Structural Machine Learning Models and Their Applications

Final Project

Name : KVDTT Abeywardhana.
Student ID : 7108053024

Selected Papers

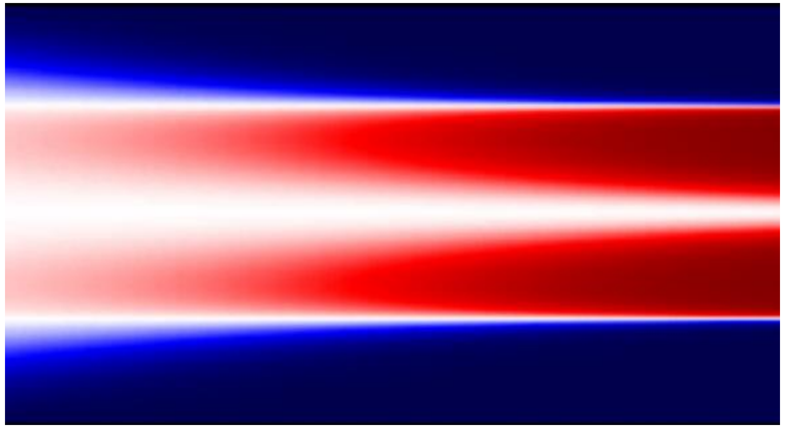
- Physics-informed neural network: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations.
- Deep Hidden Physics Models: Deep learning of Nonlinear Partial Differential Equations

1. The second paper I have already discussed in previous presentation.
2. GitHub link to the codes:

https://github.com/Thimira1992/Structural-Machine-Learning-Models-and-Their-Applications---End-final-report

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

- Abstract - Data-driven solution and data-driven discovery of partial differential equation (PDE).



$$u_t + uu_x - (0.01/\pi)u_{xx} = 0,$$
$$u(0, x) = -\sin(\pi x),$$
$$u(t, -1) = u(t, 1) = 0.$$

$$u(t, x)$$

Method: Data driven solution of PDE

Burgers' Equation

$$u_t + uu_x - (0.01/\pi)u_{xx} = 0,$$
$$u(0,x) = -\sin(\pi x),$$
$$u(t,-1) = u(t,1) = 0.$$

Import data (solved by numerically) related to this PDE equation

We define $f(t,x)$ to be given by

$$f := u_t + \mathcal{N}[u],$$

Proceed by approximating $u(t,x)$ by a deep neural network
(Python code using Tensorflow)

$u(t, x)$ can be simply defined as

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

The physics-informed neural network $f(t, x)$ takes the form

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

The shared parameters between the neural networks $u(t, x)$ and $f(t, x)$ can be learned by minimizing the mean squared error loss.

$$MSE = MSE_u + MSE_f,$$

where

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

and

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

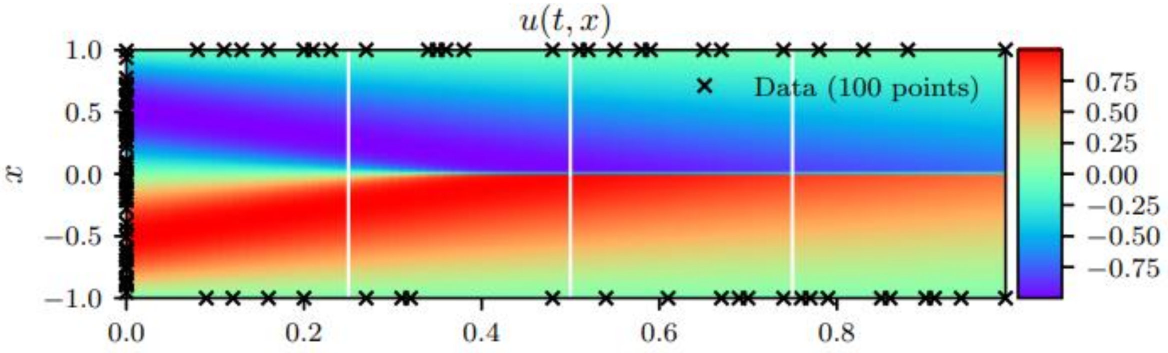
Neural network

```
def initialize_NN(layers):
    weights = []
    biases = []
    num_layers = len(layers)
    for l in range(0,num_layers-1):
        W = xavier_init(size=[layers[l], layers[l+1]])
        b = tf.Variable(tf.zeros([1,layers[l+1]], dtype=tf.float32), dtype=tf.float32)
        weights.append(W)
        biases.append(b)
    return weights, biases

def xavier_init(size):
    in_dim = size[0]
    out_dim = size[1]
    xavier_stddev = np.sqrt(2/(in_dim + out_dim))
    return tf.Variable(tf.random.truncated_normal([in_dim, out_dim], stddev=xavier_stddev, dtype=tf.float32), dtype=tf.float32)

#@tf.function
def neural_net(X, weights, biases):
    tf.config.run_functions_eagerly(False)
    num_layers = len(weights) + 1
    H = X
    for l in range(0,num_layers-2):
        W = weights[l]
        b = biases[l]
        H = tf.nn.relu(tf.add(tf.matmul(H, W), b))
    W = weights[-1]
    b = biases[-1]
    Y = tf.add(tf.matmul(H, W), b)
    return Y
```

Training data, test data and layers

```python
if __name__ == "__main__":

    nu = 0.1  #0.01, 0.001 ,0.01/np.pi
    noise = 0.0

    N_u = 100
    N_f = 10000
    layers = [2, 20, 20, 20, 20, 20, 20, 20, 20, 1]

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

    t = data['t'].flatten()[:,None]
    x = data['x'].flatten()[:,None]
    Exact = np.real(data['usol']).T
```

Construct by numerical methods
Using Matlab.



My implementations

Inside the code	Paper	My code implements
Import data	Chebfun package - Matlab	Tailored Finite Point Method - Matlab
Activation functions	Sin, Tanh	Sin. Tanh, relu
tf placeholders for Identification	Tf.placeholder	Tf.compat.v1.placeholder
Optimizer for Identification	Tf.contrib.opt.ScipyOptimizerInterface	Tf.compat.v1.train.GradientDescentOptimizer Tf.keras.optimizer.SGD
Optimizer for Identification	Tf.train.Adamoptimizer	Tf.compat.v1.train.Adamoptimizer

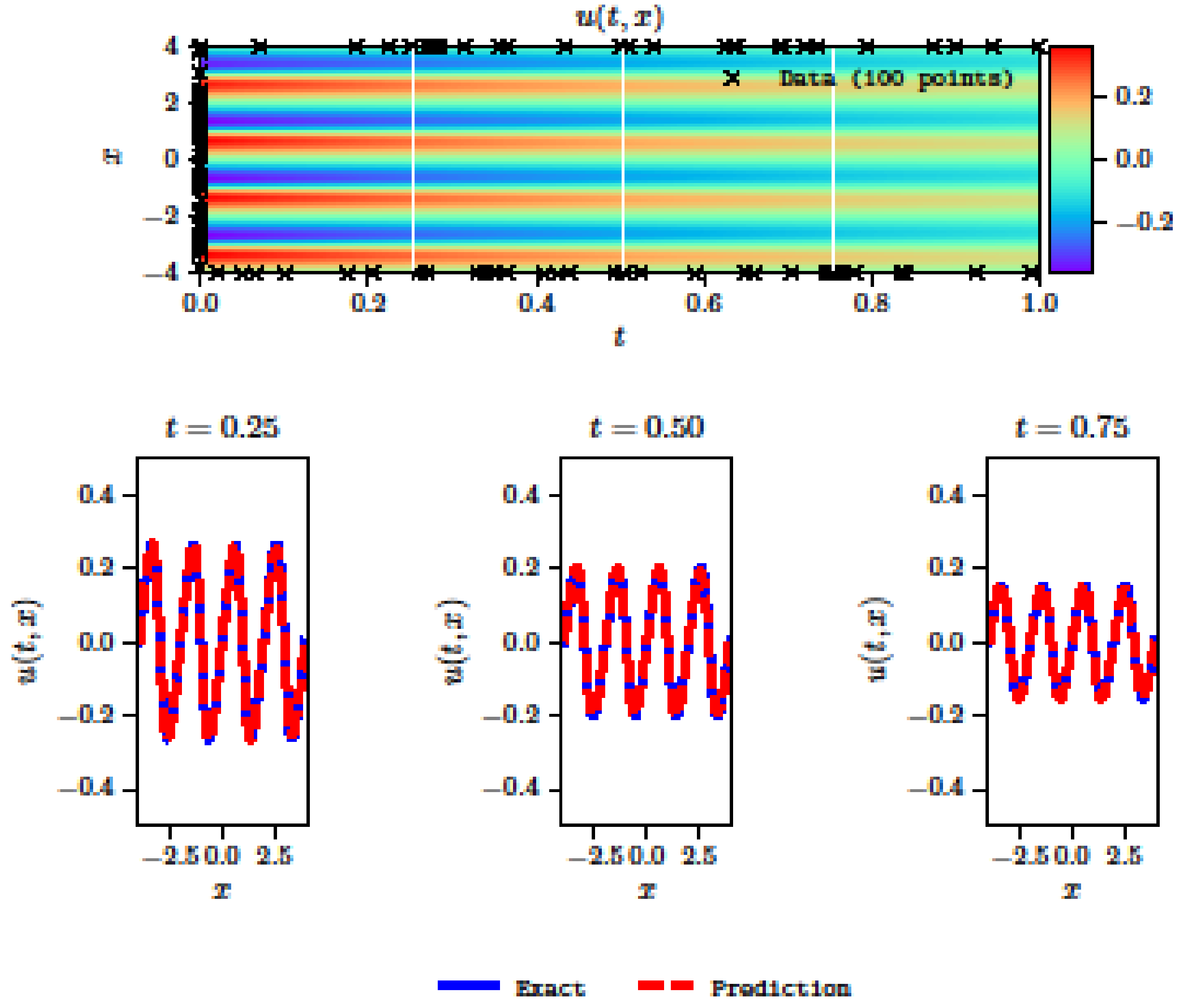
However, These errors occurs

```
type, dtype_hint, ctx, accepted_result_types)
   1463       graph = get_default_graph()
   1464       if not graph.building_function:
-> 1465         raise RuntimeError("Attempting to capture an EagerTensor without "
   1466                            "building a function.")
   1467       return graph.capture(value, name=name)

RuntimeError: Attempting to capture an EagerTensor without building a function.
```

Downgraded the Tensorflow

My results:

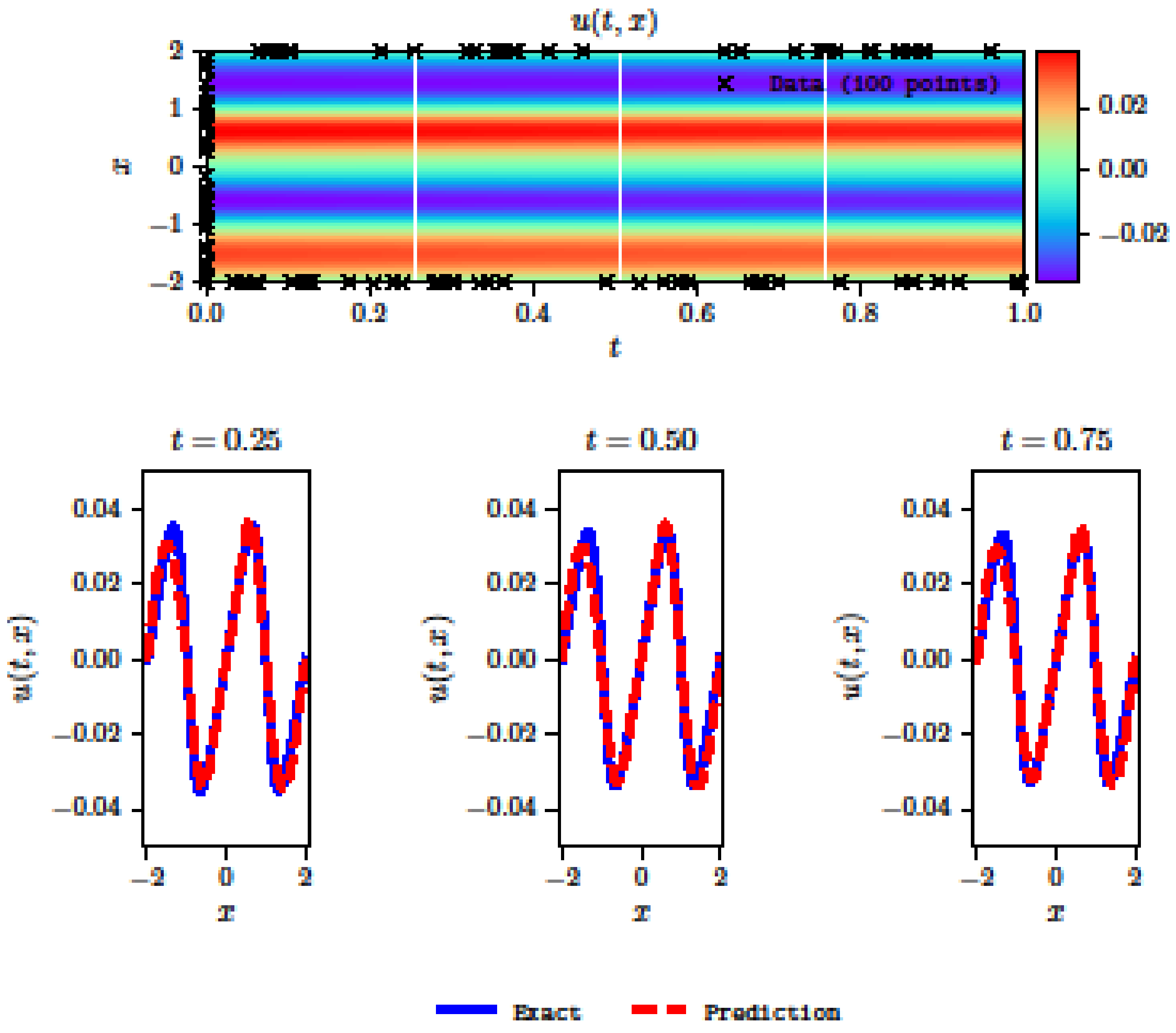
- When the diffusion = 0.1




```
Loss: 0.027766367
Loss: 1.0744218
Loss: 0.025212226
Loss: 0.02296518
Loss: 0.022312945
Loss: 0.0222618
```

```
Loss: 3.475343e-06
Loss: 3.4752852e-06
Loss: 3.4752852e-06
Loss: 3.4752852e-06
Training time: 667.8828
Error u: 3.031867e-02
```

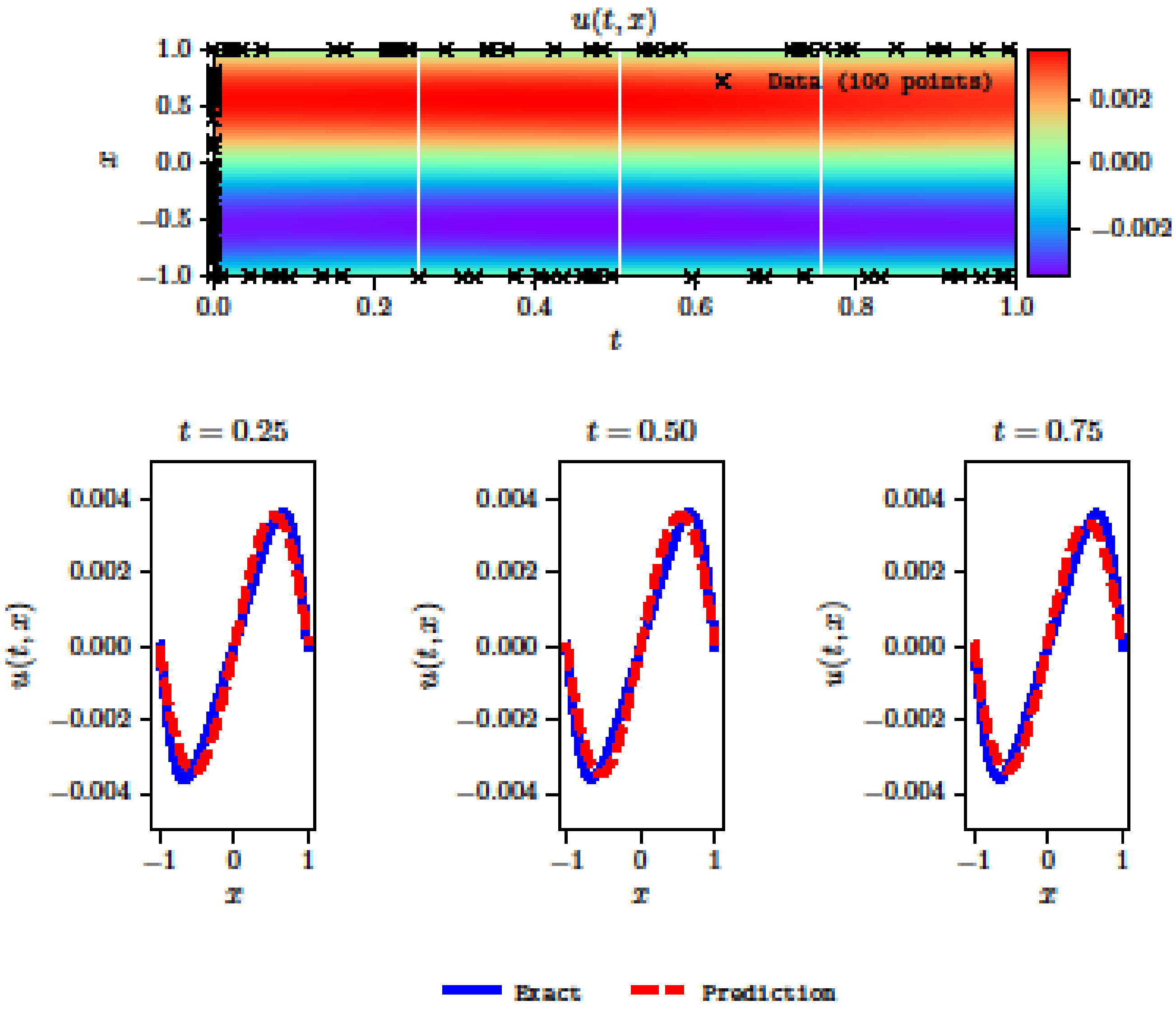
- When the diffusion = 0.01




```
Loss: 0.05958105
Loss: 0.6459013
Loss: 0.050138526
Loss: 0.28813097
Loss: 0.0028513304
Loss: 0.0021960451
```

```
Loss: 7.218465e-06
Loss: 7.216982e-06
Loss: 7.2142543e-06
Loss: 7.2124844e-06
Training time: 61.6658
Error u: 1.750651e-01
```

- When the diffusion = 0.001




```
Loss: 0.03907715
Loss: 0.10187675
Loss: 0.14202361
Loss: 0.0012227248
Loss: 0.00034217525
Loss: 0.00017722331
```

```
Loss: 3.6610425e-07
Loss: 3.6037773e-07
Loss: 3.5983453e-07
Loss: 3.589055e-07
Training time: 8.8428
Error u: 1.874392e-01
```

Deep Hidden Physics Models: Deep learning of Nonlinear Partial Differential Equations

- In this paper:
  - Method is how to discover closed form mathematical models of the physical world expressed by partial differential equations from scattered data collected in space and time.
  - we construct structured nonlinear regression models that can uncover the dynamic dependencies in a given set of spatio-temporal data, and return a closed form model that can be subsequently used to forecast future states.

Methodology

- Define the deep hidden physics model f,

$$f := u_t - \mathcal{N}(t, x, u, u_x, u_{xx}, \ldots)$$

- Parameters of the neural networks u and N can be learned by minimizing the sum of squared errors loss function

$$SSE := \sum_{i=1}^{N} \left(|u(t^i, x^i) - u^i|^2 + |f(t^i, x^i)|^2 \right)$$

- We solve the learned Partial differential equation by PINNs algorithm.

Code:

```
##############################################################################
############################## DeepHPM Class ################################
##############################################################################

class DeepHPM:
    def __init__(self, t, x, u,
                       x0, u0, tb, X_f,
                       u_layers, pde_layers,
                       layers,
                       lb_idn, ub_idn,
                       lb_sol, ub_sol):

        # Domain Boundary
        self.lb_idn = lb_idn
        self.ub_idn = ub_idn

        self.lb_sol = lb_sol
        self.ub_sol = ub_sol

        # Init for Identification
        self.idn_init(t, x, u, u_layers, pde_layers)

        # Init for Solution
        self.sol_init(x0, u0, tb, X_f, layers)

        # tf session
        self.sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
                                                     log_device_placement=True))
```

```
########################### Identifier ####################################
##########################################################################

def idn_init(self, t, x, u, u_layers, pde_layers):
    # Training Data for Identification
    self.t = t
    self.x = x
    self.u = u

    # Layers for Identification
    self.u_layers = u_layers
    self.pde_layers = pde_layers

    # Initialize NNs for Identification
    self.u_weights, self.u_biases = initialize_NN(u_layers)
    self.pde_weights, self.pde_biases = initialize_NN(pde_layers)

    # tf placeholders for Identification
    self.t_tf = tf.placeholder(tf.float32, shape=[None, 1])
    self.x_tf = tf.placeholder(tf.float32, shape=[None, 1])
    self.u_tf = tf.placeholder(tf.float32, shape=[None, 1])
    self.terms_tf = tf.placeholder(tf.float32, shape=[None, pde_layers[0]])

    # tf graphs for Identification
    self.idn_u_pred = self.idn_net_u(self.t_tf, self.x_tf)
    self.pde_pred = self.net_pde(self.terms_tf)
    self.idn_f_pred = self.idn_net_f(self.t_tf, self.x_tf)

    # loss for Identification
    self.idn_u_loss = tf.reduce_sum(tf.square(self.idn_u_pred - self.u_tf))
```

```python
### Load Data ###

data_idn = scipy.io.loadmat('../Data/burgers_sine_0.1_tfp.mat')

t_idn = data_idn['t'].flatten()[:,None]
x_idn = data_idn['x'].flatten()[:,None]
Exact_idn = np.real(data_idn['usol'])

T_idn, X_idn = np.meshgrid(t_idn,x_idn)

keep = 2/3
index = int(keep*t_idn.shape[0])
T_idn = T_idn[:,0:index]
X_idn = X_idn[:,0:index]
Exact_idn = Exact_idn[:,0:index]

t_idn_star = T_idn.flatten()[:,None]
x_idn_star = X_idn.flatten()[:,None]
X_idn_star = np.hstack((t_idn_star, x_idn_star))
u_idn_star = Exact_idn.flatten()[:,None]

#

data_sol = scipy.io.loadmat('../Data/burgers_sine_0.1_tfp.mat')

t_sol = data_sol['t'].flatten()[:,None]
x_sol = data_sol['x'].flatten()[:,None]
Exact_sol = np.real(data_sol['usol'])
```

```python
# Layers
u_layers = [2, 50, 50, 50, 50, 1]
pde_layers = [3, 100, 100, 1]

layers = [2, 50, 50, 50, 50, 1]
```

My implementations

- For diffusion = 0.001

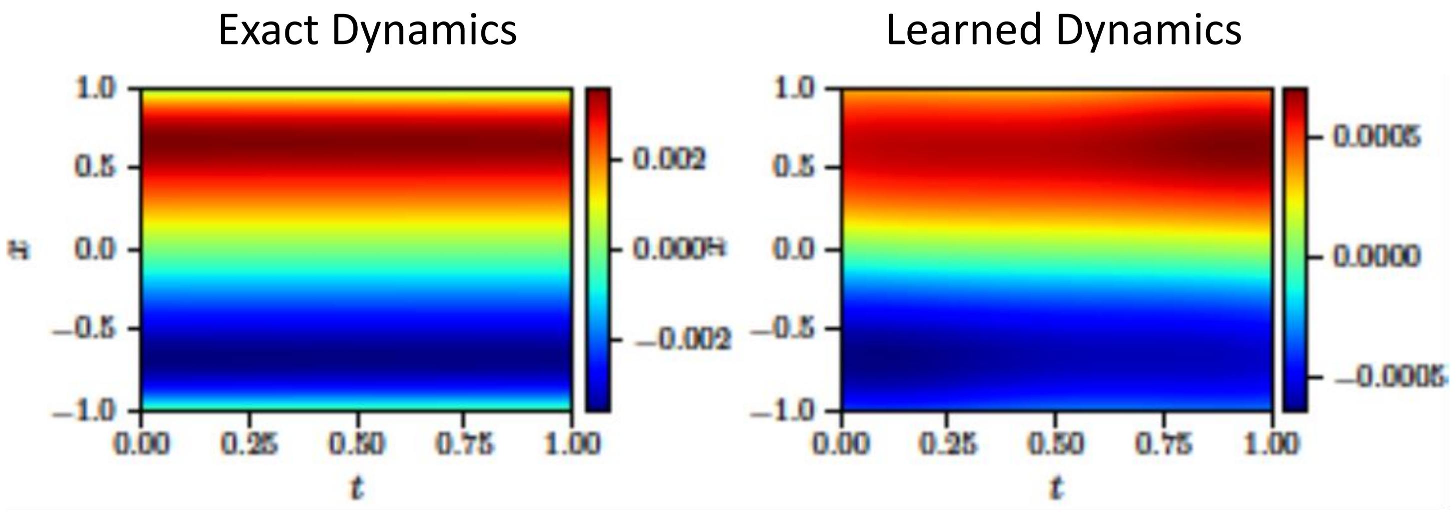
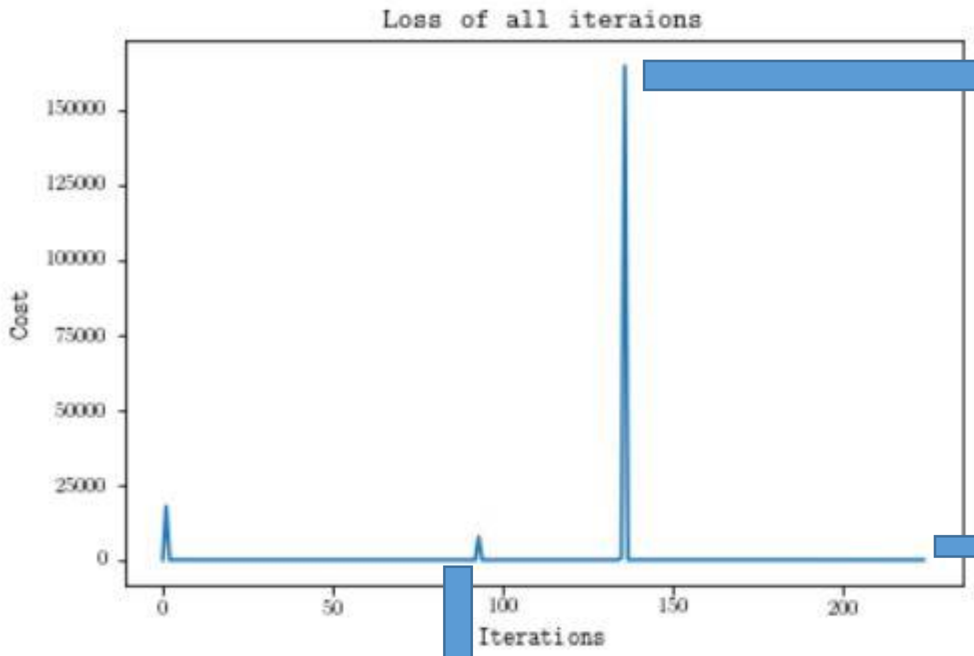


Figure: A solution to the Burger's equation (left panel) is compared to the corresponding solution of the learned partial differential equation (right panel).

- Loss of Iterations

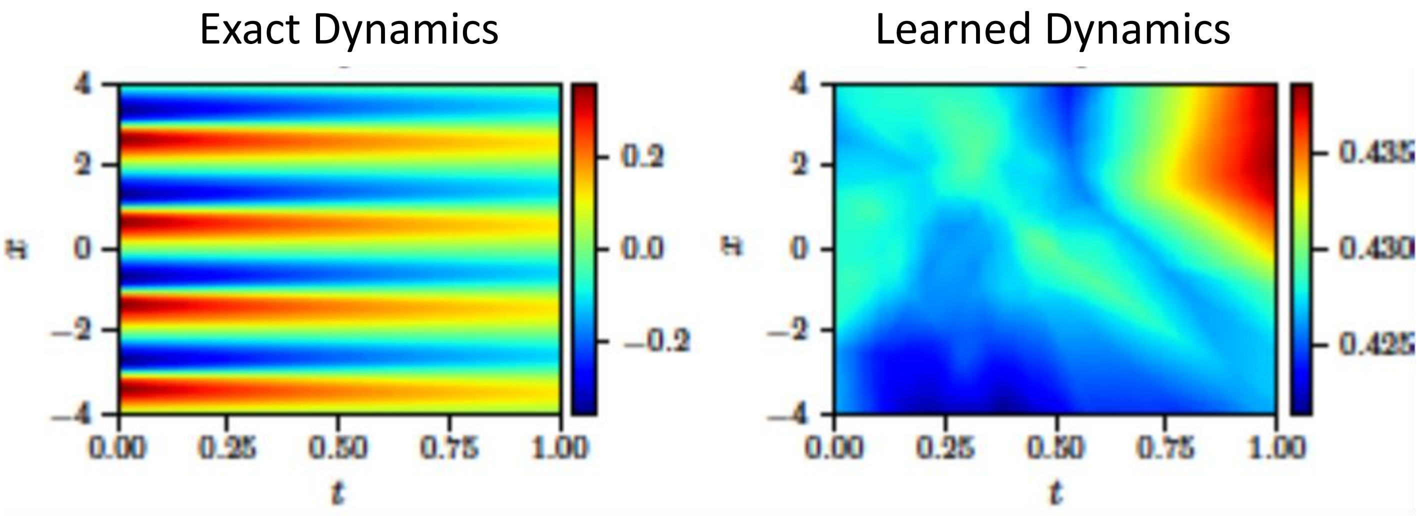
```
Loss: 1.327886e-03
Loss: 1.327886e-03
Loss: 1.327886e-03
Error u: 1.221638e-01
Loss: 7.213282e+02
Loss: 1.650634e+05
Loss: 1.107052e+01
```




```
Loss: 6.386290e-04
Loss: 6.386290e-04
Loss: 6.386290e-04
Error u: 4.242970e-01
```

```
Loss: 4.651525e+01
Loss: 1.787498e+04
Loss: 1.437561e+01
Loss: 2.620550e+00
Loss: 2.342708e-01
Loss: 1.493185e-01
```

When activation function as RELU



Conclusions:

- Activation fun ReLU is not suitable for both methods.
- Sinoid and Tanh activations functions give better results.
- Even the low diffusion terms can be approximate by the two methods by importing high resolution data.

The End!