Some General Theoretical Foundations of Machine Learning and Applications

by

Dr. Bivas Bhaumik

Department of Mathematics

National Institute of Technology, Rourkela

Important Links:

Google Scholar: FfG_kKQAAAAJ&hl=en&oi=ao

ORCID ID: 0000-0002-3168-2687

Primary Scopus ID: 57224404057

h-Index: 10

i10-Index: 10

Statistical Foundations

• Overview of descriptive statistics: Mean, variance, skewness, MSE, RMSE, R^2

Descriptive statistics provide numerical summaries of data to describe its central tendency, dispersion, and overall characteristics. These measures are fundamental in data analysis for understanding datasets.

- The **mean** is the arithmetic average of a dataset, representing the central tendency. It is calculated as: $$\text{Mean} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Use: Provides a single-value summary of the dataset.

Limitation: Sensitive to outliers.

- The **variance** measures the spread of data around the mean, reflecting its variability. It is calculated as: $$\text{Variance}(\sigma^2) = \frac{\sum_{i=1}^{n}(x_i - \text{Mean})^2}{n}$$
  - **Low variance**: Data points are close to the mean.
  - **High variance**: Data points are widely spread.

•Statistics Measures (MSE RMSE and R^2):

MSE is the average of the squared differences between the actual and predicted values and **RMSE** is the square root of the MSE.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \qquad \text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Interpretation: A lower **MSE** indicates that the predictions are closer to the actual values.
RMSE provides a measure of the model's error in the same unit as the dependent variable

$\boldsymbol{R^2}$ measures the proportion of variance in the dependent variable that is explained by the independent variable(s) in the model.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

- $R^2 = 1$: Perfect model (all variance explained).
- $R^2 = 0$: The model explains none of the variance.

Limitations: R^2 doesn't indicate whether predictions are unbiased and can be misleading with non-linear models.

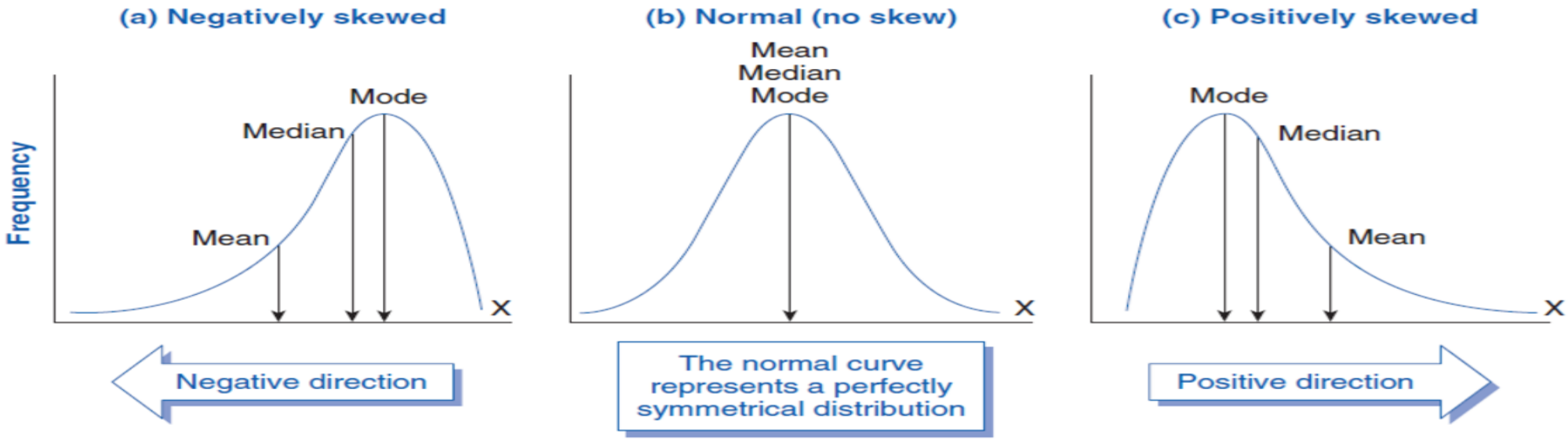
Continue...

Skewness quantifies the asymmetry of a dataset's distribution:

$$\text{Skewness} = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \text{Mean})^3}{(\text{Variance})^{3/2}}$$

- **Positive skew**: Tail extends to the right.
- **Negative skew**: Tail extends to the left.
- **Zero skew**: Symmetrical distribution.

Example: As the data becomes skewed from a normal distribution, the mean loses its ability to provide the best measure of central tendency.



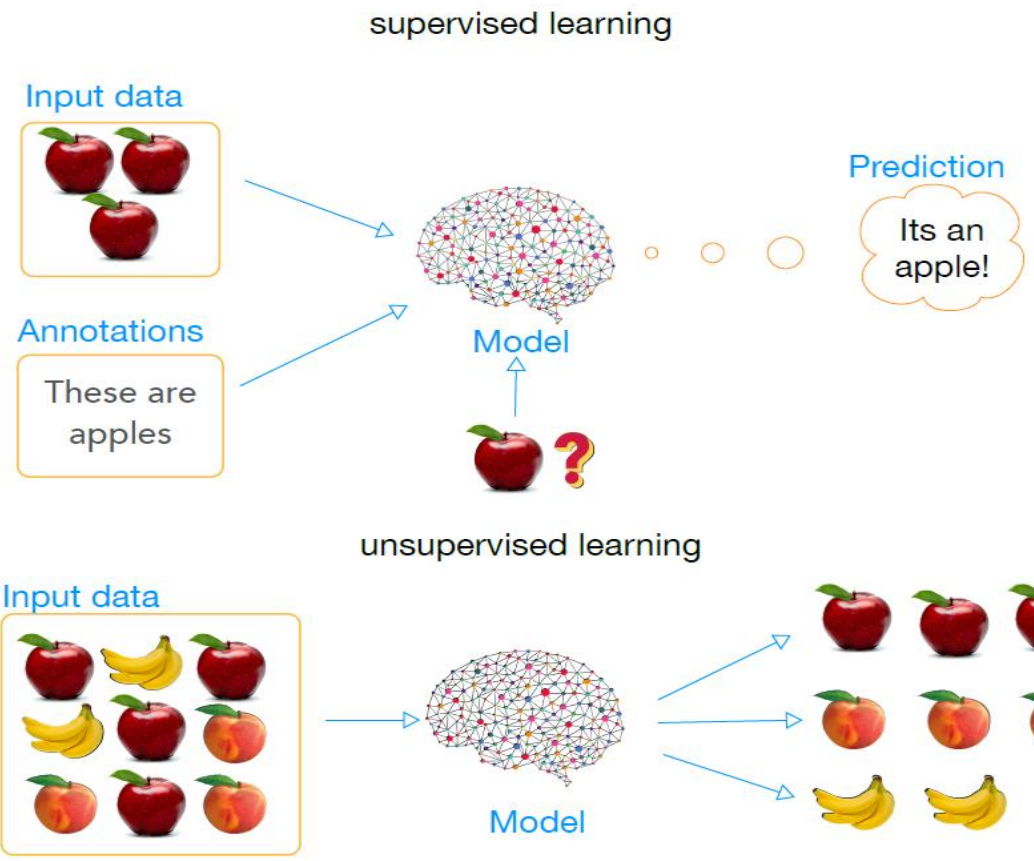
- **Connection to ML:** Training, validation, and testing

Overview of Machine Learning

- Classification using ML

Machine learning (ML) is the process of enabling systems to learn and improve from experience by analyzing data without explicit programming. It allows machines to identify patterns, make predictions, and adapt over time, solving complex problems across various domains.

- **Supervised learning .**
  - **Prediction**
  - **Classification (discrete labels), Regression (real values)**
- **Unsupervised learning.**
  - **Clustering**
  - **Probability distribution estimation**
  - **Finding association (in features)**
  - **Dimension reduction**
- **Semi-supervised learning.**
- **Reinforcement learning.**
  - **Decision making (robot, chess machine)**

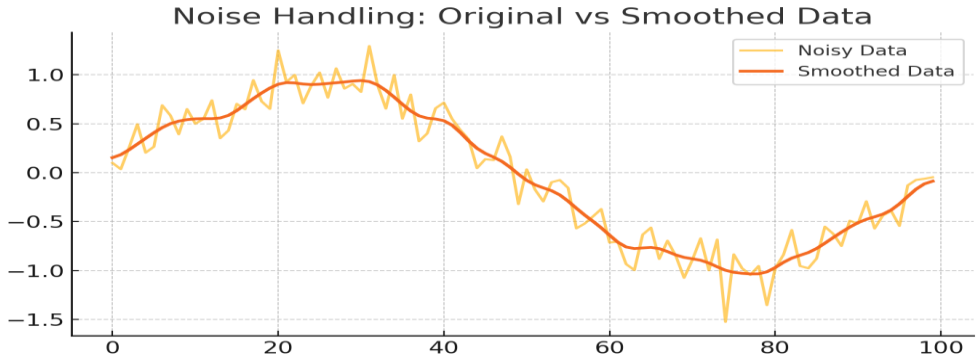


Example: Use K-Means to cluster the customers based on similarity in age, income, and spending score.

Uncertainty in Machine Learning:
(uncertainty present in the Data)

Data Preprocessing:
It is a data mining technique that involves transforming row data into an understandable format.

What & why preprocess the data?
- Incomplete: Lacking values, certain attributes of interest, etc.
- Noisy: Containing errors or outliers
- Inconsistent: Lack of compatibility of similarity between two or more facts



No quality data no quality mining results?
- Quality decision must be based on the quality data
- Data warehouse needs consistent integration of quality data

Major task in Data preprocessing?
- **Data cleaning** i.e., finding missing values, smooth noisy data, identify and remove the outliers, and resolve inconsistencies.
- **Data transformation** i.e., Normalization and aggregation.

Continue...

MISSING VALUES

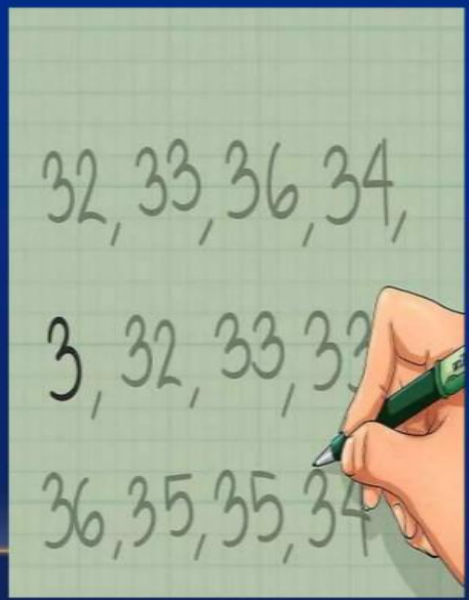
1. Removing the training example
2. Filling in missing value manually:
3. Using a standard value to replace the missing value
4. Using central tendency (mean, median, mode) for attribute to replace the missing value:
5. Using central tendency (mean, median, mode) for attribute belonging to same class to replace the

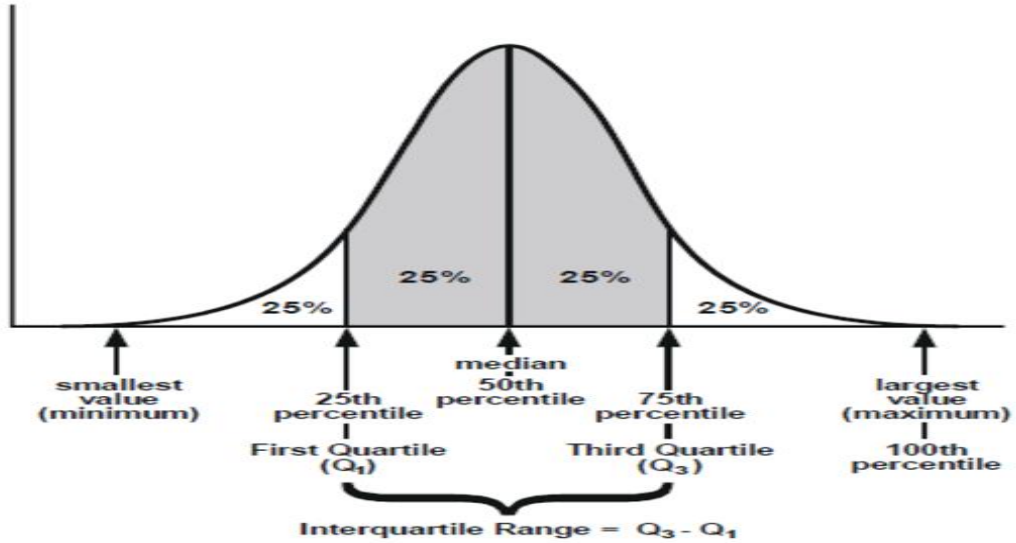
PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	male	22	1	0	A/5 21171	7.25		S
2	1	1	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	female	26	0	0	STON/O2. 3101282	7.925		S
4	1	1	female	35	1	0	113803	53.1	C123	S
5	0	3	male	35	0	0	373450	8.05		S
6	0	3	male		0	0	330877	8.4583		Q

```
df = pd.DataFrame(data)
print(df.isnull().sum())
```

OUTLIERS

Outliers are extreme values that fall a long way outside of the other observations.
For example, in a normal distribution, outliers may be values on the tails of the distribution.
An outlier is an observation that lies an abnormal distance from other values in a random sample from a population.






```
print(df.dropna())  # Drops rows with any missing values
df['Salary'] = df['Salary'].fillna(df['Salary'].median())
df['Age'] = df['Age'].fillna(df['Age'].mean())
```

Continue...

DATA TRANSFORMATION

- **MinMax Scaler**

It just scales all the data between 0 and 1. The formula for calculating the scaled value is-

x_scaled = (x – x_min)/(x_max – x_min)

- Standard Scaler

the Standard Scaler scales the values such that the mean is 0 and the standard deviation is 1(or the variance). df_std

- **MaxAbsScaler**

takes the absolute maximum value of each column and divides each value in the column by the maximum value.

scales the data between the range [-1, 1].

- **Robust Scaler**

to standardizing input variables in the presence of outliers is to ignore the outliers from the calculation of the mean and standard deviation

- **Quantile Transformer Scaler**

converts the variable distribution to a normal distribution. and scales it accordingly.

- **Log Transform**

take the log of the values in a column and use these values as the column instead.

It is primarily used to convert a skewed distribution to a normal distribution/less-skewed distribution

the log-transformed data follows a normal or near normal distribution.

Reducing the impact of too-low values

Reducing the impact of too-high values.

- **Unit Vector Scaler/Normalizer**

Normalization is the process of scaling individual samples to have unit norm.

Normalizer works on the rows

If we are using L1 norm, the values in each column are converted so that the sum of their absolute values along the row = 1

Handling Categorical Data

- Find and Replace
- Label Encoding
- Binary encoding
- One Hot Encoding

```
pd.get_dummies(obj_df, columns=["drive_wheels"]).head()
```

- OrdinalEncoder

```
from sklearn.preprocessing import OrdinalEncoder
```

NumPy and **SciPy**

- Fundamental Packages for scientific computing with Python
- Contains powerful n-dimensional array objects
- Useful linear algebra, random number and other capabilities

Pandas

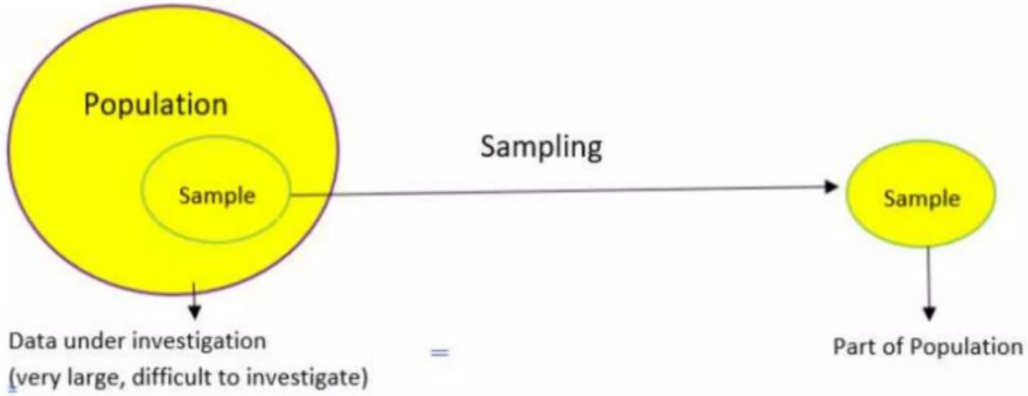
- Contains useful data structures and algorithms

Matplotlib

- Contains functions for plotting/visualizing data.

```
encoded_df = pd.get_dummies(df, columns=['Color', 'Shape'])
```

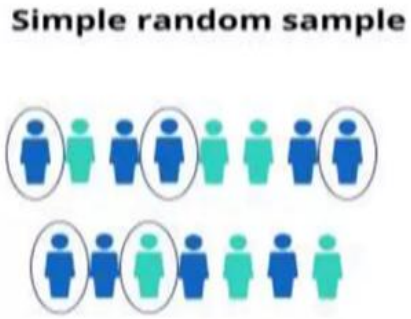
Sampling Techniques

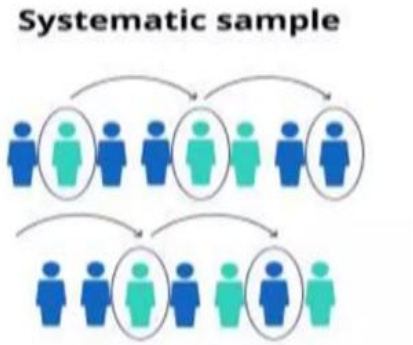


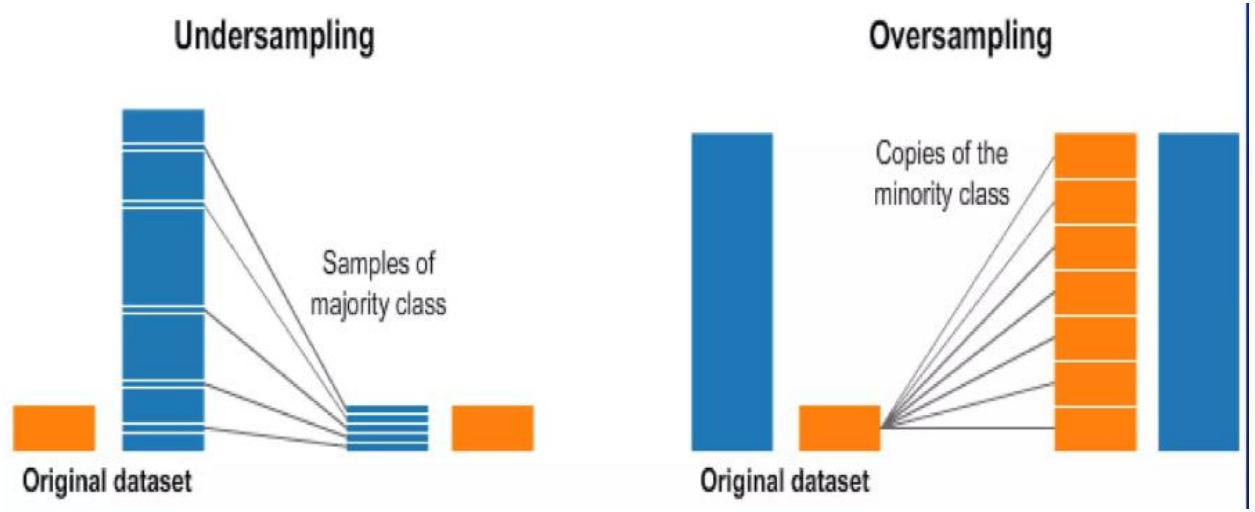
- **Sampling** is done to draw conclusions about the populations from samples. It enables us to determine a population's characteristics by directly observing only a portion (or sample) of the population

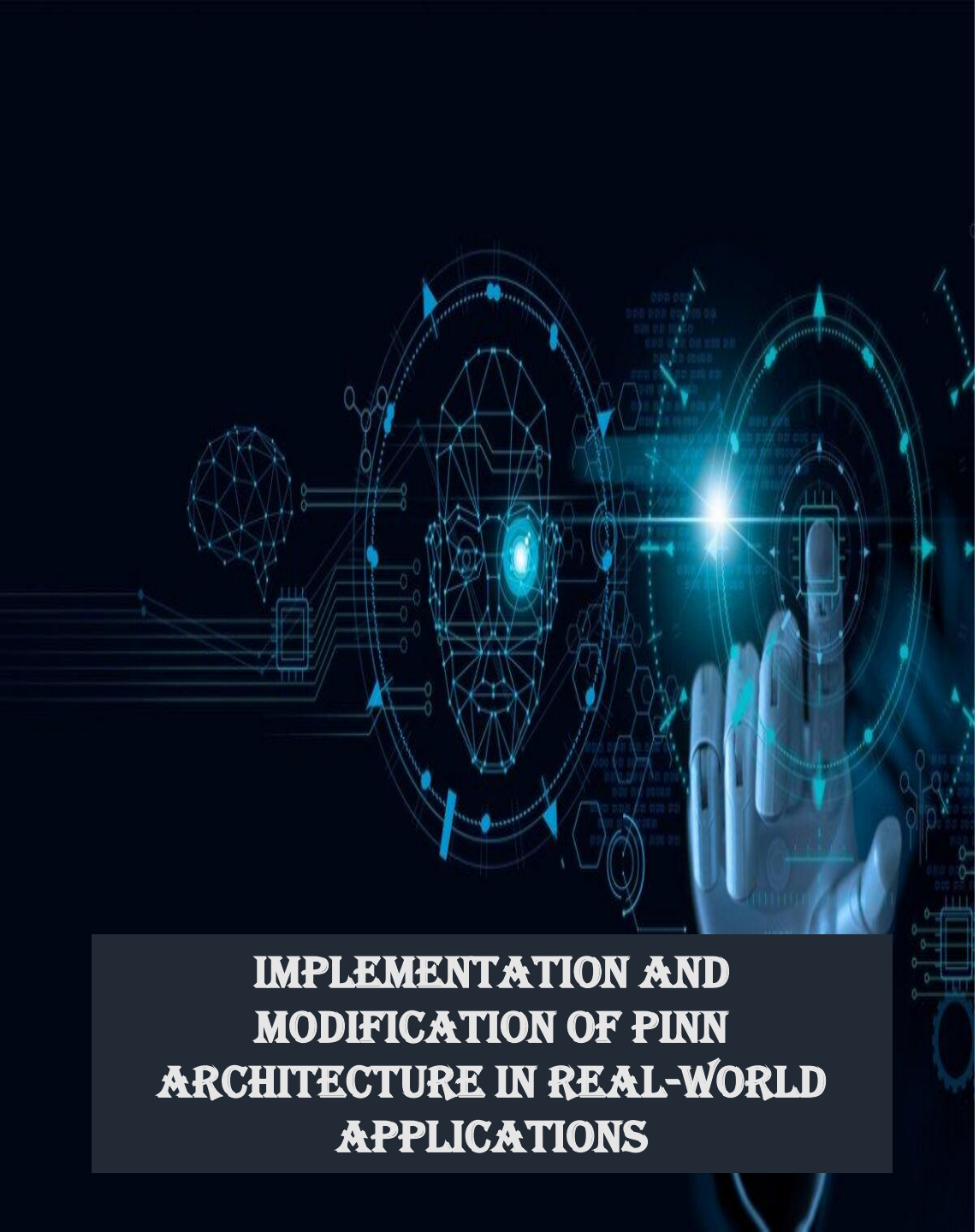
- **Types of sampling: 1. Simple Random Sampling, 2. Systematic Sampling, 3. Stratified Sampling 4. Cluster Sampling**

- **Resampling** is a series of methods to reconstruct the sample data sets including training and validation data sets.
  - **Cross-validation**
  - **Imbalance dataset**

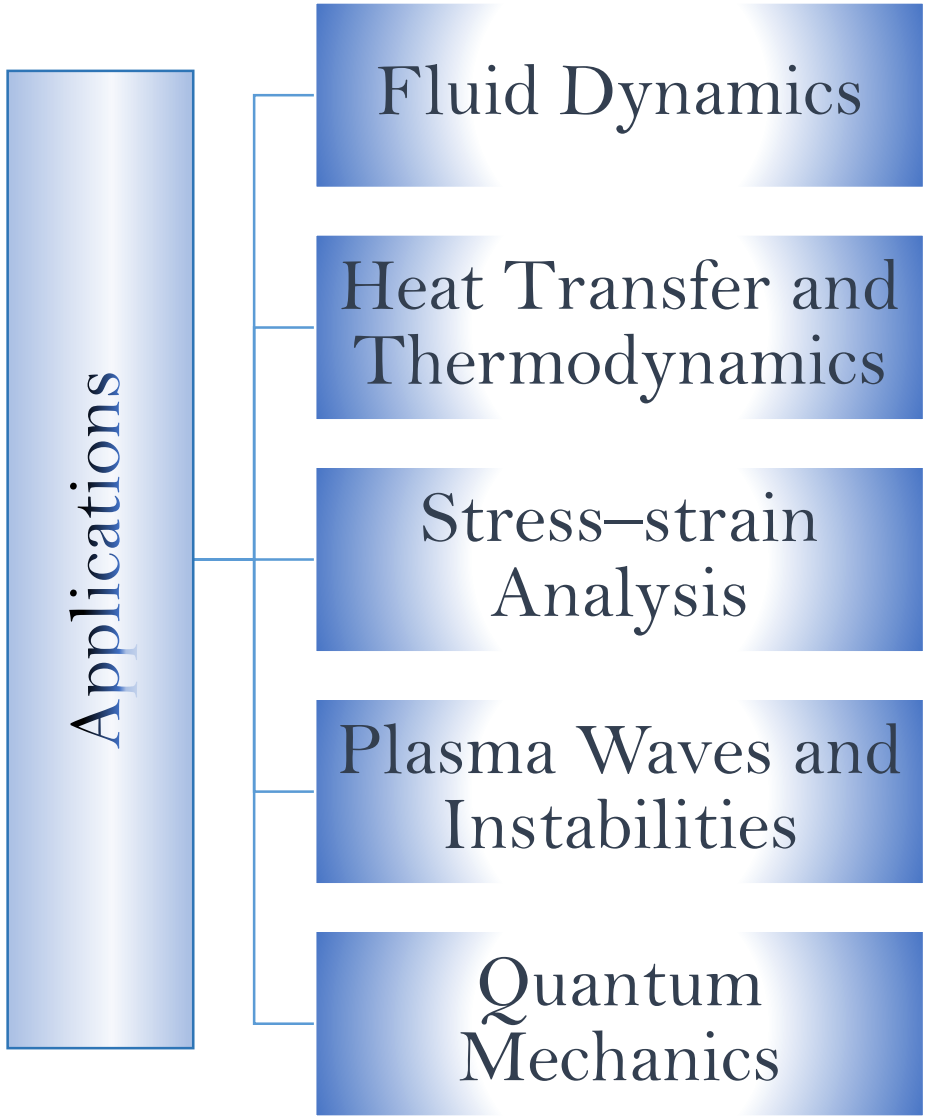






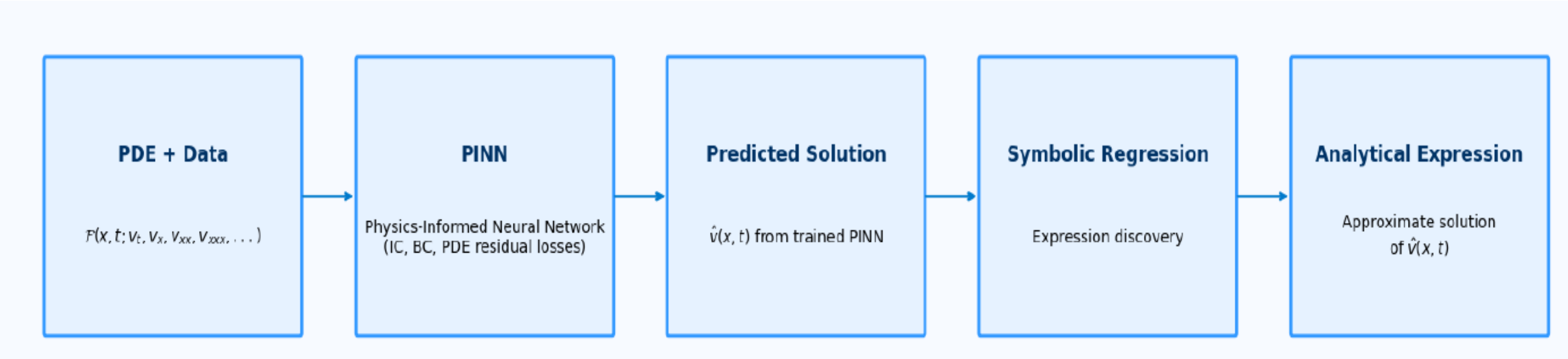


IMPLEMENTATION AND MODIFICATION OF PINN ARCHITECTURE IN REAL-WORLD APPLICATIONS



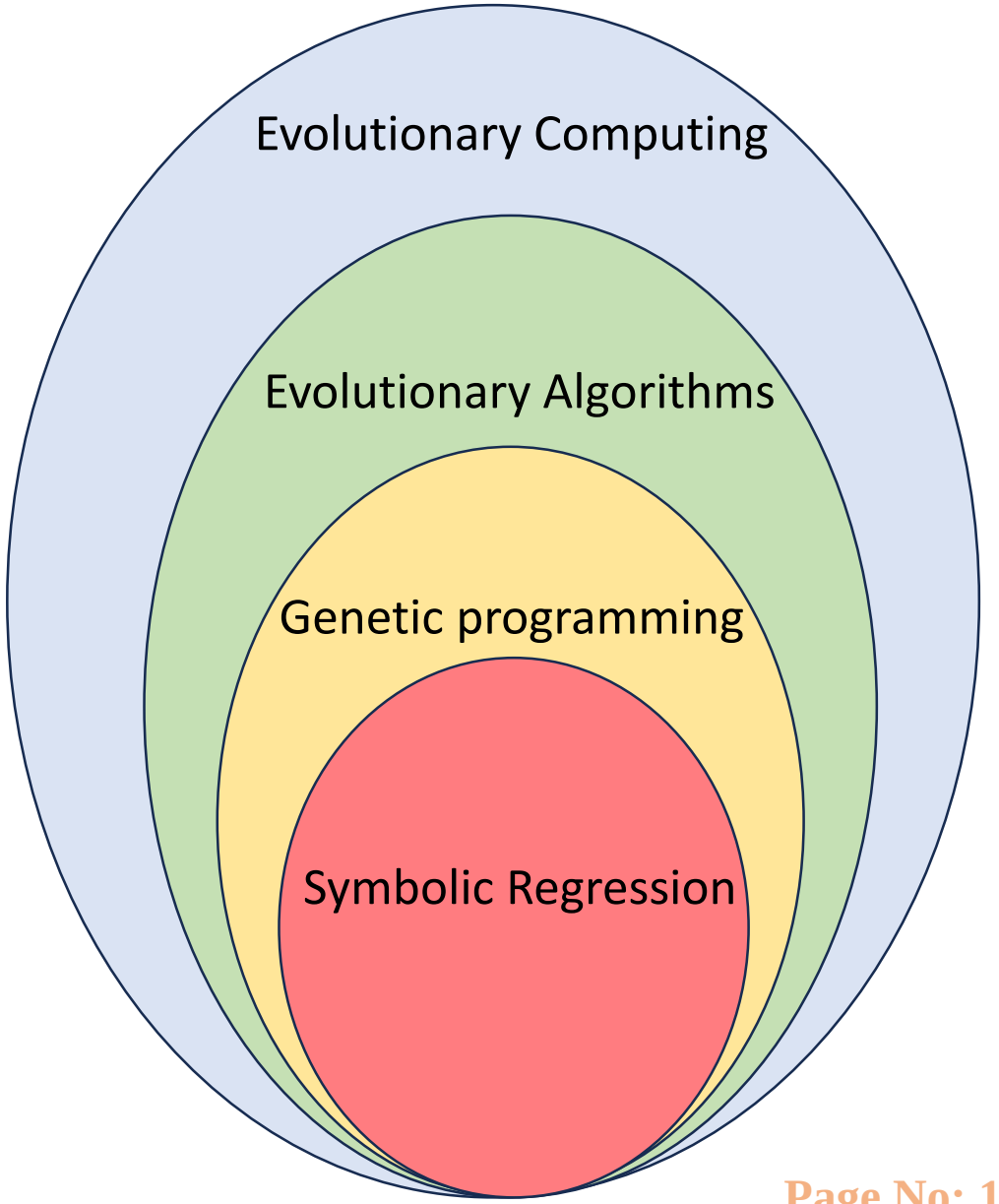
A Hybrid Symbolic Regression and Physics-informed Neural Networks Approach for deriving analytical Solution of PDEs

Methodology for Converting PINN-Based Numerical Solutions of non-linear PDEs into Interpretable Symbolic Form:



Uses of Symbolic Regression

Model discovery without present Structure	• Symbolic regression does not require predefined model structures. It automatically discovers the best-fitting equation by exploring various functional forms.
Interpretability	• The output of Symbolic Regression is an explicit mathematical equation, making it easier to understand and interpret as compared to black-box methods like Neural Networks
Adaptivity to complex relationships	• Symbolic Regression can handle non-linear, non-parametric, and multidimensional relationships • It is capable of capturing complex interactions and dependencies in data.
Automatic Feature Selection	• Through its evolutionary process, Symbolic Regression inherently selects the most relevant variables and estimates redundant ones.



Physics-Informed Neural Network with Symbolic Regression for Deriving Analytical Approximate Solutions to Nonlinear Partial Differential Equations [Published in " neural computing and applications", springer nature, 2025]

Model equations and the PINN predicted Solutions:

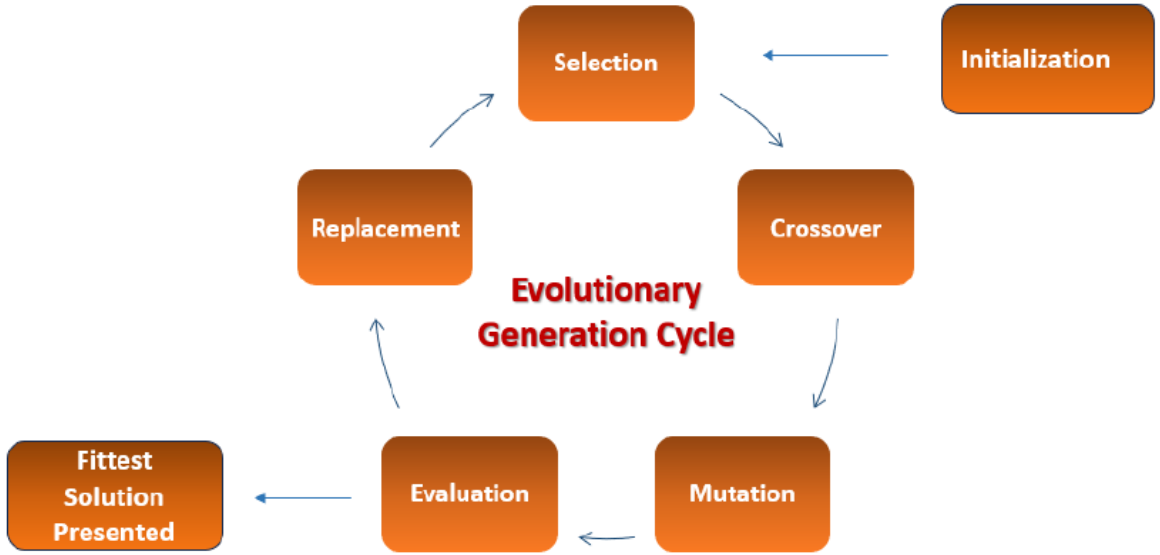
We solved a forced nonlinear evolutionary equation,

$$v_t + vv_x = \frac{1}{2}v_{xx} + \frac{M}{2}v,$$

$$v_t + vv_x + \frac{1}{2}v_{xxx} + \frac{1}{2}v_{xxxxx} = \frac{M}{2}v.$$

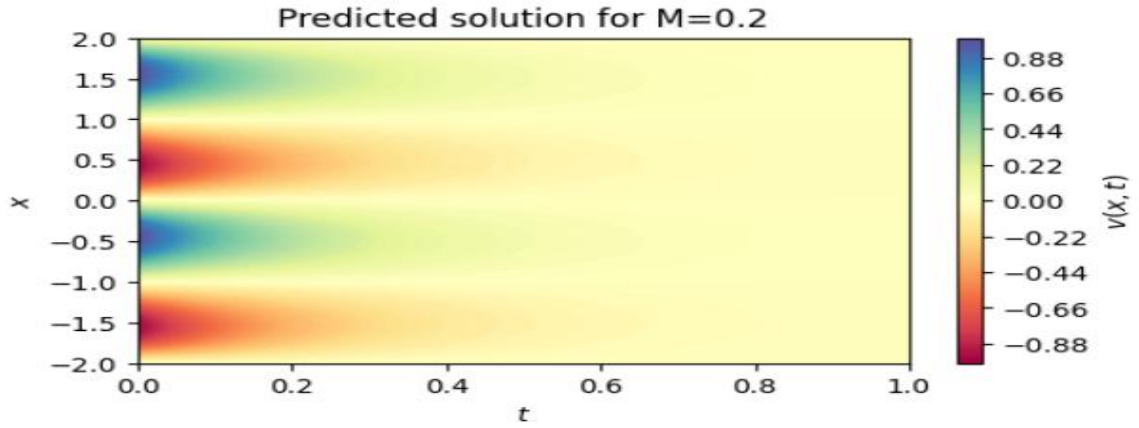
With the initial pulse wave: $v(x,0) = -sin(\pi x)$,

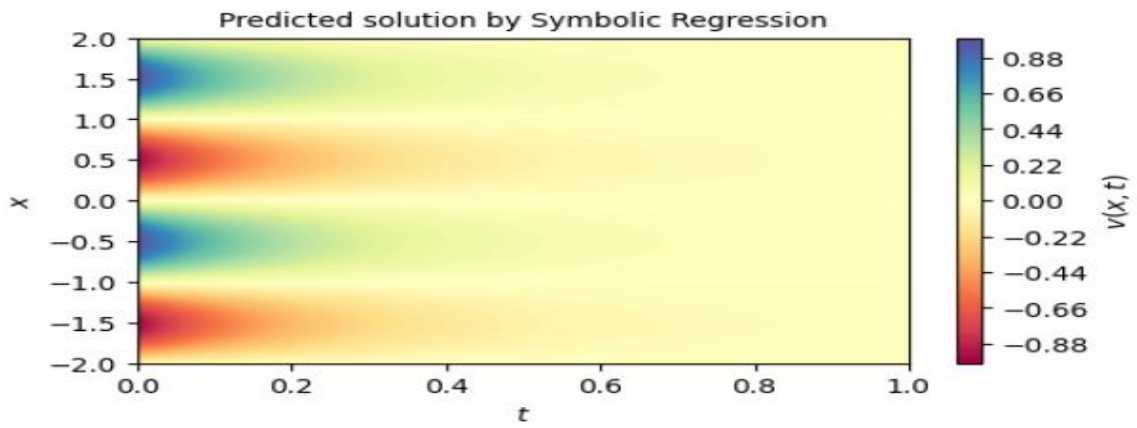
and boundary condition: $v(-2,t) = v(2,t) = 0$.



The symbolic solution for M = 0.2 is obtained as:

$$v(x,t) = e^{t(-3.76)} \sin\left(\frac{x}{-0.322}\right)(-0.321)\frac{1}{(-0.797)\,(t \cdot 0.560 + 0.411)}$$

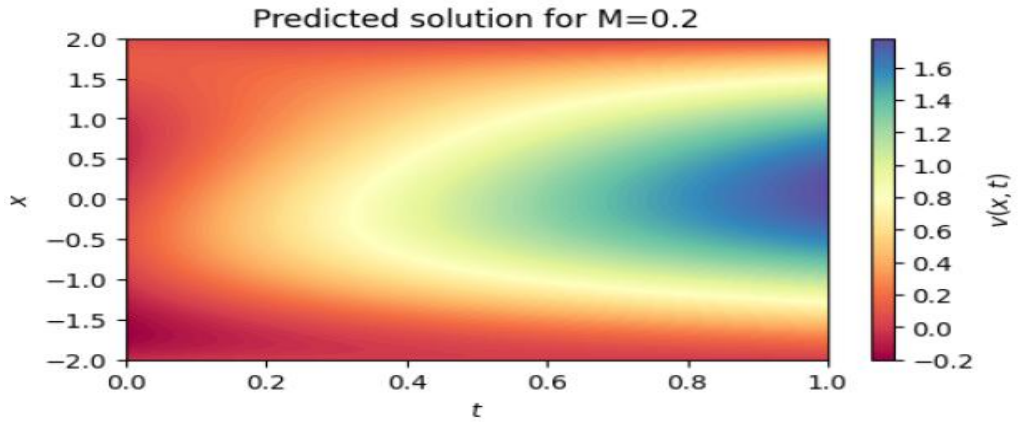


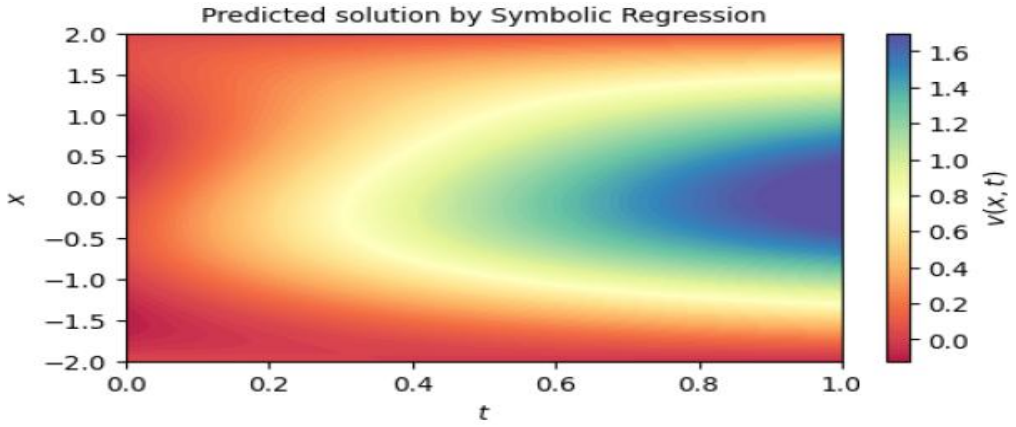


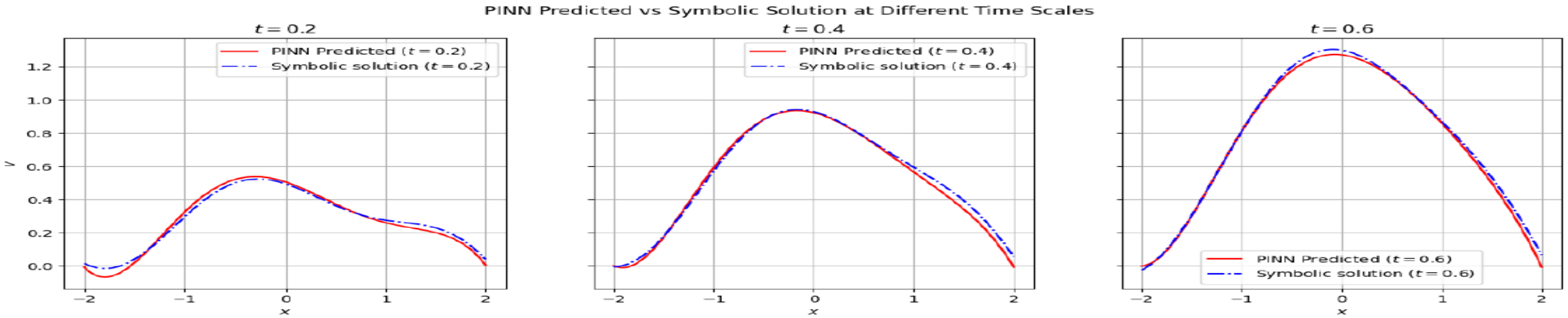
PINN-Based and Symbolic Regression-Based Solutions:

The obtained an analytical expression representing the solution to the PDE is, $v(x,t) = (\cos(0.0656 - x) + 0.414) \cdot 1.70 \sin(\sin(0.0760)x + t - \cos(0.177 - x)(x - 0.0541) \cdot 0.176)$

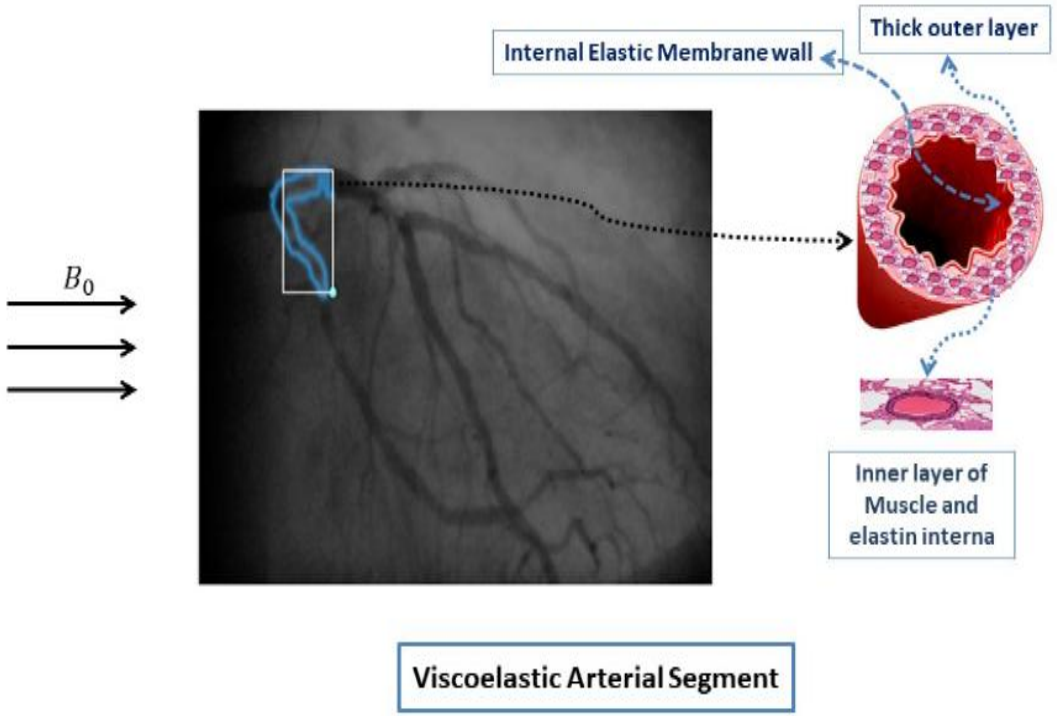
Loss Type	M = 0.2	M = 0.4
Mean Residual Loss	0.006	0.0007
MSE loss of symbolic regression	0.0008	0.0009
Mean Absolute error between PINN and SR model	0.02	0.02
Complexity of discovered equation	27	23







Effects of viscosity and induced magnetic fields on weakly nonlinear wave transmission in a viscoelastic tube using physics-informed neural networks [Published in the Journal "Physics of Fluids", AIP Publishing, 2025]



This study presents an advanced deep learning methodology that utilizes physics-informed neural networks (PINNs), to analyze the transmission of weakly nonlinear waves in a prestressed viscoelastic arterial tube. These insights aid in obtaining precise measurements of pulse wave velocity for cardiovascular health monitoring. Consequently, the application of PINN proves to be a valuable tool for solving real-world PDEs and highlights its importance in advancing medical machine learning fields.

Final Flow Equations Effects of viscosity and induced magnetic fields

The perturbed Burger, Korteweg-de Vries (Kdv), and Korteweg-de Vries-Burgers (KdVB) equations are,

$$\frac{\partial U}{\partial t} + 0.1U\frac{\partial U}{\partial X} - \Gamma\frac{\partial^2 U}{\partial X^2} + a_4 U = 0,$$

$$\frac{\partial U}{\partial t} + 0.1U\frac{\partial U}{\partial X} + 0.1\frac{\partial^3 U}{\partial X^3} + a_4 U = 0,$$

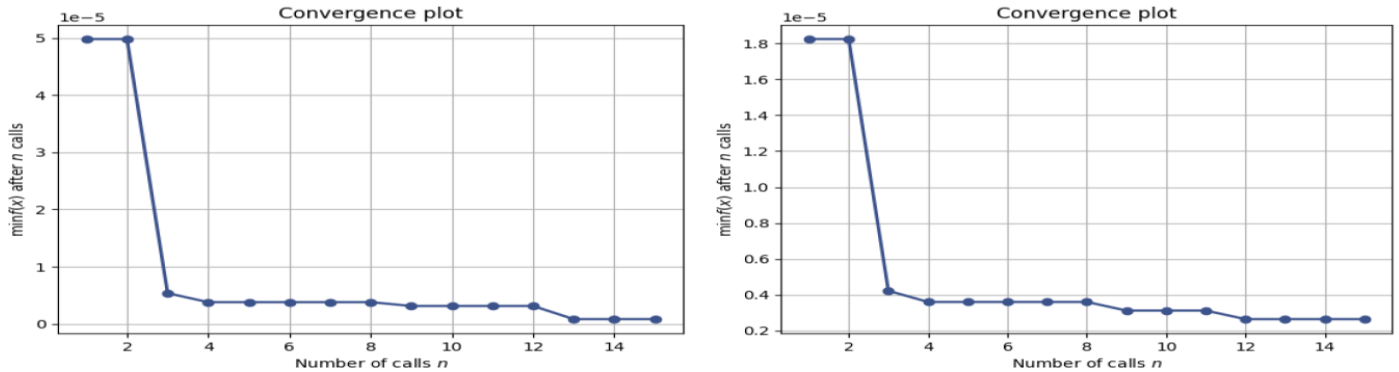
$$\frac{\partial U}{\partial t} + 0.1U\frac{\partial U}{\partial X} + 0.1\frac{\partial^3 U}{\partial X^3} - \Gamma\frac{\partial^2 U}{\partial X^2} + a_4 U = 0.$$

A PDE solution can be approximated as:

$$U(X,t) \approx \tilde{U}(X,t) = \mathbf{NN}_{\tilde{\boldsymbol{\theta}}}\left(X,t;\mathcal{T}^{Bc},\mathcal{T}^{Ic}\right).$$

Bayesian Hypermeter Optimization

Hyperparameters	Values		Optimal values for Perturbed PDEs	
	Range of parameters	Initial setting	Perturbed Burger	Perturbed KdvB
No. of layers	1 - 10	2	4	8
Learning rate	$1e^{-6}$ - $1e^{-1}$	$1.42e^{-4}$	$1.416e^{-4}$	$4.838e^{-4}$
No. of nodes	5 – 500	8	16	5
Initializer	–	–	Glorot normal	Glorot normal
Activation functions	Sigmoid sin, tanh	Sigmoid	sin	tanh
Optimizer	Adam, L-BFGS	Adam	Adam	Adam
$\boldsymbol{\lambda}_{ic} = \boldsymbol{\lambda}_{bc}$	$1e^{-2} - 1e^{-4}$	$0.2e^{-2}$	$0.311e^{-3}$	$0.1e^{-2}$



Figs.: The convergence of PINN-predicted solutions versus number of calls.

PINN based Solutions of the perturbed PDEs with residual losses:

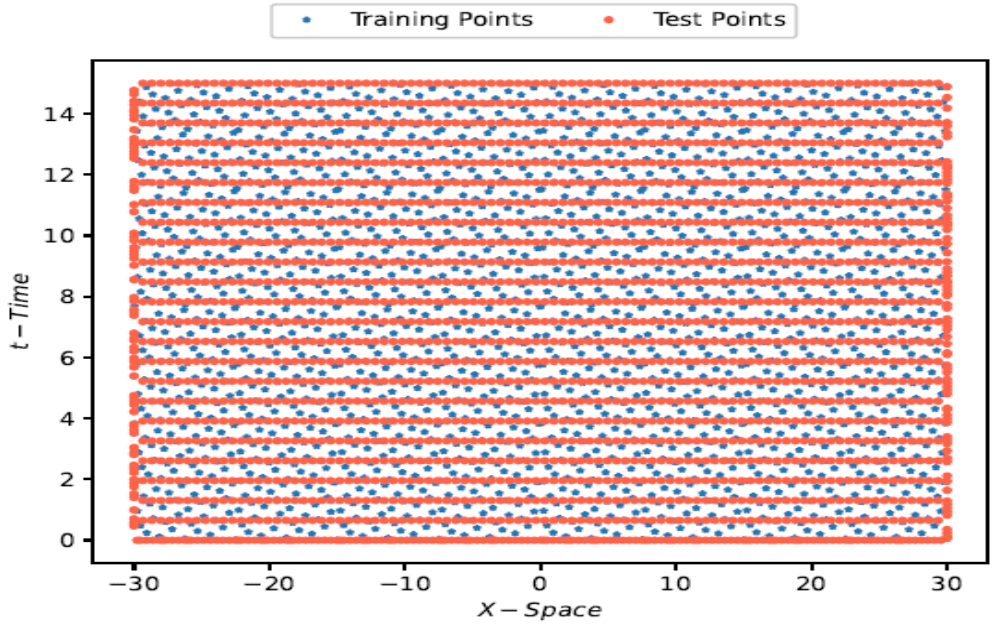
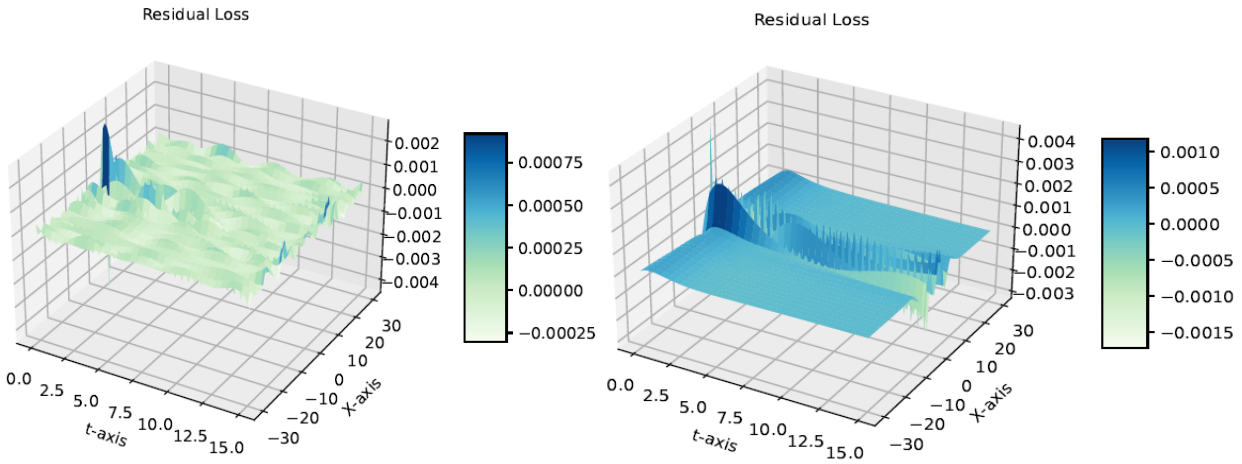
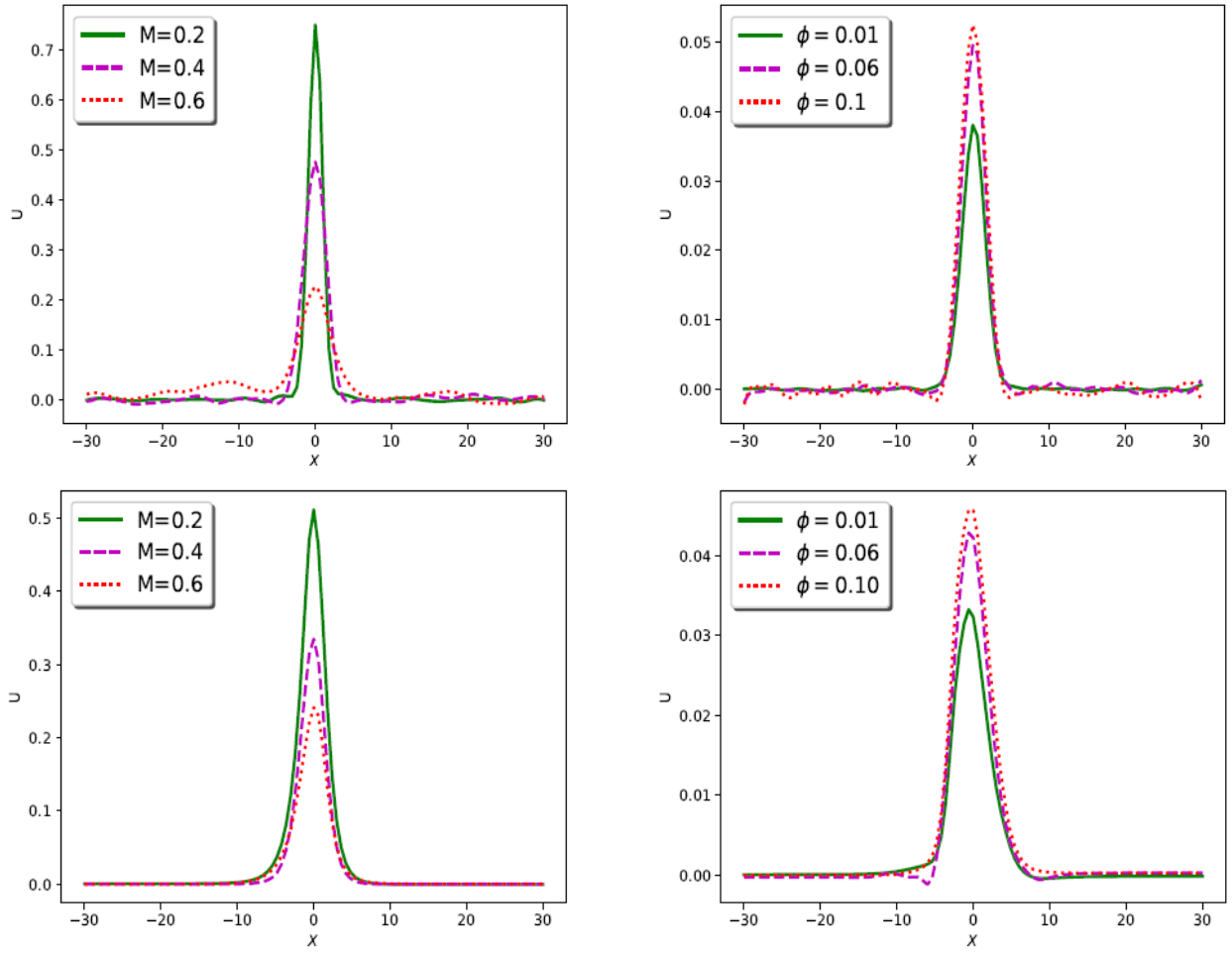


Fig.: Training and Testing Data points.

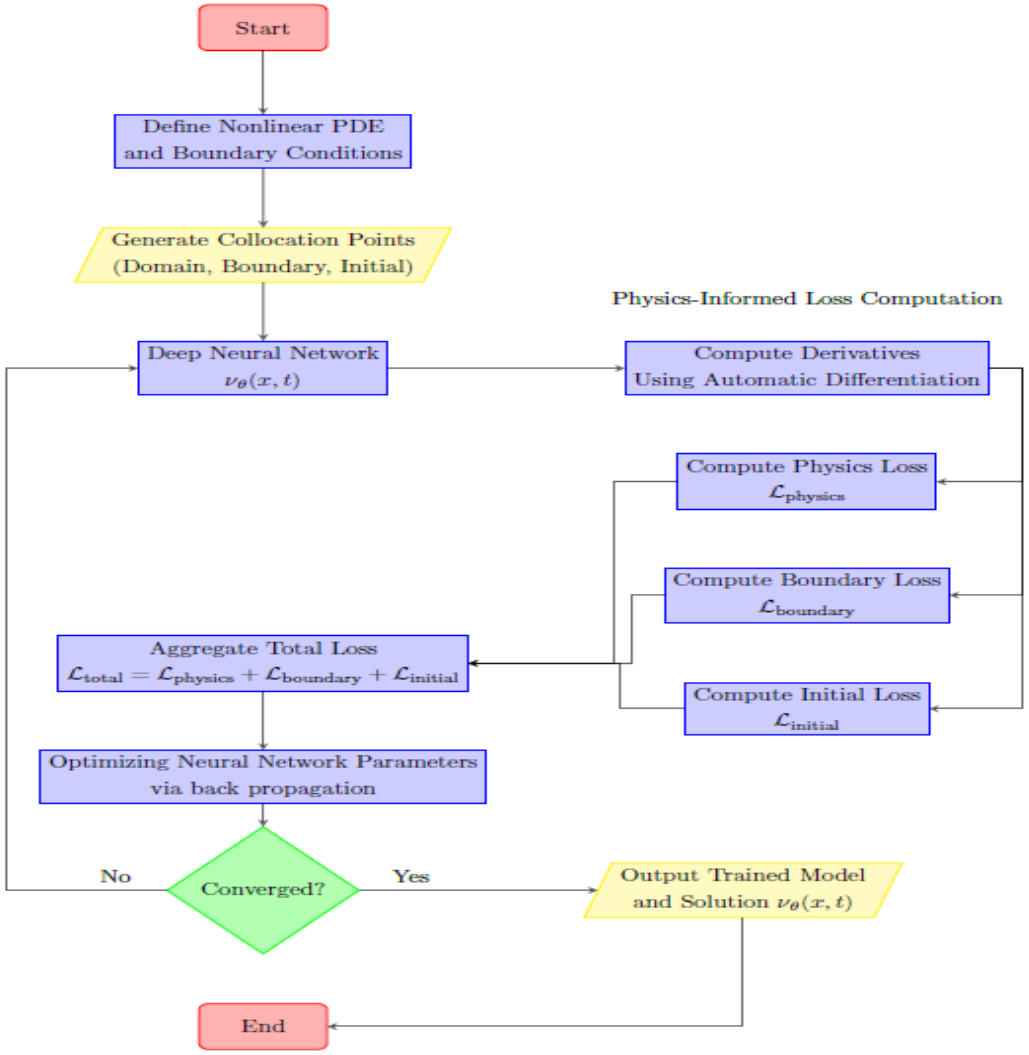


Figs.: Residual losses for perturbed Burger and KdvB equations.



Figs.: Predicted solutions of perturbed Burger and KdvB equations at time steps t = 0.6 and t = 9 for various M and ϕ with initial pulse $u(x, 0) = \operatorname{sech} X$.

Gradient Enhanced Self-Training Physics-Informed Neural Network (gST-PINN) for Solving Nonlinear Partial Differential Equations [arXiv preprint arXiv:2510.10483, 2025]



- Traditional PINNs often struggle with challenges such as limited precision, slow training dynamics, lack of labeled data availability, and inadequate handling of multi-physics interactions.

- We proposed a Gradient Enhanced Self-Training PINN (gST-PINN) method that specifically introduces a gradient based pseudo point self-learning algorithm for solving PDEs.

Strategy 1: Enhanced Self-training physics-informed neural networks (ESTPINNs) with Gradient Guidance for solving High Dimensional PDEs

The PDE with its gradients are defined as follows:

$$\mathcal{G} \equiv \frac{\partial}{\partial t}u(t,x) + \frac{\partial}{\partial x}(u^2(t,x)/2) - (\nu/\pi)\frac{\partial^2}{\partial x^2}u(t,x) = 0, \quad (x,t) \in \Omega = (0,1) \times [0,2]$$

$$\nabla\mathcal{G} = \left(\frac{\partial \mathcal{G}}{\partial t}, \frac{\partial \mathcal{G}}{\partial x}\right), \quad \nabla^2\mathcal{G} = \left(\frac{\partial^2 \mathcal{G}}{\partial t^2}, \frac{\partial^2 \mathcal{G}}{\partial x \partial t}, \frac{\partial^2 \mathcal{G}}{\partial x^2}\right) \ldots, \quad \forall (x,t) \in \Omega.$$

With the boundary conditions:

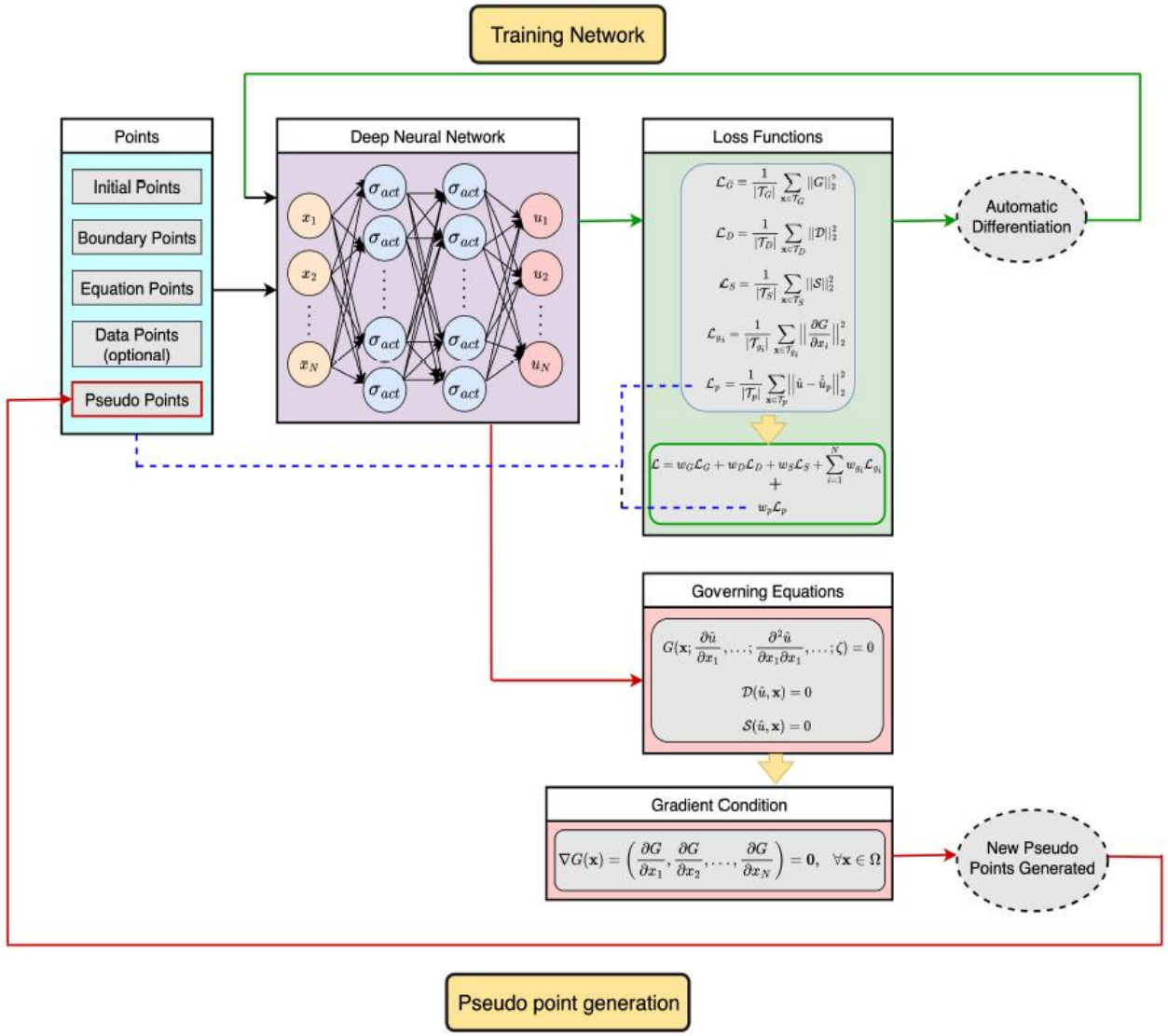
$$u(0,x) = u_0(x) = \sum_{i=1}^{N} A_i \sin(2\pi n_i)x + \Phi_i, \quad x \in (0,1),\ u(0,t) = u(1,t).$$

An additional loss term: $\mathcal{L}_{g_i}(\tilde{\theta}; T_{g_i}) = \frac{1}{|T_{g_i}|} \sum_{(x,t)\in\mathcal{T}_{g_i}} \left(\left\| \frac{\partial \mathcal{G}}{\partial x} \right\|_2^2 + \left\| \frac{\partial \mathcal{G}}{\partial t} \right\|_2^2 \right)$

Total loss: $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{PINN}}(\tilde{\theta}; x, t) + \sum w_{g_i} \cdot \mathcal{L}_{g_i}(\tilde{\theta}; T_{g_i})$

EST-PINN with Pseudo-Point Generation:

- ❑ Select top q% samples with the lowest residual and gradient values.
- ❑ Continue training with the enriched data until convergence.



Results on Burger Equations

For a field u(x, t), the general form of Burgers equation is given as,

$$\frac{\partial}{\partial t}(u(t,x)) + \frac{\partial}{\partial x}(u^2(t,x)/2) = (\eta_v/\pi)\frac{\partial^2}{\partial x^2}, (u(t,x))$$

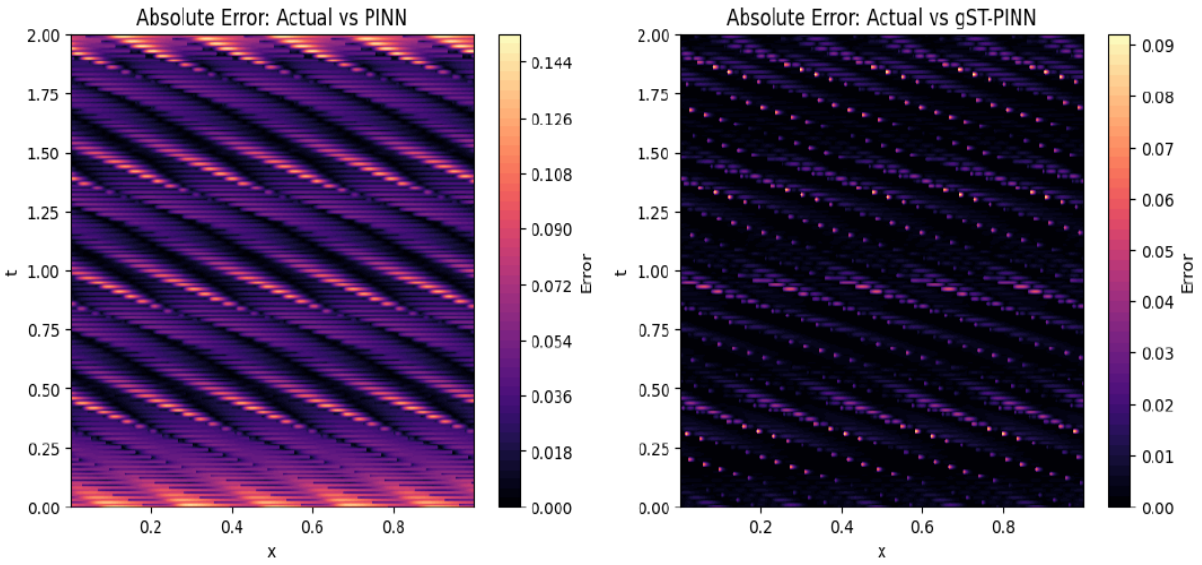
Here, η_v $(kinematic\ viscosity) \rightarrow 0.01$ & $x \in (0,1),\ t \in [0,2].$

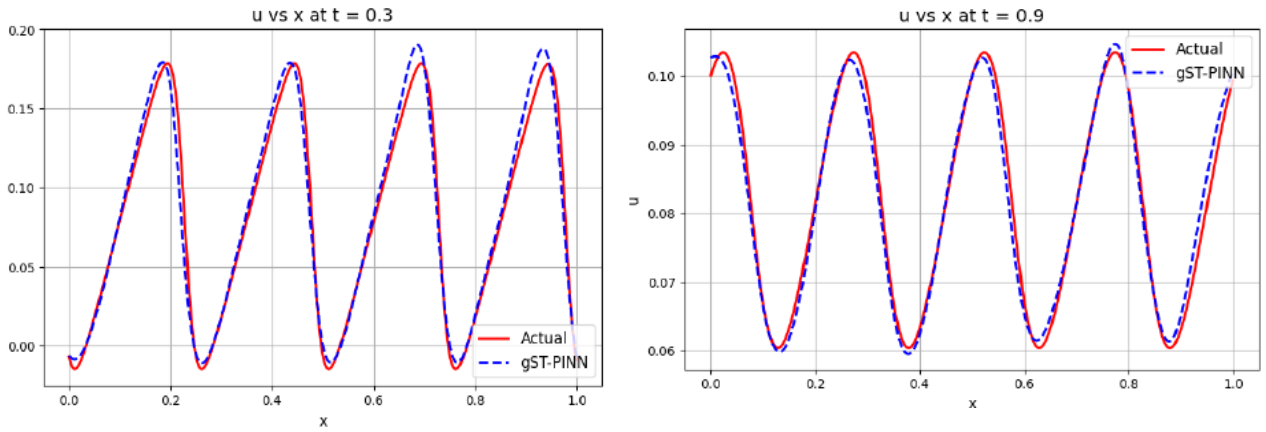
With initial and periodic boundary condition:

$$u(t,0) = u(t,1) \approx \tilde{B}_1(t)\ \forall t \in [0,2]$$

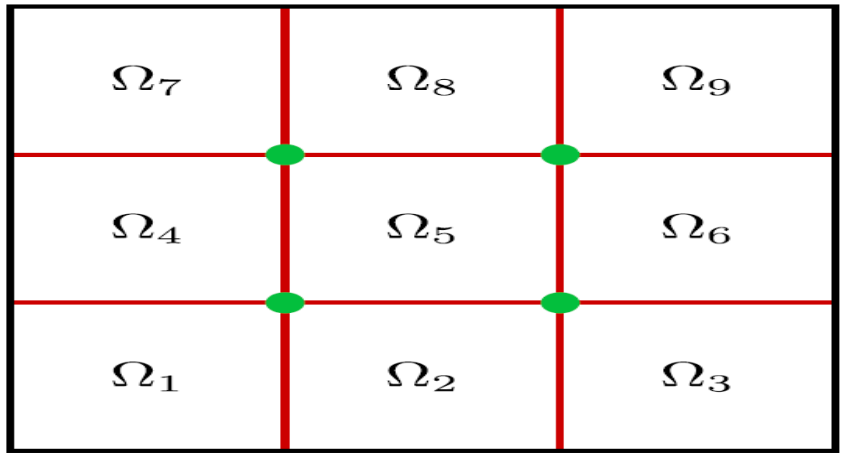
$$u(0,x) = u_0(x) = \sum_{i=1}^{N} \lambda_i \sin(2\pi f_i/L_x)x + \Psi_i \approx \tilde{I}_1(x)$$

Table 1: Burgers' Equation		
	L_2 error in u	MSE error in u
PINN with labelled data	4.051458×10^{-1}	1.488497×10^{-3}
ST-PINN with labelled data	8.154244×10^{-2}	6.029660×10^{-5}
ST-PINN without labelled data	4.056303×10^{-1}	1.492060×10^{-3}
gST-PINN with labelled data	7.732166×10^{-2}	5.421603×10^{-5}
gST-PINN without labelled data	3.979561×10^{-1}	1.436137×10^{-3}

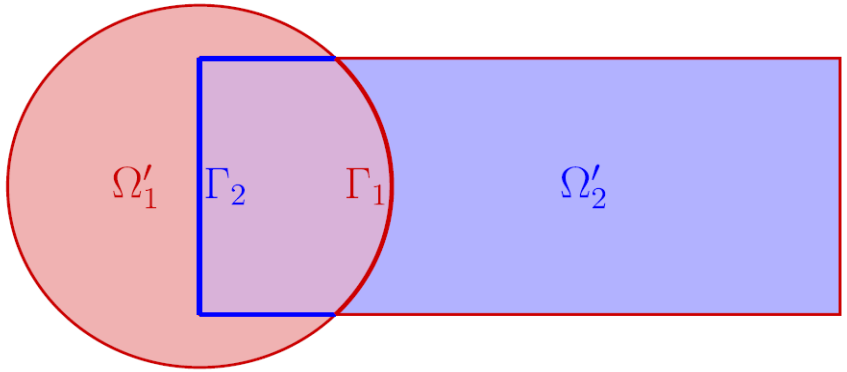




Strategy 2: Domain Decomposition Approach on Semi-supervised physics-informed neural networks with Gradient Guidance.



Decomposition of the domain Ω into nine non-overlapping subdomains Ω_i $(i = 1, 2, \dots, 9)$



Decomposition of the domain Ω into two overlapping subdomains Ω'_1 and Ω'_2. The boundary of the overlapping region is partitioned into Γ_1 and Γ_2 accordingly

- We introduce a hybrid approach combining PINNs, pseudo-point self-training, and domain decomposition for efficient nonlinear PDE solving in multi-physics and multiscale regimes.
- Our method uses irregular convex/nonconvex space–time decompositions with smoother boundaries, suitable for problems with cracks, shocks, and other non-smooth features.

Strategy 3: Analytical Expressions via Genetic-based Symbolic Regression in terms of inputs variables i.e. deriving approximate analytical solutions.

- The gST-PINN with genetic-based symbolic regression to build an interpretable and data-efficient framework for deriving approximate analytical solutions of nonlinear PDEs.

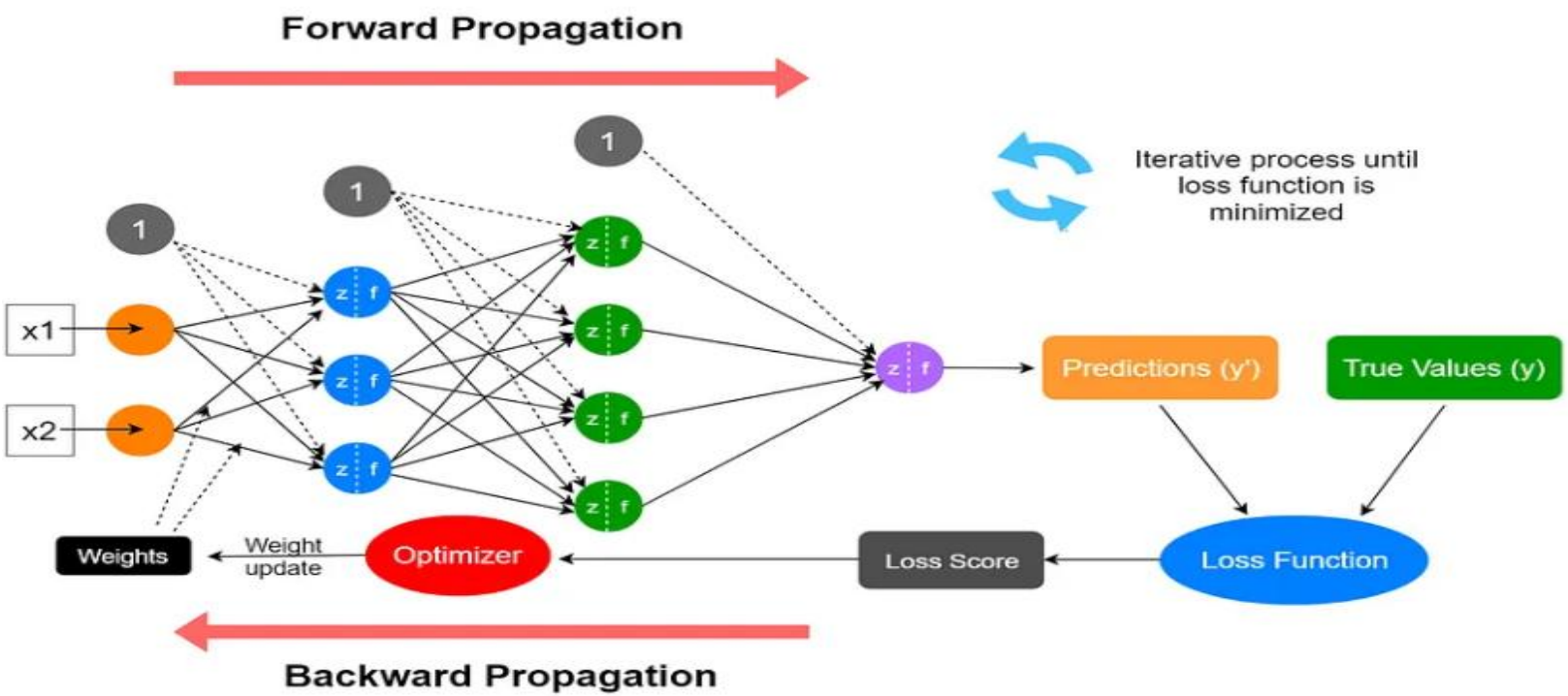
- Unlike traditional PINNs that produce black-box solutions, this hybrid approach extracts explicit analytical expressions that retain physical meaning and enhance generalizability.

- Beyond forward simulation or parameter estimation, the framework provides a transparent and compact representation of system behavior, validated through extensive numerical experiments and statistical analysis.

Outcomes:

- The proposed gST-PINN approach provides an efficient and robust framework for solving nonlinear PDEs including evolutionary flow equations.
- This gST-PINN approach is outperform standard PINNs by achieving faster convergence, reduced L2 relative and mean squared errors (MSE), and enhanced long-term predictive accuracy.
- The proposed method combines domain decomposition, gradient constraints, and adaptive data selection to leverage neural networks for solving higher-order PDEs with complex geometries.
- Finally, the gST-PINN is versatile, applicable to diffusion–reaction systems, diffusion–sorption models, the Black–Scholes equation, coupled PDEs, and multi-physics problems.

NEURAL NETWORKS CONSTRUCTION USING PYTHON AND PYTORCH LIBRARY



Example: Construction of a simple Neural Network model

```
import numpy as np
```

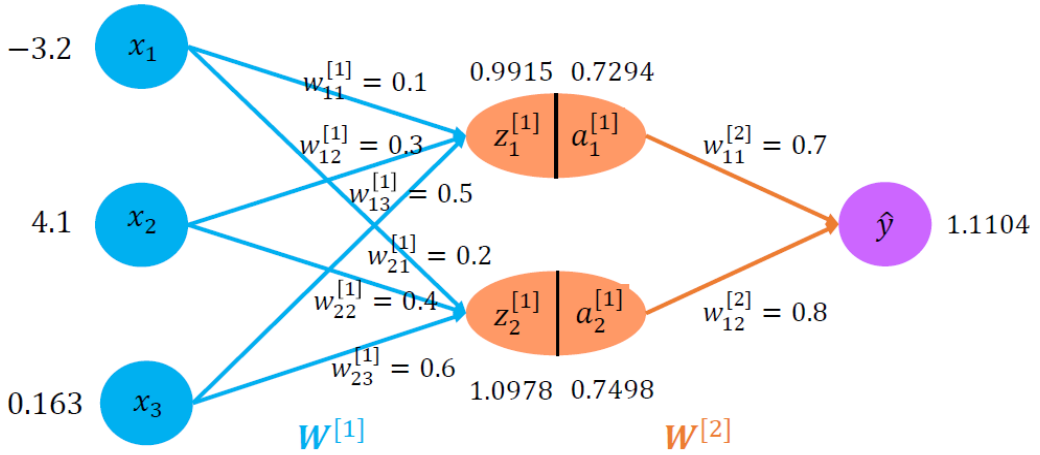
```
def sigmoid(x):
    return 1/(1+np.exp(-x))
```

```
X = np.array(([-3.2, -1.7, -4.16, -2.73],
              [4.1, 3.9, 4.33, 3.14],
              [0.163, 1.35, 0.78, 1.07]))
W1 = np.array(([0.1, 0.3, 0.5],
               [0.2, 0.4, 0.6]))
Z1 = np.dot(W1, X)
print(Z1)
```

```
[[0.9915 1.675  1.273  1.204 ]
 [1.0978 2.03   1.368  1.352 ]]
```

```
A1 = sigmoid(Z1)
print(A1)
```

```
[[0.7293841  0.84224131 0.78125587 0.7692356 ]
 [0.74984766 0.88391108 0.79705683 0.79445641]]
```



$$W^{[1]}x = z^{[1]} \Rightarrow \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} -3.2 \\ 4.1 \\ 0.163 \end{bmatrix} = \begin{bmatrix} 0.9915 \\ 1.0978 \end{bmatrix}$$

$$a^{[1]} = \sigma(z^{[1]}) \Rightarrow \sigma\left(\begin{bmatrix} 0.9915 \\ 1.0978 \end{bmatrix}\right) = \begin{bmatrix} \frac{1}{1+e^{-0.9915}} \\ \frac{1}{1+e^{-1.0978}} \end{bmatrix} = \begin{bmatrix} 0.7294 \\ 0.7498 \end{bmatrix}$$

Continue…

```
[ ] W2 = np.array(([[0.7, 0.8]]))
    y_hat = np.dot(W2, A1)
    print(y_hat)
```

```
[[1.110447   1.29669778 1.18452457 1.17403005]]
```

$$z^{[2]} = W^{[2]}a^{[1]} = \hat{y} \Rightarrow [0.7 \quad 0.8]\begin{bmatrix}0.7294\\0.7498\end{bmatrix} = 1.1104$$

```
[ ] y = np.array([[2, 6.2, 3.5, 4.2]])
    MSE = np.square(np.subtract(y, y_hat)).mean()
    print(MSE)
```

```
9.837899450853262
```

$$MSE = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(1.1104 - 2)^2 = 0.3957 \ (Actual\ value = 2, say)$$

```
[ ] diff = np.subtract(y_hat, y)
    print(diff)
```

```
[[-0.889553   -4.90330222 -2.31547543 -3.02596995]]
```

$$\delta^{[2]} = \frac{\partial \mathcal{L}}{\partial \hat{y}} = (\hat{y} - y)$$

```
[ ] dW2 = np.dot(diff, A1.T)
    print(dW2)
```

```
[[-8.91525207 -9.25067913]]
```

$$\frac{\partial \mathcal{L}}{\partial w_{11}^{[2]}} = \delta^{[2]}a_1^{[1]}$$

$$\frac{\partial \mathcal{L}}{\partial w_{12}^{[2]}} = \delta^{[2]}a_2^{[1]}$$

Continue…

```
[ ]  def d_sigmoid(x, keepdims = True):
         return x*(1-x)
```

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

```
[ ]  dZ1 = np.multiply(np.dot(W2.T, diff), d_sigmoid(A1))
     print(dZ1)
```

$$\delta^{[1]} = (W^{[2]})^T \delta^{[2]} \odot \sigma'(Z^{[1]})$$

```
[[-0.12290781 -0.45605427 -0.27699244 -0.37600259]
 [-0.13348714 -0.40251123 -0.29963593 -0.39530163]]
```

```
     dW1 = np.dot(dZ1, X.T)
     print(dW1)
```

$$\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \delta^{[1]} X^T$$

```
[[ 3.3473729  -4.66255909 -1.25408412]
 [ 3.43708686 -4.65576177 -1.22183734]]
```

```
[ ]  def gradient_descent(W, dW, learning_rate):
         return W - learning_rate*dW
```

$$W \leftarrow W - \eta \frac{\partial \mathcal{L}}{\partial W}$$

Steps

Continue…

```
[ ]  W2 = gradient_descent(W2, dW2, 0.1)
     W1 = gradient_descent(W1, dW1, 0.1)
     print(W2)
     print(W1)
```

```
[[1.59152521 1.72506791]]
[[-0.23473729  0.76625591  0.62540841]
 [-0.14370869  0.86557618  0.72218373]]
```

```
[ ]  Z1 = np.dot(W1, X)
     A1 = sigmoid(Z1)
     y_hat = np.dot(W2, A1)
     print(y_hat)
```

```
[[3.26041908 3.27655151 3.29073445 3.24398985]]
```

```
[ ]  MSE = np.square(np.subtract(y, y_hat)).mean()
     print(MSE)
```

```
2.77323870230118
```

Steps

Now we will update the weights using gradient decent algorithm

With **learning rate** $\eta = 0.1$, the weight update equations become:

$$W^{[1]} = W^{[1]} - 0.1\,\frac{\partial J}{\partial W^{[1]}}$$

$$W^{[2]} = W^{[2]} - 0.1\,\frac{\partial J}{\partial W^{[2]}}$$

- Perform forward propagation again using the **updated weights.**
- This procedure is repeated until the NN achieves a **sufficiently small MSE**.

Example 1: (Construction of Neural Network using Pytorch)

```
import torch
import torch.nn as nn
import torch.optim as optim
```

LIBRARIES TO IMPORT

```
class RegressionNet(nn.Module):
    def __init__(self, input_size, hidden1, hidden2, output_size=1):
        super(RegressionNet, self).__init__()

        # Fully connected layers
        self.fc1 = nn.Linear(input_size, hidden1)
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.fc3 = nn.Linear(hidden2, output_size)

        # Activation function
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.relu(self.fc1(x))
        out = self.relu(self.fc2(out))
        out = self.fc3(out)     # No activation in output for regression
        return out
```

FEED FORWARD NEURAL NETWORK

Continue…

```
# Example dimensions
input_size = 5   # e.g., 5 input features
hidden1 = 64
hidden2 = 32
output_size = 1

# Create model
model = RegressionNet(input_size, hidden1, hidden2, output_size)

# Mean Squared Error Loss for regression
criterion = nn.MSELoss()

# Optimizer – Adam (you can use SGD too)
optimizer = optim.Adam(model.parameters(), lr=0.001)

 # Random dataset (100 samples, 5 features)
 X = torch.randn(100, input_size)
 y = torch.randn(100, 1)
```

MODEL HYPER-PARAMETERS

Continue…

```python
epochs = 500

for epoch in range(epochs):
    # Forward pass
    outputs = model(X)
    loss = criterion(outputs, y)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 20 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
```

MODEL TRAINING WITH BACKWARD PROPAGATION

```python
model.eval()  # set to evaluation mode
with torch.no_grad():
    test_input = torch.randn(3, input_size)
    predicted = model(test_input)
    print("Predicted values:\n", predicted)
```

EVALUATION ON TESTING DATASET

Example 2: (Solution of Heat equation using PINN)

PDE

$$u_t = \alpha u_{xx}, \qquad (x,t) \in (0,1) \times (0,1)$$

Initial Condition

$$u(x,0) = \sin(\pi x), \qquad x \in [0,1]$$

Boundary Conditions

$$u(0,t) = 0, \qquad u(1,t) = 0, \qquad t \in [0,1]$$

LIBRARIES TO IMPORT

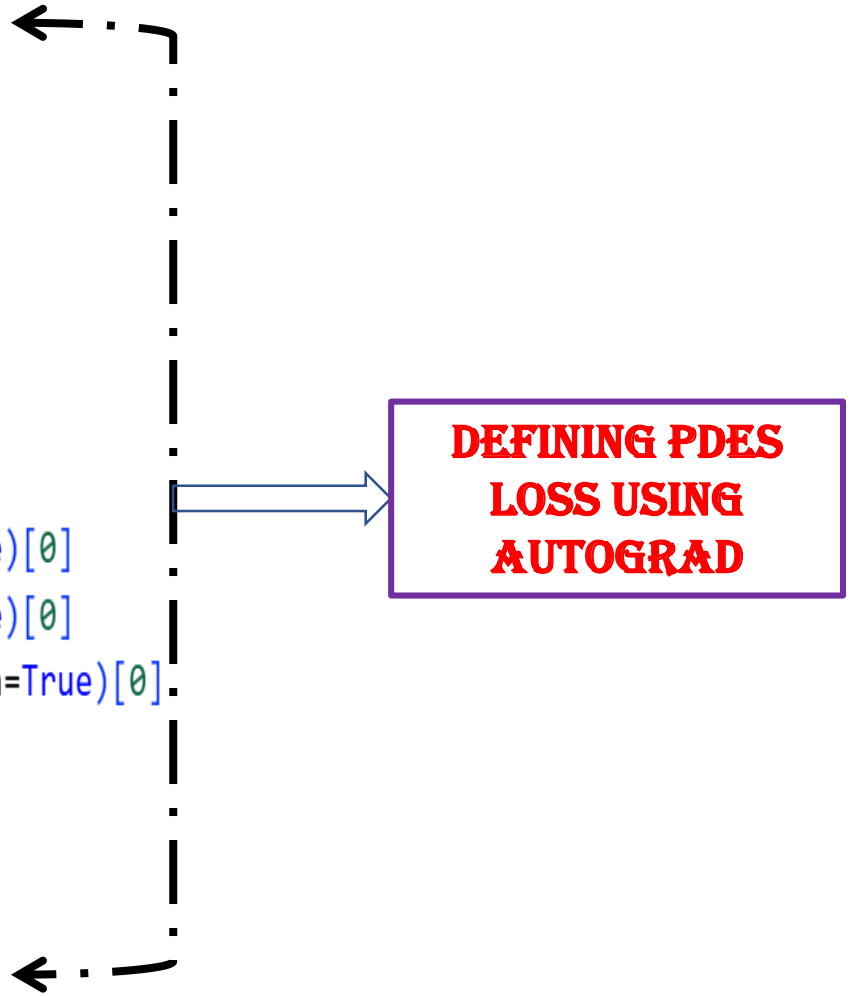
```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
# Set random seeds for reproducibility
torch.manual_seed(0)
np.random.seed(0)
```

FORWARD PROPAGATION OF NEURAL NETWORK

```
class PINN(nn.Module):
    def __init__(self, layers):
        super(PINN, self).__init__()
        self.layers = nn.ModuleList()
        for i in range(len(layers)-1):
            self.layers.append(nn.Linear(layers[i], layers[i+1]))
        self.activation = nn.Tanh()

    def forward(self, x, t):
        # Combine x and t
        X = torch.cat((x, t), dim=1)
        for i in range(len(self.layers)-1):
            X = self.activation(self.layers[i](X))
        X = self.layers[-1](X)
        return X
```

Continue…

```python
def pde_residual(model, x, t, alpha=0.01):
    # Enable gradient computation
    x.requires_grad = True
    t.requires_grad = True
    # Forward pass
    u = model(x, t)
    # Compute partial derivatives
    u_t = torch.autograd.grad(u, t, grad_outputs=torch.ones_like(u), create_graph=True)[0]
    u_x = torch.autograd.grad(u, x, grad_outputs=torch.ones_like(u), create_graph=True)[0]
    u_xx = torch.autograd.grad(u_x, x, grad_outputs=torch.ones_like(u_x), create_graph=True)[0]

    # PDE residual: u_t - alpha * u_xx = 0
    f = u_t - alpha * u_xx
    return f
```



Continue…

```
# Collocation points (for PDE)
N_f = 10000
x_f = np.random.rand(N_f, 1)
t_f = np.random.rand(N_f, 1)
# Boundary points
N_b = 200
t_b = np.linspace(0, 1, N_b).reshape(-1, 1)
x_b0 = np.zeros_like(t_b)
x_b1 = np.ones_like(t_b)
# Initial condition points
N_i = 200
x_i = np.linspace(0, 1, N_i).reshape(-1, 1)
t_i = np.zeros_like(x_i)
u_i = np.sin(np.pi * x_i)
# Convert all to tensors
to_tensor = lambda x: torch.tensor(x, dtype=torch.float32)
x_f, t_f = to_tensor(x_f), to_tensor(t_f)
x_b0, t_b, x_b1 = to_tensor(x_b0), to_tensor(t_b), to_tensor(x_b1)
x_i, t_i, u_i = to_tensor(x_i), to_tensor(t_i), to_tensor(u_i)
```

CONSTRUCTING TRAINING POINTS AND TRANSFORMING IT TO TENSORS

Continue…

```python
def loss_function(model):
    # PDE residual loss
    f = pde_residual(model, x_f, t_f)
    loss_f = torch.mean(f**2)

    # Initial condition loss
    u_pred_i = model(x_i, t_i)
    loss_i = torch.mean((u_pred_i - u_i)**2)

    # Boundary condition loss
    u_b0 = model(x_b0, t_b)
    u_b1 = model(x_b1, t_b)
    loss_b = torch.mean(u_b0**2) + torch.mean(u_b1**2)

    return loss_f + loss_i + loss_b
```

PINN LOSS INCLUDING PDE LOSS, INITIAL AND BOUNDARY LOSSES

```python
layers = [2, 50, 50, 50, 1]  # (x,t) → 50 → 50 → 50 → u
model = PINN(layers)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_history = []
```

LAYERS AND OPTIMIZERS

Continue…

MODEL TRAINING WITH BACKWARD PROPAGATION

```
epochs = 5000
for epoch in range(epochs):
    optimizer.zero_grad()
    loss = loss_function(model)
    loss.backward()
    optimizer.step()
    loss_history.append(loss.item())

    if (epoch+1) % 500 == 0:
        print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.6f}')
```

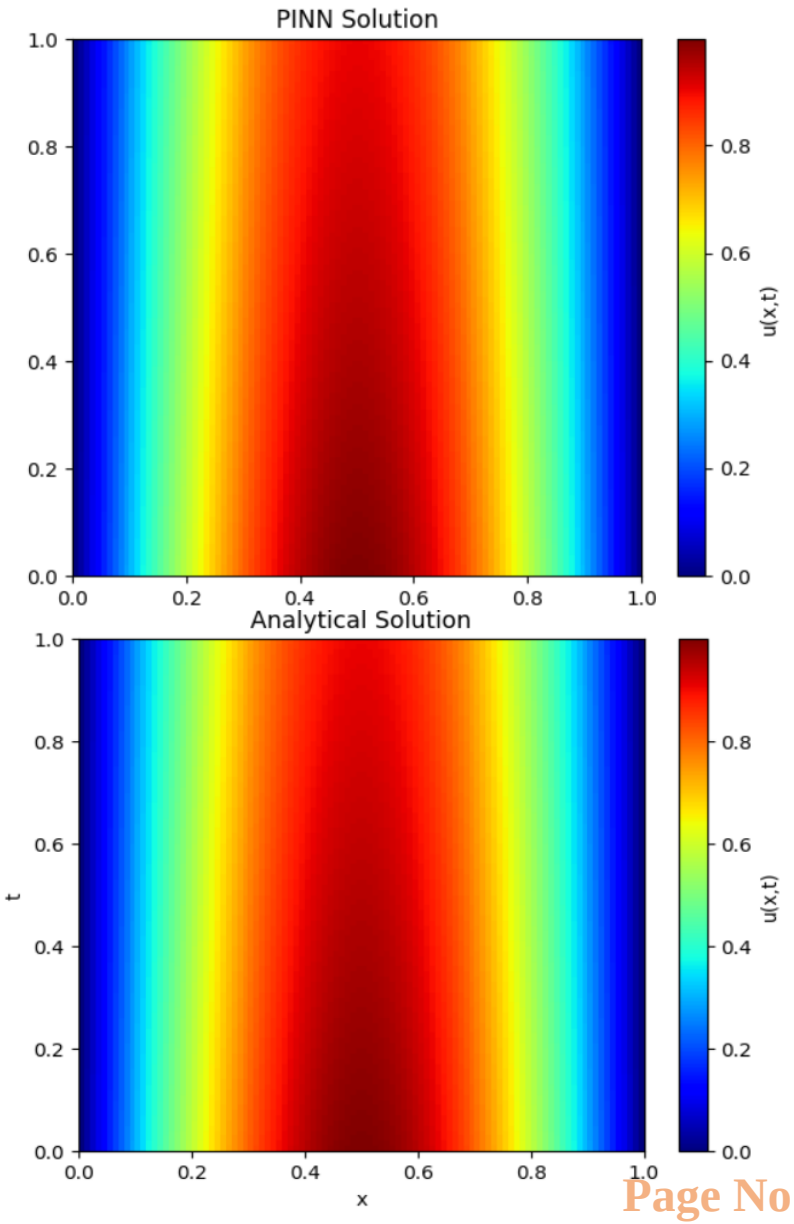
CREATING TEST DATASET AND DEFINING EXACT SOLUTION OF HEAT EQUATION

```
# Create test grid
x_test = np.linspace(0, 1, 100)
t_test = np.linspace(0, 1, 100)
X, T = np.meshgrid(x_test, t_test)
x_test_t = torch.tensor(X.flatten()[:, None], dtype=torch.float32)
t_test_t = torch.tensor(T.flatten()[:, None], dtype=torch.float32)


# Predict using trained model
u_pred = model(x_test_t, t_test_t).detach().numpy()
test_loss_history.append(loss.item())
U_pred = u_pred.reshape(100, 100)
# Exact analytical solution
alpha=0.01
U_exact = np.exp(-alpha * np.pi**2 * T) * np.sin(np.pi * X)
```

Continue…

PLOTS OF PINN PREDICTED AND EXACT SOLUTION OF HEAT EQUATION

```
# Plot results
plt.figure(figsize=(14,5))
plt.subplot(1,2,1)
plt.imshow(U_pred, extent=[0,1,0,1], origin='lower', aspect='auto', cmap='jet')
plt.colorbar(label='u(x,t)')
plt.title('PINN Solution')

plt.subplot(1,2,2)
plt.imshow(U_exact, extent=[0,1,0,1], origin='lower', aspect='auto', cmap='jet')
plt.colorbar(label='u(x,t)')
plt.title('Analytical Solution')
plt.show()
```



Example 3: (Solution of wave equation using PINN)

$$u_{tt} = c^2 u_{xx}$$

on $(x,t) \in [0,1] \times [0,1]$ with Dirichlet boundary conditions
$u(0,t) = u(1,t) = 0$, and example initial conditions
$u(x,0) = \sin(\pi x), \quad u_t(x,0) = 0.$

LIBRARIES TO IMPORT

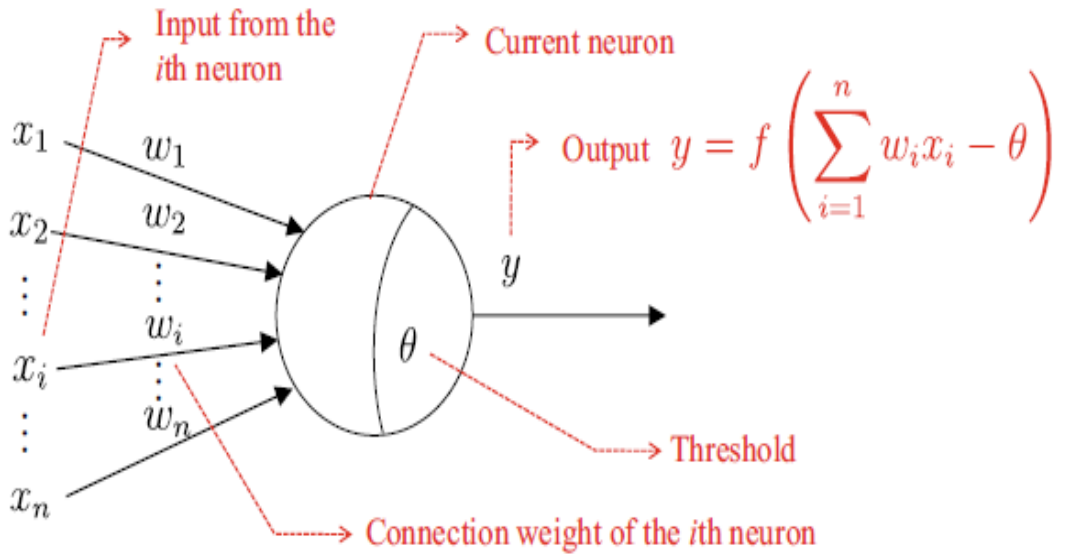
```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from math import pi
```

DEFINING CONSTANTS INCLUDING TRAINING POINTS

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Parameters in the PDE
c = 1.0              # wave speed
x_min, x_max = 0.0, 1.0
t_min, t_max = 0.0, 1.0

# Training point counts
N_f = 20000    # collocation (interior) points
N_bc = 2000    # boundary points (x=0 and x=1)
N_ic = 2000    # initial condition points (t=0)
```

Continue…

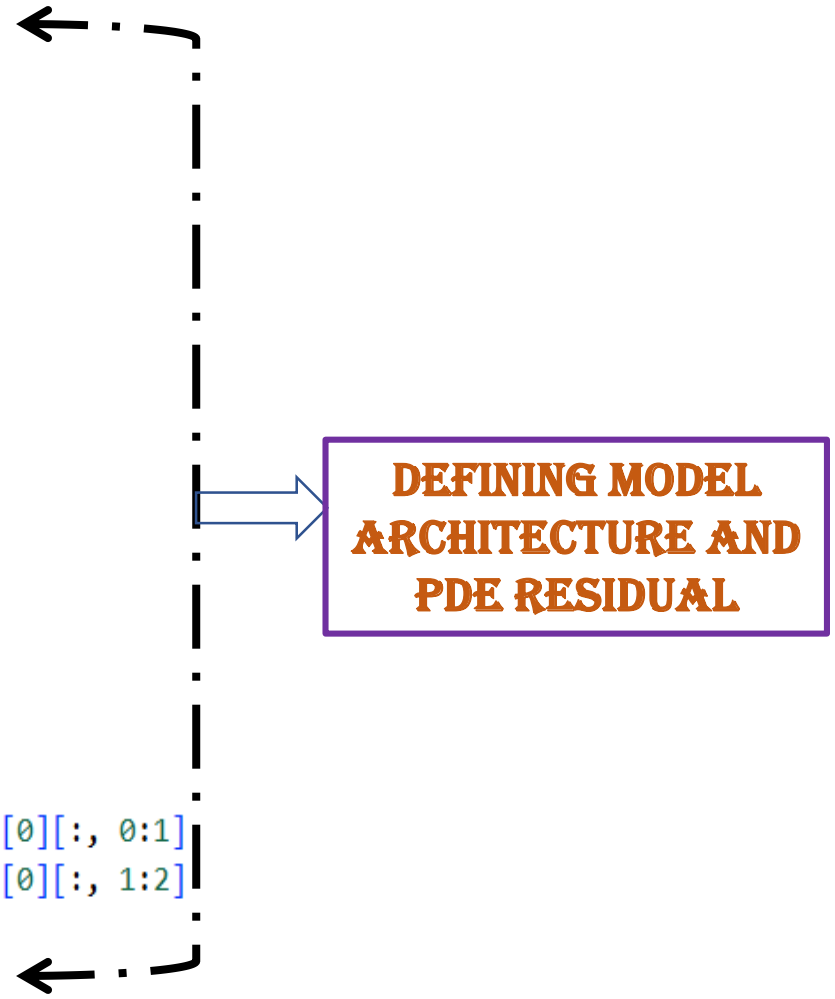
```python
# Neural Network (fully connected)
class PINN(nn.Module):
    def __init__(self, layers):
        super().__init__()
        self.layers = nn.ModuleList()
        for i in range(len(layers)-1):
            self.layers.append(nn.Linear(layers[i], layers[i+1]))
        self.activation = nn.Tanh()

        # xavier init
        for m in self.layers:
            if isinstance(m, nn.Linear):
                nn.init.xavier_normal_(m.weight)
                nn.init.zeros_(m.bias)

    def forward(self, x):
        # x: [N, 2] -> (x,t)
        y = x
        for i, layer in enumerate(self.layers[:-1]):
            y = layer(y)
            y = self.activation(y)
        y = self.layers[-1](y)
        return y
```



DEFINING FORWARD PROPAGATION WITH WEIGHTS INITIALIZATION

Continue…

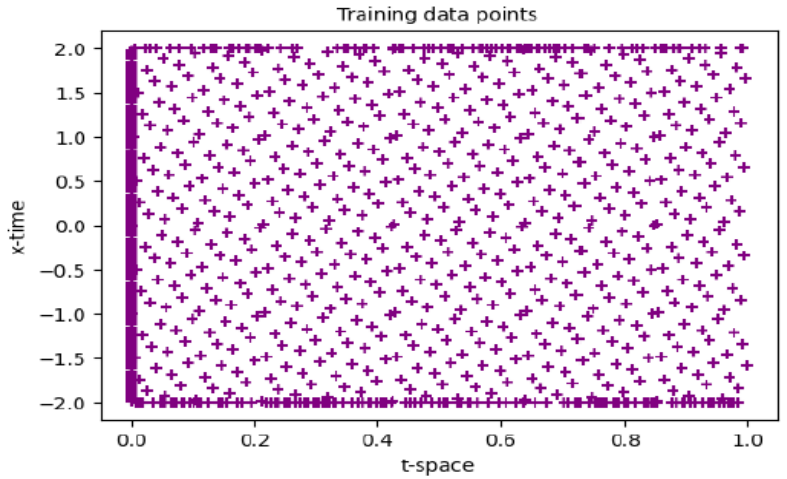
```python
# Network Configuration
layers = [2, 64, 64, 64, 64, 1]  # input: (x,t), output: u
model = PINN(layers).to(device)

# PDE residuals using autograd
def pde_residual(model, x_t):
    """
    Compute residual r = u_tt - c^2 u_xx for (x,t) points.
    x_t: tensor shape [N,2] with requires_grad=True
    """
    u = model(x_t)
    # 1st derivatives
    grads = torch.autograd.grad(u, x_t, grad_outputs=torch.ones_like(u), create_graph=True)[0]
    u_x = grads[:, 0:1]
    u_t = grads[:, 1:2]
    # 2nd derivatives
    u_xx = torch.autograd.grad(u_x, x_t, grad_outputs=torch.ones_like(u_x), create_graph=True)[0][:, 0:1]
    u_tt = torch.autograd.grad(u_t, x_t, grad_outputs=torch.ones_like(u_t), create_graph=True)[0][:, 1:2]
    residual = u_tt - (c**2) * u_xx
    return residual
```



Continue…

CONSTRUCTION OF TRAINING DATASET WITH THE HELP OF INITIAL AND BOUNDARY FUNCTIONS

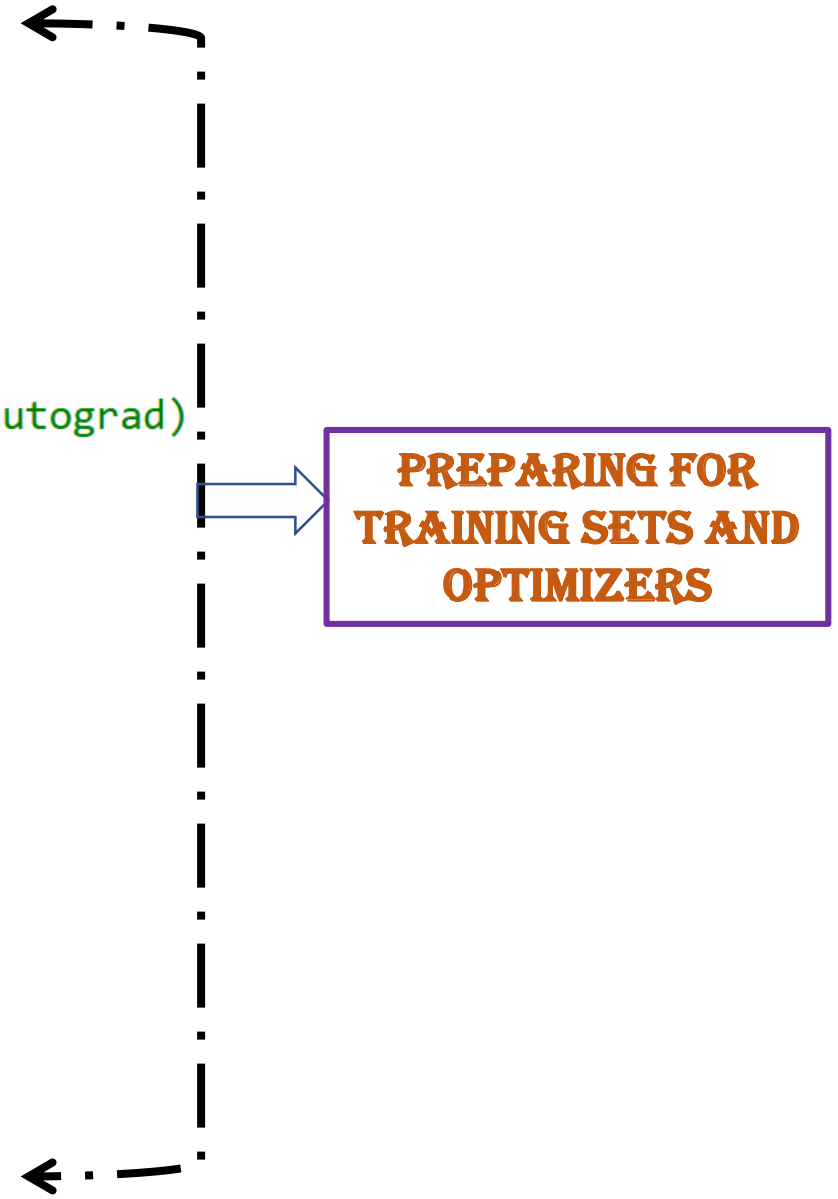
```
# Initial and boundary condition functions
def u_ic(x):
    # u(x,0) = sin(pi x)
    return torch.sin(pi * x)
def ut_ic(x):
    # u_t(x,0) = 0
    return torch.zeros_like(x)
# Training data sampling
def sample_collocation(N):
    x = torch.rand(N,1)*(x_max-x_min) + x_min
    t = torch.rand(N,1)*(t_max-t_min) + t_min
    xt = torch.cat([x,t], dim=1)
    return xt.to(device)
def sample_initial(N):
    x = torch.rand(N,1)*(x_max-x_min) + x_min
    t = torch.zeros_like(x) + t_min
    xt = torch.cat([x,t], dim=1)
    ux = u_ic(x).to(device)
    utx = ut_ic(x).to(device)
    return xt.to(device), ux.to(device), utx.to(device)
```




```
def sample_boundary(N):
    # sample t in (t_min,t_max), x at boundaries 0 and 1
    t = torch.rand(N//2,1)*(t_max-t_min) + t_min
    x0 = torch.zeros_like(t) + x_min
    x1 = torch.zeros_like(t) + x_max
    xt0 = torch.cat([x0, t], dim=1)
    xt1 = torch.cat([x1, t], dim=1)
    return xt0.to(device), xt1.to(device)
```

Continue…

```
# Prepare Training sets
torch.manual_seed(1234)
xt_f = sample_collocation(N_f)
xt_ic, u0_ic, ut0_ic = sample_initial(N_ic)
xt_b0, xt_b1 = sample_boundary(N_bc)

# set requires_grad for PDE collocation and IC (needed for autograd)
xt_f.requires_grad = True
xt_ic.requires_grad = True
xt_b0.requires_grad = True
xt_b1.requires_grad = True

# Optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
# Optionally use LBFGS later (uncomment to use)
use_lbfgs = False

# Training loop
mse_loss = nn.MSELoss()
```



Continue…

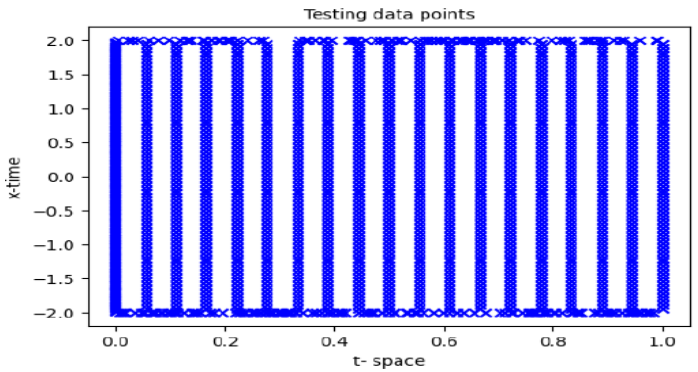
```
print_every = 1000
n_epochs = 10000
if not use_lbfgs:
    for epoch in range(1, n_epochs+1):
        model.train()
        optimizer.zero_grad()
        # PDE residual loss
        r = pde_residual(model, xt_f)
        loss_r = mse_loss(r, torch.zeros_like(r))
        # Initial condition u(x,0)
        u_pred_ic = model(xt_ic)
        loss_ic_u = mse_loss(u_pred_ic, u0_ic)
        # Initial condition u_t(x,0)
        grads_ic = torch.autograd.grad(u_pred_ic, xt_ic, grad_outputs=torch.ones_like(u_pred_ic), create_graph=True)[0]
        u_t_pred_ic = grads_ic[:,1:2]
        loss_ic_ut = mse_loss(u_t_pred_ic, ut0_ic)
        # Boundary conditions
        u_b0 = model(xt_b0)
        u_b1 = model(xt_b1)
        loss_bc = mse_loss(u_b0, torch.zeros_like(u_b0)) + mse_loss(u_b1, torch.zeros_like(u_b1))
        loss = loss_r + loss_ic_u + loss_ic_ut + loss_bc
        loss.backward()
        optimizer.step()
```

TRAINING PINN WITH LOSS MINIMIZATION TO SATISFYING THE PDE ASSOCIATED WITH INITIAL AND BOUNDARY CONDITIONS

Continue…

MODEL EVALUATION FOR TEST DATASET

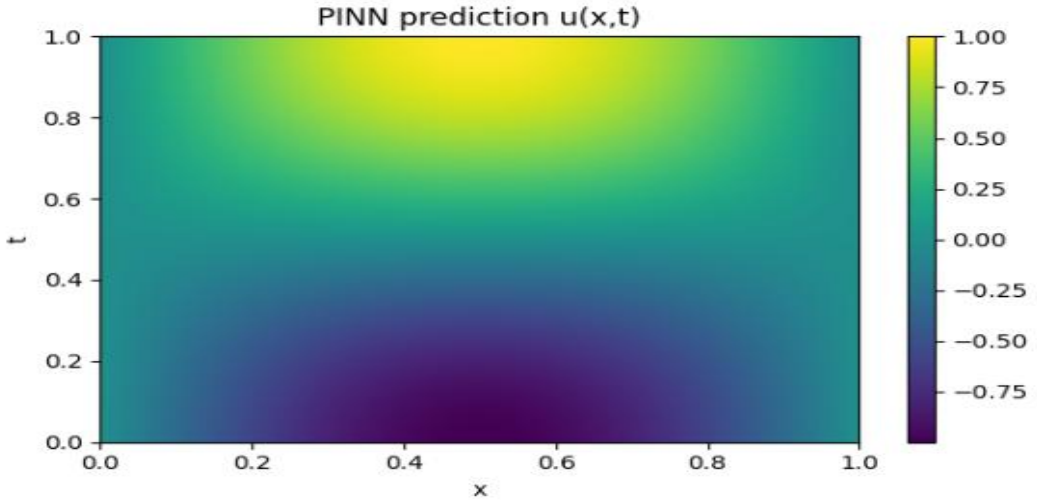
```
model.eval()
# Evaluation grid
nx = 200
nt = 200
x_vals = np.linspace(x_min, x_max, nx)
t_vals = np.linspace(t_min, t_max, nt)
X, T = np.meshgrid(x_vals, t_vals)
XT = np.hstack((X.reshape(-1,1), T.reshape(-1,1)))
XT_t = torch.tensor(XT, dtype=torch.float32).to(device)
with torch.no_grad():
    u_pred = model(XT_t).cpu().numpy().reshape(nt, nx)

# Analytic solution:
u_exact = np.cos(pi * T) * np.sin(pi * X)
```

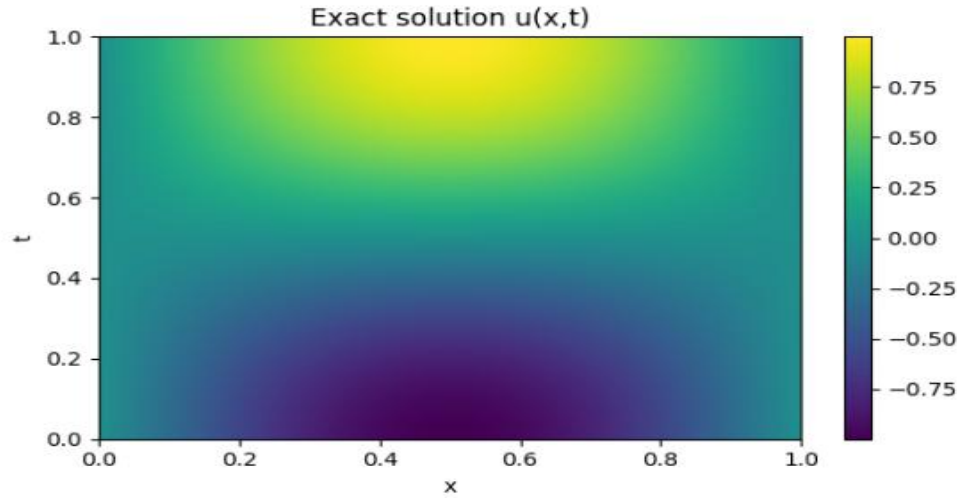


PLOTS OF PINN PREDICTED SOLUTION AND ANALYTICAL SOLUTION

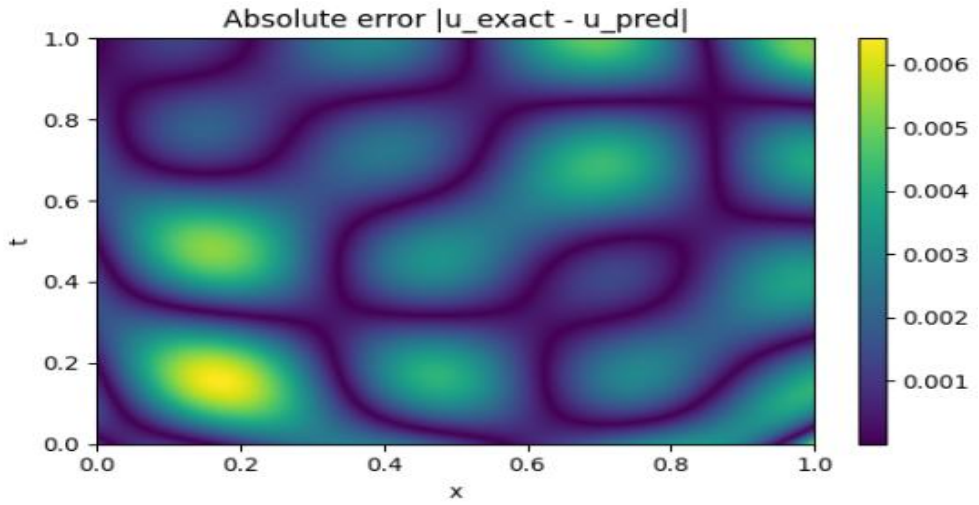
```
# plot predicted and error
fig, axes = plt.subplots(1,3, figsize=(18,4))
im0 = axes[0].imshow(u_pred, extent=[x_min,x_max,t_max,t_min], aspect='auto')
axes[0].set_title("PINN prediction u(x,t)")
axes[0].set_xlabel("x"); axes[0].set_ylabel("t")
fig.colorbar(im0, ax=axes[0])

im1 = axes[1].imshow(u_exact, extent=[x_min,x_max,t_max,t_min], aspect='auto')
axes[1].set_title("Exact solution u(x,t)")
axes[1].set_xlabel("x"); axes[1].set_ylabel("t")
fig.colorbar(im1, ax=axes[1])

im2 = axes[2].imshow(np.abs(u_exact - u_pred), extent=[x_min,x_max,t_max,t_min], aspect='auto')
axes[2].set_title("Absolute error |u_exact - u_pred|")
axes[2].set_xlabel("x"); axes[2].set_ylabel("t")
fig.colorbar(im2, ax=axes[2])

plt.tight_layout()
plt.show()
```

PLOTS OF PINN PREDICTED, ANALYTICAL SOLUTIONS AND THEIR ABSOLUTE DIFFERENCES





ABSOLUTE ERROR BETWEEN EXACT AND PREDICTED SOLUTION:



Thank You For Listening

Any Query ?

You can use the following code for providing data set manually for regression problem:

```
import pandas as pd
df = pd.read_csv('data.csv')
df.head()

from sklearn.model_selection import train_test_split

# Features and target
X = df.drop('target', axis=1)   # replace 'target' with your column name
y = df['target']

# Train-test split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```