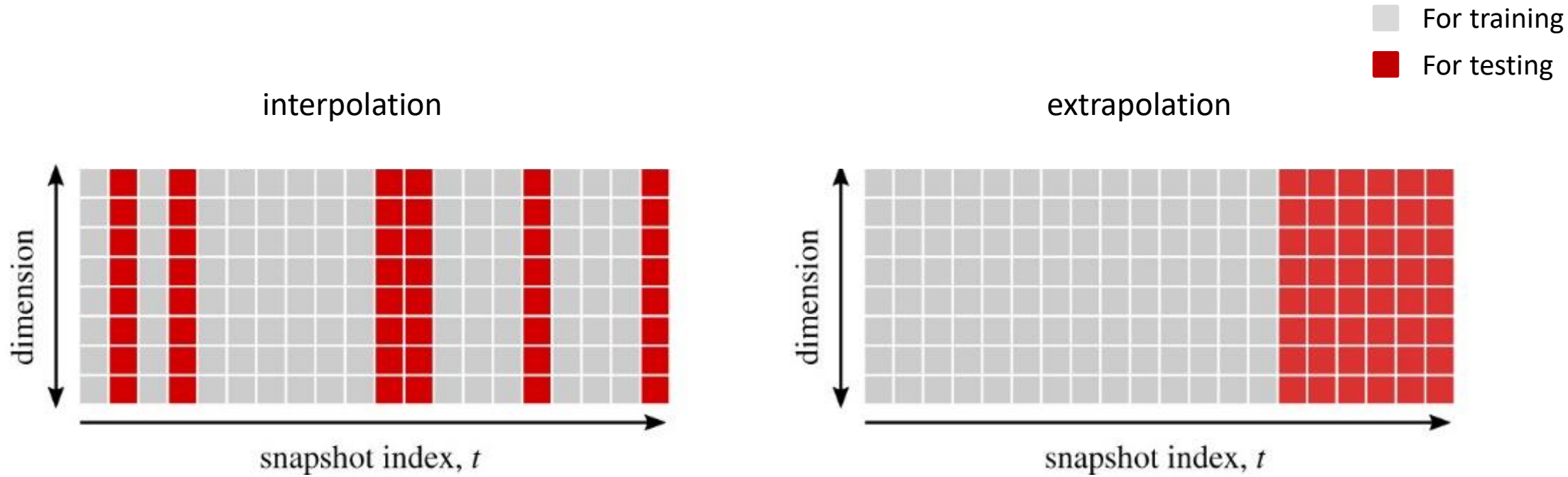
Physics-informed Neural Networks

Prof. Seungchul Lee, Industrial AI Lab.

Why Deep Learning Needs Physics?

- Why do data-driven 'black-box' methods fail?
  - May output result that is physically inconsistent
  - Easy to learn spurious relationships that look good only on training and test data
    - Can lead to poor generalization outside the available data (out-of-sample prediction tasks)
  - Interpretability is absent
    - Discovering the mechanism of an underlying process is crucial for scientific advancements

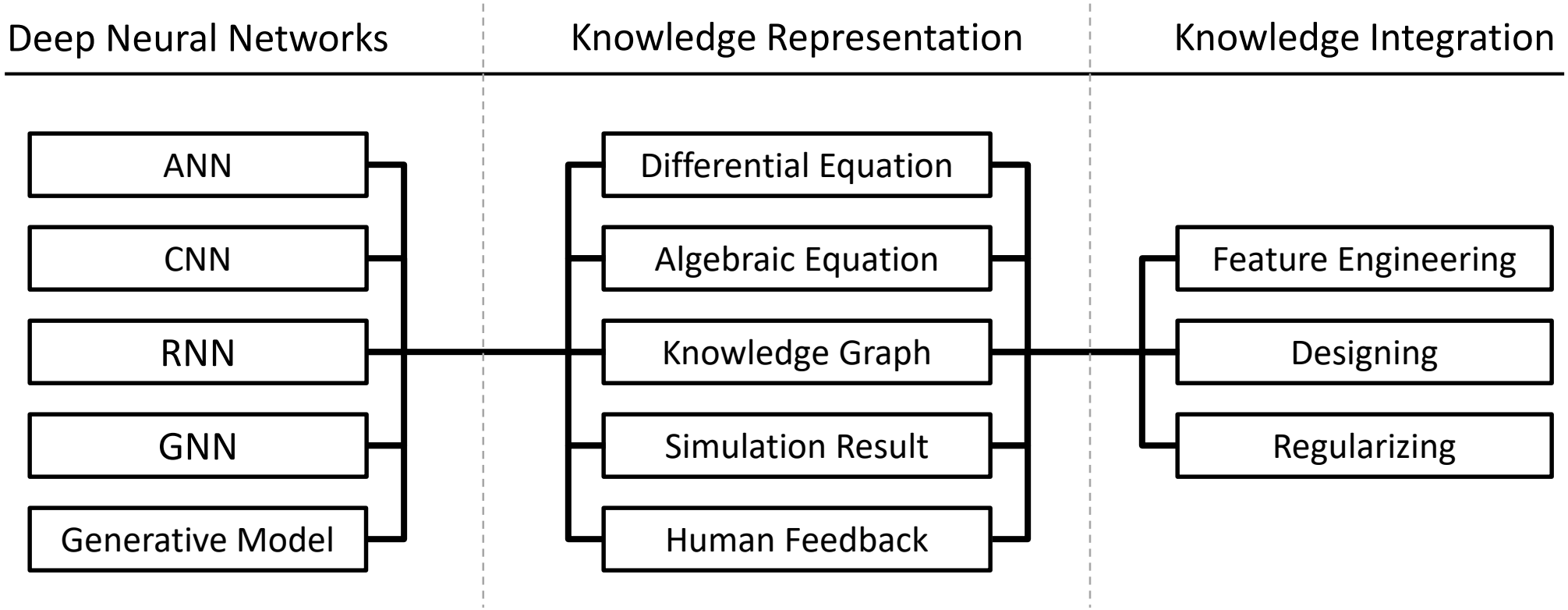


Why Deep Learning Needs Physics?

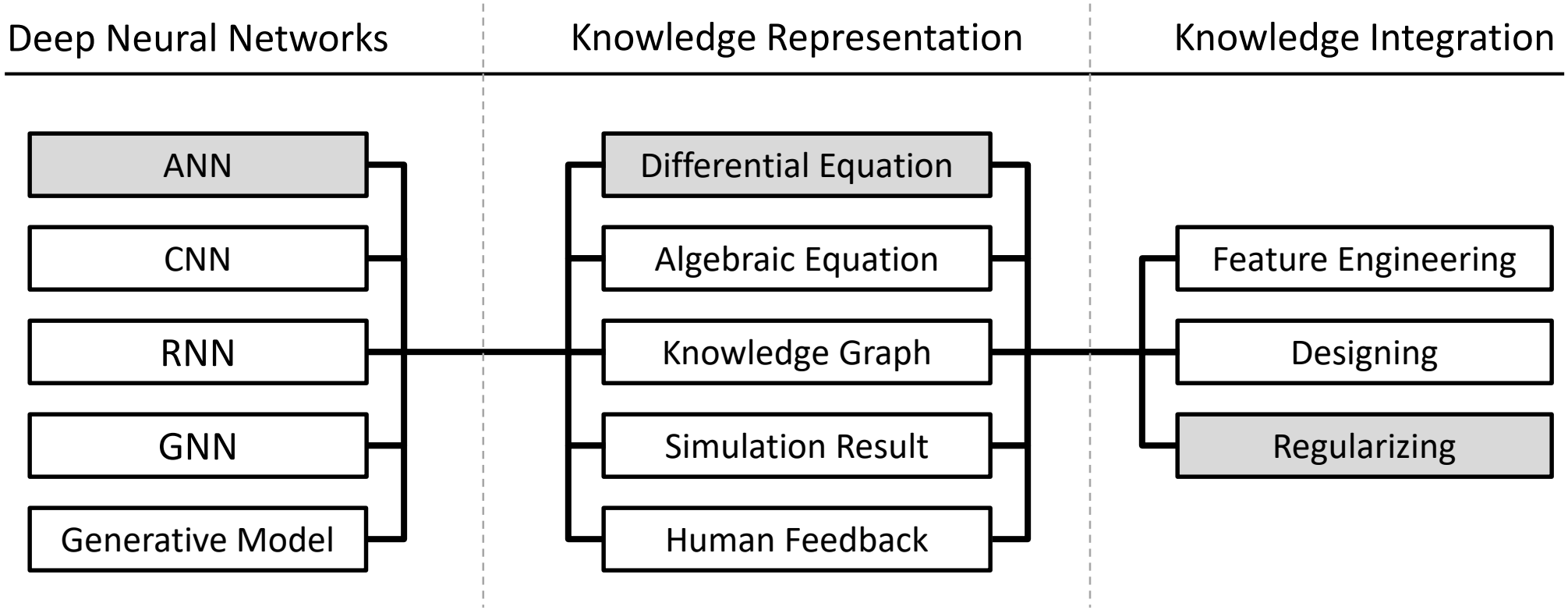
- Why do data-driven 'black-box' methods fail?
  - May output result that is physically inconsistent
  - Easy to learn spurious relationships that look good only on training and test data
    - Can lead to poor generalization outside the available data (out-of-sample prediction tasks)
  - Interpretability is absent
    - Discovering the mechanism of an underlying process is crucial for scientific advancements

- Physics-Informed Neural Networks (PINNs)
  - Take full advantage of data science methods with the accumulated prior knowledge of scientific theories → Improve predictive performance
  - Integration of domain knowledge to overcome the issue of imbalanced data & data shortage

Taxonomy of Informed Deep Learning



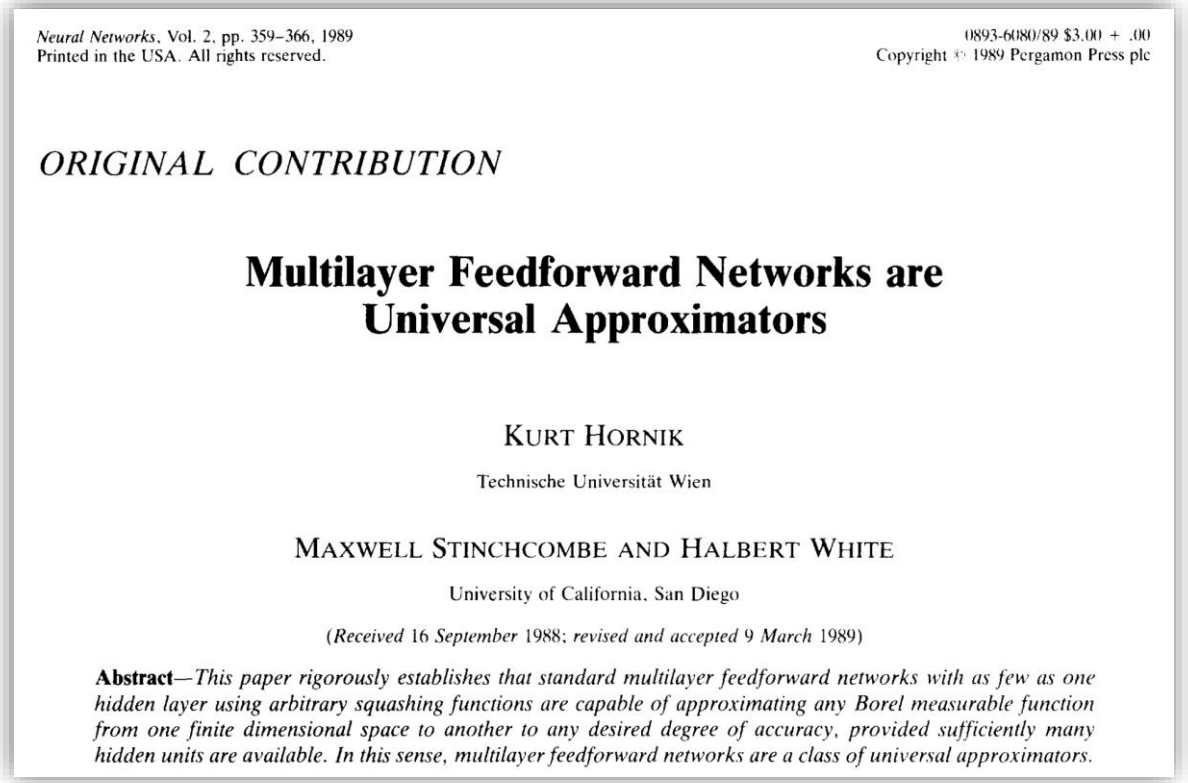
Taxonomy of Informed Deep Learning



Deep Learning for Solving Differential Equations

Multilayer Feedforward Networks are Universal Approximators

- The Universal Approximation Theorem
  - Neural Networks are capable of approximating any Borel measurable function
  - Neural Networks (1989)



Neural Networks, Vol. 2, pp. 359–366, 1989
Printed in the USA.

0893-6080/89 $3.00 + .00

ORIGINAL CONTRIBUTION

Multilayer Feedforward Networks are Universal Approximators

KURT HORNIK

Technische Universität Wien

MAXWELL STINCHCOMBE AND HALBERT WHITE

University of California, San Diego

(*Received* 16 *September* 1988; *revised and accepted* 9 *March* 1989)

Abstract—*This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.*

Differential Equations

- Types of ODEs
  - Linear

$$f'' + y = f(t)$$

  - Nonlinear

$$x^2 y'' + y = f(t) \qquad y' y'' + y = f(t)$$

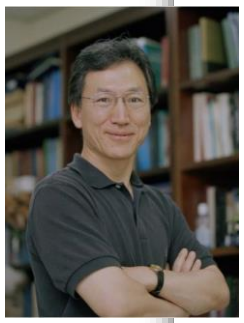
- Partial Differential Equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \qquad \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = u(x, y)$$

- Types of ODE / PDE problems
  - Initial Value Problem (IVP)
  - Boundary Value Problem (BVP)

Neural Networks for Solving Differential Equations

- Neural Algorithm for Solving Differential Equations
  - Journal of Computational Physics (1990)
  - Neural minimization for finite difference equation



JOURNAL OF COMPUTATIONAL PHYSICS **91**, 110–131 (1990)

Neural Algorithm for Solving Differential Equations

HYUK LEE

Department of Electrical Engineering, Polytechnic Institute of New York, Brooklyn, New York 11201

AND

IN SEOK KANG

Department of Chemical Engineering, California Institute of Technology, Pasadena, California 91125

Received August 17, 1988; revised October 6, 1989

Finite difference equations are considered to solve differential equations numerically by utilizing minimization algorithms. Neural minimization algorithms for solving the finite difference equations are presented. Results of numerical simulation are described to demonstrate the method. Methods of implementing the algorithms are discussed. General features of the neural algorithms are discussed.

- ANN for ODE and PDE
  - IEEE on Neural Networks (1998)

IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 9, NO. 5, SEPTEMBER 1998 987

Artificial Neural Networks for Solving Ordinary and Partial Differential Equations

Isaac Elias Lagaris, Aristidis Likas, *Member, IEEE,* and Dimitrios I. Fotiadis

To illustrate the method, we consider the *first-order ODE*

$$\frac{d\Psi(x)}{dx} = f(x, \Psi) \tag{11}$$

with $x \in [0, 1]$ and the IC $\Psi(0) = A$.
A trial solution is written as

$$\Psi_t(x) = A + xN(x, \vec{p}) \tag{12}$$

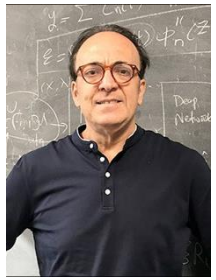
where $N(x, \vec{p})$ is the output of a feedforward neural network with one input unit for x and weights $\vec{p}$. Note that $\Psi_t(x)$ satisfies the IC by construction. The error quantity to be minimized is given by

$$E[\vec{p}] = \sum_i \left\{ \frac{d\Psi_t(x_i)}{dx} - f(x_i, \Psi_t(x_i)) \right\}^2 \tag{13}$$

Journal of Computational Physics (2019)

- M. Raissi, P. Perdikaris, G.E. Karniadakis



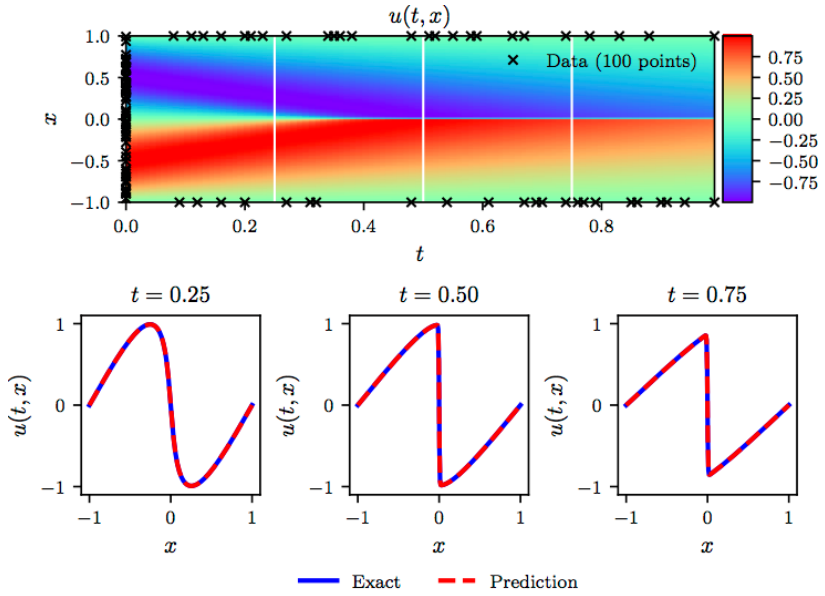


Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations

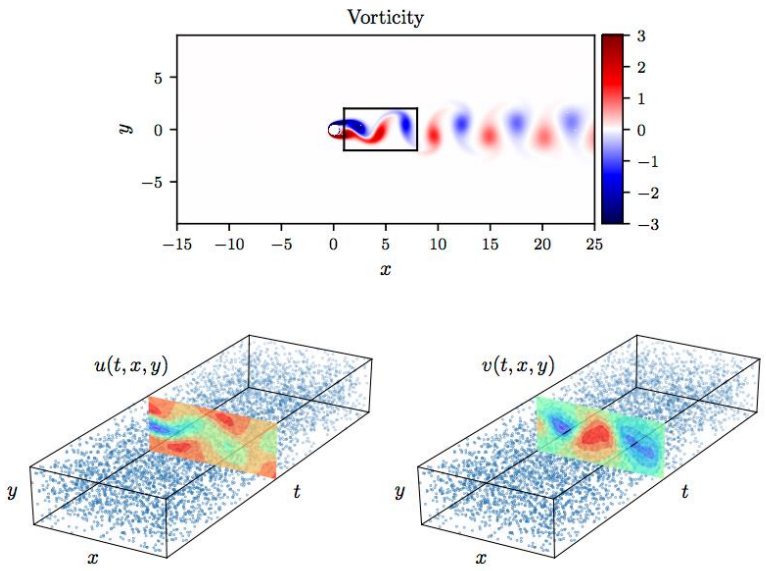
M. Raissi [a], P. Perdikaris [b,*], G.E. Karniadakis [a]

[a] Division of Applied Mathematics, Brown University, Providence, RI, 02912, USA
[b] Department of Mechanical Engineering and Applied Mechanics, University of Pennsylvania, Philadelphia, PA, 19104, USA

Burgers' equation



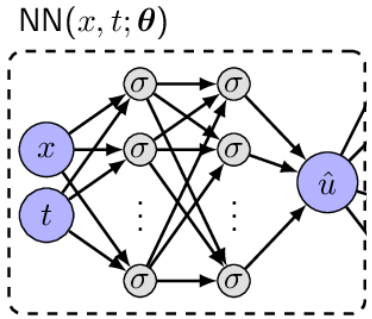
Navier-Stokes equation



Architecture of Physics-informed Neural Networks (PINN)

- NN as an universal function approximator

- Given
  - ODE or PDE
  - Some measured data from initial and boundary conditions



$$\frac{\partial u}{\partial t} - \lambda \frac{\partial^2 u}{\partial x^2} = 0$$

- Adding constraints for regularization
  - Regularized by physics, but matched with sparse data

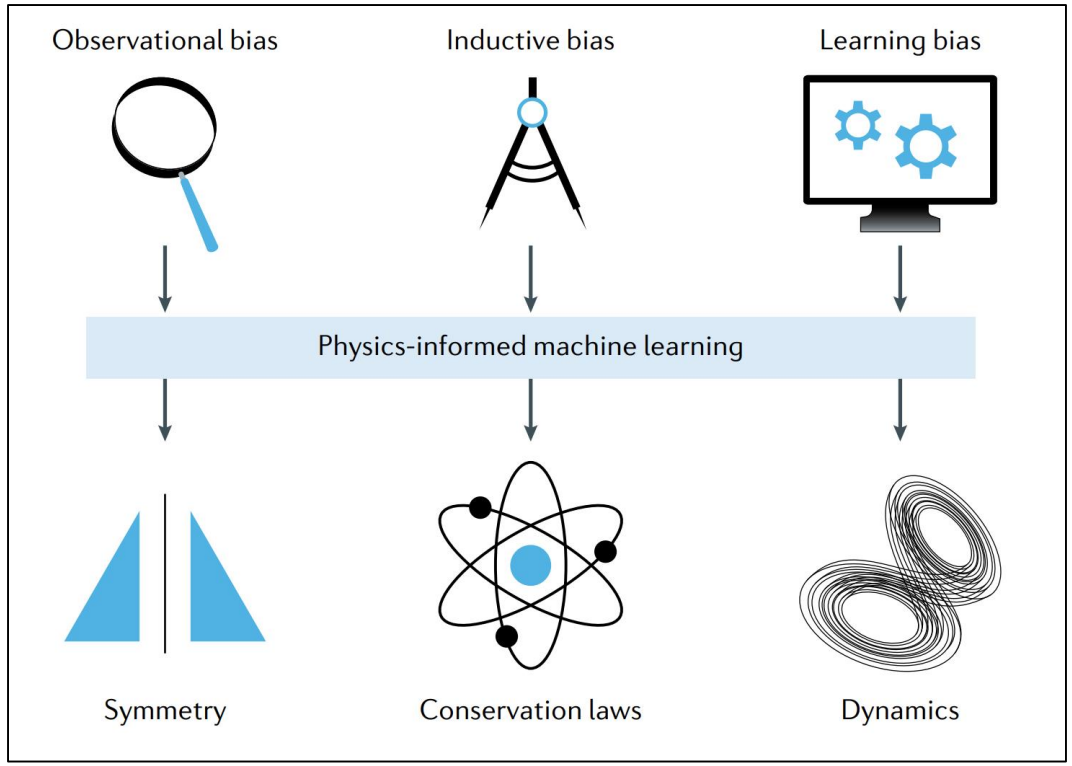
$$\mathcal{L} = \omega_{\text{data}} \mathcal{L}_{\text{data}} + \omega_{\text{PDE}} \mathcal{L}_{\text{PDE}}$$

$$\mathcal{L}_{\text{data}} = \frac{1}{N_{\text{data}}} \sum (u(x,t) - u)^2$$

$$\mathcal{L}_{\text{PDE}} = \frac{1}{N_{\text{PDE}}} \sum \left(\frac{\partial u}{\partial t} - \lambda \frac{\partial^2 u}{\partial x^2} \right)^2$$

Nature Reviews Physics (2021)

- Physics-informed machine learning
- How to embed physics in ML



nature reviews physics

Explore content ⌄ About the journal ⌄ Publish with us ⌄

nature › nature reviews physics › review articles › article

Review Article | Published: 24 May 2021

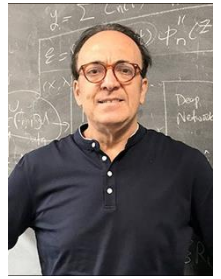
Physics-informed machine learning

George Em Karniadakis ✉, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang & Liu Yang

Nature Reviews Physics **3**, 422–440 (2021) | Cite this article

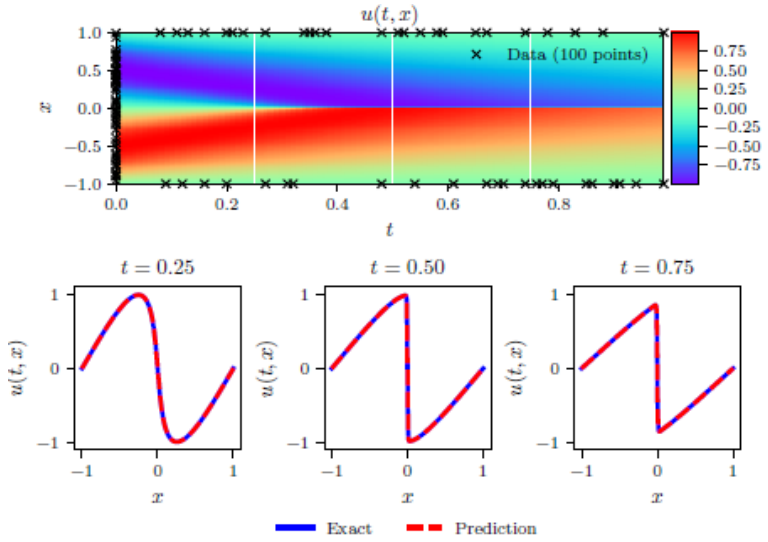
6882 Accesses | **3** Citations | **40** Altmetric | Metrics

- Observation bias
- Inductive bias
- Learning bias

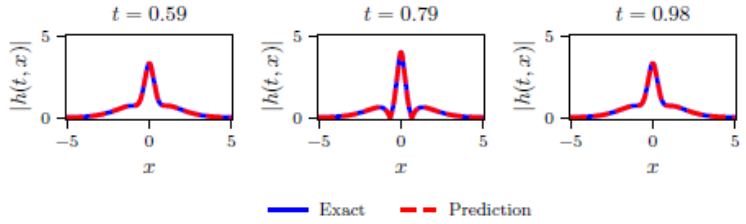


Examples

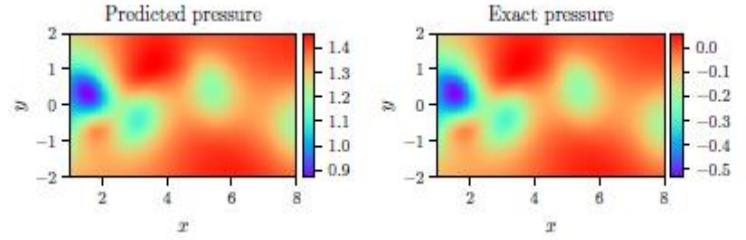
- 1D Burgers' Equation
  - $u_t + uu_x - (0.01/\pi)u_{xx} = 0, \quad x \in [-1, 1],\ t \in [0, 1]$
  - $u(0, x) = -\sin(\pi x)$
  - $u(t, -1) = u(t, 1) = 0$



- Schrodinger Equation
  - $ih_t + 0.5h_{xx} + |h|^2 h = 0, \quad x \in [-5, 5],\ t \in [0, \frac{\pi}{2}]$
  - $h(0, x) = 2\,\mathrm{sech}(x)$
  - $h(t, -5) = h(t, 5)$
  - $h_x(t, -5) = h_x(t, 5)$



- 2D Navier-Stokes Equation
  - $u_t + \lambda_1(uu_x + vu_y) = -p_x + \lambda_2(u_{xx} + u_{yy})$
  - $v_t + \lambda_1(uv_x + vv_y) = -p_y + \lambda_2(v_{xx} + v_{yy})$



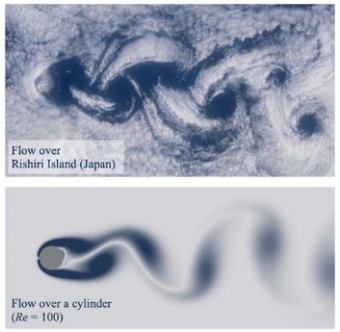
Example: Vortex Shedding

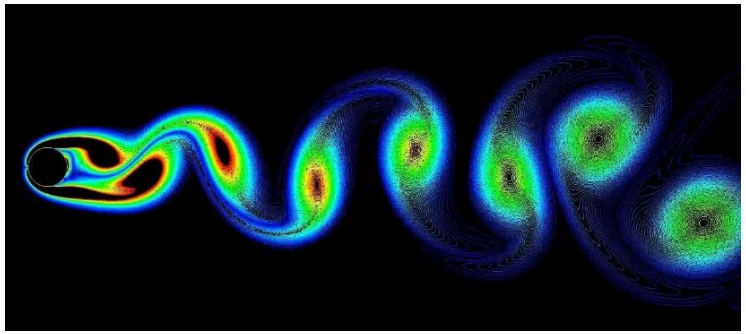
- Navier-Stocks equation (momentum)
- PDE

$$\frac{du}{dt} + \lambda_1 \left(u\frac{du}{dx} + v\frac{du}{dy} \right) = -\frac{dp}{dx} + \lambda_2 \left(\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} \right)$$

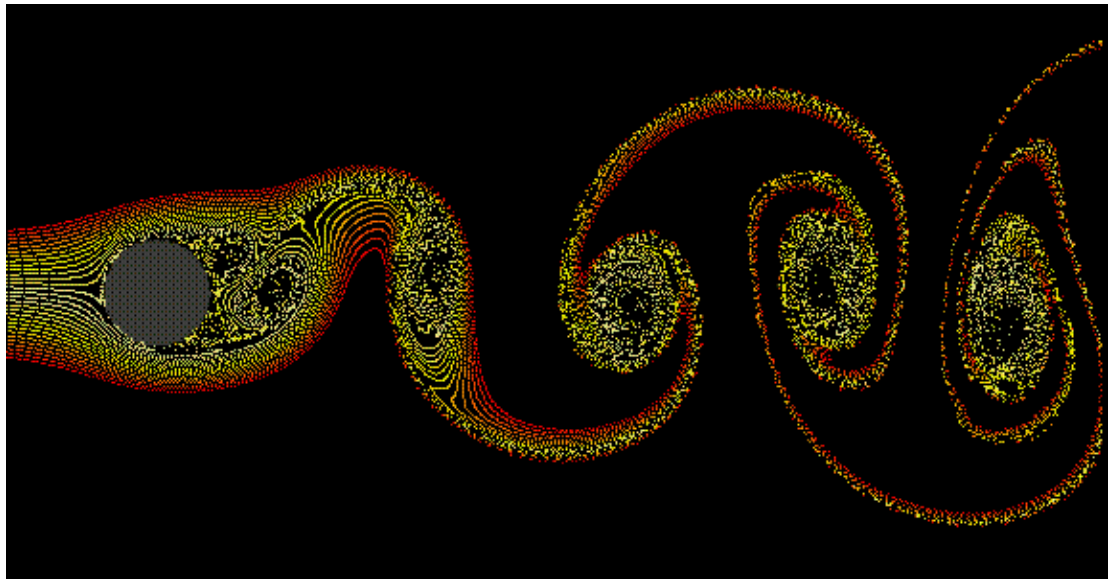
$$\frac{dv}{dt} + \lambda_1 \left(u\frac{dv}{dx} + v\frac{dv}{dy} \right) = -\frac{dp}{dy} + \lambda_2 \left(\frac{d^2v}{dx^2} + \frac{d^2v}{dy^2} \right)$$

- Boundary conditions
- Initial conditions (5000 points used)

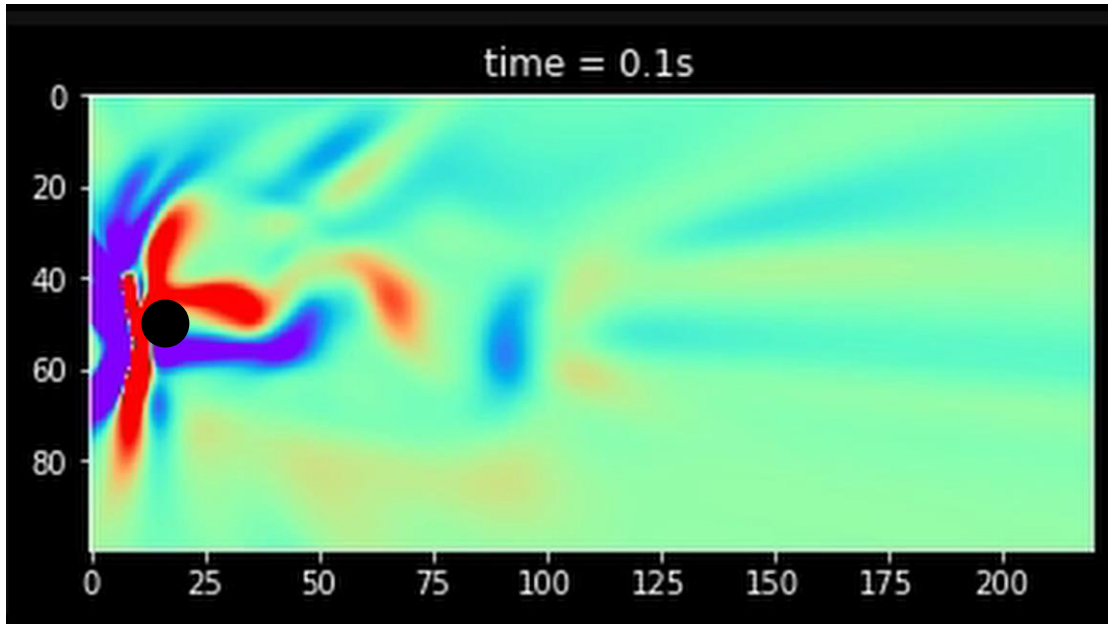




CFD simulation

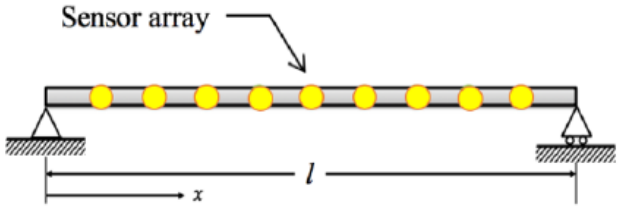


Deep Learning

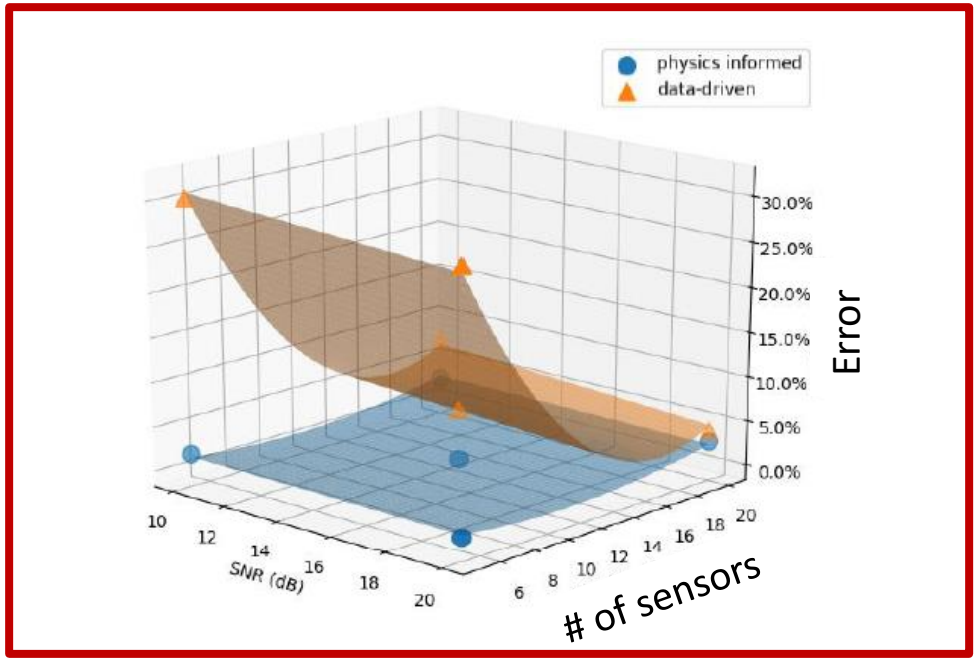


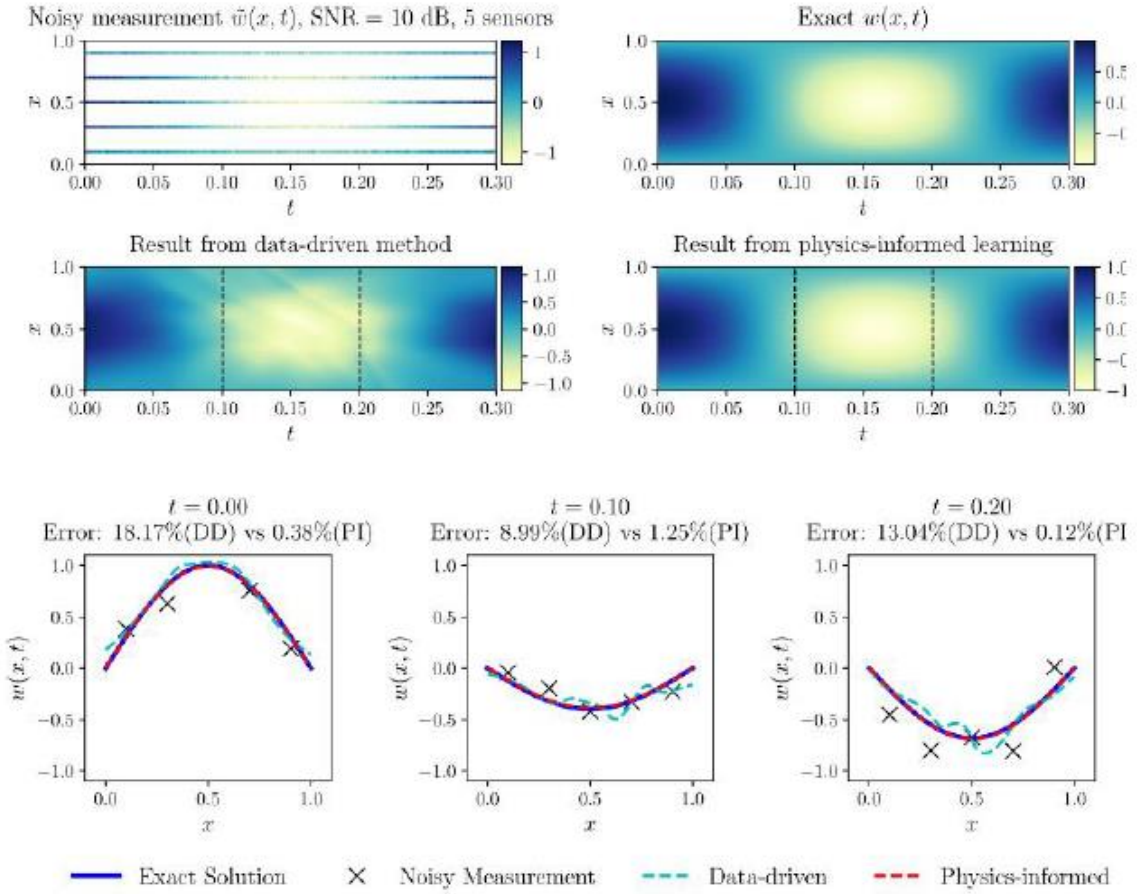
Displacement Field Reconstruction

- Reconstruct full displacement field from the sparse measurements and embedding the governing equations into loss functions



$$EI\frac{\partial^4\omega}{\partial x^4}+\mu\frac{\partial^2\omega}{\partial t^2}=q$$

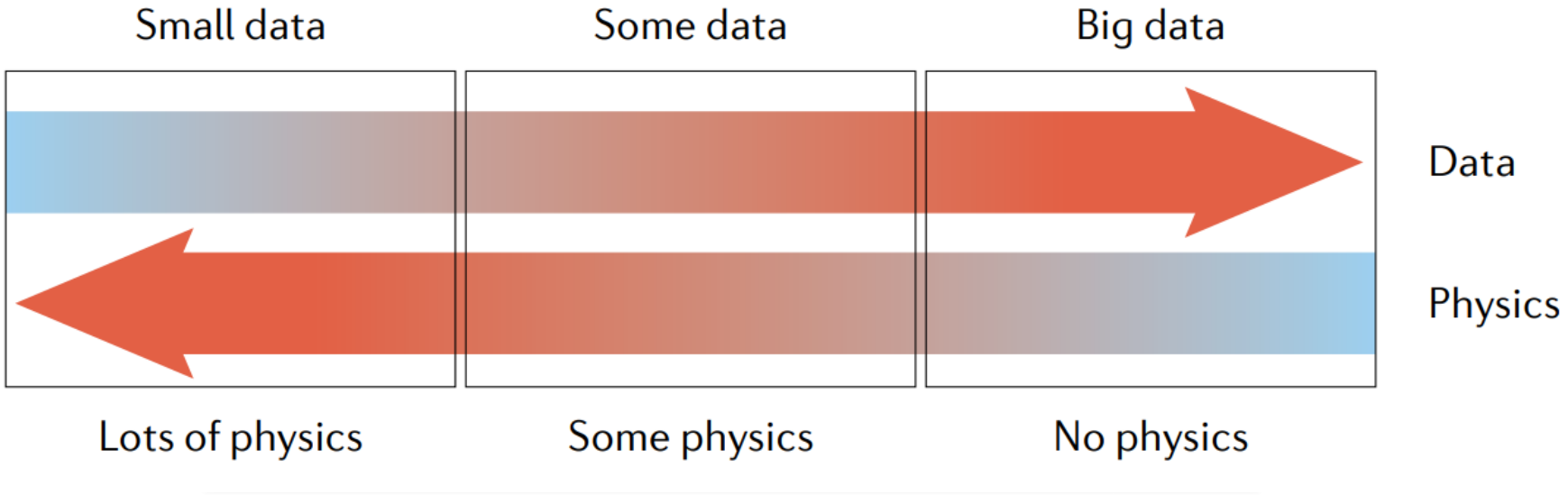




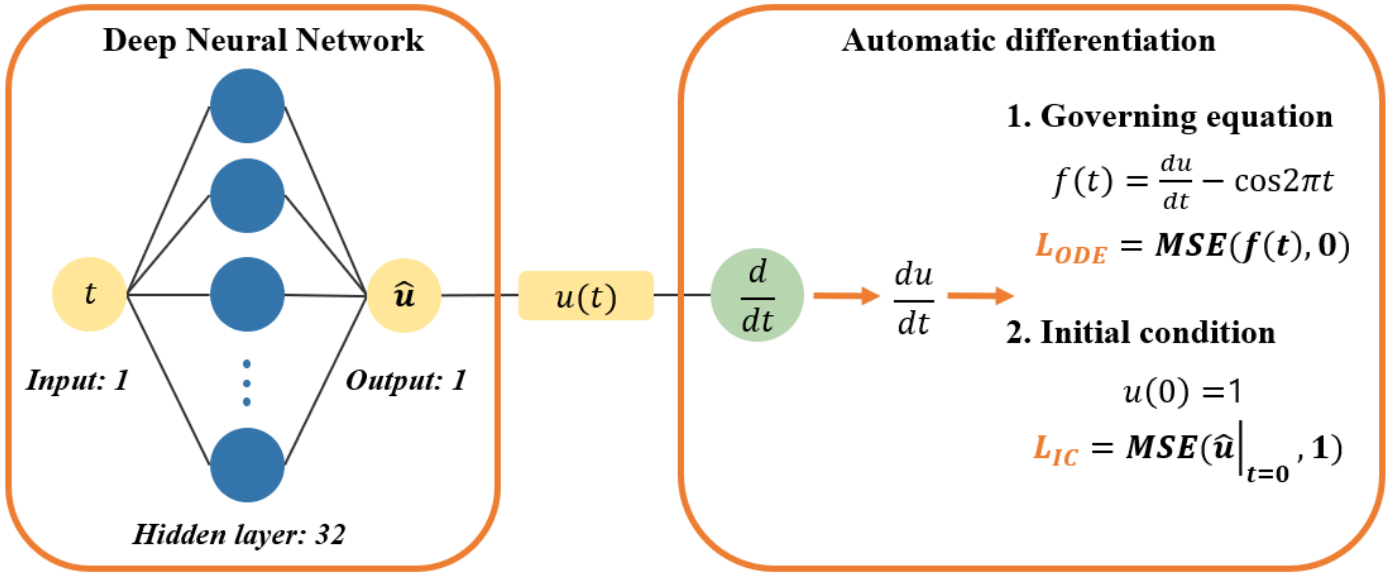
Yuan et al. Machine learning for structural health monitoring: challenges and opportunities, 2020

Summary of Physics-informed AI

- Leverage inductive bias and sparsity



Lab 1: Simple Example



$L_{total} = L_{ODE} + L_{IC}$

- Let's look at the ODE

$$\frac{du}{dt} = \cos 2\pi t$$

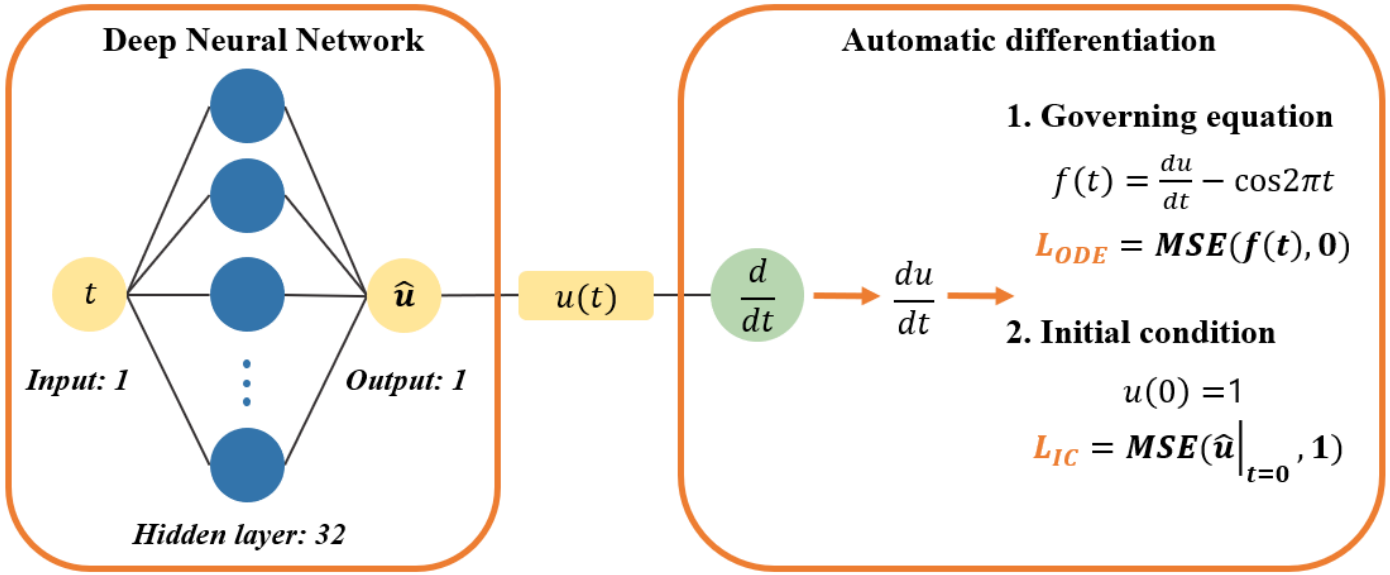
- Initial condition

$$u(0) = 1$$

- The exact solution

$$u(t) = \frac{1}{2\pi}\sin 2\pi t + 1$$

Lab 1: Simple Example



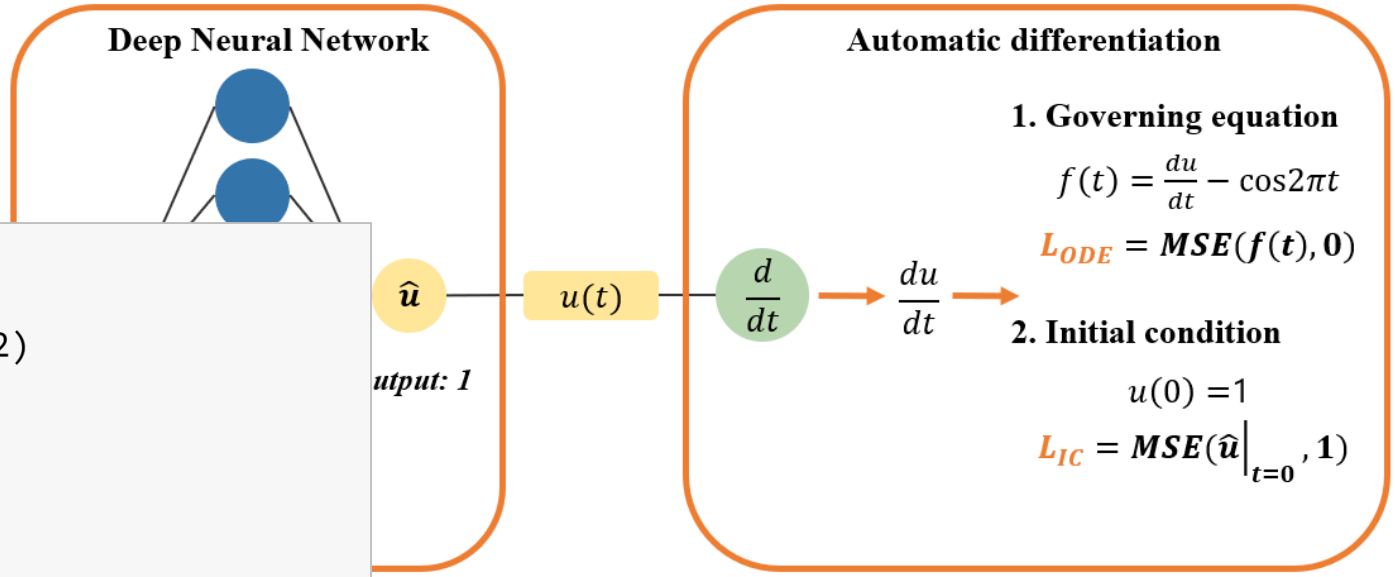
$$L_{total} = L_{ODE} + L_{IC}$$

```
NN = tf.keras.models.Sequential([
    tf.keras.layers.Input((1,)),
    tf.keras.layers.Dense(units = 32, activation = 'tanh'),
    tf.keras.layers.Dense(units = 1)
])
```

```
optm = tf.keras.optimizers.Adam(learning_rate = 0.001)
```

Lab 1: Simple Example

```python
def ode_system(t, net):
    t = t.reshape(-1,1)
    t = tf.constant(t, dtype = tf.float32)
    t_0 = tf.zeros((1,1))
    one = tf.ones((1,1))

    with tf.GradientTape() as tape:
        tape.watch(t)

        u = net(t)
        u_t = tape.gradient(u, t)

    ode_loss = u_t - tf.math.cos(2*np.pi*t)
    IC_loss = net(t_0) - one

    square_loss = tf.square(ode_loss) + tf.square(IC_loss)
    total_loss = tf.reduce_mean(square_loss)

    return total_loss
```



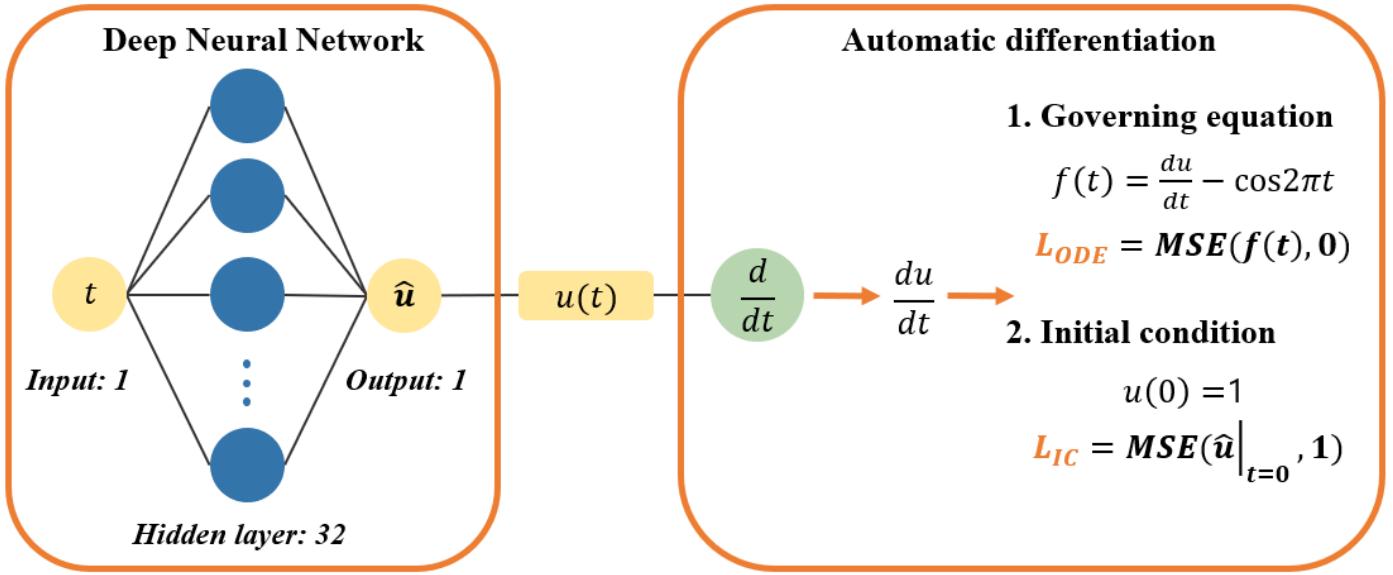
$$L_{total} = L_{ODE} + L_{IC}$$

$$L_{\mathrm{ODE}} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{d\mathrm{NN}(t_i)}{dt} - \cos 2\pi t_i\right)^2$$

$$L_{IC} = \frac{1}{n}\sum_{i=1}^{n}(\mathrm{NN}(0) - 1)^2$$

$$L_{\mathrm{Total}} = L_{\mathrm{ODE}} + L_{\mathrm{IC}}$$

Lab 1: Simple Example




```
train_t = (np.random.rand(30)*2).reshape(-1, 1)
train_loss_record = []

for itr in range(6000):
    with tf.GradientTape() as tape:
        train_loss = ode_system(train_t, NN)
        train_loss_record.append(train_loss)

        grad_w = tape.gradient(train_loss, NN.trainable_variables)
        optm.apply_gradients(zip(grad_w, NN.trainable_variables))
```

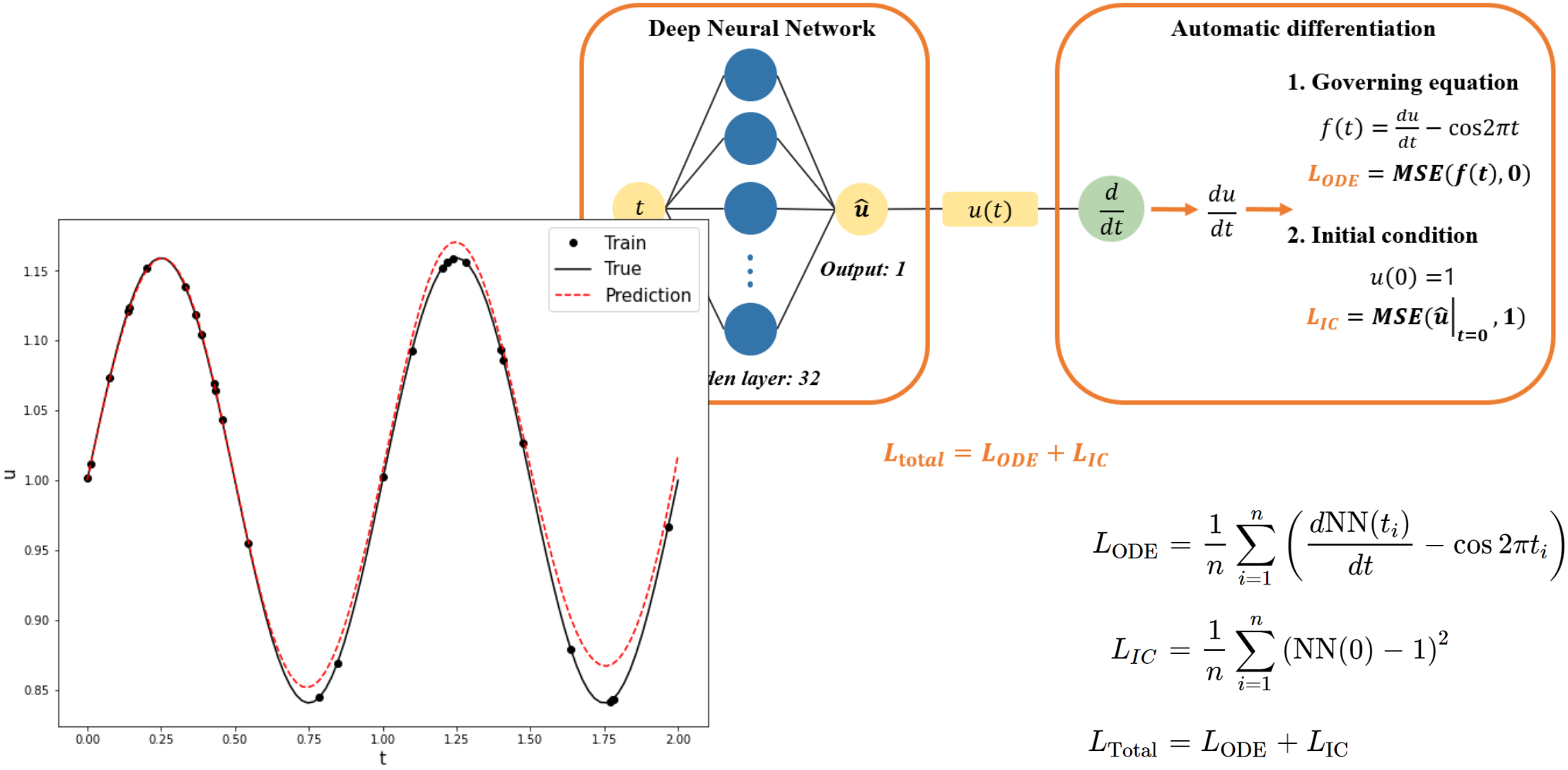
$= L_{ODE} + L_{IC}$

$$L_{\text{ODE}} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{d\text{NN}(t_i)}{dt} - \cos 2\pi t_i\right)^2$$

$$L_{IC} = \frac{1}{n}\sum_{i=1}^{n}(\text{NN}(0) - 1)^2$$

$$L_{\text{Total}} = L_{\text{ODE}} + L_{\text{IC}}$$

Lab 1: Simple Example



$$L_{total} = L_{ODE} + L_{IC}$$

$$L_{\text{ODE}} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{d\text{NN}(t_i)}{dt} - \cos 2\pi t_i\right)^2$$

$$L_{IC} = \frac{1}{n}\sum_{i=1}^{n}(\text{NN}(0) - 1)^2$$

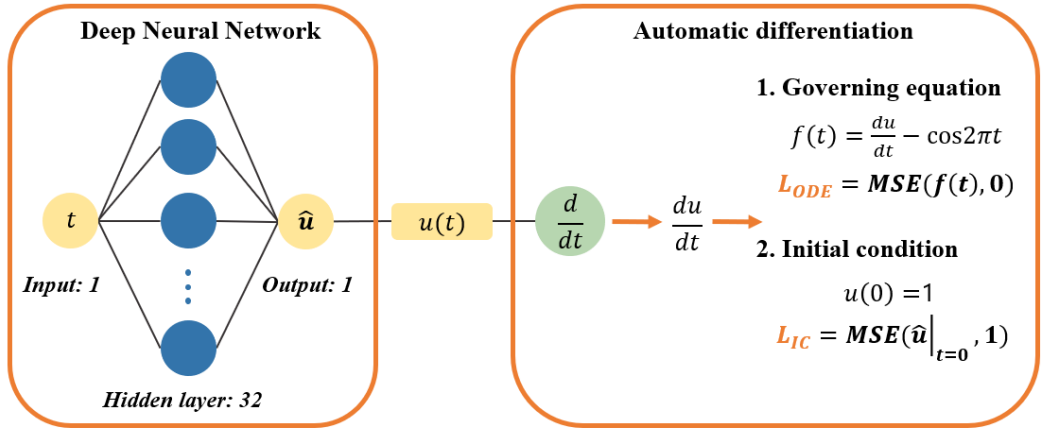
$$L_{\text{Total}} = L_{\text{ODE}} + L_{\text{IC}}$$

Lab 2: Solve Lab 1 Again using DeepXDE

- Embed a PDE into the loss of the neural network using automatic differentiation
- Compare the PINN algorithm to a standard finite element method (FEM)

- DeepXDE: A Deep Learning Library for Solving Differential Equations
  - DeepXDE — DeepXDE 0.14.0 documentation
  - Library for solving forward and inverse partial differential equations (PDEs) via physics-informed neural network (PINN)

Lab 2: Solve Lab 1 Again using DeepXDE



$$L_{total} = L_{ODE} + L_{IC}$$

```
NN = tf.keras.models.Sequential([
    tf.keras.layers.Input((1,)),
    tf.keras.layers.Dense(units = 32, activation = 'tanh'),
    tf.keras.layers.Dense(units = 1)
])
```

```
optm = tf.keras.optimizers.Adam(learning_rate = 0.001)
```

```
layer_size = [1] + [32] + [1]
activation = "tanh"
initializer = "Glorot uniform"

NN = dde.maps.FNN(layer_size, activation, initializer)
```

```
model = dde.Model(data, NN)
model.compile("adam", lr = 0.001)
```

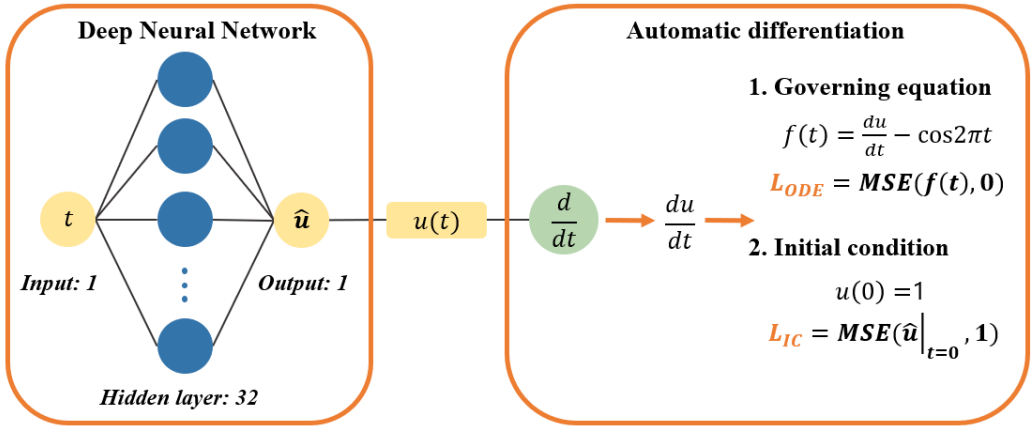
Lab 2: Solve Lab 1 Again using DeepXDE

```python
def ode_system(t, net):
    t = t.reshape(-1,1)
    t = tf.constant(t, dtype = tf.float32)
    t_0 = tf.zeros((1,1))
    one = tf.ones((1,1))

    with tf.GradientTape() as tape:
        tape.watch(t)

        u = net(t)
        u_t = tape.gradient(u, t)

    ode_loss = u_t - tf.math.cos(2*np.pi*t)
    IC_loss = net(t_0) - one

    square_loss = tf.square(ode_loss) + tf.square(IC_loss)
    total_loss = tf.reduce_mean(square_loss)

    return total_loss
```

```python
def ode_system(t, u):
    du_t = dde.grad.jacobian(u, t)
    return du_t - tf.math.cos(2*pi*t)
```

```python
def boundary(t, on_initial):
    return on_initial and np.isclose(t[0], 0)
```

Lab 2: Solve Lab 1 Again using DeepXDE



$$L_{total} = L_{ODE} + L_{IC}$$

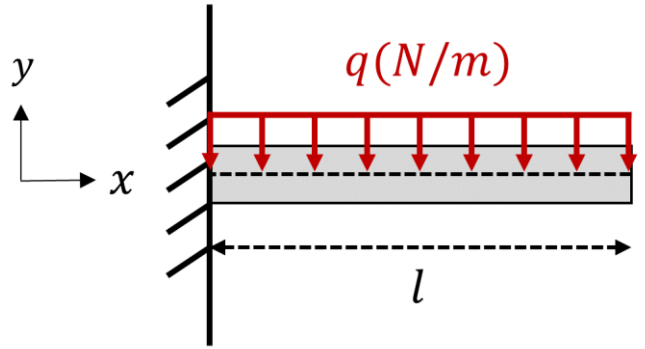
```
train_t = (np.random.rand(30)*2).reshape(-1, 1)
train_loss_record = []

for itr in range(6000):
    with tf.GradientTape() as tape:
        train_loss = ode_system(train_t, NN)
        train_loss_record.append(train_loss)

        grad_w = tape.gradient(train_loss, NN.trainable_variables)
        optm.apply_gradients(zip(grad_w, NN.trainable_variables))
```

```
geom = dde.geometry.TimeDomain(0, 2)

ic = dde.IC(geom, lambda t: 1, boundary)

# Reference solution to compute the error
def true_solution(t):
    return np.sin(2*np.pi*t)/(2*np.pi) + 1

data = dde.data.PDE(geom,
                    ode_system,
                    ic,
                    num_domain = 30,
                    num_boundary = 2,
                    solution = true_solution,
                    num_test = 100)
```

```
losshistory, train_state = model.train(epochs = 6000)
```

Lab 3: Euler Beam (Solid Mechanics)

$$E = 1\,\mathrm{pa}, \quad I = 1\,\mathrm{kg}\cdot\mathrm{m}^2, \quad q = 1\,\mathrm{N/m}, \quad l = 1\,\mathrm{m}$$



- Partial differential equations & boundary conditions

$$\frac{\partial^4 y}{\partial x^4} + 1 = 0, \qquad \text{where} \quad x \in [0, 1]$$

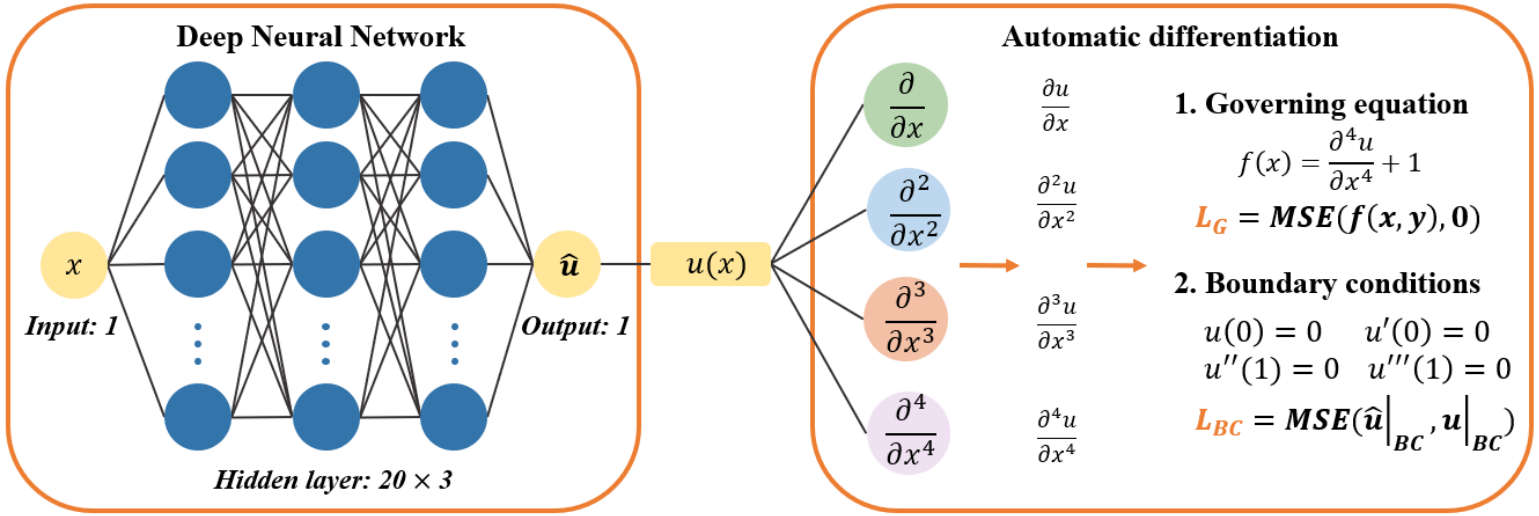
- One Dirichlet boundary condition on the left boundary: $y(0) = 0$
- One Neumann boundary condition on the left boundary: $y'(0) = 0$
- Two boundary conditions on the right boundary: $y''(1) = 0, \quad y'''(1) = 0$

- The exact solution is

$$y(x) = -\frac{1}{24}x^4 + \frac{1}{6}x^3 - \frac{1}{4}x^2$$

Lab 3: Euler Beam (Solid Mechanics)

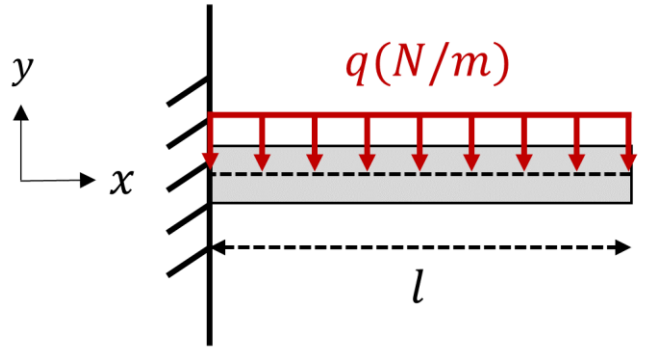
- Make a neural network and loss functions



$$L_{total} = L_G + L_{BC}$$

```
layer_size = [1] + [20] * 3 + [1]
activation = "tanh"
initializer = "Glorot uniform"

net = dde.maps.FNN(layer_size, activation, initializer)
```

Lab 3: Euler Beam (Solid Mechanics)



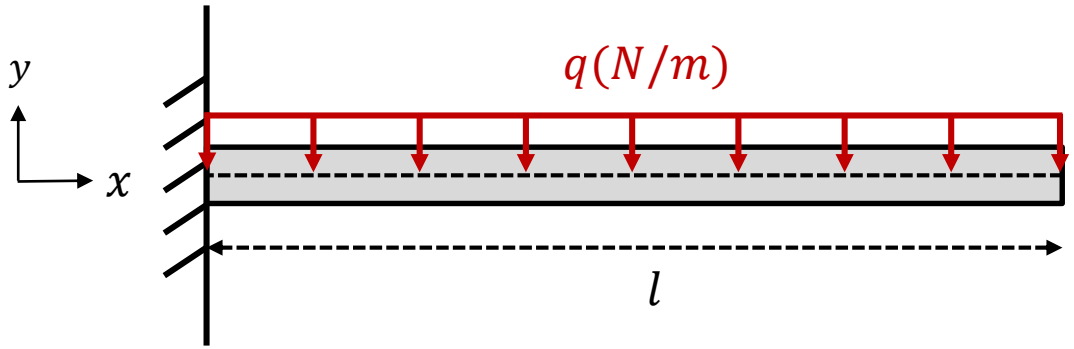

```python
def ddy(x, y):
    return dde.grad.hessian(y, x)

def dddy(x, y):
    return dde.grad.jacobian(ddy(x, y), x)

def pde(x, y):
    dy_xx = ddy(x, y)
    dy_xxxx = dde.grad.hessian(dy_xx, x)
    return dy_xxxx + 1
```

$$\frac{\partial^4 y}{\partial x^4} + 1 = 0$$

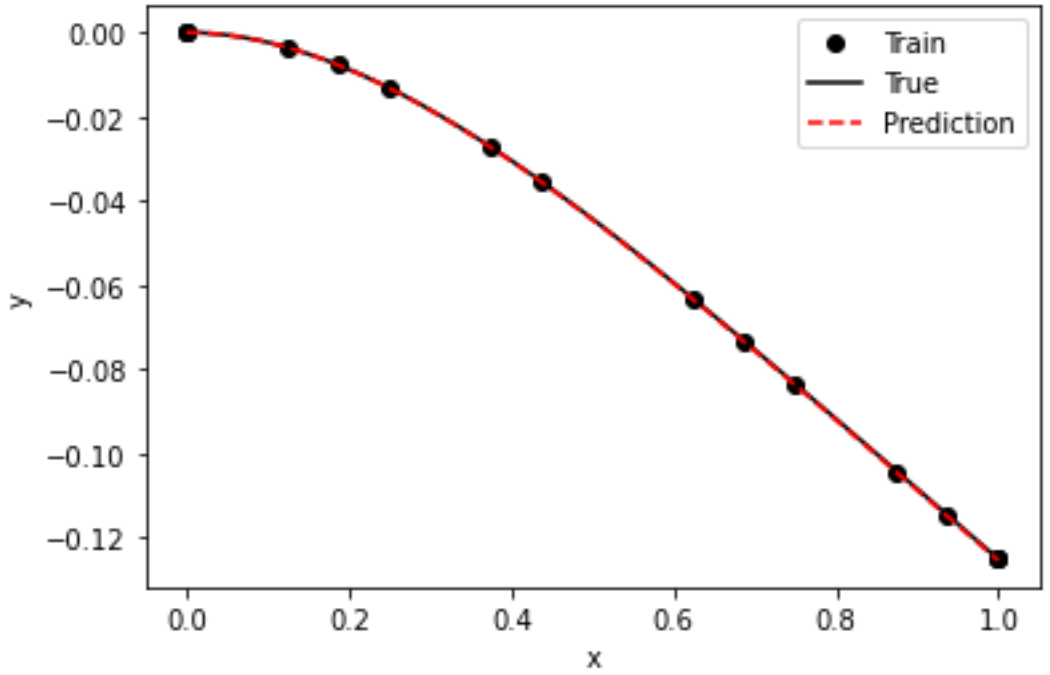
```python
def boundary_left(x, on_boundary):
    return on_boundary and np.isclose(x[0], 0)

def boundary_right(x, on_boundary):
    return on_boundary and np.isclose(x[0], 1)
```

```python
geom = dde.geometry.Interval(0, 1)

bc1 = dde.DirichletBC(geom, lambda x: 0, boundary_left) # u(0) = 0
bc2 = dde.NeumannBC(geom, lambda x: 0, boundary_left) # u'(0) = 0
bc3 = dde.OperatorBC(geom, lambda x, y, _: ddy(x, y), boundary_right) # u''(1) = 0
bc4 = dde.OperatorBC(geom, lambda x, y, _: dddy(x, y), boundary_right) # u'''(1) = 0

# Reference solution to compute the error
def true_solution(x):
    return -(x ** 4) / 24 + x ** 3 / 6 - x ** 2 / 4

data = dde.data.PDE(geom,
                    pde,
                    [bc1, bc2, bc3, bc4],
                    num_domain = 10,
                    num_boundary = 2,
                    solution = true_solution,
                    num_test = 100)
```

Lab 3: Euler Beam (Solid Mechanics)

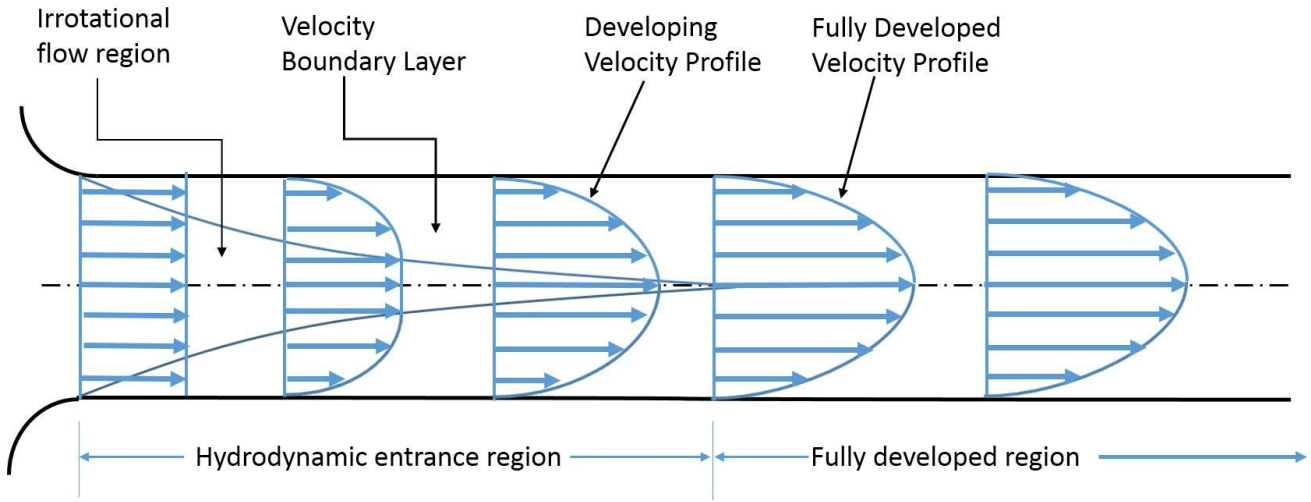


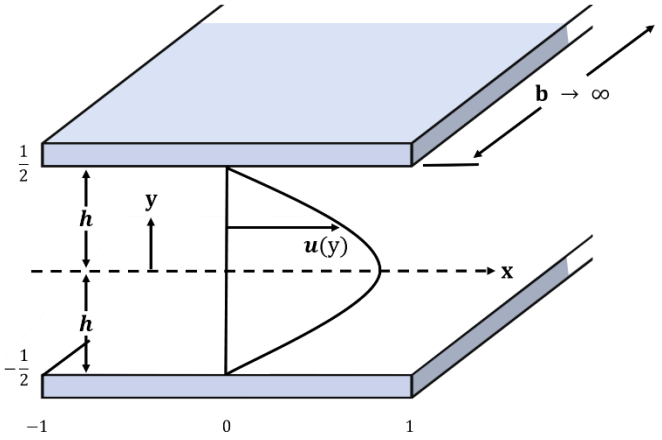
$$y(x) = -\frac{1}{24}x^4 + \frac{1}{6}x^3 - \frac{1}{4}x^2$$



Lab 4: Navier-Stokes Equations (Fluid Mechanics)

- To solve 2D Navier-Stokes Equations to find velocity profile in infinite parallel plates flow
  - The fluid typically enters the plates with a nearly uniform velocity profile
  - As the fluid moves through the plates, viscous effects cause it to stick to the plates wall (no-slip boundary condition)
  - Thus, a boundary layer is produced along the plates wall such that the initial velocity profile changes with distance along the plates, x , until the fluid reaches the end of the hydrodynamic entrance region (which is also called entrance length) beyond which the velocity profile does not vary with x





Lab 4: Navier-Stokes Equations (Fluid Mechanics)

$$\rho = 1\,\mathrm{kg/m^3}, \quad \mu = 1\,\mathrm{N\cdot s/m^2}, \quad D = 2h = 1\,\mathrm{m}, \quad L = 2\,\mathrm{m}, \quad u_{in} = 1\,\mathrm{m/s}, \quad \nu = \frac{\mu}{\rho}$$

- Hydraulic diameter is $D_h = \lim_{b\to\infty} \dfrac{4(2bh)}{2b+4h} = 4h = 2\,\mathrm{m}$

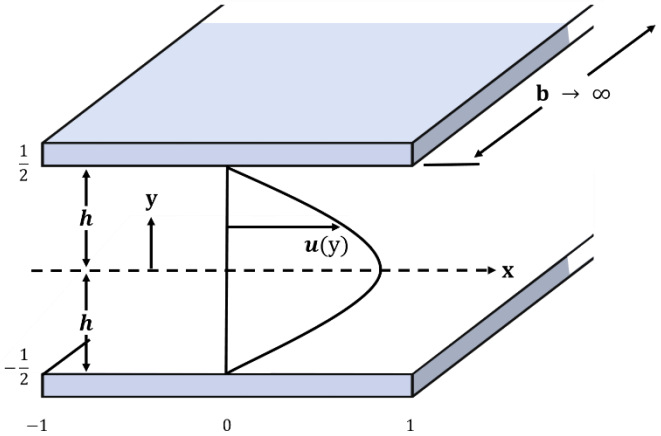
- The Reynolds number of this system is $Re = \dfrac{\rho u_{in} D_h}{\mu} = 2$

- 2D Navier-Stokes Equations & boundary conditions (for steady state)

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + \frac{1}{\rho}\frac{\partial p}{\partial x} - \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0$$

$$u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + \frac{1}{\rho}\frac{\partial p}{\partial y} - \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) = 0$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$$



Lab 4: Navier-Stokes Equations (Fluid Mechanics)

- Two Dirichlet boundary conditions on the plate boundary (no-slip condition)

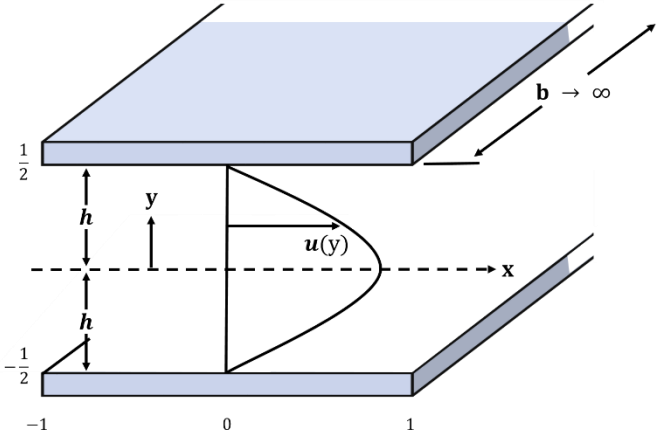
$$u(x,y) = 0, \quad v(x,y) = 0 \qquad \text{at} \quad y = \frac{D}{2} \ \text{or} \ -\frac{D}{2}$$

- Two Dirichlet boundary conditions at the inlet boundary

$$u(-1,y) = u_{\text{in}}, \quad v(-1,y) = 0$$

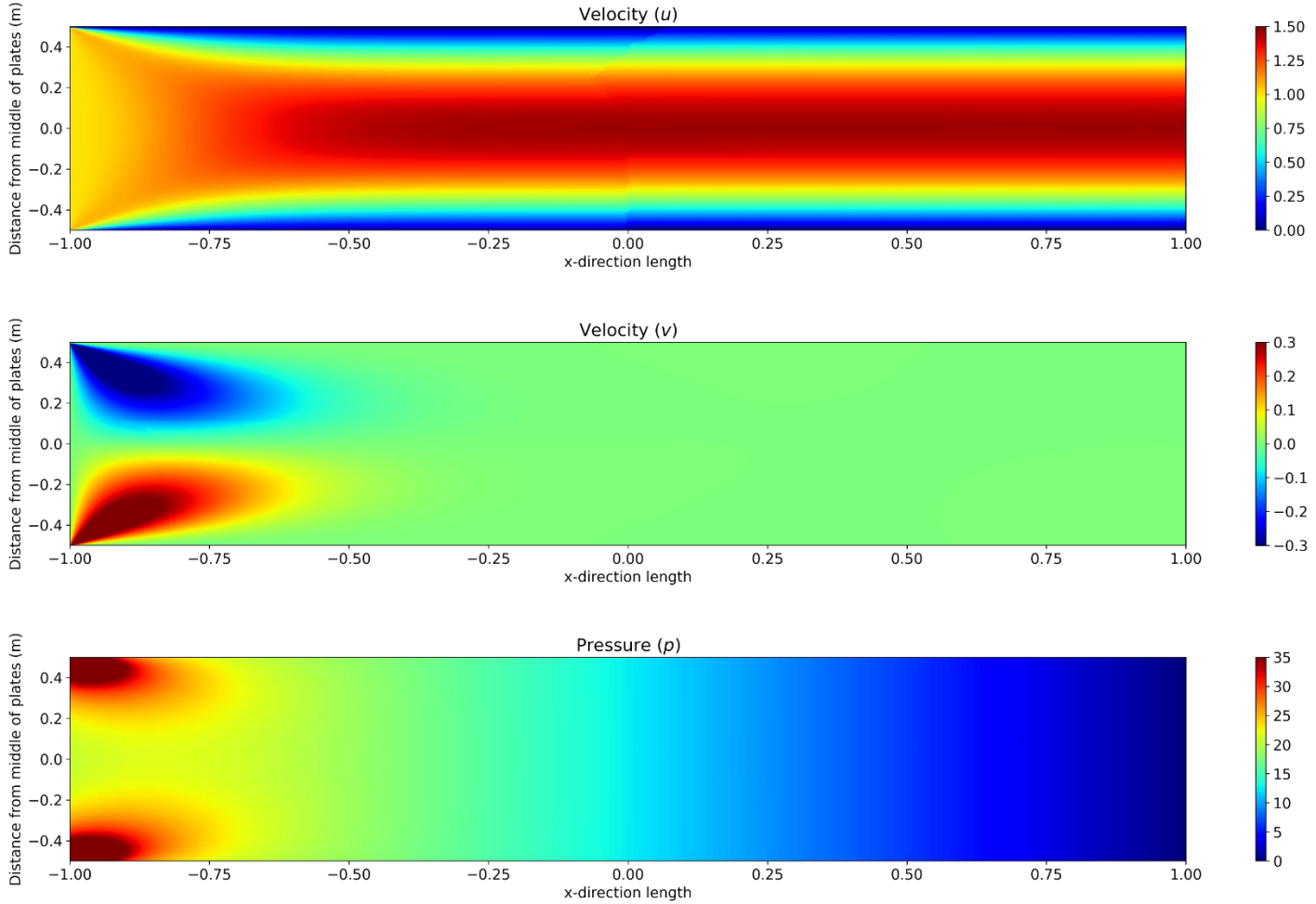
- Two Dirichlet boundary conditions at the outlet boundary

$$p(1,y) = 0, \quad v(1,y) = 0$$



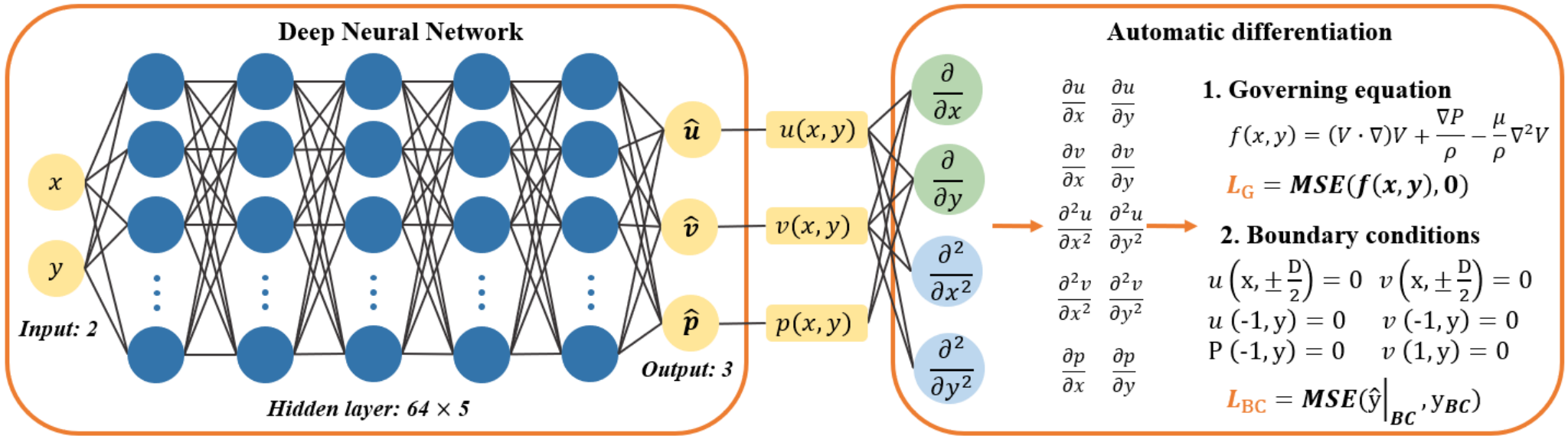
CFD Solution

- Velocity u and velocity v, and pressure p, respectively



Lab 4: Navier-Stokes Equations (Fluid Mechanics)

- Make a neural network and loss functions



$X\ (input)$ $\quad$ $Y\ (ouput)$ $\quad$ $L_{total} = L_G + L_{BC}$

$$\begin{bmatrix} x \\ y \end{bmatrix} \qquad \begin{bmatrix} \hat{u} \\ \hat{v} \\ \hat{p} \end{bmatrix}$$

Lab 4: Navier-Stokes Equations (Fluid Mechanics)

```python
def boundary_wall(X, on_boundary):
    on_wall = np.logical_and(np.logical_or(np.isclose(X[1], -D/2), np.isclose(X[1], D/2)), on_boundary)
    return on_wall

def boundary_inlet(X, on_boundary):
    return on_boundary and np.isclose(X[0], -L/2)

def boundary_outlet(X, on_boundary):
    return on_boundary and np.isclose(X[0], L/2)
```

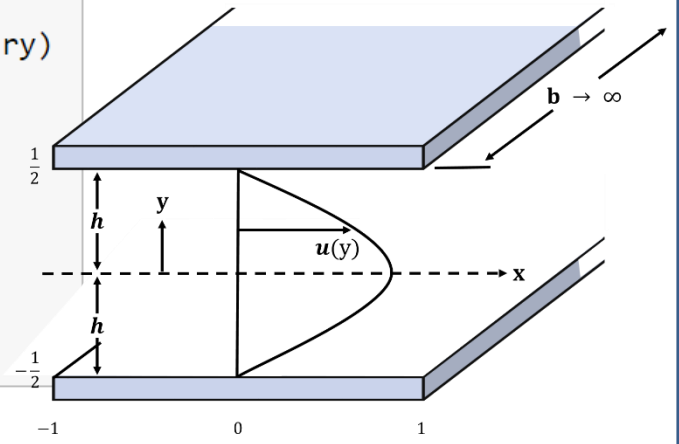
```python
def pde(X, Y):
    du_x = dde.grad.jacobian(Y, X, i = 0, j = 0)
    du_y = dde.grad.jacobian(Y, X, i = 0, j = 1)
    dv_x = dde.grad.jacobian(Y, X, i = 1, j = 0)
    dv_y = dde.grad.jacobian(Y, X, i = 1, j = 1)
    dp_x = dde.grad.jacobian(Y, X, i = 2, j = 0)
    dp_y = dde.grad.jacobian(Y, X, i = 2, j = 1)
    du_xx = dde.grad.hessian(Y, X, i = 0, j = 0, component = 0)
    du_yy = dde.grad.hessian(Y, X, i = 1, j = 1, component = 0)
    dv_xx = dde.grad.hessian(Y, X, i = 0, j = 0, component = 1)
    dv_yy = dde.grad.hessian(Y, X, i = 1, j = 1, component = 1)

    pde_u = Y[:,0:1]*du_x + Y[:,1:2]*du_y + 1/rho * dp_x - (mu/rho)*(du_xx + du_yy)
    pde_v = Y[:,0:1]*dv_x + Y[:,1:2]*dv_y + 1/rho * dp_y - (mu/rho)*(dv_xx + dv_yy)
    pde_cont = du_x + dv_y

    return [pde_u, pde_v, pde_cont]
```

$$Input\ X = [x, y]$$
$$Output\ Y = [u, v\ p]$$

$$\frac{\partial Y_i}{\partial X_j}$$

$$\frac{\partial Y_{i=1}}{\partial X_{j=1}} = \frac{\partial v}{\partial y}$$

$$\frac{\partial^2 Y_{component}}{\partial X_i \partial X_j} = \frac{\partial^2 Y_0}{\partial X_{i=0} \partial X_{j=0}} = \frac{\partial^2 u}{\partial x^2}$$



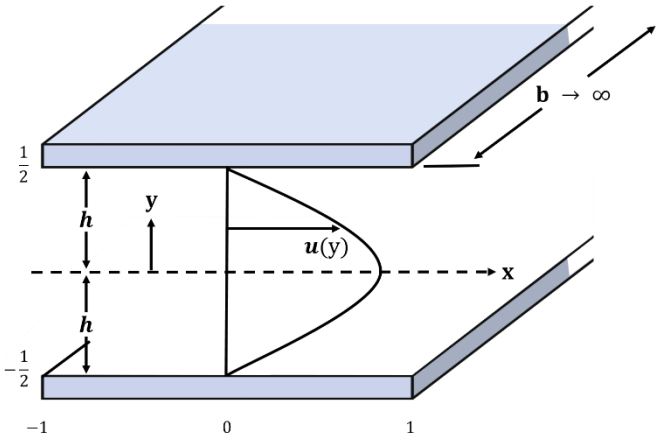
$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + \frac{1}{\rho}\frac{\partial p}{\partial x} - \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0$$

$$u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + \frac{1}{\rho}\frac{\partial p}{\partial y} - \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) = 0$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$$

Lab 4: Navier-Stokes Equations (Fluid Mechanics)

```
geom = dde.geometry.Rectangle(xmin=[-L/2, -D/2], xmax=[L/2, D/2])

bc_wall_u = dde.DirichletBC(geom, lambda X: 0., boundary_wall, component = 0)
bc_wall_v = dde.DirichletBC(geom, lambda X: 0., boundary_wall, component = 1)

bc_inlet_u = dde.DirichletBC(geom, lambda X: u_in, boundary_inlet, component = 0)
bc_inlet_v = dde.DirichletBC(geom, lambda X: 0., boundary_inlet, component = 1)

bc_outlet_p = dde.DirichletBC(geom, lambda X: 0., boundary_outlet, component = 2)
bc_outlet_v = dde.DirichletBC(geom, lambda X: 0., boundary_outlet, component = 1)
```

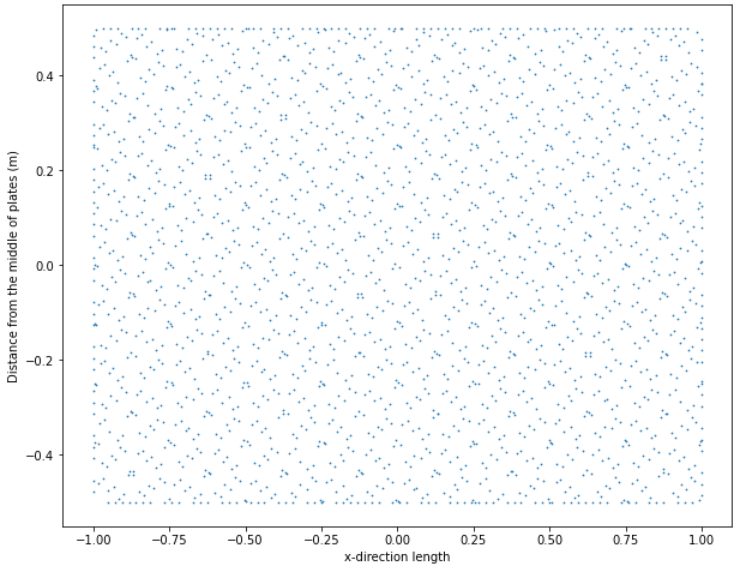



```
data = dde.data.PDE(geom,
                    pde,
                    [bc_wall_u, bc_wall_v, bc_inlet_u, bc_inlet_v, bc_outlet_p, bc_outlet_v],
                    num_domain = 2000,
                    num_boundary = 200,
                    num_test = 100)
```

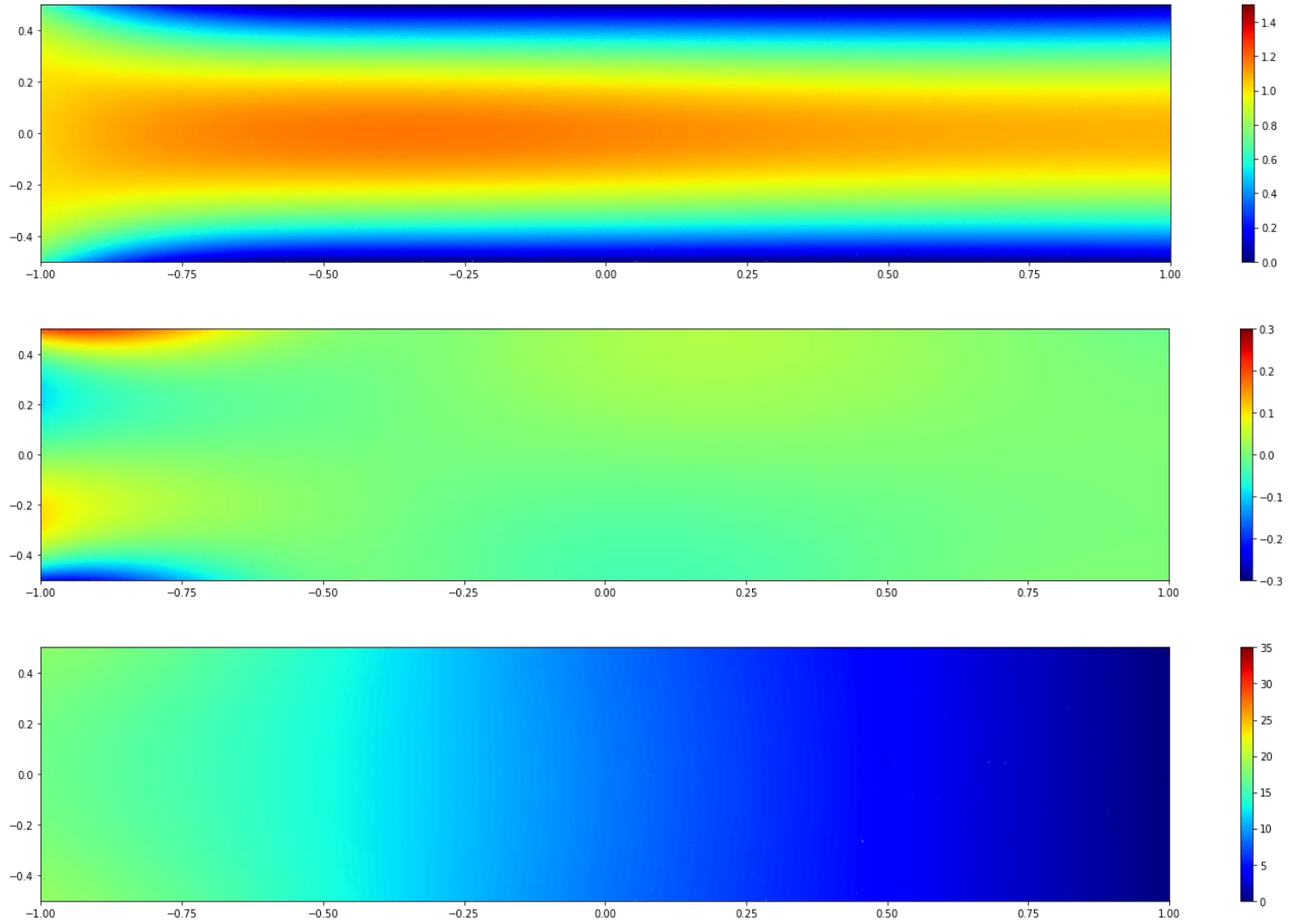
Lab 4: Navier-Stokes Equations (Fluid Mechanics)

```
plt.figure(figsize = (10,8))
plt.scatter(data.train_x_all[:,0], data.train_x_all[:,1], s = 0.5)
plt.xlabel('x-direction length')
plt.ylabel('Distance from the middle of plates (m)')
plt.show()
```

```
layer_size = [2] + [64] * 5 + [3]
activation = "tanh"
initializer = "Glorot uniform"

net = dde.maps.FNN(layer_size, activation, initializer)

model = dde.Model(data, net)
model.compile("adam", lr = 1e-3)
```



Plot Results (Adam Optimizer)

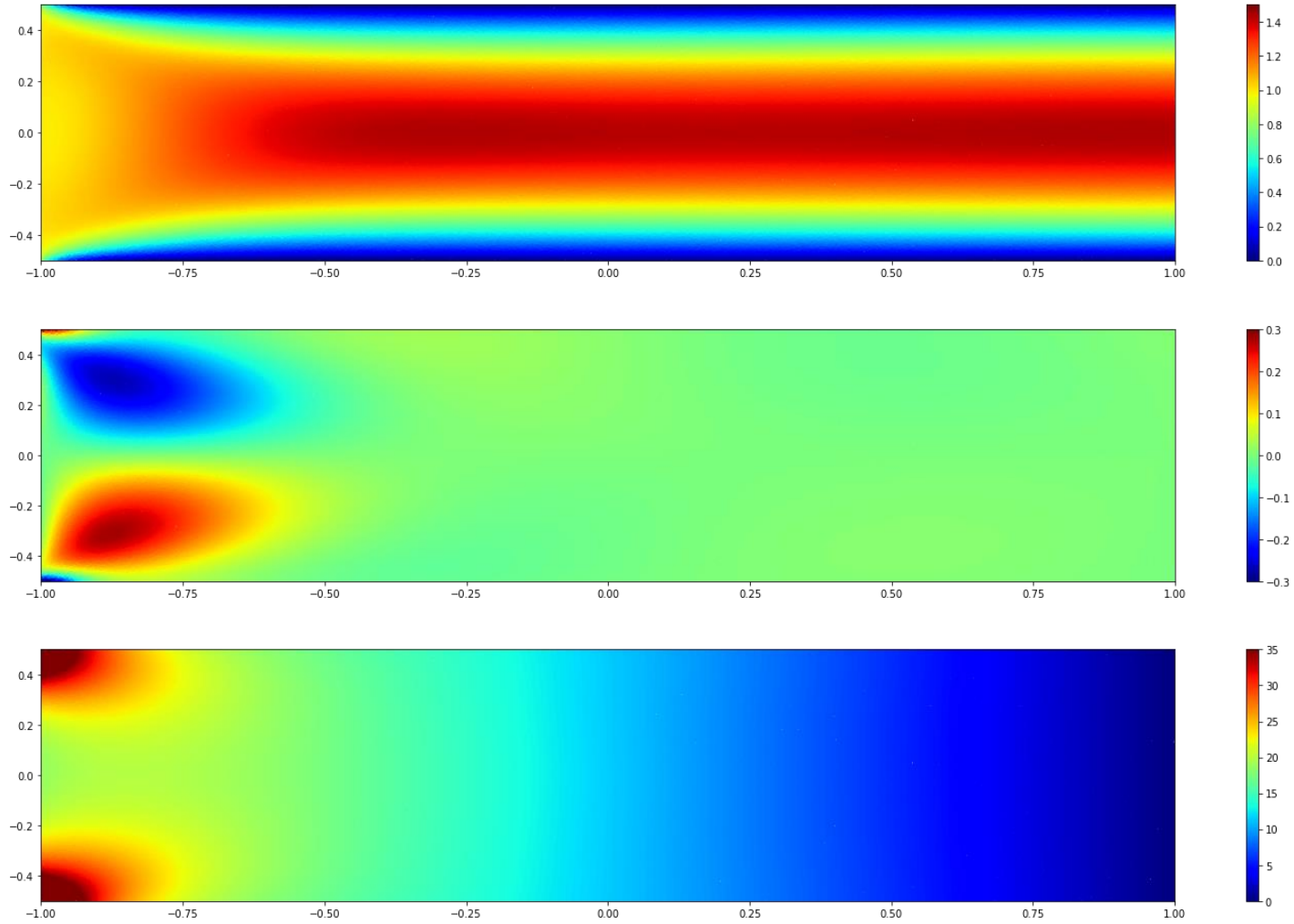


Train More (L-BFGS Optimizer)

- Limited-memory BFGS (L-BFGS or LM-BFGS) is an optimization algorithm in the family of quasi-Newton methods that approximates the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) using a limited amount of computer memory

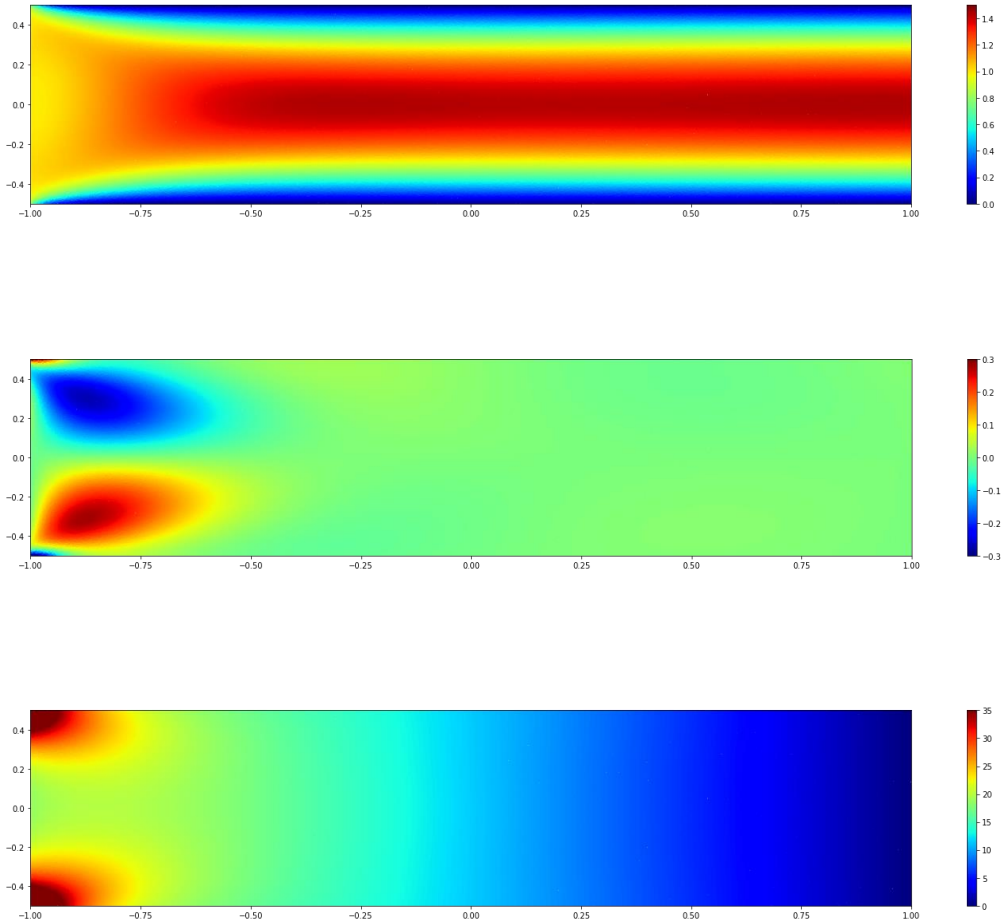
```
dde.optimizers.config.set_LBFGS_options(maxiter = 3000)

model.compile("L-BFGS")
losshistory, train_state = model.train()
dde.saveplot(losshistory, train_state, issave = False, isplot = True)
```

Plot Results (Adam + L-BFGS)

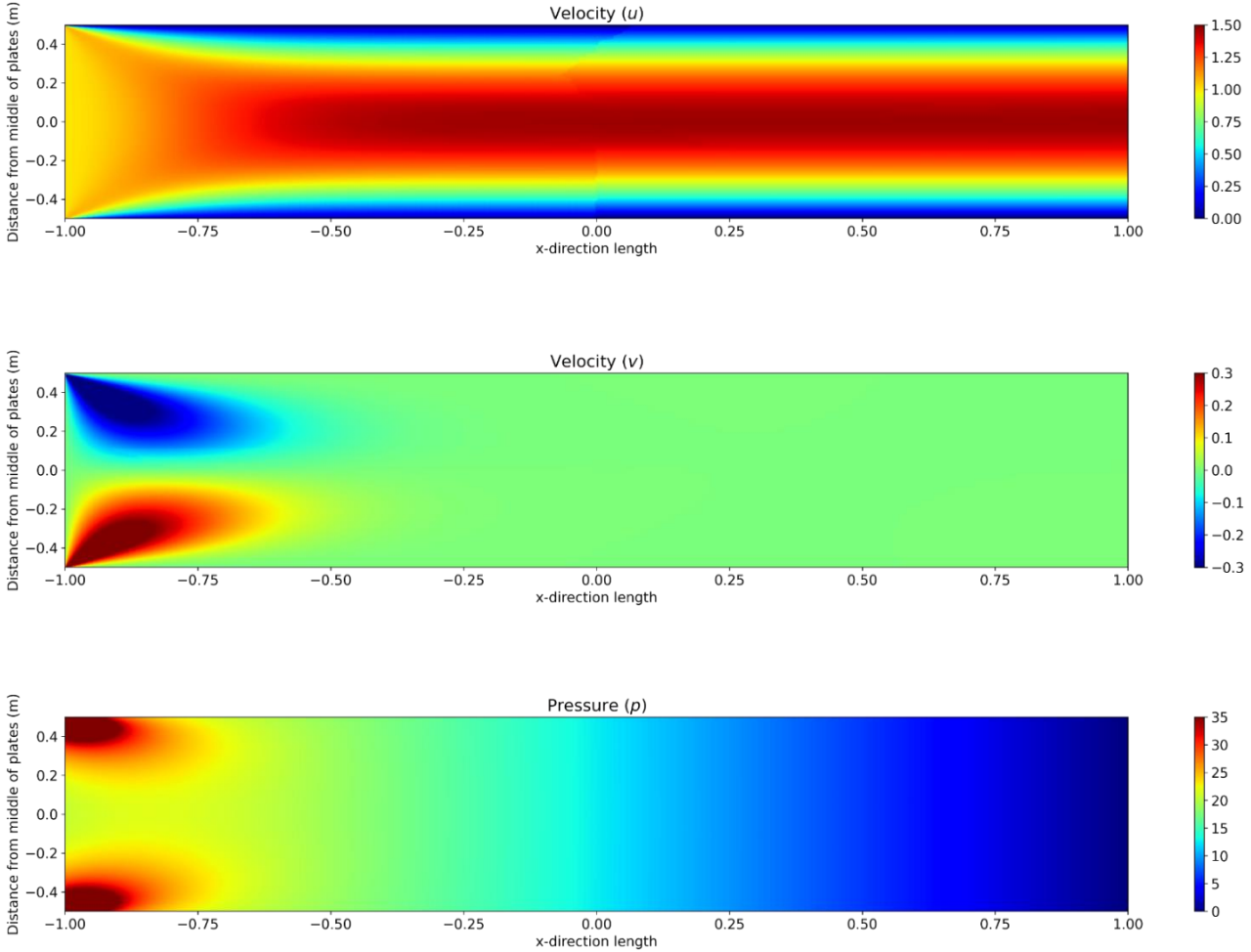


PINN vs. CFD

PINN



CFD



Validation: Velocity Profile at the End of the Plate

- Fully developed velocity profile at the infinite parallel plates flow are known as

$$u(y) = \frac{3V_{avg}}{2}\left[1 - \left(\frac{y}{h}\right)^2\right]$$