UK Atomic Energy Authority

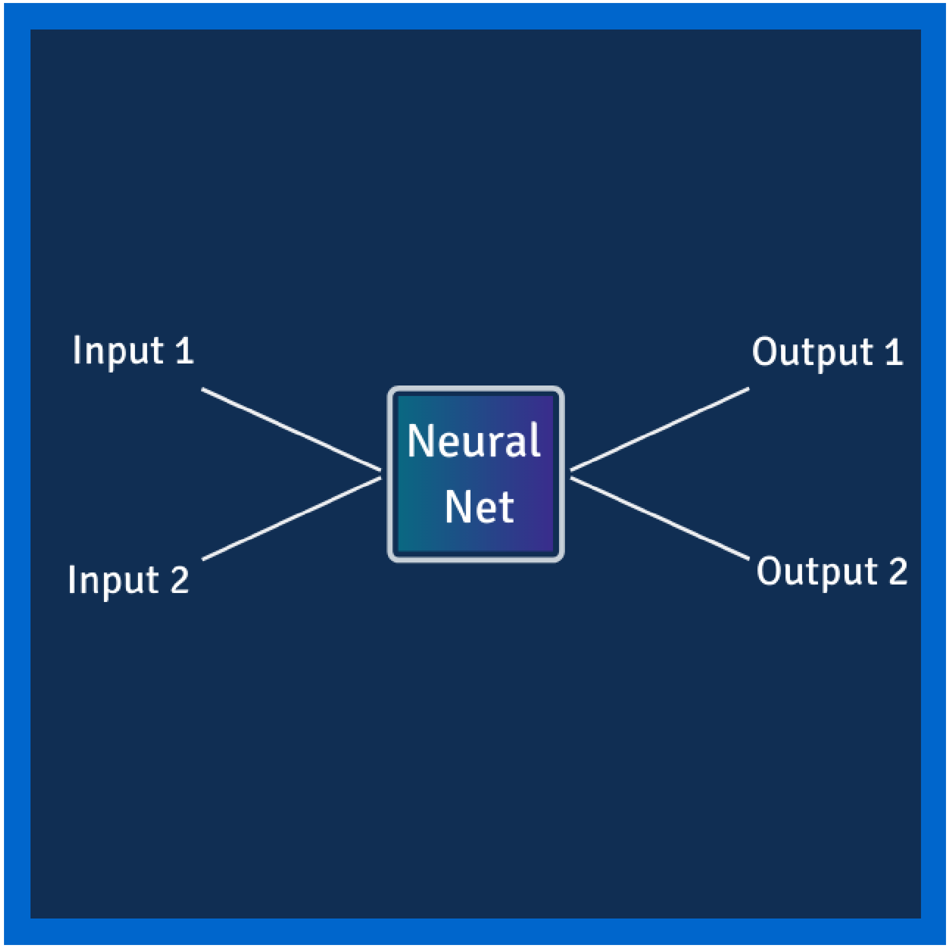
UKAEA

Neural PDEs

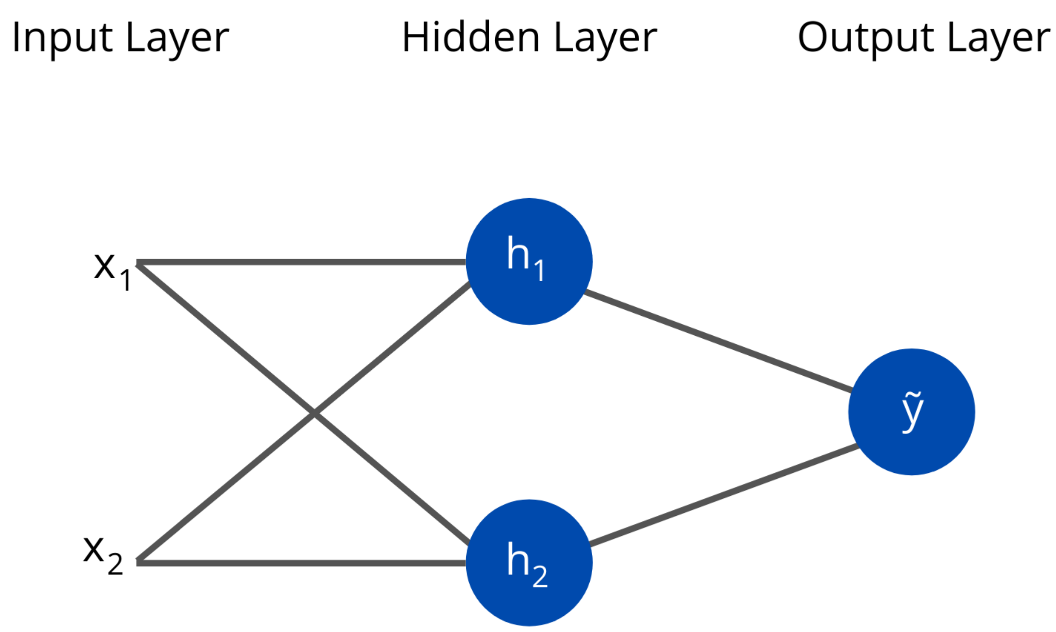
Vignesh Gopakumar

Fusion-EP Talks **28th May, 2020**

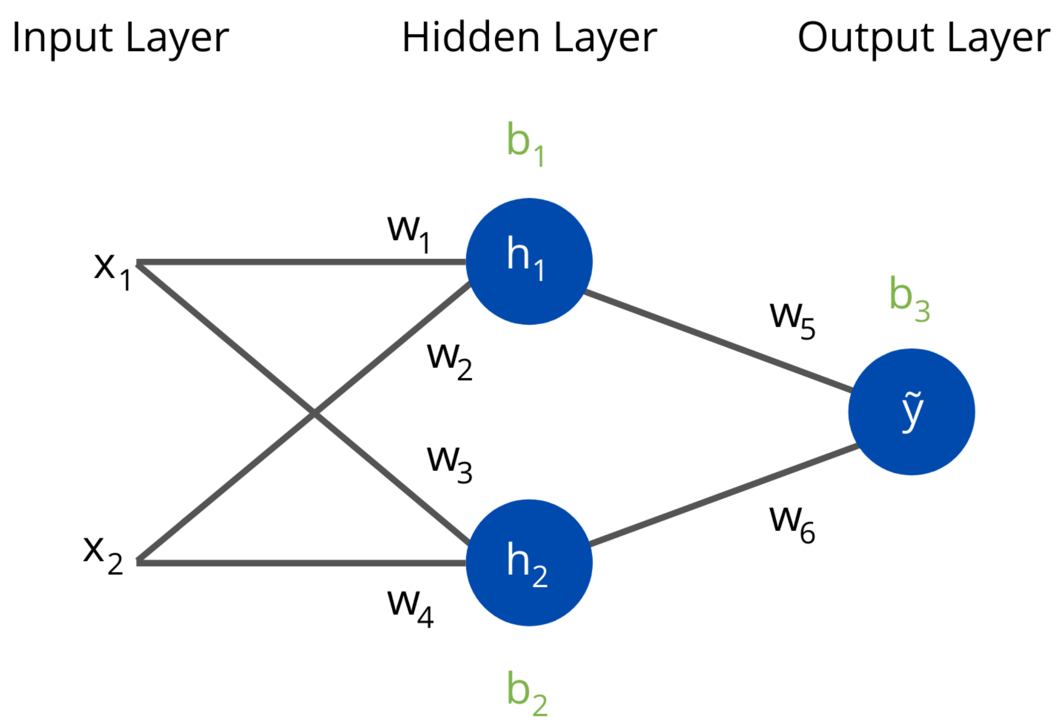
Regular NNs



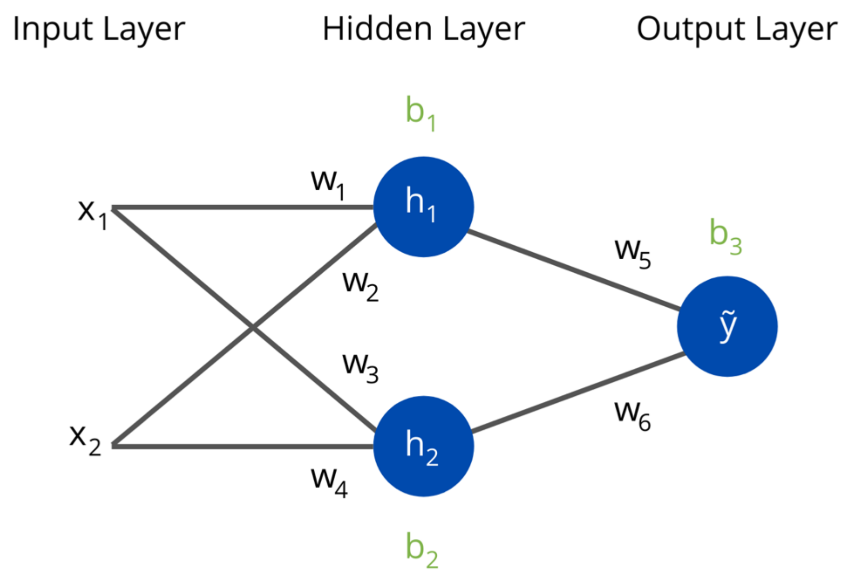
Feedforward Structure



Feedforward Structure



Feedforward Structure



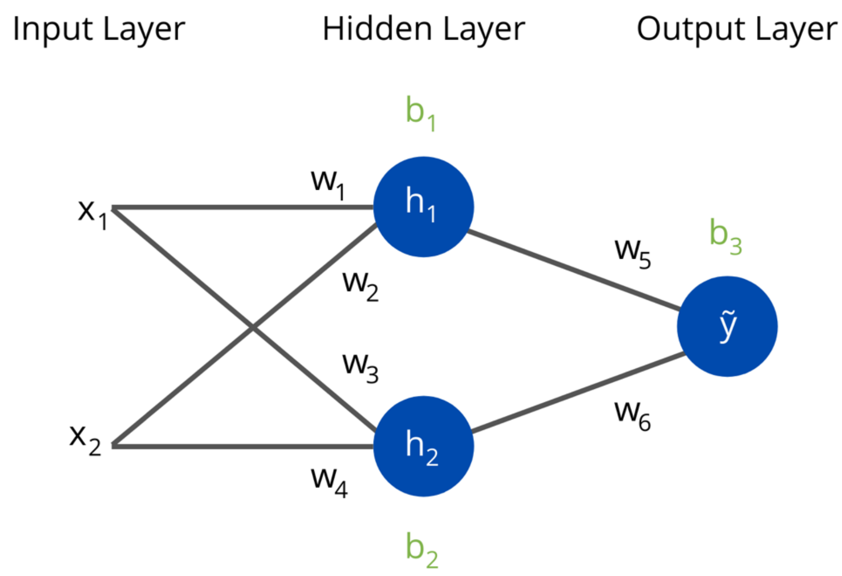
$$h_1 = f(b_1 + w_1 * x_1 + w_2 * x_2)$$

$$h_2 = f(b_2 + w_3 * x_1 + w_4 * x_2)$$

$$\tilde{y} = f(b_3 + w_5 * h_1 + w_6 * h_2)$$

Backpropagation

UK Atomic Energy Authority



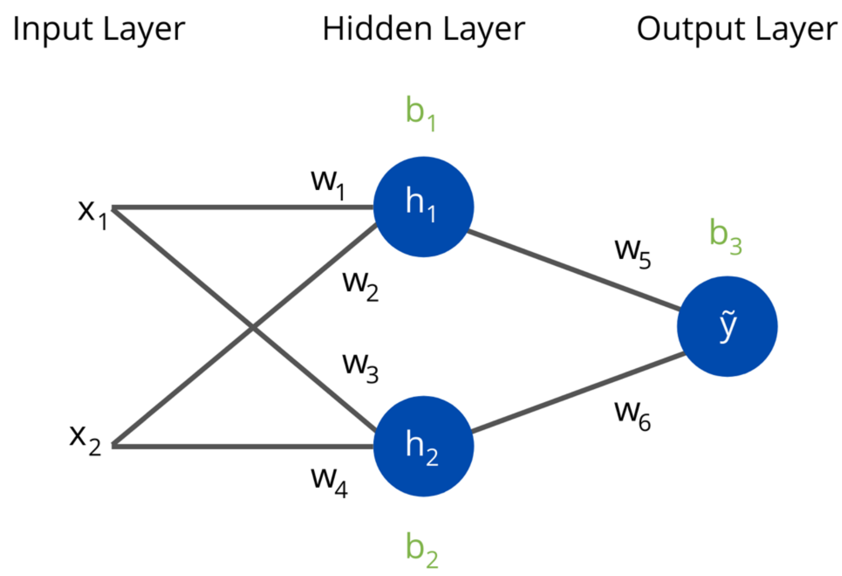
$$L = (y - \tilde{y})^2$$

$$h_1 = f(b_1 + w_1 * x_1 + w_2 * x_2)$$

$$h_2 = f(b_2 + w_3 * x_1 + w_4 * x_2)$$

$$\tilde{y} = f(b_3 + w_5 * h_1 + w_6 * h_2)$$

Backpropagation



$$L = (y - \tilde{y})^\wedge 2$$

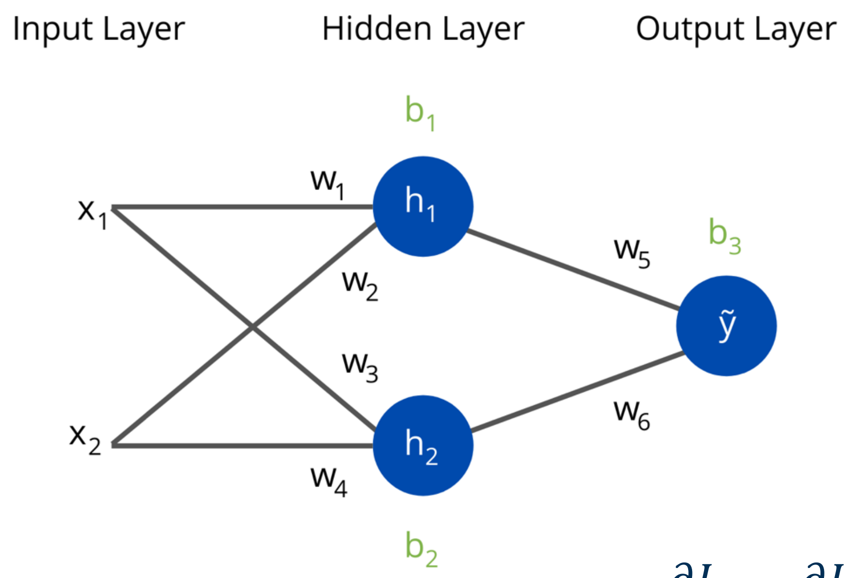
$$h_1 = f(b_1 + w_1 * x_1 + w_2 * x_2)$$

$$h_2 = f(b_2 + w_3 * x_1 + w_4 * x_2)$$

$$\tilde{y} = f(b_3 + w_5 * h_1 + w_6 * h_2)$$

$$\frac{\partial L}{\partial w} = ?$$

Backpropagation



$$L = (y - \tilde{y})^2$$

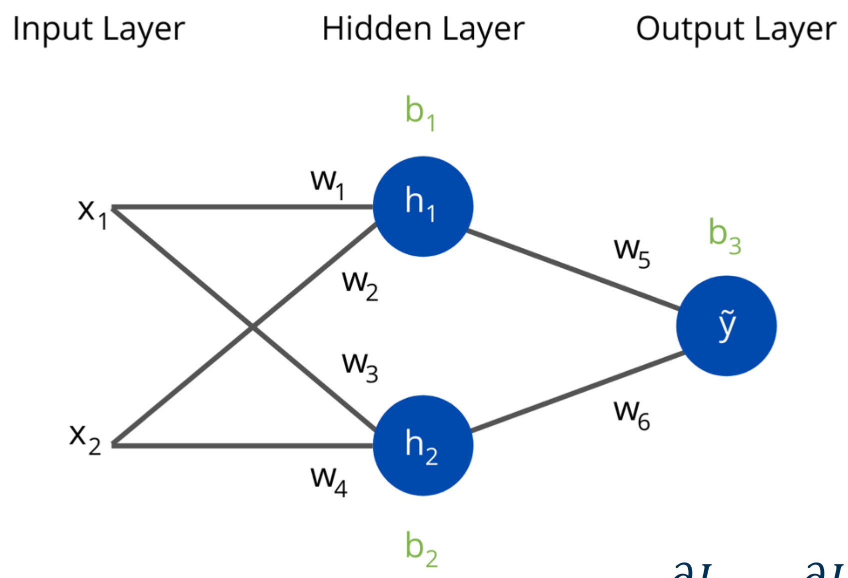
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \tilde{y}} * \frac{\partial \tilde{y}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$h_1 = f(b_1 + w_1 * x_1 + w_2 * x_2)$$

$$h_2 = f(b_2 + w_3 * x_1 + w_4 * x_2)$$

$$\tilde{y} = f(b_3 + w_5 * h_1 + w_6 * h_2)$$

Backpropagation



$$L = (y - \tilde{y})^2$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \tilde{y}} * \frac{\partial \tilde{y}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$h_1 = f(b_1 + w_1 * x_1 + w_2 * x_2)$$

$$h_2 = f(b_2 + w_3 * x_1 + w_4 * x_2)$$

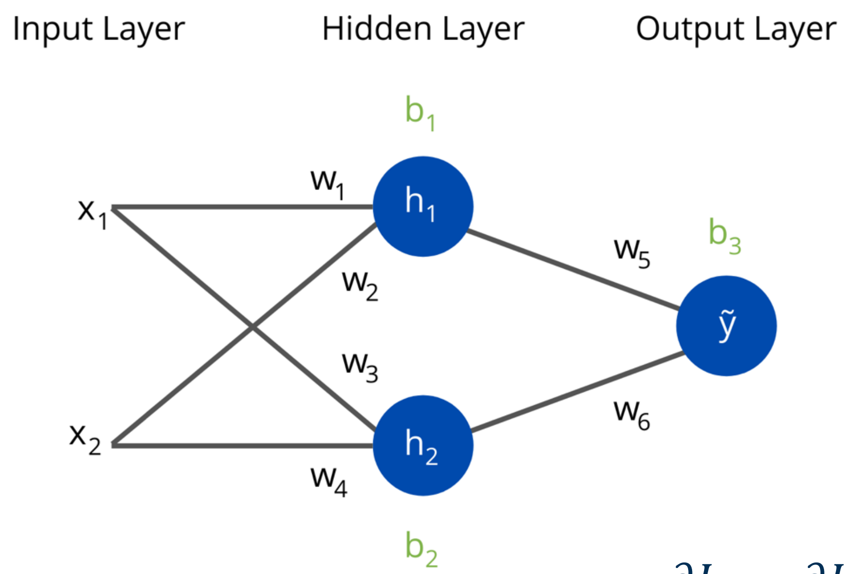
$$\tilde{y} = f(b_3 + w_5 * h_1 + w_6 * h_2)$$

$$\frac{\partial L}{\partial \tilde{y}} = -2(y - \tilde{y})$$

$$\frac{\partial \tilde{y}}{\partial h_1} = w_5 * f'(b_3 + w_5 * h_1 + w_6 * h_2)$$

$$\frac{\partial h_1}{\partial w_1} = x_1 * f'(b_1 + w_1 * x_1 + w_2 * x_2)$$

Backpropagation



$$L = (y - \tilde{y})^2$$

$$h_1 = f(b_1 + w_1 * x_1 + w_2 * x_2)$$

$$h_2 = f(b_2 + w_3 * x_1 + w_4 * x_2)$$

$$\tilde{y} = f(b_3 + w_5 * h_1 + w_6 * h_2)$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \tilde{y}} * \frac{\partial \tilde{y}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

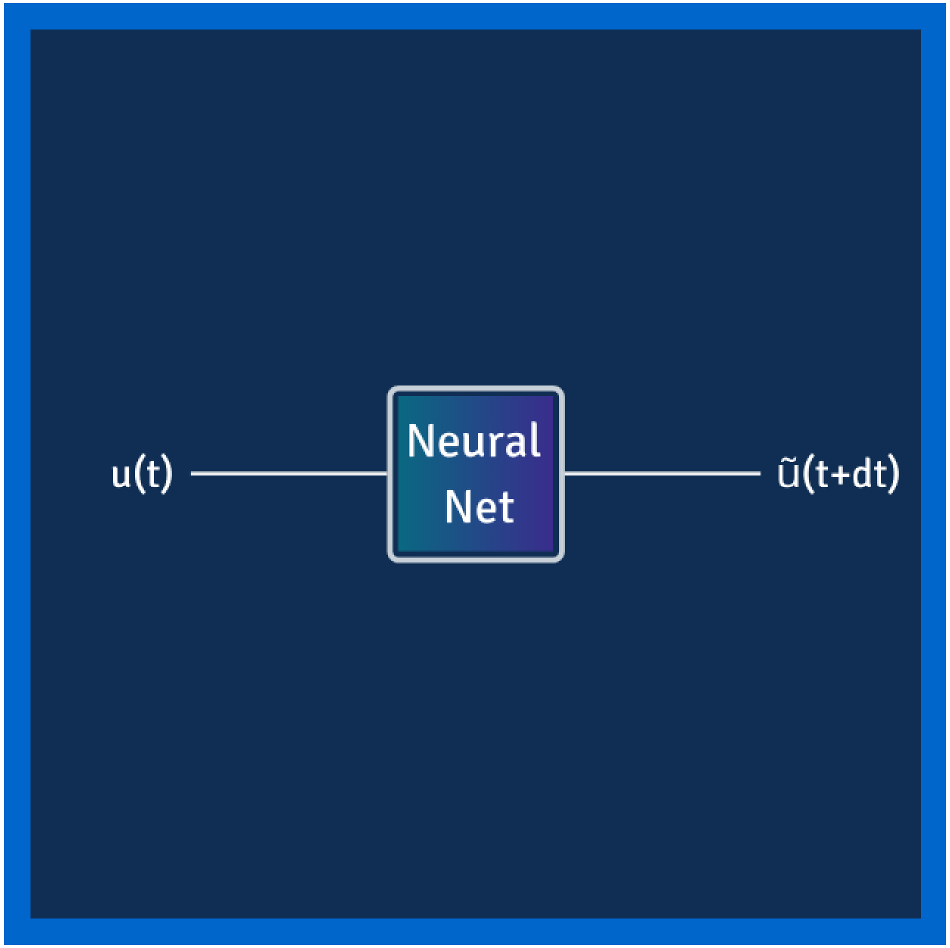
$$\frac{\partial L}{\partial \tilde{y}} = -2(y - \tilde{y})$$

$$\frac{\partial \tilde{y}}{\partial h_1} = w_5 * f'(b_3 + w_5 * h_1 + w_6 * h_2)$$

$$\frac{\partial h_1}{\partial w_1} = x_1 * f'(b_1 + w_1 * x_1 + w_2 * x_2)$$

$$w_1 = w_1 - \gamma \frac{\partial L}{\partial w_1}$$

Surrogate Model Layout

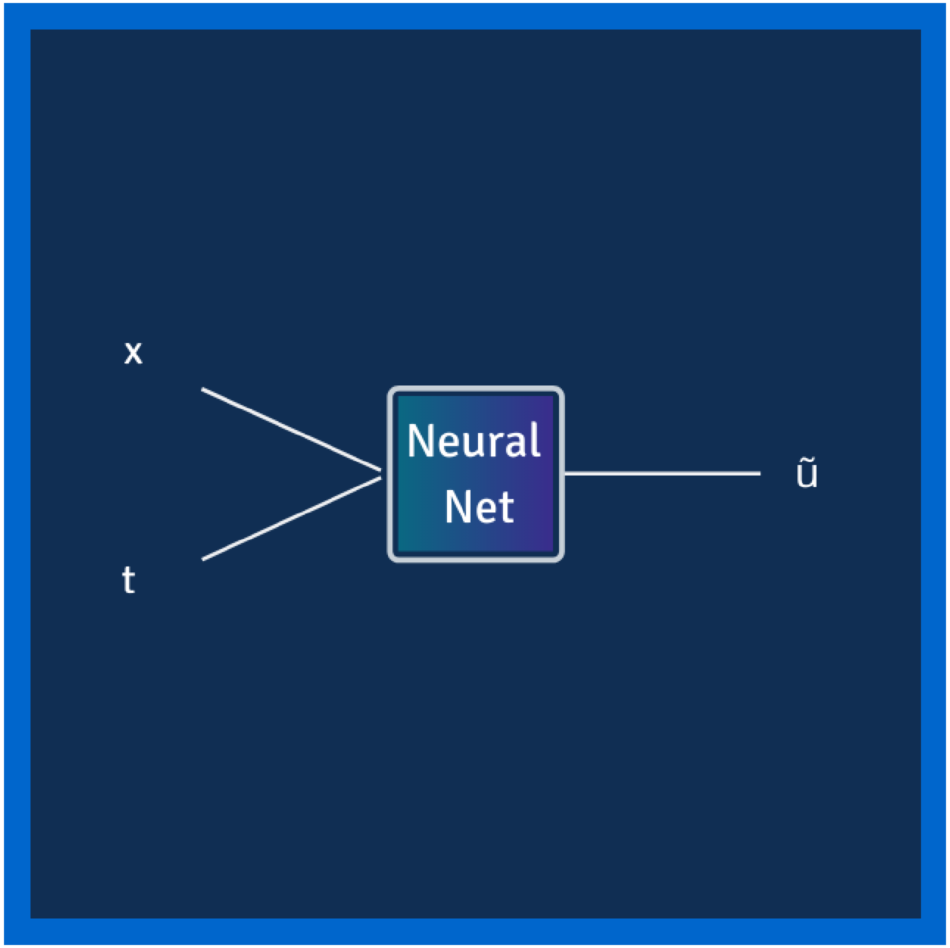


Loss Function:

$$\frac{1}{N}\sum(u - \tilde{u})$$

aka reconstruction error.

Surrogate Model Layout

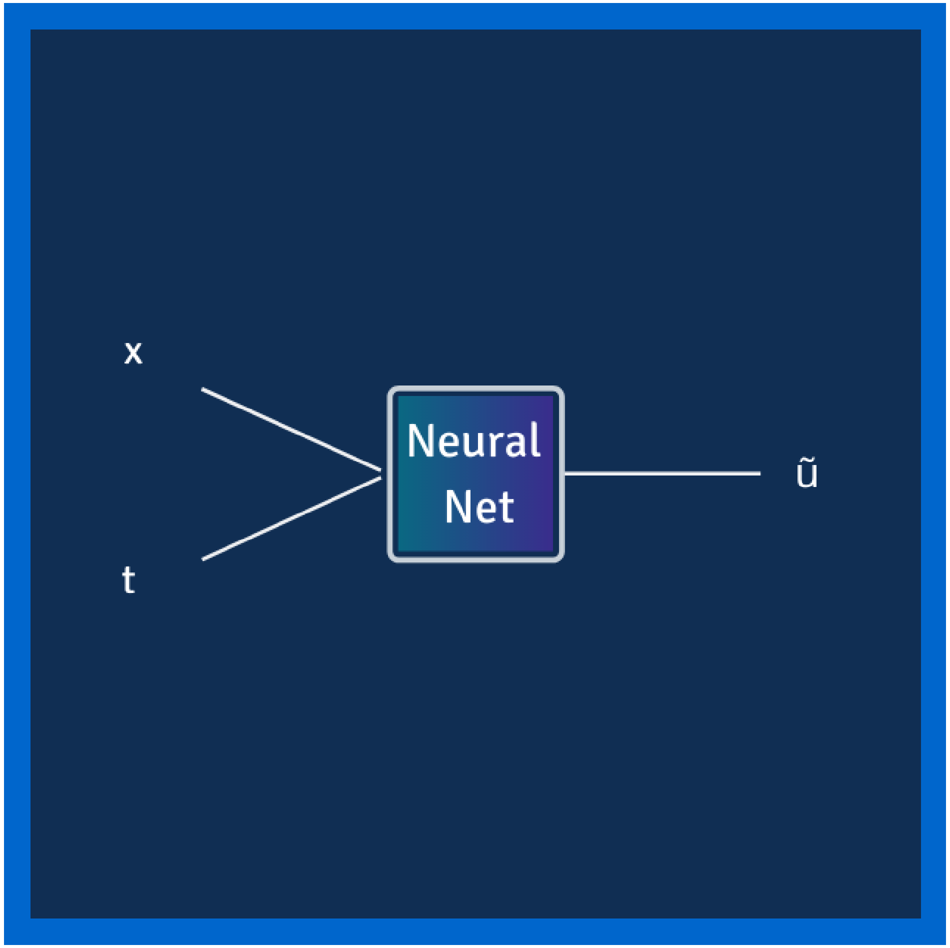


Loss Function:

$$\frac{1}{N}\sum(u - \tilde{u})$$

aka reconstruction error.

Surrogate Model with Physical Penalty

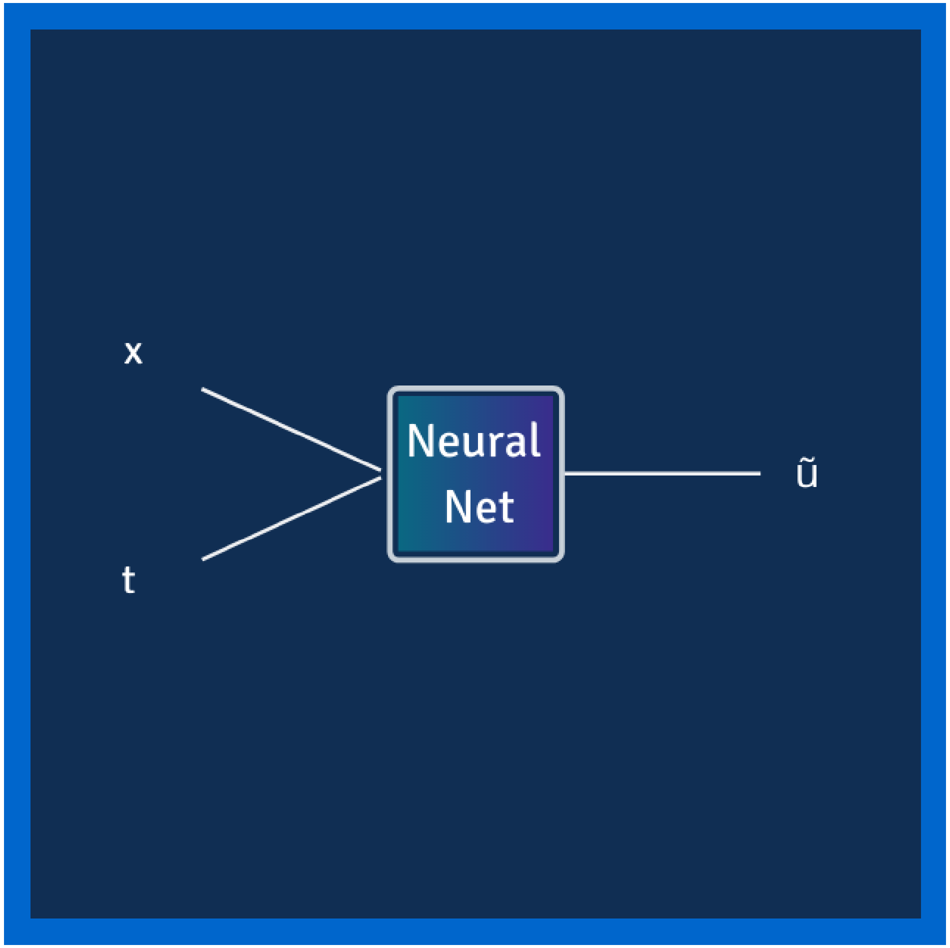


Loss Function:

$$\frac{1}{N}\sum(u - \tilde{u}) + \sum(m\tilde{u} - mu)$$

Momentum Conservation Equation playing an additional constraint (assuming $\tilde{u}$ is velocity in this case.)

Neural PDE Layout



Loss Function:

Initial Loss
\+ *Boundary Loss*
\+ *Domain Loss*

Consider a PDE written in the form:

$$f = u_t + \Lambda[u] = 0, \qquad x \in \Omega, \qquad t \in [0, T]$$

$$\mathrm{Initial\_Loss} = \mathrm{MSE} \sum \left(\mathrm{u}_{(\mathrm{x},\, \mathrm{t}=0)} - \tilde{u}_{(\mathrm{x},\, \mathrm{t}=0)} \right)$$

$$Boundary\_Loss = MSE \sum \left(BoundaryCondition\left(\tilde{u}_{(X\_lim,\, t)} \right) \right)$$

$$Domain\_Loss = MSE \sum \left(f(x, t) \right)$$

Consider the Korteweg-de Vries Equation :

$$f = u_t + u * u_x + \alpha * u_{xxx} = 0, \qquad x \epsilon [-1, 1], \qquad t \epsilon [0, 1]$$

with Periodic Boundary Conditions

$$u_{x=-1} = u_{x=1}$$

$$\frac{\partial u}{\partial x}_{x=-1} = \frac{\partial u}{\partial x}_{x=1}$$

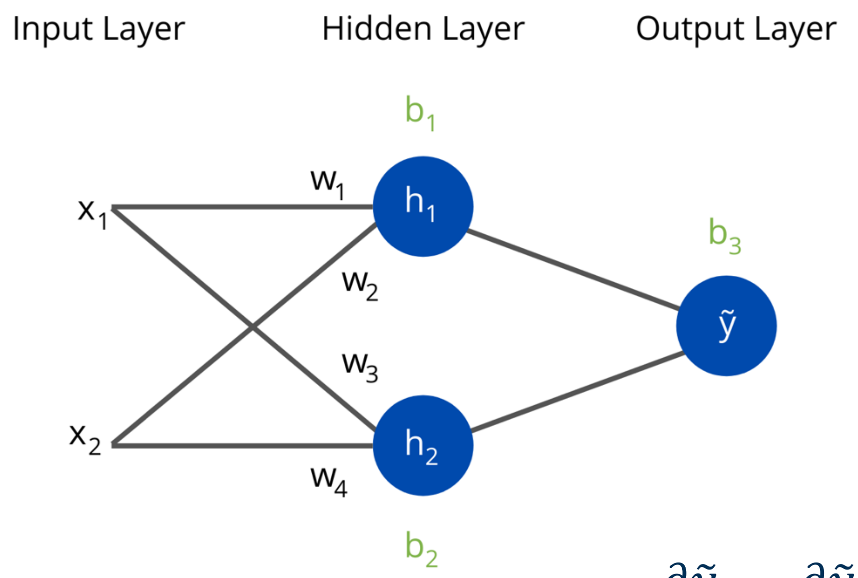
Loss Function Entities:

$$\text{Initial\_Loss} = \text{MSE} \sum \left(IC(x, 0) - \tilde{u}_{(\text{x, t=0})} \right)$$

$$Boundary\_Loss = MSE \sum \left(\frac{\partial u}{\partial x}_{x=-1,} - \frac{\partial u}{\partial x}_{x=1} + u_{x=-1} - u_{x=1} \right)$$

$$Domain\_Loss = MSE \sum \left(f(x, t) \right), \qquad x \epsilon (-1,1), \qquad t \epsilon (0,1)$$

Partial Derivatives via Backprop



$$h_1 = f(b_1 + w_1 * x_1 + w_2 * x_2)$$

$$h_2 = f(b_2 + w_3 * x_1 + w_4 * x_2)$$

$$\tilde{y} = f(b_3 + w_5 * h_1 + w_6 * h_2)$$

$$\frac{\partial \tilde{y}}{\partial x_1} = \frac{\partial \tilde{y}}{\partial h_1} * \frac{\partial h_1}{\partial x_1}$$

$$\frac{\partial \tilde{y}}{\partial h_1} = w_5 * f'(b_3 + w_5 * h_1 + w_6 * h_2)$$

$$\frac{\partial h_1}{\partial x_1} = w_1 * f'(b_1 + w_1 * x_1 + w_2 * x_2)$$

NPDE Package – PDE_Kozhi

Neural PDE Parameters :

$$N_i : Number\ of\ Initial\ Points$$
$$N_b : Number\ of\ Boundary\ Points$$
$$N_f : Number\ of\ Domain\ Points$$

Each collocation point for each loss entity is obtained by calling upon a quasi-random sequence within the boundaries of each region.

PDE Parameters :

$$Equation\ (as\ a\ string)$$
$$Lower\ and\ Upper\ bounds$$
$$Initial\ Condition$$
$$Boundary\ Condition\ and\ Value$$

NN Parameters :

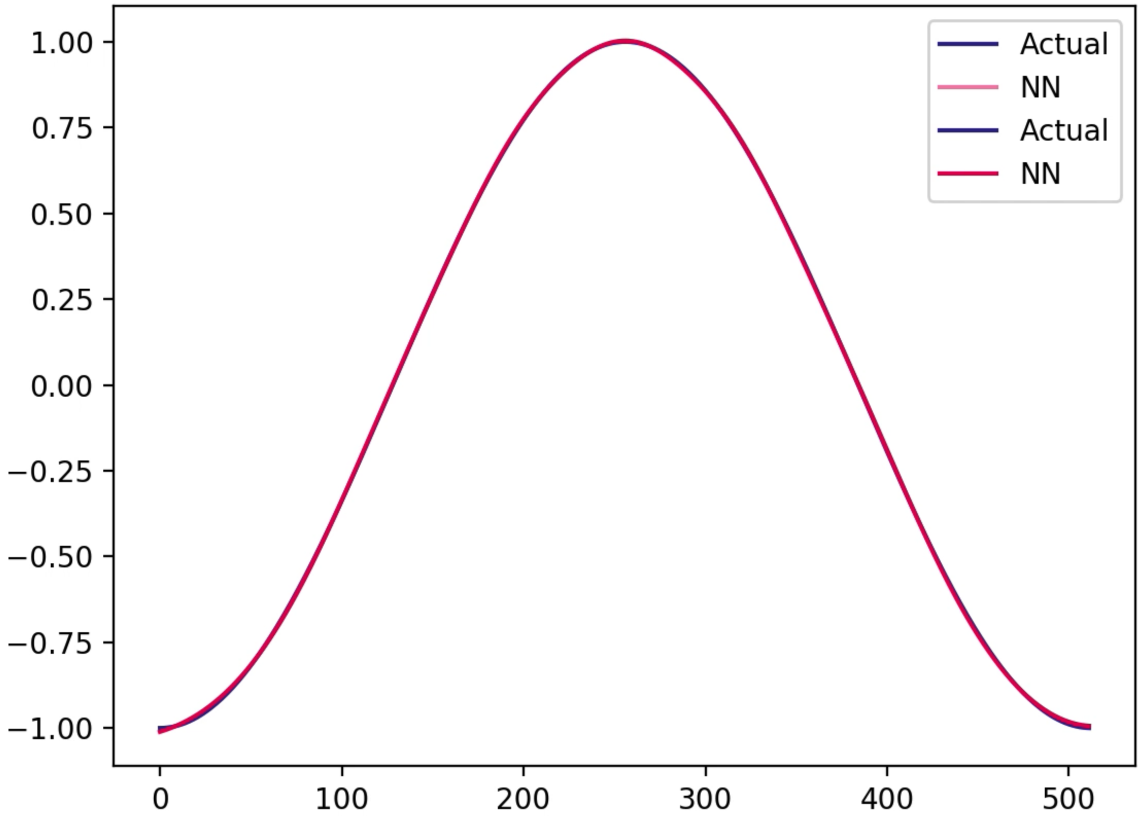
$$Number\ of\ layers\ and\ neurons$$

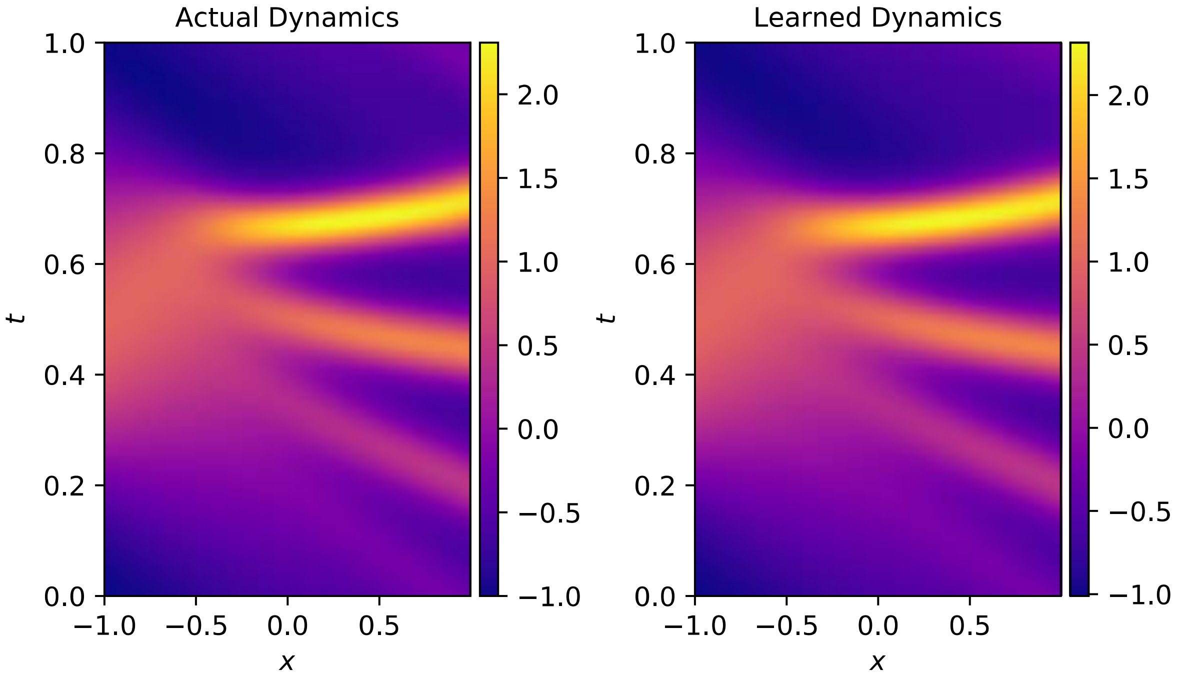
In [9]:
```python
#Neural Network Hyperparameters
NN_parameters = {
                'input_neurons' : 2,
                'output_neurons' : 1,
                'num_layers' : 4,
                'num_neurons' : 100,
                }


#Neural PDE Hyperparameters
NPDE_parameters = {'Sampling_Method': 'Random',
                   'N_initial' : 300, #Number of Randomly sampled Data points from the IC vector
                   'N_boundary' : 300, #Number of Boundary Points
                   'N_domain' : 20000 #Number of Domain points generated
                  }


#PDE
PDE_parameters = {'Equation': ' u_t + u*u_x + 0.0025*u_xxx',
                  'order': 3,
                  'lower_range': [-1., 0.],
                  'upper_range': [1., 1.],
                  'Boundary_Condition': "Periodic",
                  'Boundary_Vals' : None,
                  'Initial_Condition': lambda x: np.cos(np.pi*x)
                 }
```

In [10]:
```python
#Obtaining the training data
soln_loc = '/Examples/Data/KdV.mat'
x, t, training_data, testing_input, testing_output  = npde.Main.solution_data(soln_loc, NN_parameters, PDE_parameters,

params = npde.Parameters.parameters(PDE_parameters, NN_parameters, NPDE_parameters, Model_Name, Equation_Name)
```

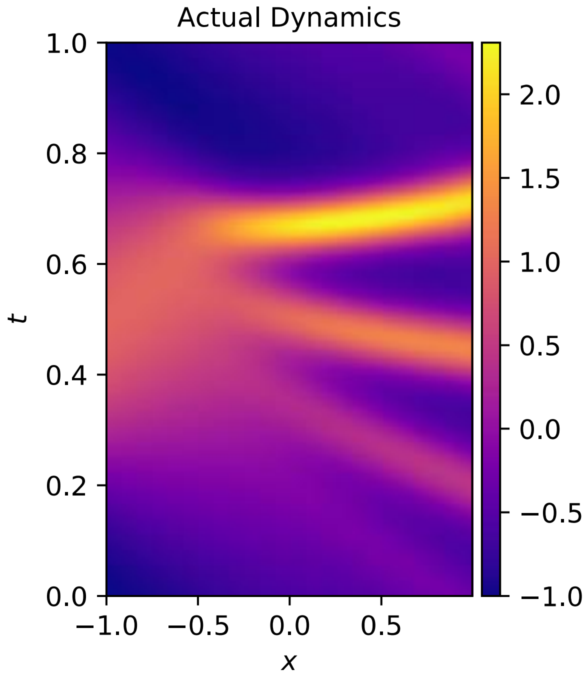
In [ ]:
```python
#Initialising the Model
model = npde.Main.setup(params, training_data)
```

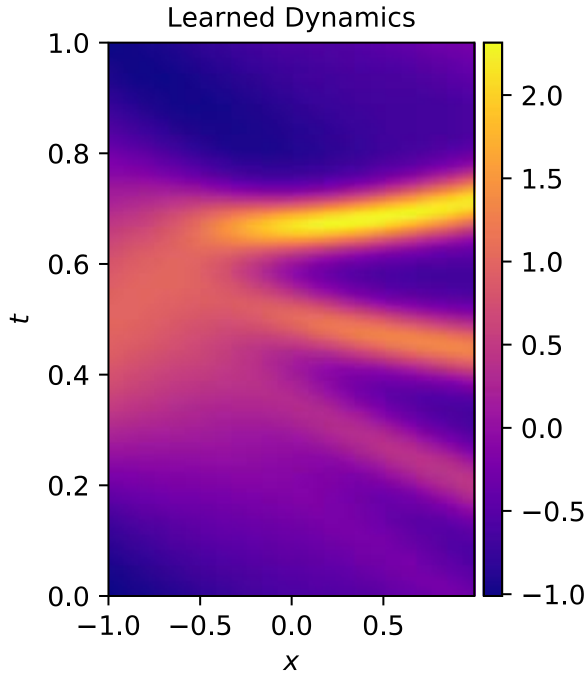
In [ ]:
```python
#Training Conditions -----------------------------------------------------
optimiser = {
            'opt_type' : "GD",
            'optimizer' : "adam",
            'learning_rate' : 0.001,
            'nIter' : 2000,
            'qn_source' : None
            }

start_time = time.time()
loss_GD = model.train(optimiser, Model_Name)
time_GD = time.time() - start_time


optimiser = {
            'opt_type' : "QN",
            'optimizer' : "L-BFGS-B",
            'learning_rate' : None,
            'nIter' : None,
            'qn_source' : "Scipy"
            }

start_time = time.time()
loss_Scipy = model.train(optimiser, Model_Name)
time_Scipy = time.time() - start_time
```

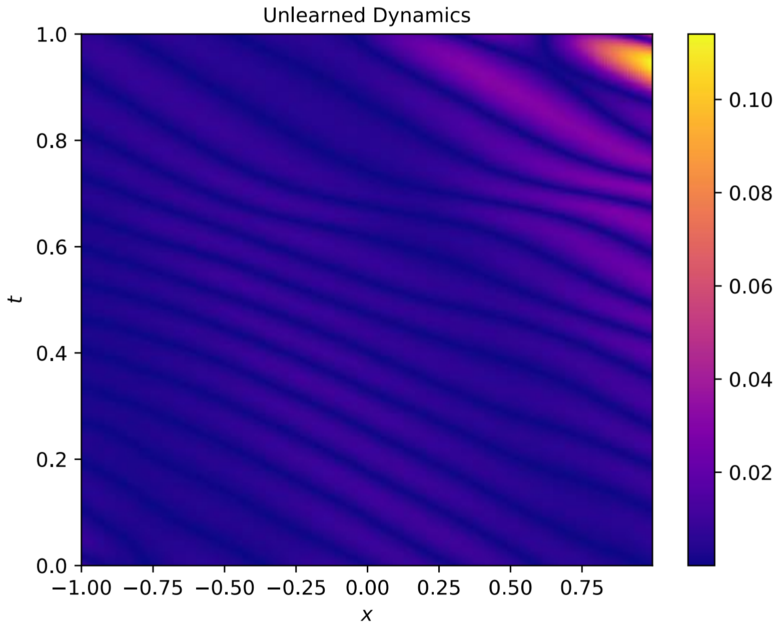

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
100
200
300
400
500
Actual
NN
Actual
NN



Actual Dynamics
Learned Dynamics
1.0
0.8
0.6
0.4
0.2
0.0
t
−1.0
−0.5
0.0
0.5
x
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0

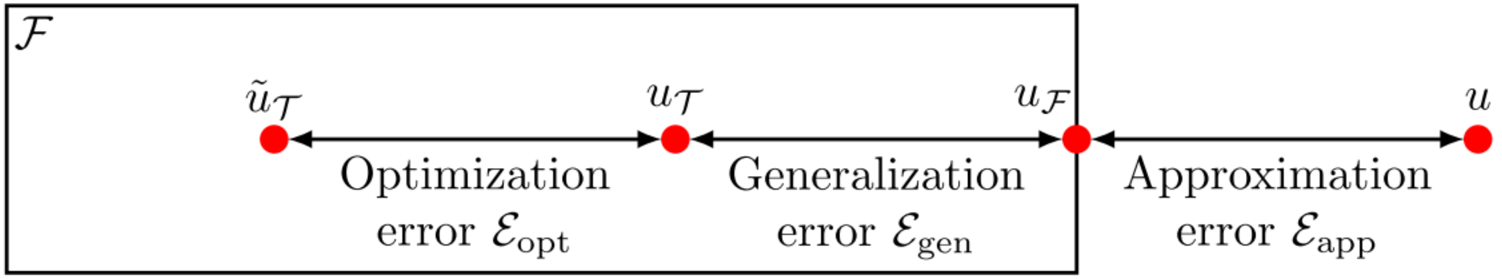

Actual Dynamics
t
x
1.0
0.8
0.6
0.4
0.2
0.0
−1.0
−0.5
0.0
0.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0


Learned Dynamics
t
x
1.0
0.8
0.6
0.4
0.2
0.0
−1.0
−0.5
0.0
0.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0


Unlearned Dynamics
t
x
1.0
0.8
0.6
0.4
0.2
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
0.10
0.08
0.06
0.04
0.02

Approx. Theory and Error Analysis for Neural PDEs

- Approximation Error (Best function close to u in the Function Space F – Global Minimum)
- Generalisation Error (Governed by the number of Points)
- Optimisation Error (Network stuck at local minimum)
- Networks with larger size have smaller approximation errors but could lead to higher generalization errors (Bias-Variance Tradeoff).



Source: DeepXdE

Numerical Solvers Compared with Neural PDEs

- Traditional Solvers have high round-off and truncation errors.
- Expensive at Higher Dimensions (Curse of Dimensionality)
- Confined to a Mesh
- Neural PDEs can be be accelerated on GPUs and TPUs

Still this isn't extremely cheap to run.

UK Atomic Energy Authority

Took approximately two hours to get to the final solution on a single CPU.

But accelerated by a single GPU, converges within 10 minutes.

Throwing away 'learned general dynamics' being thrown away with this case-specific approach.

Deep Hidden Physics Models

UK Atomic Energy Authority

Learn the general behavior of the PDE by mapping inputs to outputs of an already known solution.

Two different Neural Networks are used to attain this characterized by :

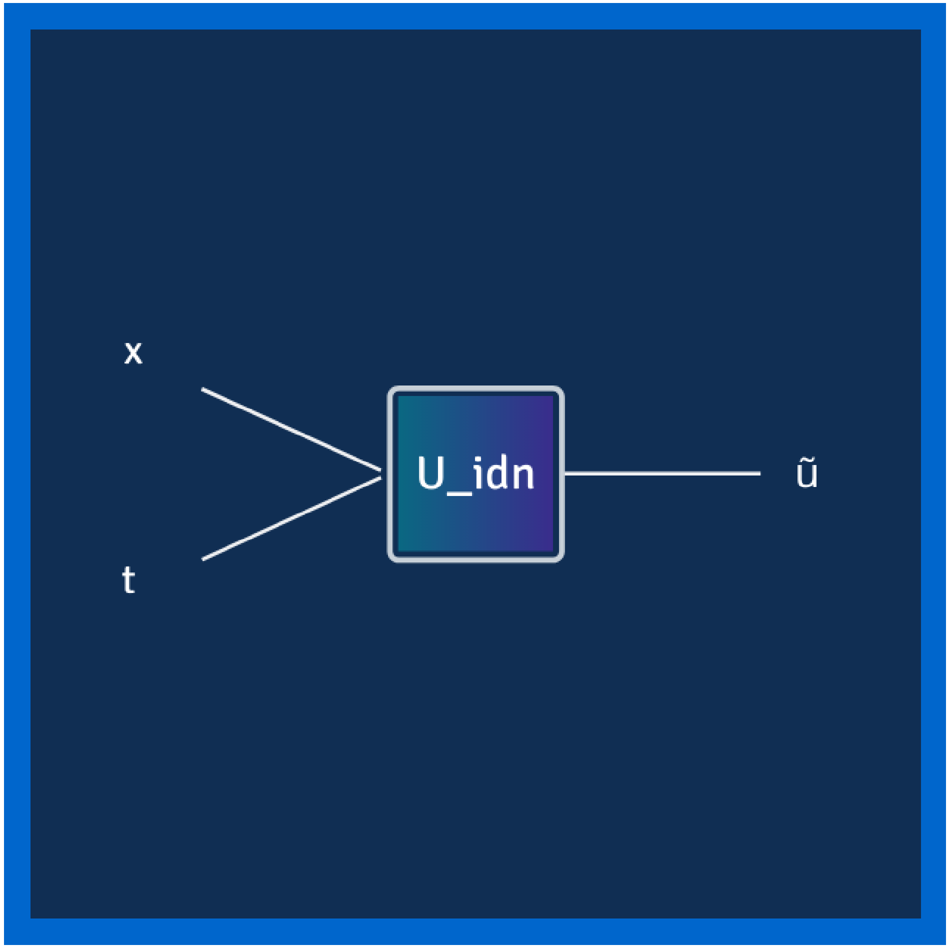
$$\boldsymbol{u_{idn}} \;\&\; \boldsymbol{\Lambda}$$

Where, u_{idn} learns the mapping of the input coordinates to the known solution, while Λ learns the general behaviour of the PDE using u_{idn}.

The final solution is attained by using another neural network $\boldsymbol{u_{soln}}$ by using the function Λ.

Source: Deep hidden physics models: Deep learning of nonlinear partial differential equations – M. Raissi

Identifier

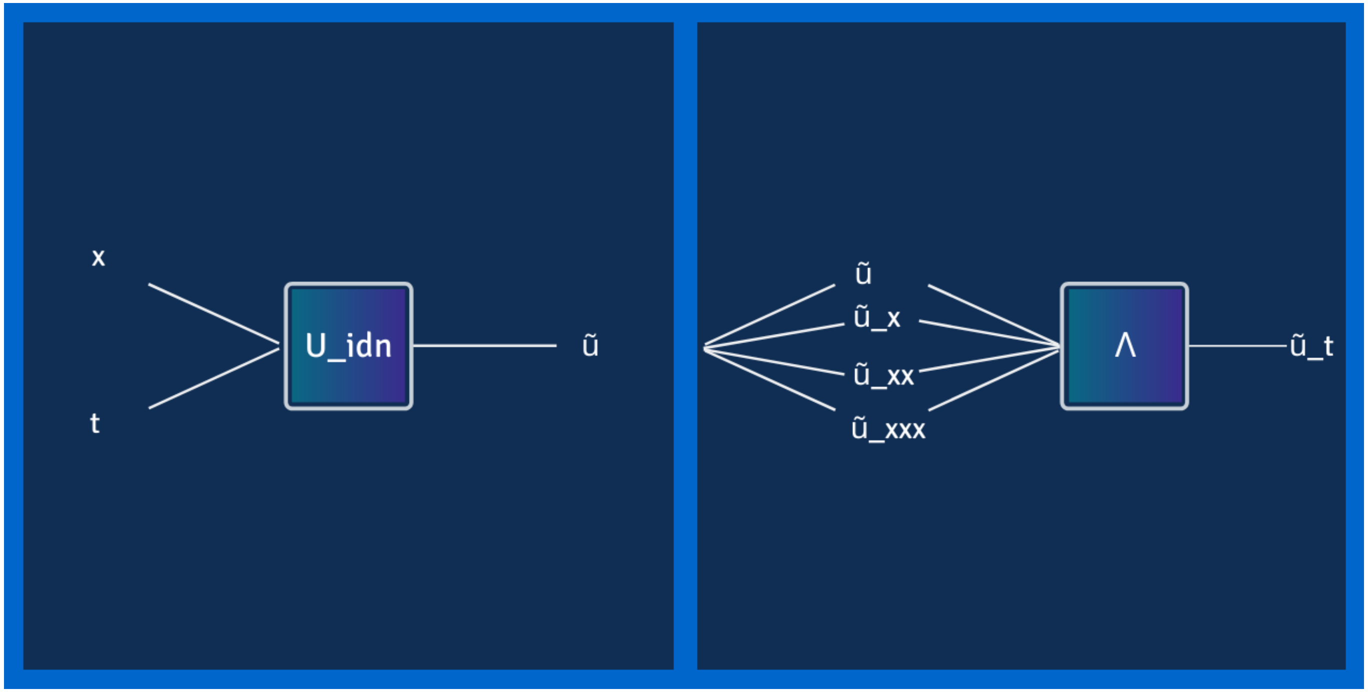


Loss Function:

$$\frac{1}{N}\sum(u - \tilde{u})$$

aka reconstruction error.

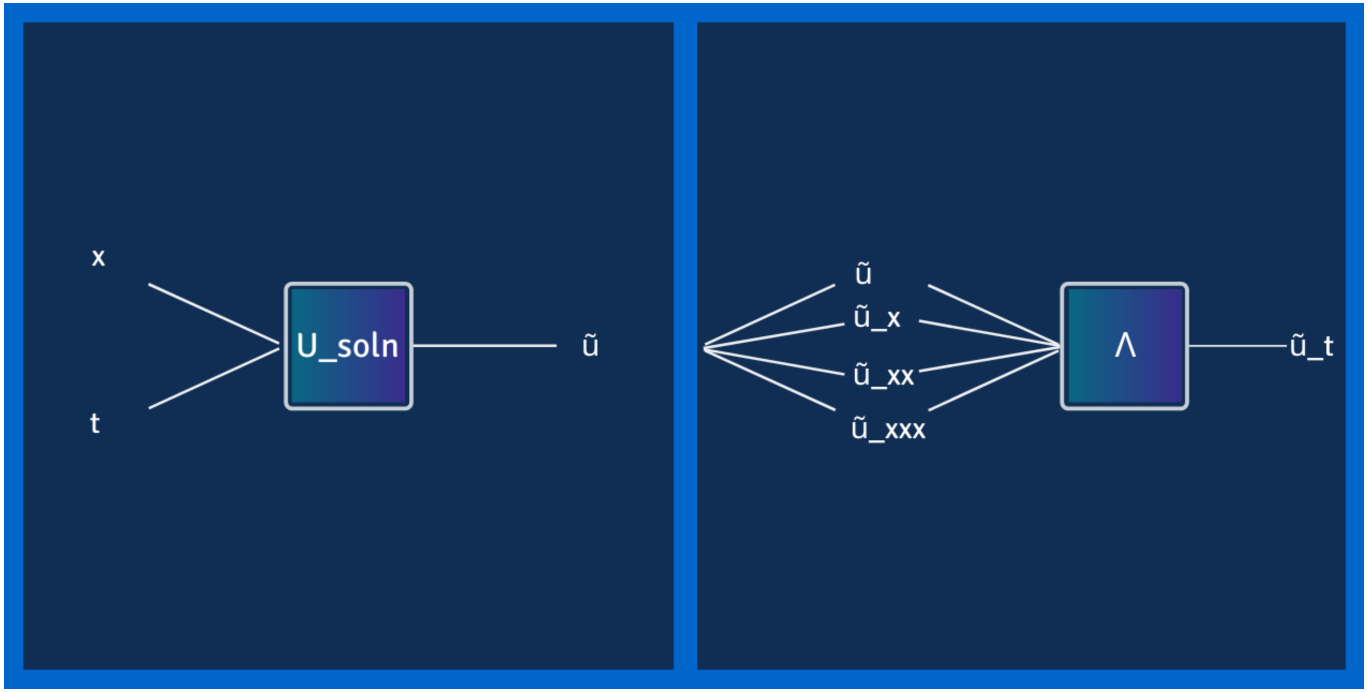
Lambda



Loss Function:

$$MSE((\tilde{u}_t - tf.concat(\tilde{u}, \tilde{u}_x, \tilde{u}_{xx}, \tilde{u}_{xxx})))$$

Solution



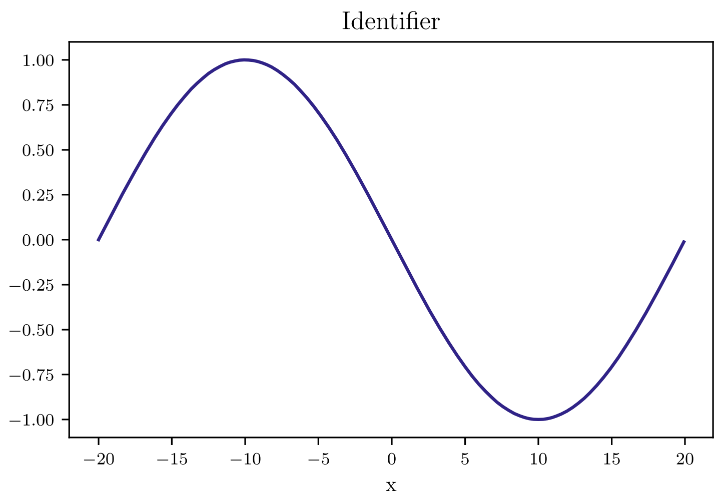
Loss Function:

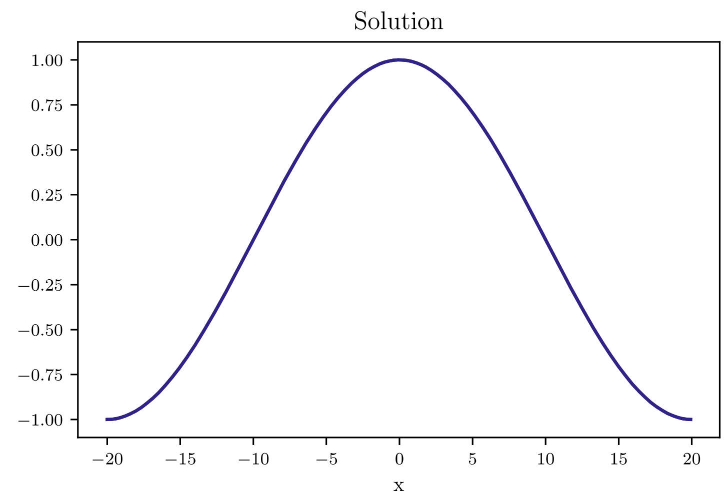
$$MSE(u_{ic} - \tilde{u}_{ic})$$

$$MSE(u_{bc} - \tilde{u}_{bc})$$

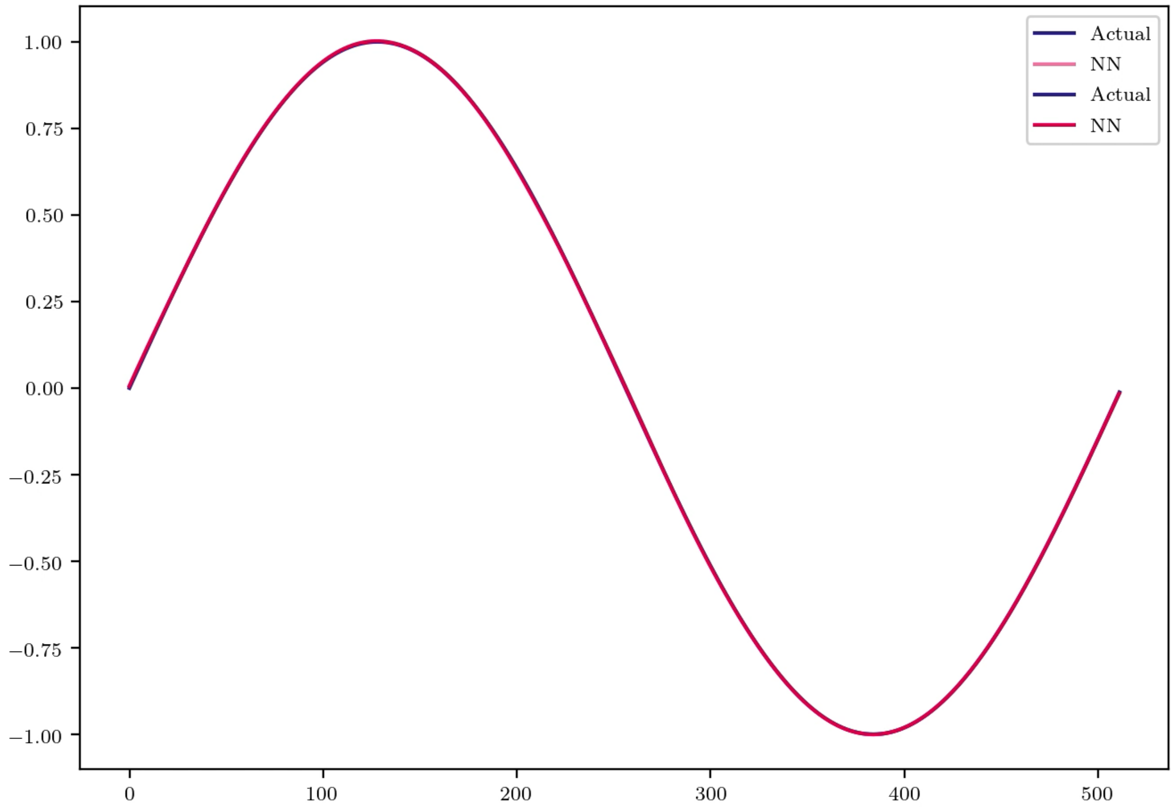
$$MSE((\tilde{u}_t \ - tf.concat(\tilde{u}, \tilde{u}_x \ , \tilde{u}_{xx}, \tilde{u}_{xxx})))$$

Identifier and Solution – Initial Condition

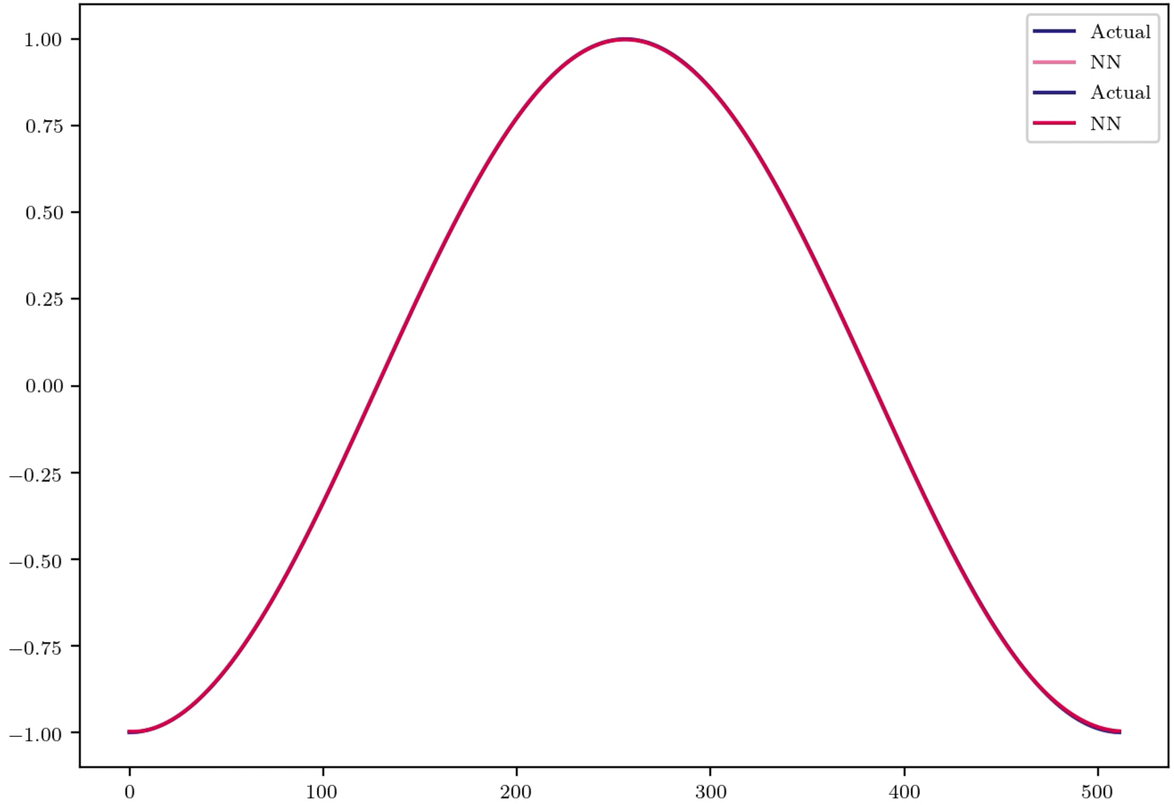


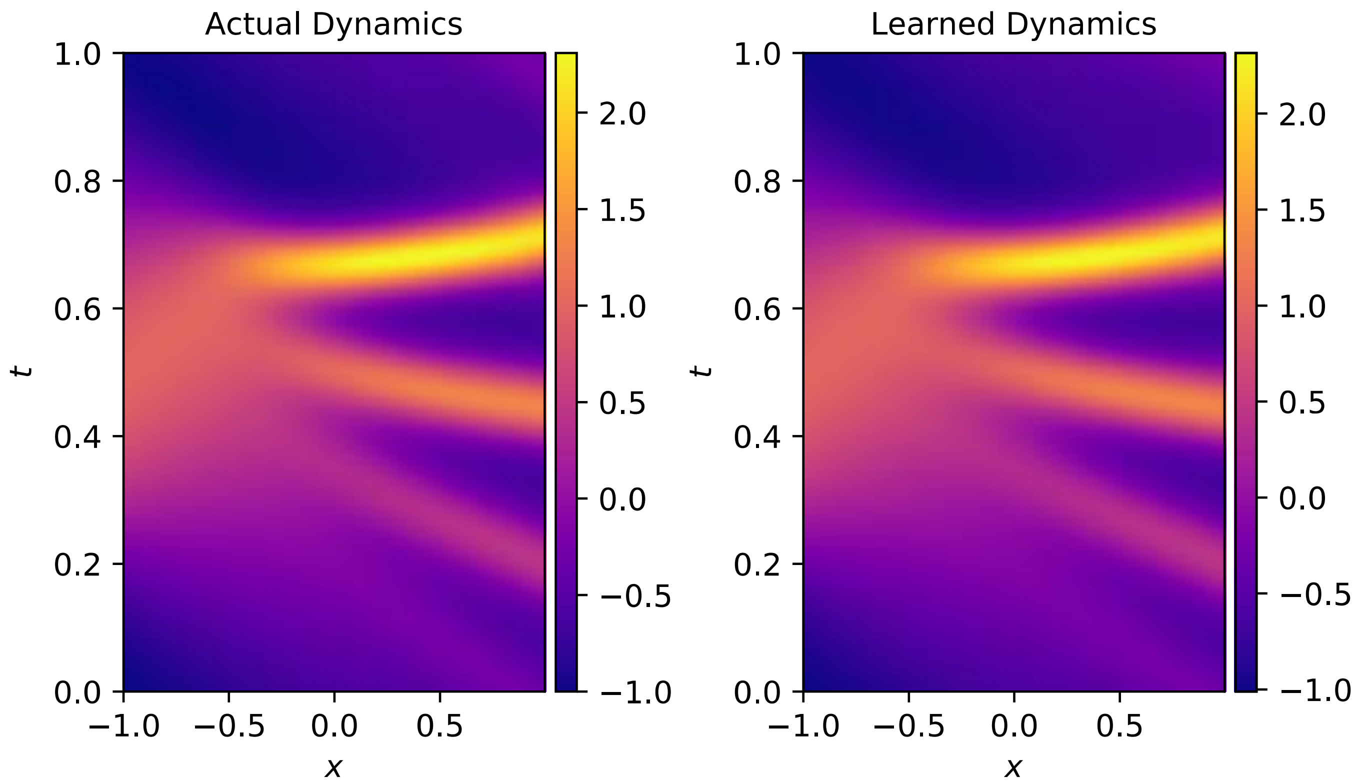


Identifier - U_{idn}



Solution - U_{soln}




Actual Dynamics
Learned Dynamics
1.0
0.8
0.6
0.4
0.2
0.0
t
−1.0
−0.5
0.0
0.5
x
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0

Additional Functionality:

UK Atomic Energy Authority

- Better Sampling Strategies : Spatio-Temporal Residual Based
- Higher Order Approximations represented by perturbations – Deep Galerkin Methods
- More complex Network Elements – Batch Normalisation, Sparse Nets, Resnets
- Case-agnostic modelling

References :

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, *378*, 686–707. https://doi.org/10.1016/j.jcp.2018.10.045

Raissi, M. (2018). Deep hidden physics models: Deep learning of nonlinear partial differential equations. In *Journal of Machine Learning Research* (Vol. 19, pp. 1–24).

Michoski, C., Milosavljevic, M., Oliver, T., & Hatch, D. (2019). *Solving Irregular and Data-enriched Differential Equations using Deep Neural Networks*. *78712*, 1–22. http://arxiv.org/abs/1905.04351

Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. (2020). DeepXdE: A deep learning library for solving differential equations. *CEUR Workshop Proceedings*, *2587*, 1–17.

Sirignano, J., & Spiliopoulos, K. (2018). DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, *375*(Dms 1550918), 1339–1364. https://doi.org/10.1016/j.jcp.2018.08.029

Koryagin, A., Khudorozkov, R., & Tsimfer, S. (2019). *PyDEns: a Python Framework for Solving Differential Equations with Neural Networks*. *i*. http://arxiv.org/abs/1909.11544

Rackauckas, C., Innes, M., Ma, Y., Bettencourt, J., White, L., & Dixit, V. (2019). *DiffEqFlux.jl - A Julia Library for Neural Differential Equations*. 1–17. http://arxiv.org/abs/1902.02376

Gopakumar, V., & Samaddar, D. (2020). Image mapping the temporal evolution of edge characteristics in tokamaks using neural networks. *Machine Learning: Science and Technology*, *1*(1), 015006. https://doi.org/10.1088/2632-2153/ab5639

Jiang, C. M., Esmaeilzadeh, S., Azizzadenesheli, K., Kashinath, K., Mustafa, M., Tchelepi, H. A., Marcus, P., Prabhat, & Anandkumar, A. (2020). *MeshfreeFlowNet: A Physics-Constrained Deep Continuous Space-Time Super-Resolution Framework*. http://arxiv.org/abs/2005.01463