \cdots, x_d\right)$$



can use as few terms as $\{\cos(\frac{2\pi x_j}{P}), sin(\frac{2\pi x_j}{P})\}$

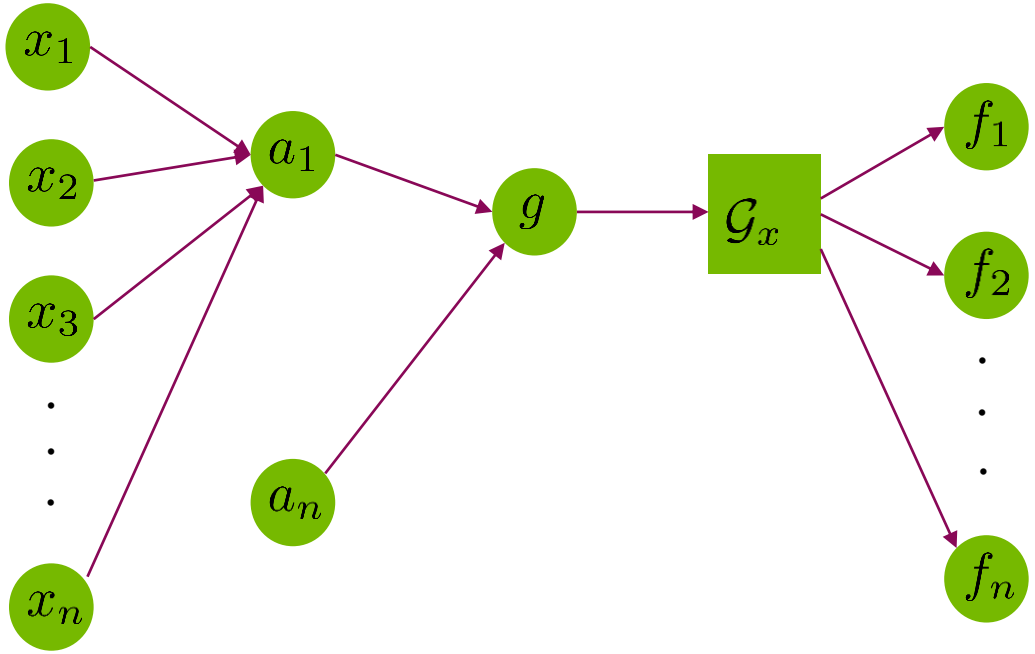
Linearly Constrained Neural Networks

For example: divergence-free neural networks

$$\nabla \cdot \mathbf{f} = \frac{\partial f_1}{\partial x} + \frac{\partial f_2}{\partial y} = 0$$

Divergence of the curl is 0 : $\nabla \cdot (\nabla \times g) \equiv 0$, so choose

$$\mathbf{f} = \nabla \times g$$



Hendriks J, Jidling C, Wills A, Schön T. Linearly constrained neural networks. arXiv preprint arXiv:2002.01600. 2020 Feb 5.

Linearly Constrained Neural Networks: Implementation

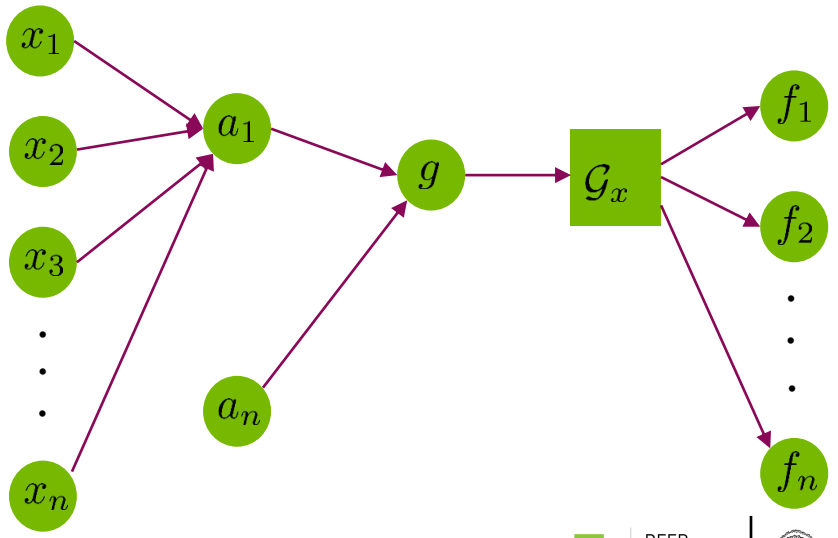
- Problem definition

$$f_1(x_1, x_2) = \exp(-ax_1x_2)(ax_1 \sin(x_1x_2) - x_1 \cos(x_1x_2)),$$
$$f_2(x_1, x_2) = \exp(-ax_1x_2)(x_2 \cos(x_1x_2) - ax_2 \sin(x_1x_2)),$$

  where a is a constant.

- This vector field satisfies the constraint $\frac{\partial f_1}{\partial x_1} + \frac{\partial f_2}{\partial x_2} = 0$.
- A neural network based model satisfying these constraints is given by

$$\mathbf{f} = \begin{bmatrix} \frac{\partial}{\partial x_2} \\ -\frac{\partial}{\partial x_1} \end{bmatrix} g$$



Linearly Constrained Neural Networks: Code

1. Import Modules and Initial Setup

```
import torch
from matplotlib import pyplot as plt
from torch.utils import data
import numpy as np
import torch.autograd as ag
import torch.nn as nn
torch.manual_seed(1234)
epochs = 400
n_data = 200
### Input Data
def vector_field(x, y, a=0.01):
    v1 = torch.exp(-a*x*y)*(a*x*torch.sin(x*y) - x*torch.cos(x*y))
    v2 = torch.exp(-a*x*y)*(y*torch.cos(x*y) - a*y*torch.sin(x*y))
    return (v1, v2)


## ------------------ set up models-------------------------- ##
# set network size
n_in = 2
n_h1 = 100
n_h2 = 50
n_o = 1

# two outputs for the unconstrained network
n_o_uc = 2
```

2. Model class for divergence free system

```
# define model class
class DivFree2D(torch.nn.Module):
    def __init__(self, base_net):
        super(DivFree2D, self).__init__()
        self.base_net = base_net

    def forward(self, x):
        x.requires_grad = True
        y = self.base_net(x)
        dydx = ag.grad(outputs=y, inputs=x, create_graph=True, grad_outputs=torch.ones(y.size()),
                       retain_graph=True, only_inputs=True)[0]
        return y, dydx[:,1].unsqueeze(1), -dydx[:,0].unsqueeze(1)

model = DivFree2D(nn.Sequential(nn.Linear(n_in,n_h1),nn.Tanh(),nn.Linear(n_h1,n_h2),
                                nn.Tanh(),nn.Linear(n_h2,n_o)))
```

3. Unconstrained Model for Vanilla PINN

```
model_uc = torch.nn.Sequential(
    torch.nn.Linear(n_in, n_h1),
    torch.nn.Tanh(),
    torch.nn.Linear(n_h1, n_h2),
    torch.nn.Tanh(),
    torch.nn.Linear(n_h2, n_o_uc),
)
```

Linearly Constrained Neural Networks: Code

4. Training and Testing data Generation

```
# pregenerate validation data
x_val = 4.0 * torch.rand(2000, 2)
x1_val = x_val[:, 0].unsqueeze(1)
x2_val = x_val[:, 1].unsqueeze(1)

(v1, v2) = vector_field(x1_val, x2_val)
y1_val = v1 + 0.1 * torch.randn(x1_val.size())
y2_val = v2 + 0.1 * torch.randn(x1_val.size())
y_val = torch.cat((y1_val, y2_val), 1)


# Get the true function values on a grid
xv, yv = torch.meshgrid([torch.arange(0.0, 20.0) * 4.0 / 20.0, torch.arange(0.0, 20.0) * 4.0 / 20.0])
(v1, v2) = vector_field(xv, yv)

# generate training data
x_train = 4.0 * torch.rand(n_data, 2)
x1_train = x_train[:, 0].unsqueeze(1)
x2_train = x_train[:, 1].unsqueeze(1)

(v1_t, v2_t) = vector_field(x1_train, x2_train)
y1_train = v1_t + 0.1 * torch.randn(x1_train.size())
y2_train = v2_t + 0.1 * torch.randn(x1_train.size())
```

Linearly Constrained Neural Networks: Code

5. Data Class for Automatic Batching

```python
# define Dataset class
class Dataset(data.Dataset):
    'Characterizes a dataset for PyTorch'

    def __init__(self, x1, x2, y1, y2):
        'Initialization'
        self.x1 = x1
        self.x2 = x2
        self.y1 = y1
        self.y2 = y2

    def __len__(self):
        'Denotes the total number of samples'
        return len(self.x1)

    def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        x1 = self.x1[index]
        x2 = self.x2[index]
        y1 = self.y1[index]
        y2 = self.y2[index]

        return x1, x2, y1, y2

training_set = Dataset(x1_train, x2_train, y1_train, y2_train)

# data loader Parameters
DL_params = {'batch_size': 100,
             'shuffle': True,
             'num_workers': 0,
             'pin_memory': False}
training_generator = data.DataLoader(training_set, **DL_params)
```

6. Loss and LR scheduler

```python
# ---------------  Set up and train the constrained model -------
criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau\
                                        (optimizer, patience=10,
                                         min_lr=1e-10,
                                         factor=0.5, cooldown=15)
```

7. Train and Eval routine for Constrained Model

```python
def train(epoch):
    model.train()
    total_loss = 0
    n_batches = 0
    for x1_train, x2_train, y1_train, y2_train in training_generator:
        optimizer.zero_grad()
        x_train = torch.cat((x1_train, x2_train), 1)
        (yhat, v1hat, v2hat) = model(x_train)
        loss = (criterion(y1_train, v1hat) + criterion(y2_train, v2hat)) / 2
        loss.backward()
        optimizer.step()
        total_loss += loss
        n_batches += 1
    return total_loss / n_batches

def eval(epoch):
    model.eval()
    # with torch.no_grad():
    (yhat, v1hat, v2hat) = model(x_val)
    loss = (criterion(y1_val, v1hat) + criterion(y2_val, v2hat)) / 2
    return loss.cpu()
```

Linearly Constrained Neural Networks: Code

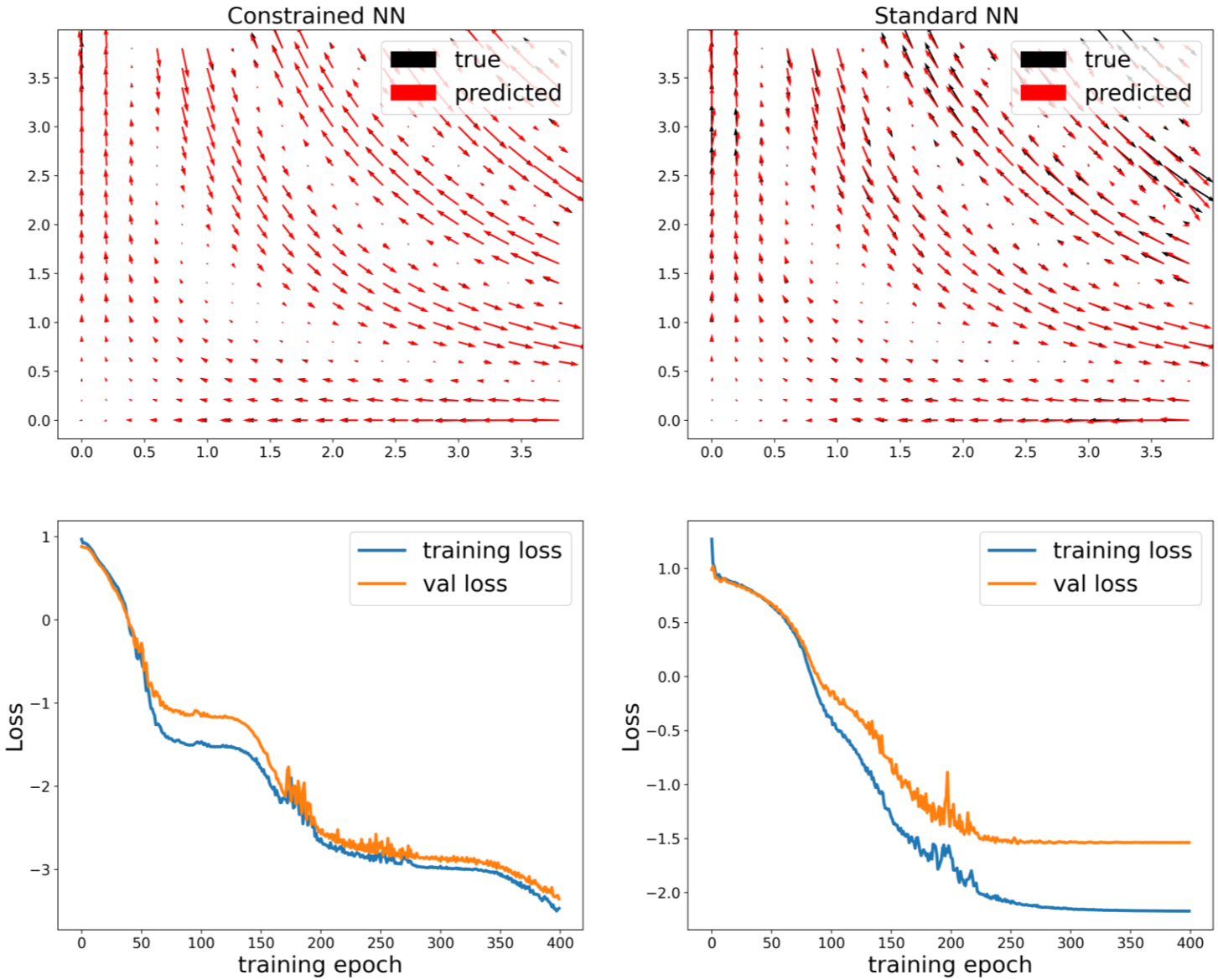
8. Training of Constrained Model

```
train_loss = np.empty([epochs, 1])
val_loss = np.empty([epochs, 1])

print('Training Constrained NN')

for epoch in range(epochs):
    train_loss[epoch] = train(epoch).detach().numpy()
    v_loss = eval(epoch)
    scheduler.step(v_loss)
    val_loss[epoch] = v_loss.detach().numpy()
    print('Constrained NN: epoch: ', epoch, 'training loss ', train_loss[epoch], \
          'validation loss', val_loss[epoch])


# work out the rms error for this one
x_pred = torch.cat((xv.reshape(20 * 20, 1), yv.reshape(20 * 20, 1)), 1)
(f_pred, v1_pred, v2_pred) = model(x_pred)
error_new = torch.cat((v1.reshape(400, 1) - v1_pred.detach(), v2.reshape(400, 1) - v2_pred.detach()), 0)
rms_error = torch.sqrt(sum(error_new * error_new) / 800)

# ---------------  Set up and train the uncconstrained model ------------------------------
optimizer_uc = torch.optim.Adam(model_uc.parameters(), lr=0.01)
scheduler_uc = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer_uc, patience=10,
                                                          min_lr=1e-10,
                                                          factor=0.5,
                                                          cooldown=15)
```

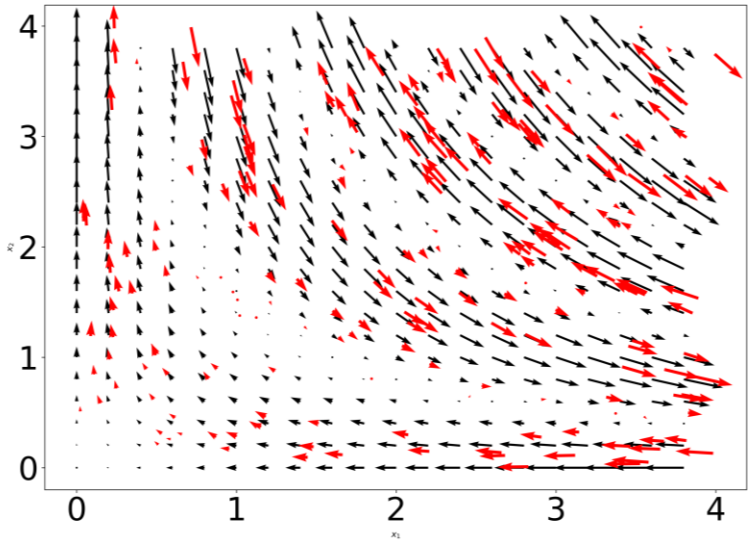
9. Training of Unconstrained Model

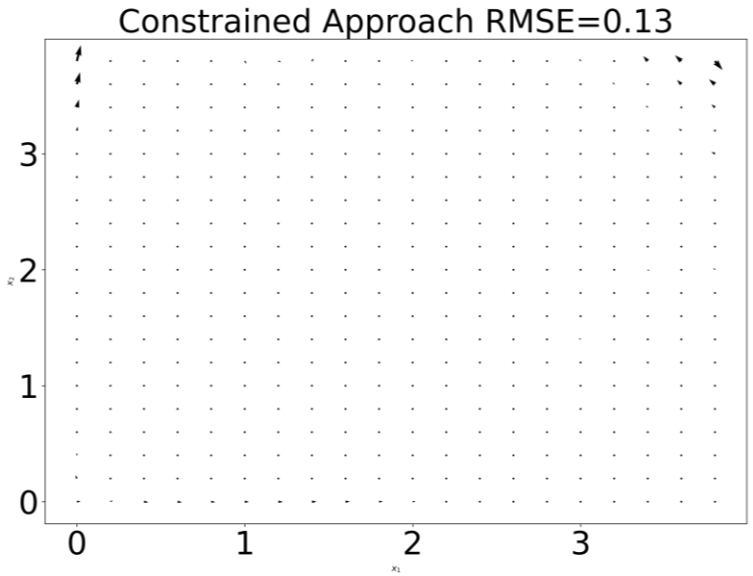
```
def train_uc(epoch):
    model_uc.train()
    total_loss = 0
    n_batches = 0
    for x1_train, x2_train, y1_train, y2_train in training_generator:
        optimizer_uc.zero_grad()
        x_train = torch.cat((x1_train, x2_train), 1)
        vhat = model_uc(x_train)
        y_train = torch.cat((y1_train, y2_train), 1)
        loss = criterion(y_train, vhat)
        loss.backward()
        optimizer_uc.step()
        total_loss += loss.cpu()
        n_batches += 1
    return total_loss / n_batches

def eval_uc(epoch):
    model_uc.eval()
    with torch.no_grad():
        (vhat) = model_uc(x_val)
        loss = criterion(y_val, vhat)
    return loss.cpu()


train_loss_uc = np.empty([epochs, 1])
val_loss_uc = np.empty([epochs, 1])


print('Training standard NN')

for epoch in range(epochs):
    train_loss_uc[epoch] = train_uc(epoch).detach().numpy()
    v_loss = eval_uc(epoch)
    scheduler_uc.step(v_loss)
    val_loss_uc[epoch] = v_loss.detach().numpy()
    print('Standard NN: epoch: ', epoch, 'training loss ', \
          train_loss_uc[epoch], 'validation loss', val_loss_uc[epoch])
```

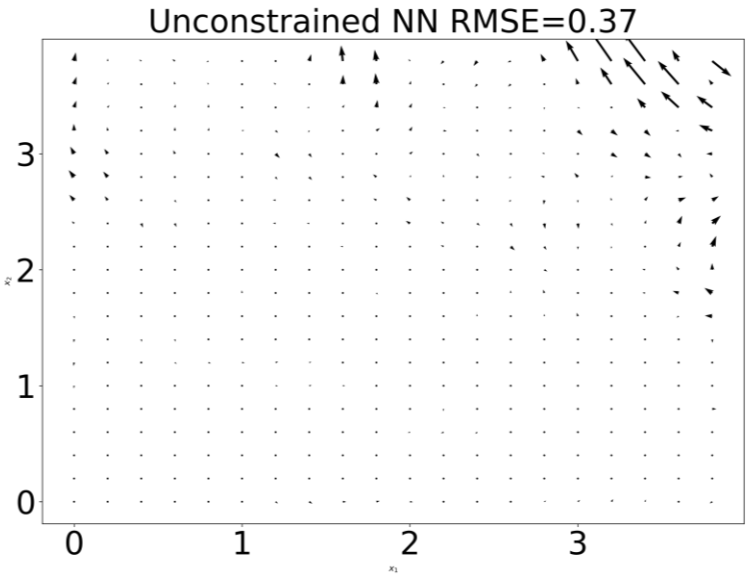
Linearly Constrained Neural Networks: Results



Linearly Constrained Neural Networks: Results







PINNs Family: Weighted Residual Methods

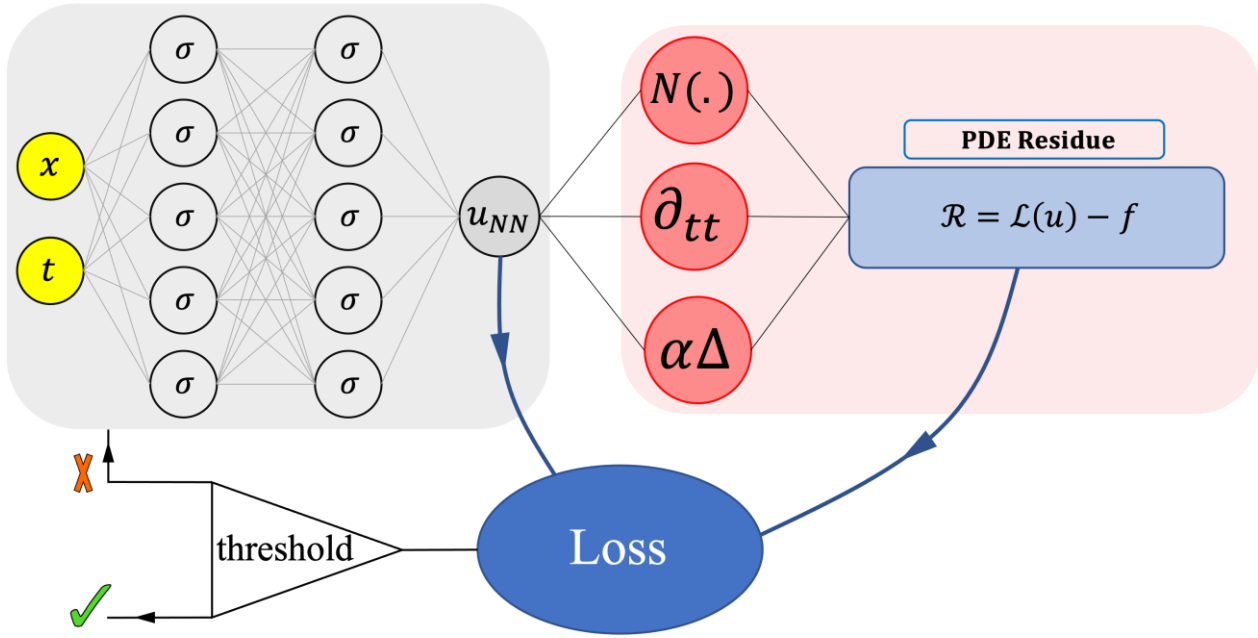
- Approximation: $u \approx \tilde{u} = u_{NN}$
- Strong-Form Residuals:

$$
\begin{aligned}
r(\tilde{u}) &= \mathcal{L}^q \tilde{u} - f && : \text{PDE residual} \\
r_b(\tilde{u}) &= \tilde{u} - h && : \text{boundary residual} \\
r_0(\tilde{u}) &= \tilde{u} - g && : \text{initial residual}
\end{aligned}
$$

Loss Function:

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

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



PINNs Family: Weighted Residual Methods

- Weighted/Varialtional Residuals:
  projections onto space of test function V

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

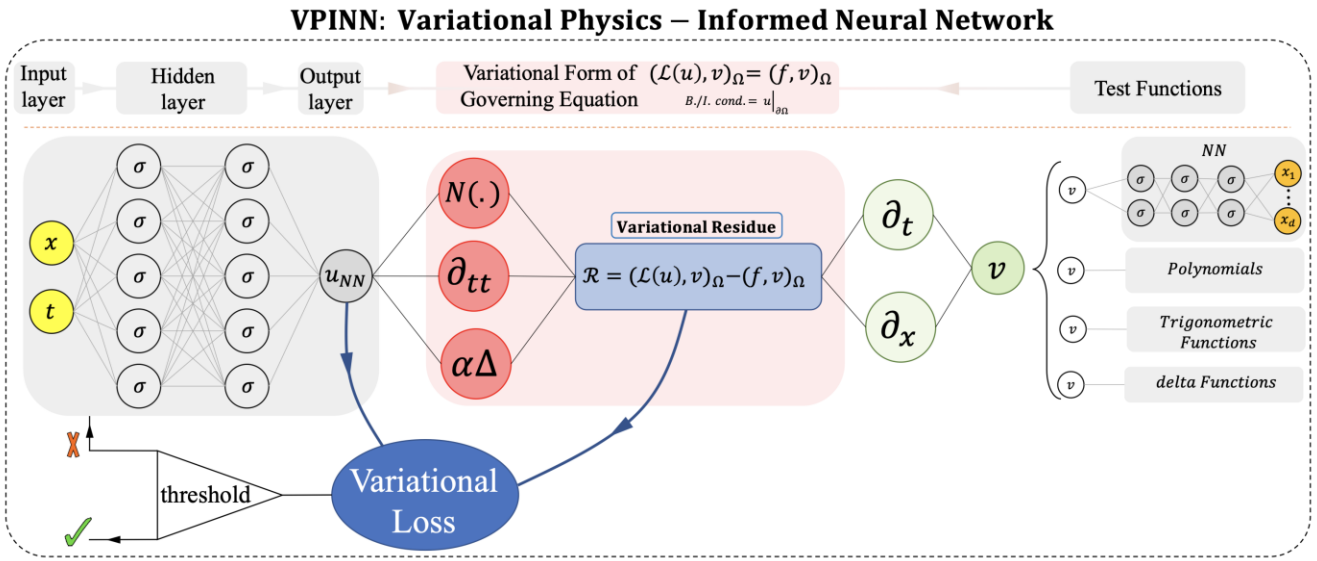
$$\mathcal{R}_{b,j}(\tilde{u}) = \int_{\partial\Omega\times(0,T]} r_b(\tilde{u})v_j dxdt$$

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

Loss Function:

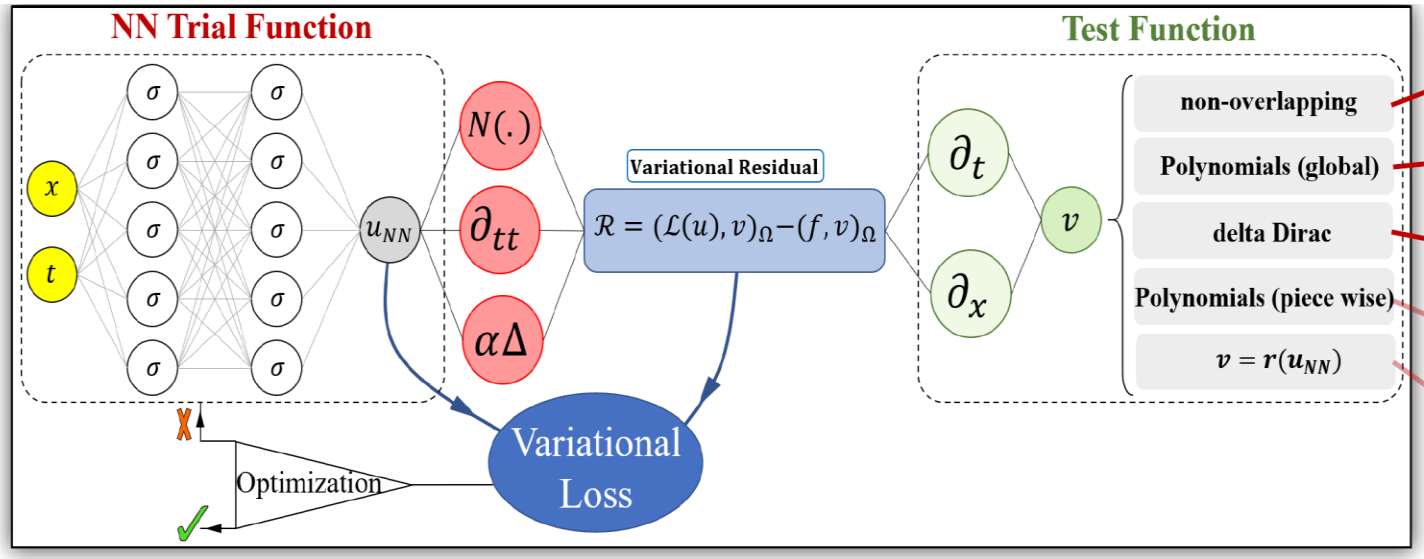
$$L^{\mathfrak{v}}(\tilde{u}, v) = w\sum_{j=1}^{N_r}\mathcal{R}_j^2(\tilde{u}) + w_b\sum_{j=1}^{N_b}\mathcal{R}_{b,j}^2(\tilde{u}) + w_0\sum_{j=1}^{N_0}\mathcal{R}_{0,j}^2(\tilde{u})$$

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.

hp-VPINNs: A General Framework for Solving PDEs



hp-VPINNs [1]
VPINNs [2]
PINNs [3]
VarNet [4]
DGM [5]

	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

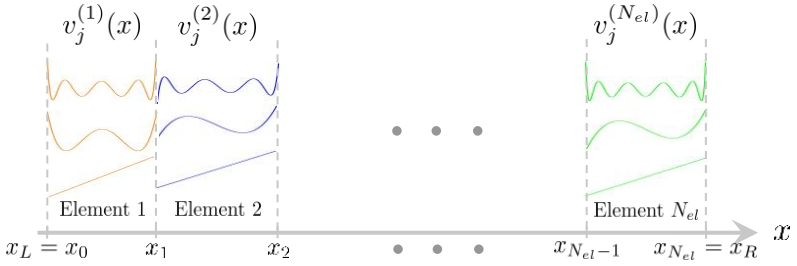
[1] Kharazmi et al., CMAME (2020).
[2] Kharazmi et al., *arXiv* (2019).
[3] Raissi et al., JCP (2019)
[4] Khodayi-Mehr et al., arXiv (2019).
[5] Sirignano et al., JCP (2018)
[6] Jagtap et al., CMAME (2020)
[7] Li et al., arXiv (2019)

hp-VPINNs: A General Framework for Solving PDEs

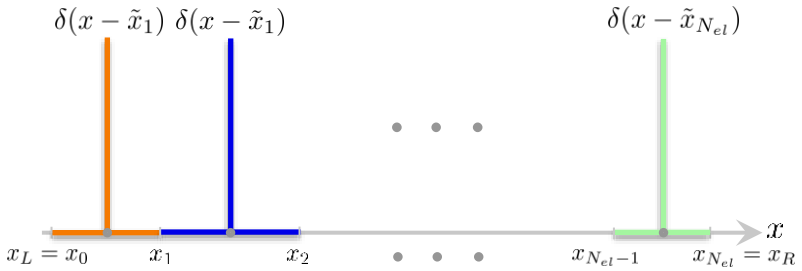
h-refinement & p-refinement

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

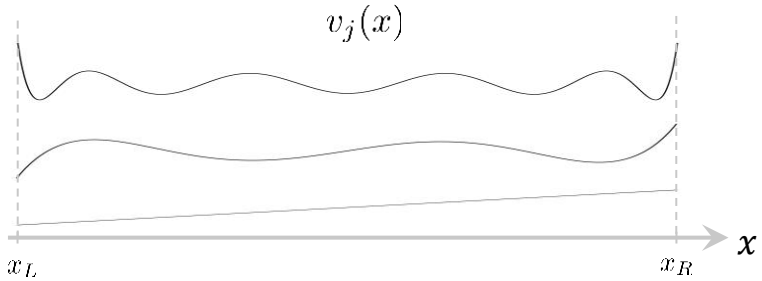
$$v_j^{(e)} = \begin{cases} \bar{v}_j \neq 0, & x \in \Omega^{(e)} \\ 0, & \text{otherwise} \end{cases}$$



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

hp-VPINNs: A General Framework for Solving PDEs

Loss Function

elemental/local variational loss

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

global variational loss

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

strong-form loss

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

Variational Neural Networks (VNNs): Function Approximation

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

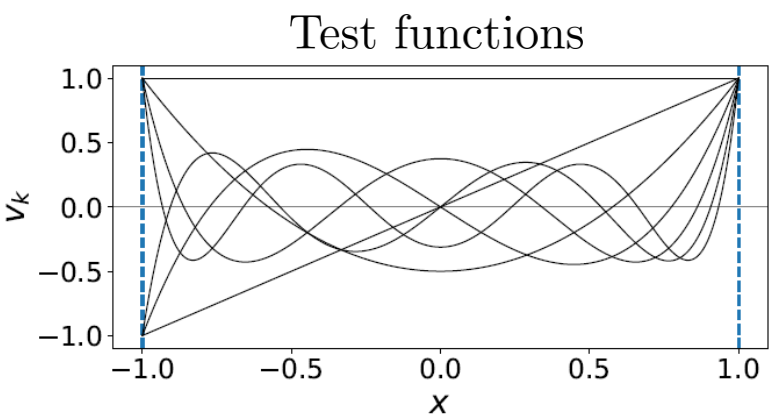
$$\text{Variational loss:} \qquad L^{\mathfrak{v}} = \sum_{e=1}^{N_{el}} \frac{1}{K^{(e)}} \sum_{k=1}^{K^{(e)}} \left| \mathcal{R}_k^{(e)} \right|^2$$

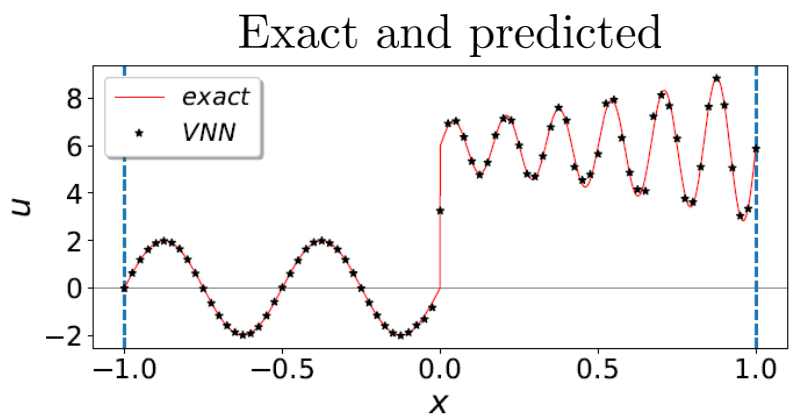
Example (discontinuous function approximation): the target function is

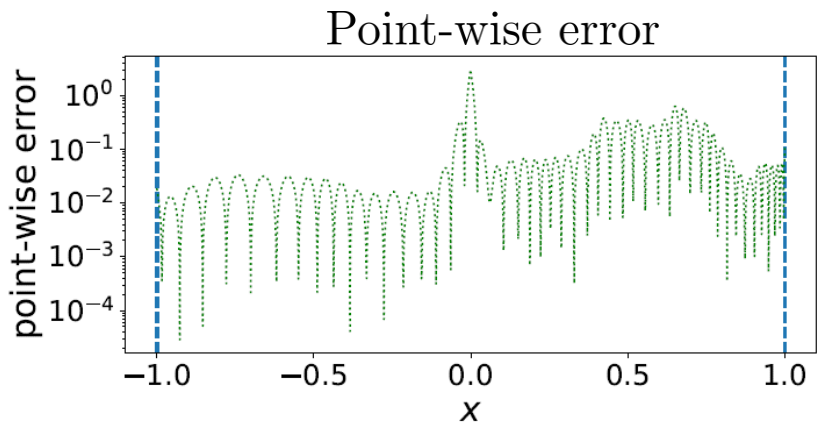
$$u^{exact} = \begin{cases} 2\sin(4\pi x) & x \in [-1, 0], \\ 6 + \exp^{1.2x} \sin(12\pi x) & x \in (0, 1] \end{cases}$$

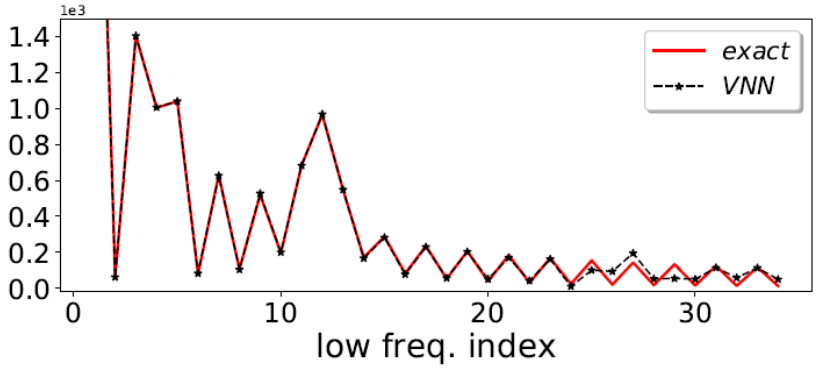
Variational Neural Networks (VNNs): Function Approximation

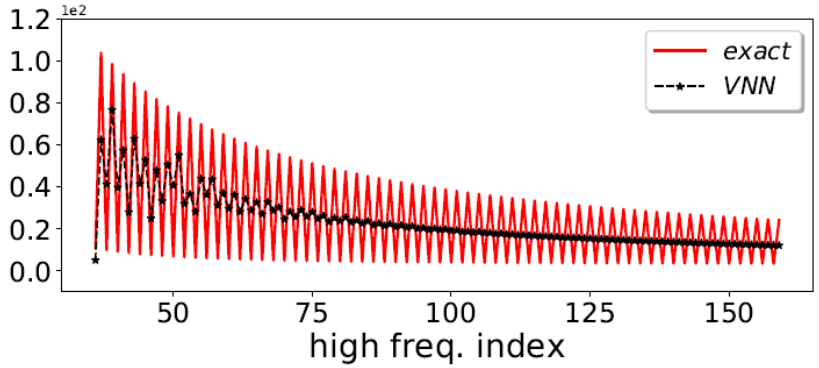
Global test functions (single element):





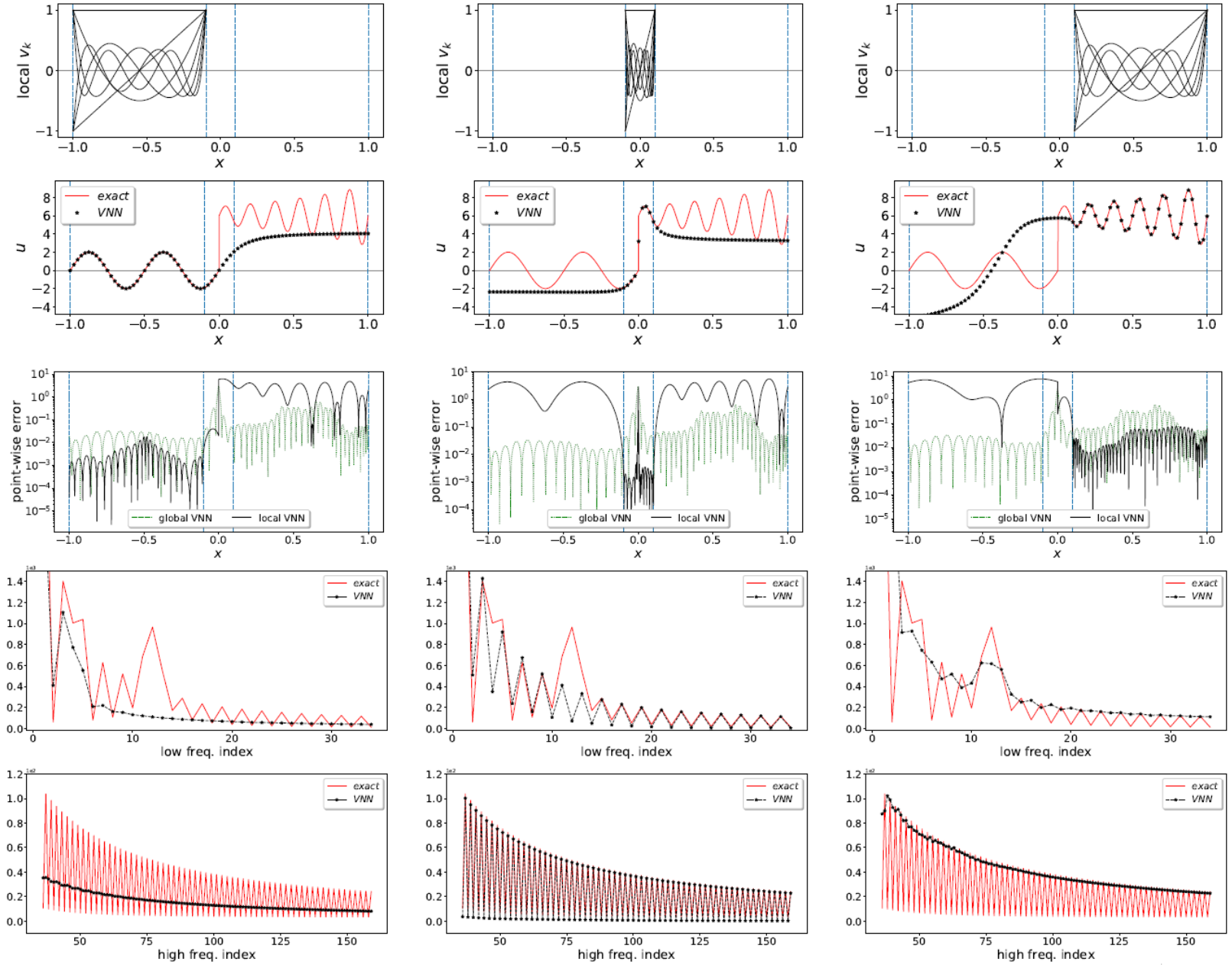






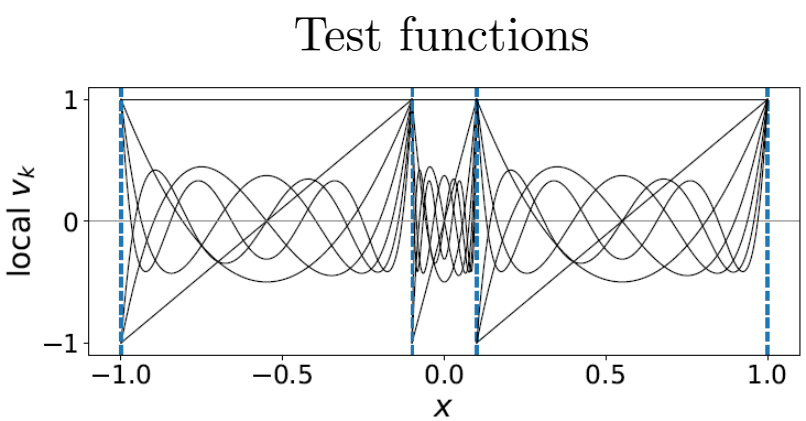
Variational Neural Networks (VNNs): Function Approximation

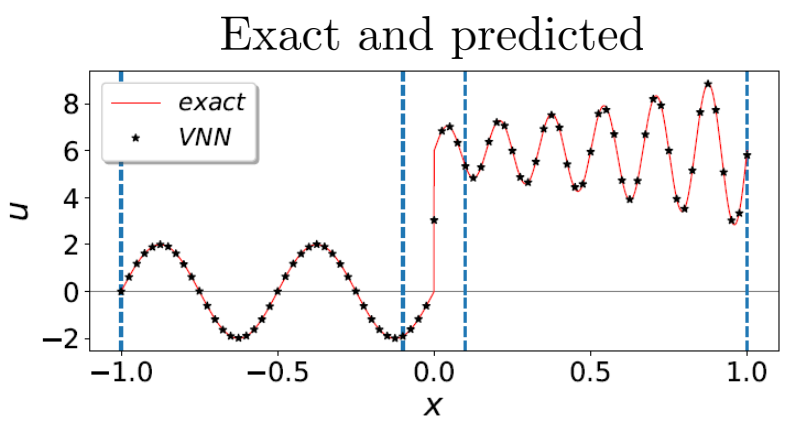
Local (elemental) test functions:

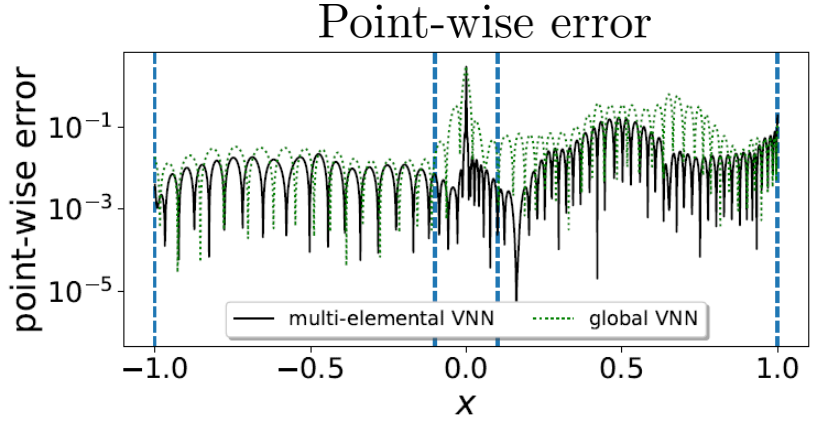


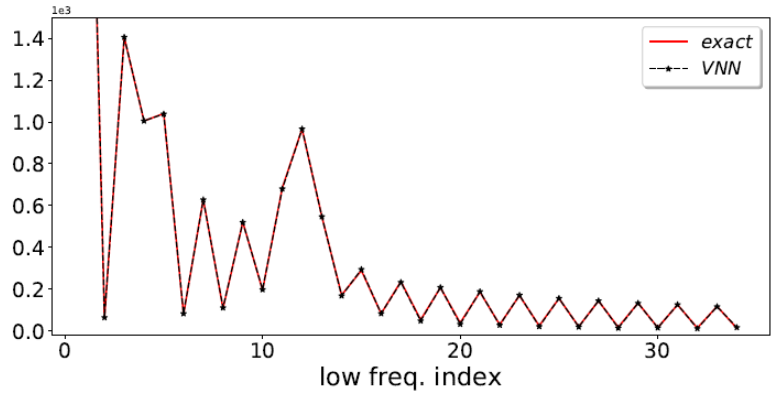
Variational Neural Networks (VNNs): Function Approximation

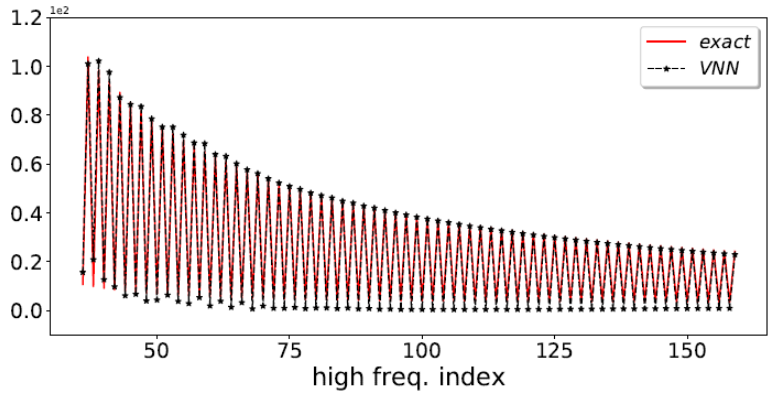
Multi-elemental test functions:





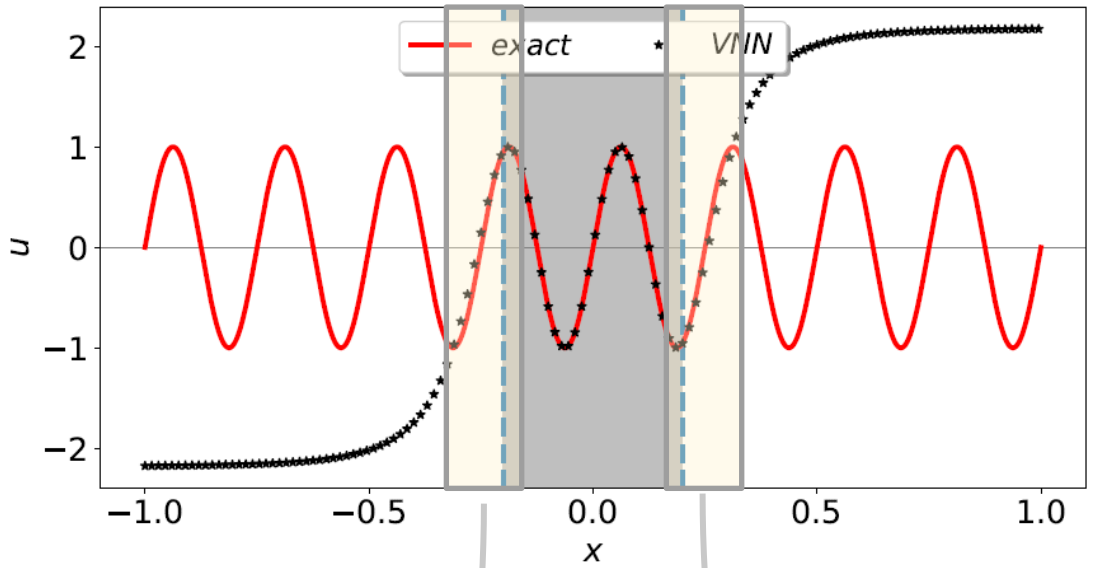


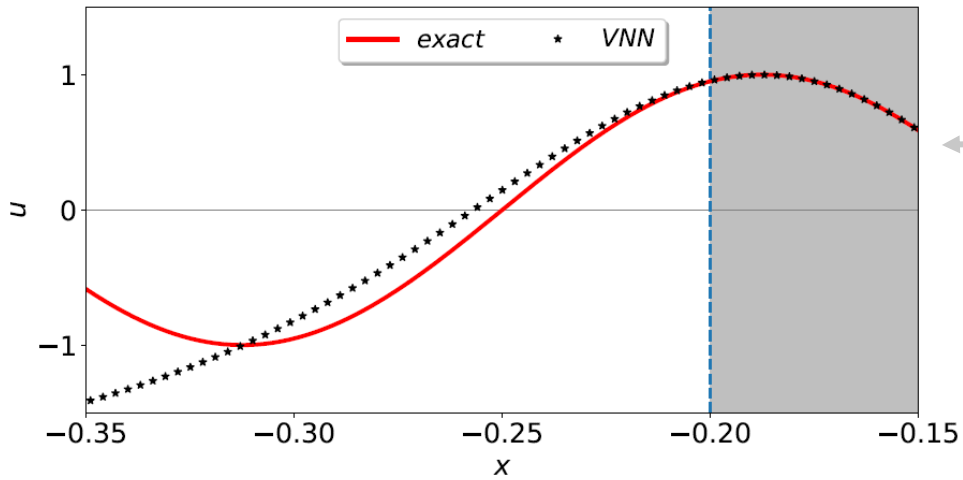


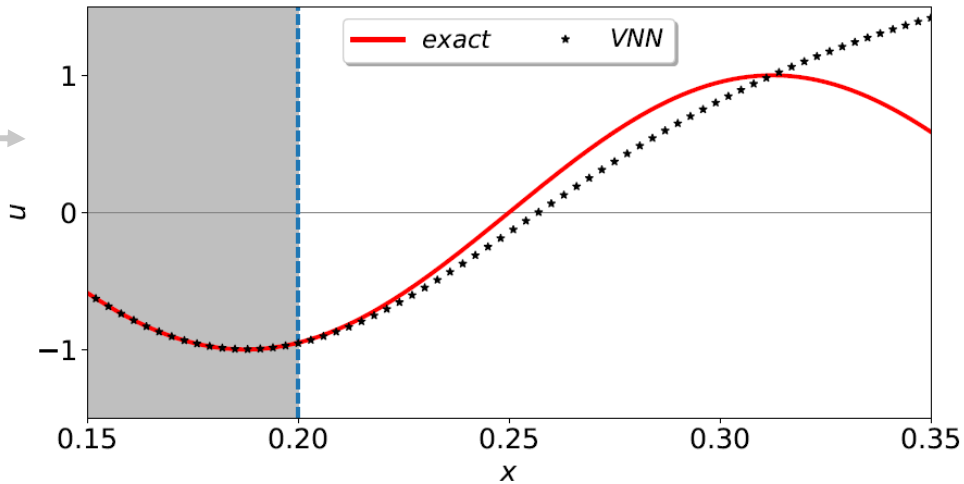


Variational Neural Networks (VNNs): Function Approximation

Localized learning & learning out-of-the-box







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

hp-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:

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

hp-VPINNs: Poisson's Equation (1D)

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

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

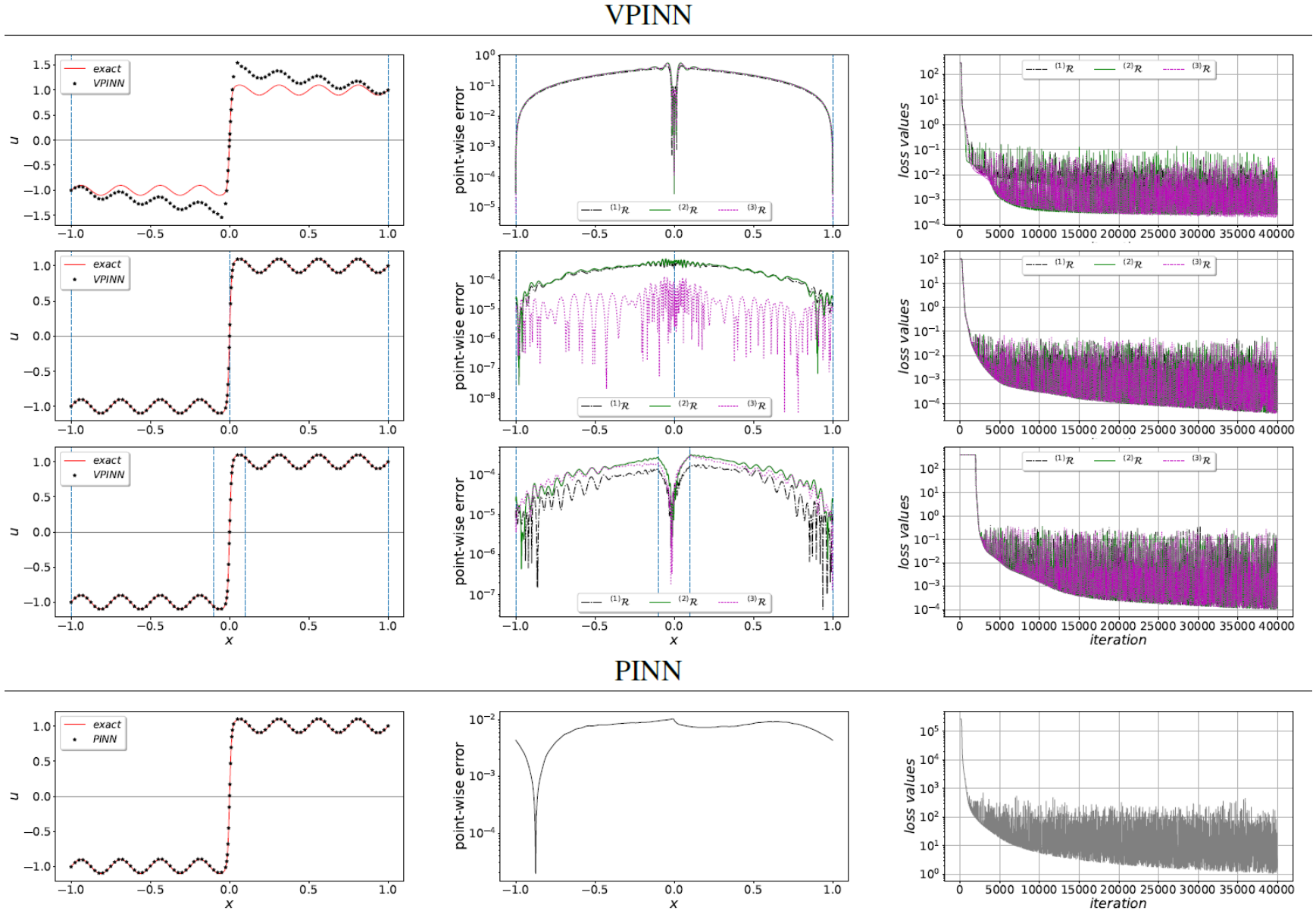
- Variational loss:

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

hp-VPINNs: Poisson's Equation (1D)

Example: **Steep Solution**

$$u^{exact}(x) = 0.1\sin(8\pi x) + \tanh(80x)$$

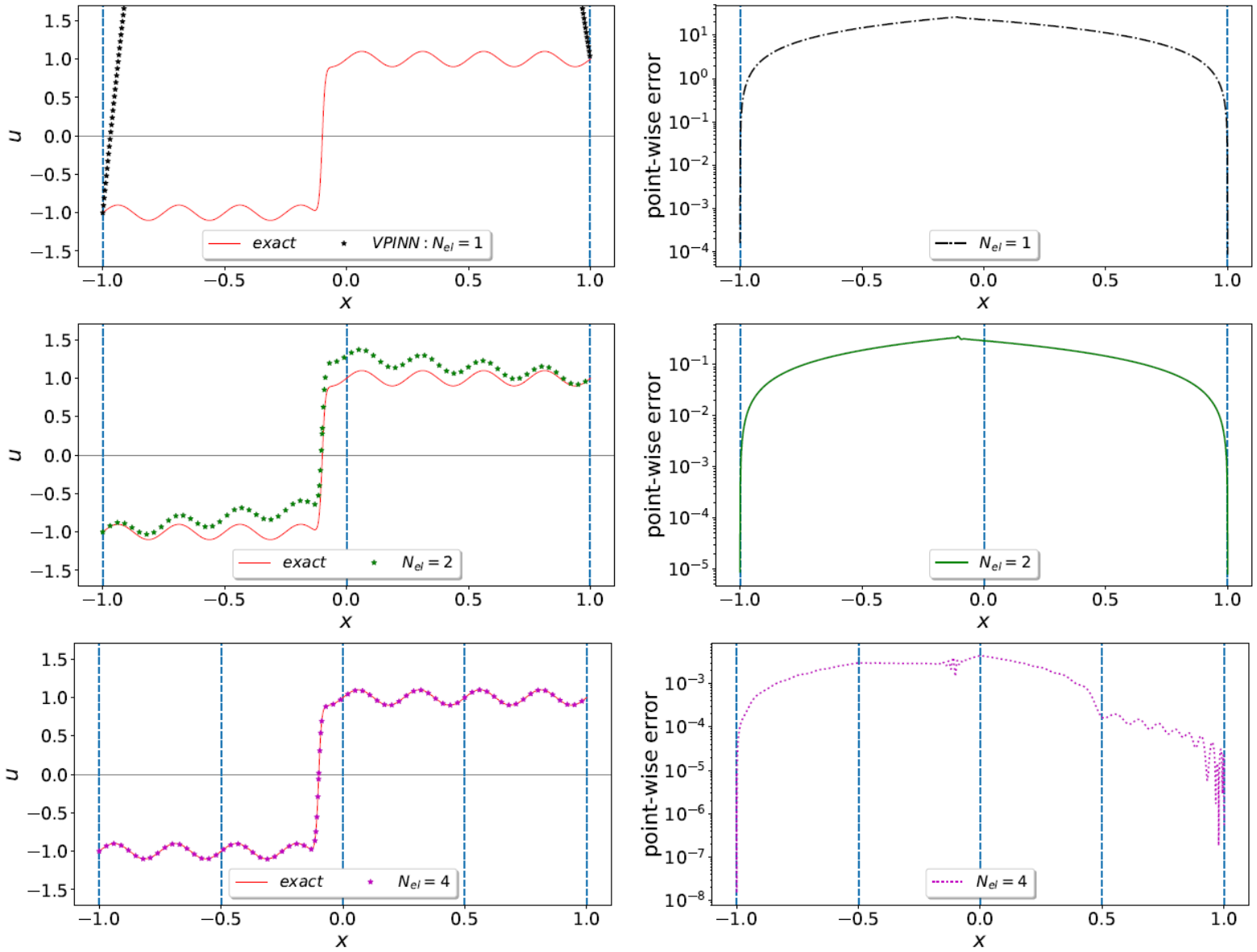


hp-VPINNs: Poisson's Equation (1-D)

Example: **Singularity Capturing**

$$u^{\text{exact}}(x) = 0.1\sin(8\pi x) + \tanh(80(x+0.1))$$

asymmetric steep part



Kharazmi E, Zhang Z, Karniadakis GE. hp-VPINNs: Variational physics-informed neural networks with domain decomposition. Computer Methods in Applied Mechanics and Engineering. 2021 Feb 1;374:113547.

hp-VPINNs: Poisson's Equation (2-D)

$$\nabla^2 u(x,y) = f(x,y), \qquad (x,y) \in \Omega = [-1,1] \times [-1,1]$$

Discrete finite dimensional space, given by

$$\tilde{V} = \text{span}\left\{v_{k_1 k_2}(x,y) = \phi_{k_1}(x)\phi_{k_2}(y), k_m = 1,2,\cdots,K_m, m = 1,2\right\}$$
$$\phi_{k_1}^{(e_x)}\left(x_{e_x-1}\right) = \phi_{k_1}^{(e_x)}\left(x_{e_x}\right) = 0, \quad k_1 = 1,2,\cdots,K_1,$$
$$\phi_{k_2}^{(e_y)}\left(y_{e_y-1}\right) = \phi_{k_2}^{(e_y)}\left(y_{e_y}\right) = 0, \quad k_2 = 1,2,\cdots,K_2.$$

The variational residual becomes

$$\mathcal{R}_{k_1 k_2}^{(e_e e_y)} = \int_{x_{e_x-1}}^{x_{e_x}} \int_{y_{e_y-1}}^{y_{e_y y}} \left(\nabla^2 u_{NN}(x,y) - f(x,y)\right) \phi_{k_1}^{(e_{x_x})}(x)\phi_{k_2}^{(e_y)}(y)dxdy$$

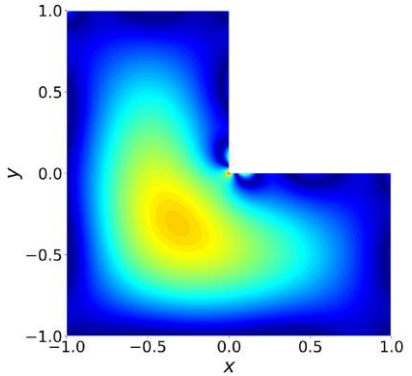
The variational loss

$$L^{\mathfrak{v}(i)} = \sum_{e_r=1}^{N_{el_r}} \sum_{e_y=1}^{N_{el_y}} \frac{1}{K_1 K_2} \sum_{k=1}^{K_1 K_2} \left|{}^{(i)}\mathcal{R}_k^{(e_e e_y)}\right|^2 + \tau_b \frac{1}{N_b} \sum_{i=1}^{N_b} \left|r_b\left(x_b^i, y_b^i\right)\right|^2, \quad i = 1,2,3$$

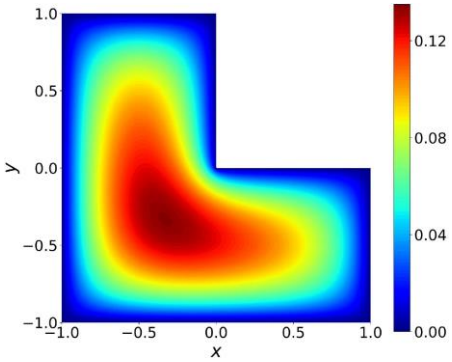
Kharazmi E, Zhang Z, Karniadakis GE. hp-VPINNs: Variational physics-informed neural networks with domain decomposition. Computer Methods in Applied Mechanics and Engineering. 2021 Feb 1;374:113547.

hp-VPINNs: Poisson's Equation (2-D)

PINNs point-wise error



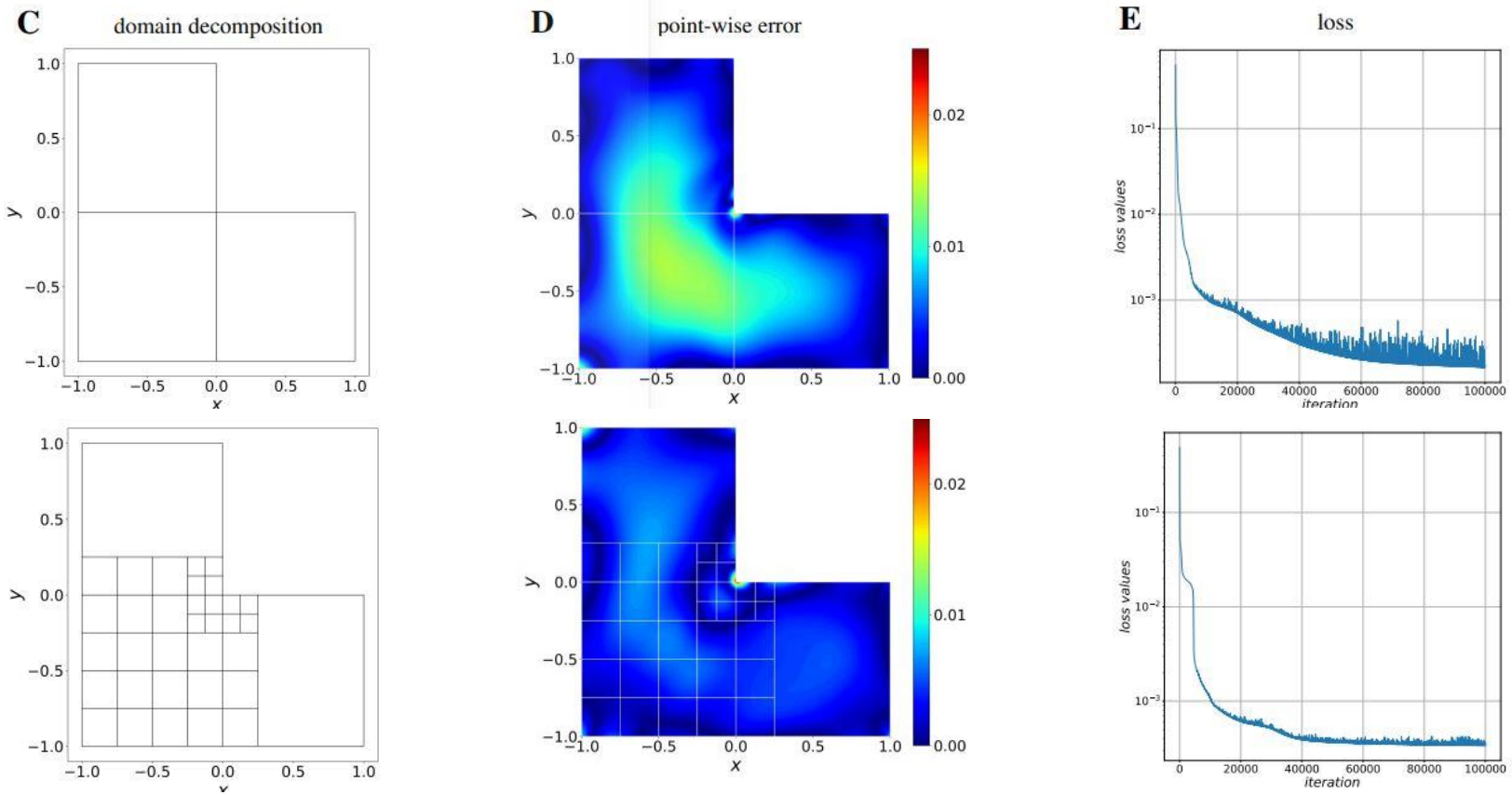
reference solution



2-D PDE: $\begin{cases} \nabla^2 u(x,y) = f(x,y) \\ \text{Homogeneous Dirichlet B.C.} \end{cases}$

The homogeneous case $f(x,y) = 0$ gives the Laplace equation

hp-VPINN:[1] $\mathcal{R}$ formulation

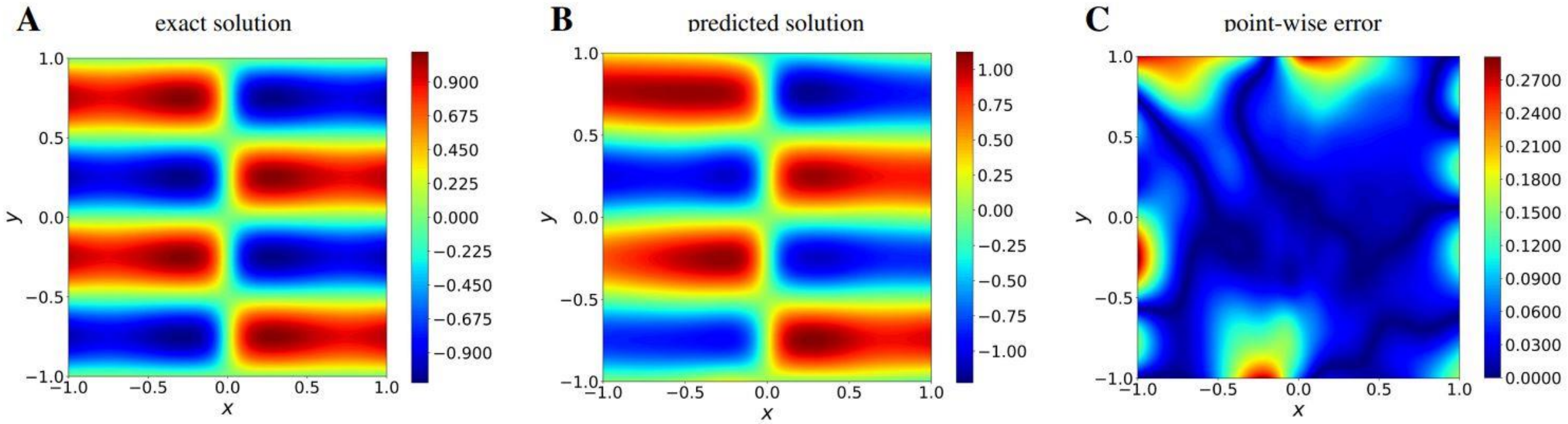


Kharazmi E, Zhang Z, Karniadakis GE. hp-VPINNs: Variational physics-informed neural networks with domain decomposition. Computer Methods in Applied Mechanics and Engineering. 2021 Feb 1;374:113547.

hp-VPINNs: Poisson's Equation

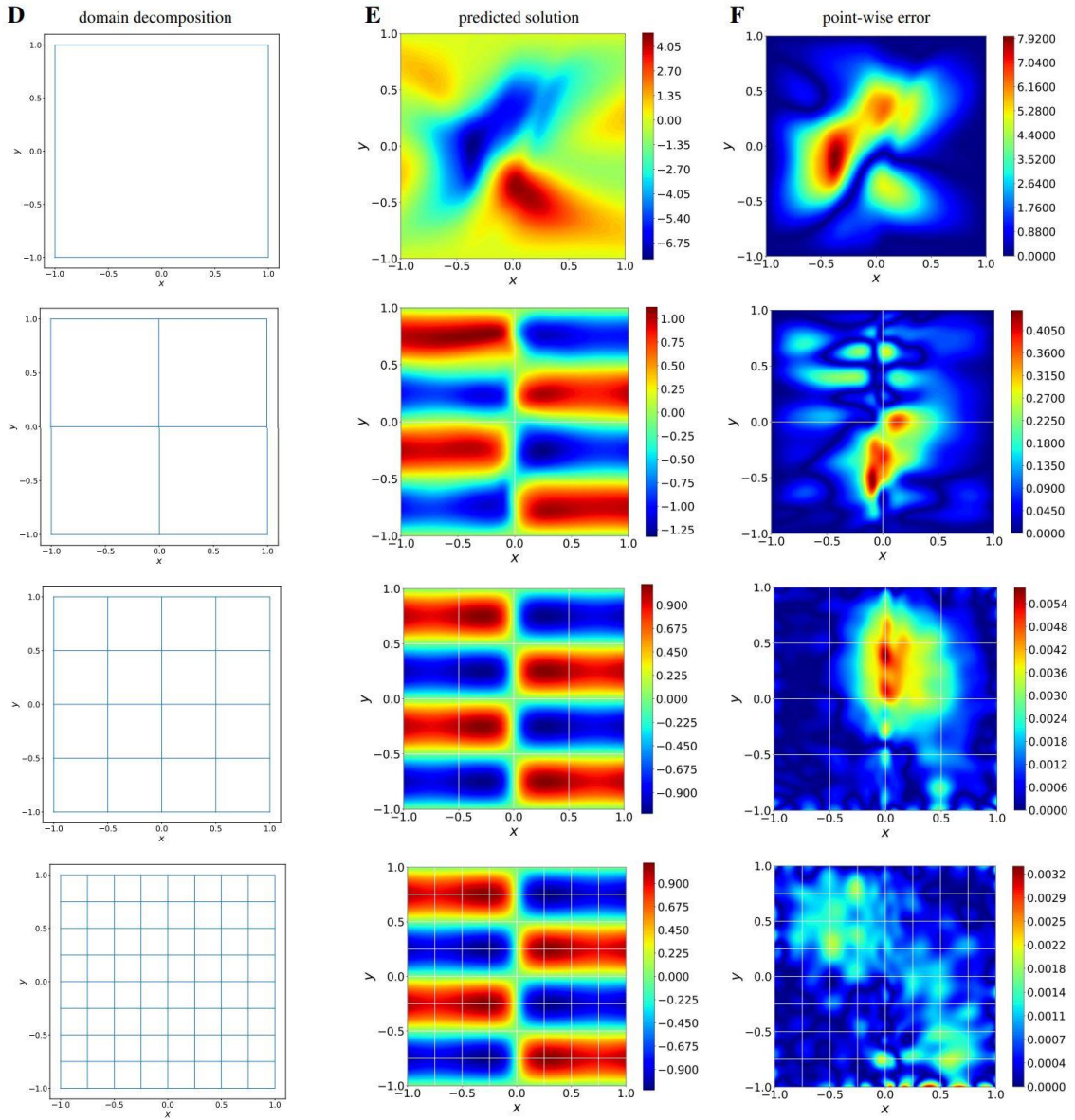
$$\text{2-D PDE: } \begin{cases} \nabla^2 u(x,y) = f(x,y) \\ \text{Homogeneous Dirichlet B.C.} \end{cases}, \qquad u^{\text{exact}}(x,y) = \sin(2\pi y) \times (0.1\sin(2\pi x) + \tanh(10x))$$

PINN



Kharazmi E, Zhang Z, Karniadakis GE. hp-VPINNs: Variational physics-informed neural networks with domain decomposition. Computer Methods in Applied Mechanics and Engineering. 2021 Feb 1;374:113547.

hp-VPINNs: Poisson's Equation



Kharazmi E, Zhang Z, Karniadakis GE. hp-VPINNs: Variational physics-informed neural networks with domain decomposition. Computer Methods in Applied Mechanics and Engineering. 2021 Feb 1;374:113547.

hp-VPINNs: Advection-Diffusion Equation

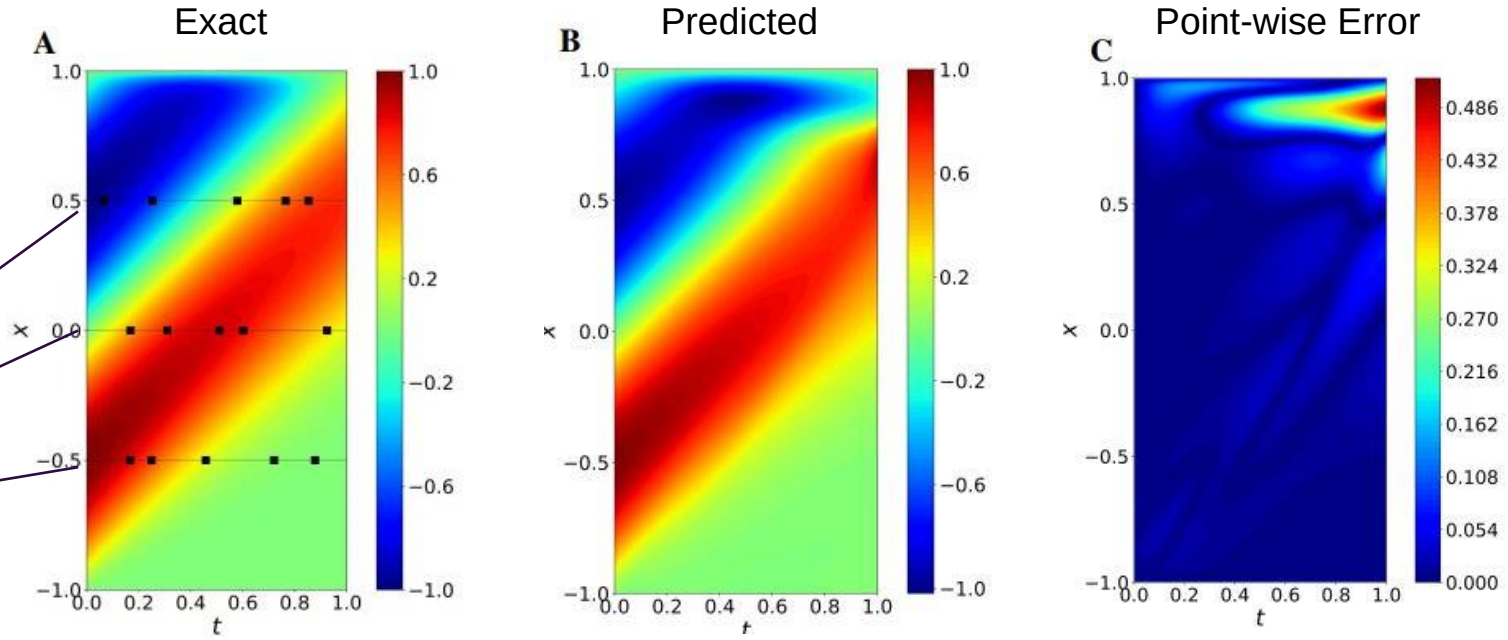
$$\Omega = [0,1] \times [-1,1]$$

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = \kappa \frac{\partial^2 u}{\partial x^2},$$

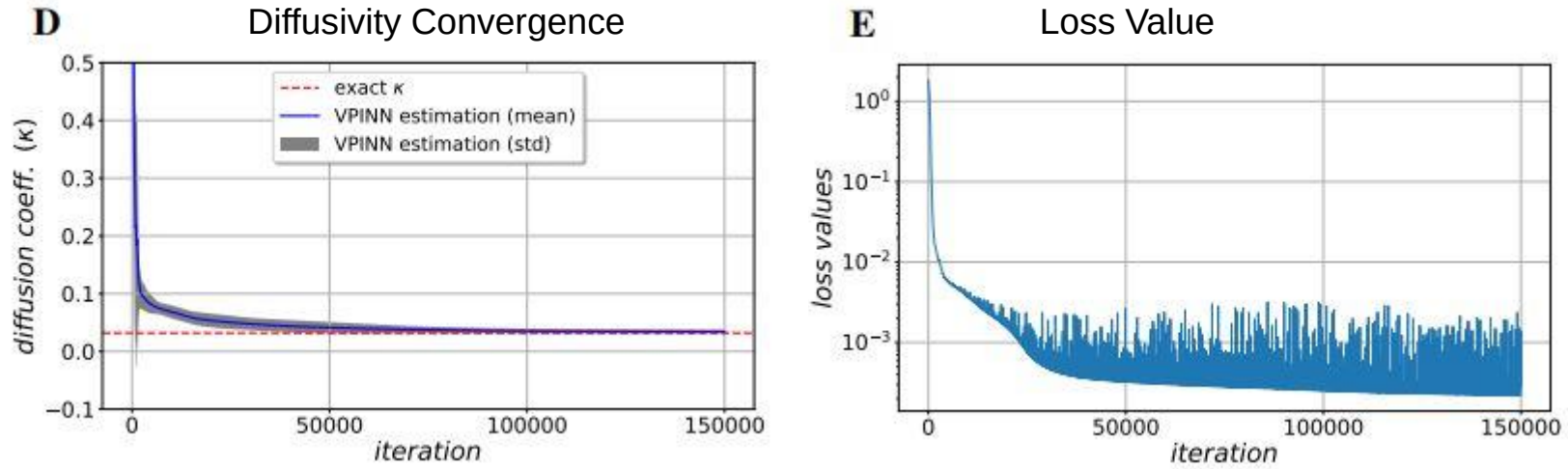
$$u(-1,t) = u(1,t) = 0,$$

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

Tree locations of sparse measurements in time



Estimation of diffusivity coefficient



Kharazmi E, Zhang Z, Karniadakis GE. hp-VPINNs: Variational physics-informed neural networks with domain decomposition. Computer Methods in Applied Mechanics and Engineering. 2021 Feb 1;374:113547.

Deep Learning for Science and Engineering Teaching Kit

Thank You