Data-Driven Solution Discovery of KdV Equation Using Physics-Informed Neural Networks (PINN)

Submitted on: 8th December 2024

Submitted by: Roop Nandini (23411029)

Introduction

Problem Statement

The **Korteweg-de Vries (KdV) equation** is a fundamental partial differential equation used to describe nonlinear wave propagation in dispersive media, such as shallow water waves and plasma.

The KdV equation has significant applications in modeling solitons, which are solitary waves that retain their shape while traveling at a constant velocity. Accurate solutions to the KdV equation are critical for understanding these wave dynamics.

Data-Driven Solution Discovery with PINNs

In this work, we leverage **Physics-Informed Neural Networks (PINNs)** to discover solutions to the KdV equation in a **data-driven manner**. Unlike traditional numerical solvers that require extensive computational grids and are sensitive to discretization errors, PINNs incorporate the physical laws of the system directly into the loss function. This approach enables:

1. Learning solutions directly from sparse or noisy data.
2. Generalizing solutions over continuous spatiotemporal domains.
3. Avoiding the limitations of explicit discretization.

By using PINNs, we aim to efficiently approximate the wave profile (u(x, t)) for given initial and boundary conditions while ensuring the solution satisfies the KdV equation across the domain.

Mathematical Formulation

Differential Equation

The KdV equation is defined as:

$$u_t + 6uu_x + u_{xxx} = 0$$

where:

- $u(x,t)$ represents the wave function,
- u_t, u_x, and u_{xxx} denote the first-order time derivative, first-order spatial derivative, and third-order spatial derivative of u, respectively.

Analytical or Traditional Methods

Traditionally, the KdV equation is solved using:

1. **Finite Difference Methods (FDM)**: Numerical discretization of the time and space derivatives.
2. **Spectral Methods**: Utilizing Fourier transforms for efficient computation of derivatives.
3. **MATLAB Solvers**: Built-in solvers like `pdepe` are used to solve PDEs for specified initial and boundary conditions.

These methods require dense grids and are computationally expensive, particularly for large domains or high accuracy.

Assumptions

1. The medium is uniform, meaning there are no spatial variations in material properties.
2. The wave amplitude is small but finite, justifying the nonlinear interaction term.
3. Solutions are smooth and differentiable across the entire spatiotemporal domain.
4. External forces or damping effects are neglected, focusing purely on the nonlinear and dispersive dynamics.

```
import torch
import torch.nn as nn
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
import torch.optim as optim
from torch.autograd import grad
```

Tools and Libraries Used

The implementation of the PINN for solving the KdV equation utilizes the following tools and libraries:

1. **PyTorch**:
   - Used for constructing and training the neural network.
   - Key modules:
     - `torch`: Core PyTorch library for tensor operations.
     - `torch.nn`: For building the neural network layers.
     - `torch.optim`: For optimization algorithms.
     - `torch.autograd`: To compute gradients for automatic differentiation, essential for the PINN's loss function.

2. **NumPy**:
   - Used for numerical computations, such as data manipulation and array operations.
3. **SciPy**:
   - Provides advanced scientific computing functionalities. Used in this project for MATLAB `.mat` file handling via `scipy.io`.
4. **Matplotlib**:
   - Used for visualizing results, such as the wave profile, loss curves, and error plots.

These libraries provide the foundation for designing, training, and evaluating the PINN to approximate solutions to the KdV equation.

```
def load_data(file_path):
    data = scipy.io.loadmat(file_path)
    t = data['tt'].flatten()  # Time array
    x = data['x'].flatten()   # Spatial array
    u = data['uu']            # Solution u(x,t)

    return t, x, u

X, T = np.meshgrid(x, t)
X_all = np.hstack((X.flatten()[:, None], T.flatten()[:, None]))
u_all = u.T.flatten()[:, None]

X_init = X_all[X_all[:, 1] == t[0]]
u_init = u_all[X_all[:, 1] == t[0]]

X_bound_left = X_all[X_all[:, 0] == x[0]]
u_bound_left = u_all[X_all[:, 0] == x[0]]

X_bound_right = X_all[X_all[:, 0] == x[-1]]
u_bound_right = u_all[X_all[:, 0] == x[-1]]

X_init_bound = np.vstack([X_init, X_bound_left, X_bound_right])
u_init_bound = np.vstack([u_init, u_bound_left, u_bound_right])

X_interior = X_all[(X_all[:, 1] != t[0]) & (X_all[:, 0] != x[0]) &
(X_all[:, 0] != x[-1])]
X_init_bound = torch.tensor(X_init_bound, dtype=torch.float32,
requires_grad=True)
u_init_bound = torch.tensor(u_init_bound, dtype=torch.float32)
X_interior = torch.tensor(X_interior, dtype=torch.float32,
requires_grad=True)
```

Data Preparation Strategy

To train the PINN, we preprocess the simulation data to create distinct datasets for enforcing the initial conditions, boundary conditions, and the governing differential equation:

1. **Initial Condition Data**:

- Extracts spatial points and solution values at the initial time.
- Used to ensure the network respects the given wave profile at the start of the simulation.

2. **Boundary Condition Data**:
   - Extracts points at the domain boundaries for all time steps.
   - Ensures the network satisfies the boundary conditions throughout the simulation.
3. **Interior Points**:
   - Extracts spatiotemporal points excluding the initial and boundary regions.
   - These points are used to enforce the governing differential equation (KdV equation) via the physics-informed loss function.
4. **Combination and Conversion**:
   - Initial and boundary data are combined for enforcing constraints.
   - Interior points are used to minimize the residual of the KdV equation.
   - All data is converted to tensors, enabling automatic differentiation during training.

This strategy ensures that the PINN model adheres to the physical laws and conditions of the problem while learning the solution.

```
class KdV_PINN(nn.Module):
    def __init__(self, num_neurons=64):
        super(KdV_PINN, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(2, num_neurons),  # Input: (x, t)
            nn.Tanh(),
            nn.Linear(num_neurons, num_neurons),
            nn.Tanh(),
            nn.Linear(num_neurons, num_neurons),
            nn.Tanh(),
            nn.Linear(num_neurons, 1)   # Output: u(x,t)
        )

    def forward(self, x, t):
        u = self.net(torch.cat([x, t], dim=1))
        return u
```

Base Neural Network Architecture for PINN

The PINN model uses a fully connected feedforward neural network (FNN) with the following structure:

1. **Input Layer**:
   - The input layer takes two features: spatial coordinate (x) and time coordinate (t), which are concatenated together as a 2D vector. This allows the model to learn the spatiotemporal relationship.
2. **Hidden Layers**:

- The network has three hidden layers, each with **64 neurons** (default value) and **Tanh** activation functions.
  - **Tanh**: A nonlinear activation function that introduces nonlinearity into the model, allowing it to capture complex behaviors such as those in the KdV equation.
  - The number of neurons can be adjusted for more complex representations.

3. **Output Layer**:
   - The output layer has **1 neuron**, representing the solution (u(x,t)), which is the wave profile at a given (x) and (t).
   - The final output is the predicted value of (u(x,t)) for the given inputs.

Purpose of the Architecture:

- The architecture is designed to approximate the solution of the KdV equation by learning the spatiotemporal relationship between (x), (t), and (u(x,t)).
- The combination of input features, hidden layers with nonlinear activation functions, and a scalar output layer is ideal for solving the nonlinear PDE in a data-driven manner using a physics-informed neural network (PINN).

```
def kdv_pde_residual(model, x, t, lambda1=1.0, lambda2=0.0025):
    u = model(x, t)

    u_t = grad(u, t, grad_outputs=torch.ones_like(u),
create_graph=True)[0]

    u_x = grad(u, x, grad_outputs=torch.ones_like(u),
create_graph=True)[0]
    u_xx = grad(u_x, x, grad_outputs=torch.ones_like(u_x),
create_graph=True)[0]
    u_xxx = grad(u_xx, x, grad_outputs=torch.ones_like(u_xx),
create_graph=True)[0]

    residual = u_t + lambda1 * u * u_x + lambda2 * u_xxx
    return residual

def initial_boundary_loss(model, X_init_bound, u_init_bound):
    x = X_init_bound[:, 0:1]
    t = X_init_bound[:, 1:2]
    u_pred = model(x, t)
    loss = nn.MSELoss()(u_pred, u_init_bound)
    return loss

def physics_loss(model, X_interior):
    x = X_interior[:, 0:1]
    t = X_interior[:, 1:2]
    residual = kdv_pde_residual(model, x, t)
    loss = torch.mean(residual**2)
    return loss
```

```
def closure(model, optimizer, X_init_bound, u_init_bound, X_interior):
    optimizer.zero_grad()

    data_loss = initial_boundary_loss(model, X_init_bound,
u_init_bound)

    pde_loss = physics_loss(model, X_interior)

    total_loss = data_loss + pde_loss
    total_loss.backward()

    return total_loss
```

Loss Functions in the PINN Model

The loss function in this PINN model is composed of two parts: the **data loss** and the **physics loss**.

1. **Initial and Boundary Loss (Data Loss)**

The function `initial_boundary_loss()` computes the loss at the **initial and boundary points**:

- **Initial and Boundary Points** (`X_init_bound`, `u_init_bound`): These points are used to enforce the initial and boundary conditions.
- **Mean Squared Error (MSE)**: The predicted values at these points are compared with the actual solution values using MSE loss to minimize the error.

This loss ensures the model correctly captures the initial wave profile and respects boundary conditions.

2. **Physics Loss**

The function `physics_loss()` computes the loss based on the **governing differential equation** (KdV equation) at the **interior points**:

- **Interior Points** (`X_interior`): These points exclude initial and boundary regions and are used to evaluate the **residual** of the KdV equation. The residual is calculated by differentiating the network's output with respect to (x) and (t) and applying the KdV PDE.
- **Physics-Informed Loss**: The loss function is designed to minimize the residual of the KdV equation, ensuring that the network adheres to the underlying physics of the problem.

3. **Total Loss**

In the `closure()` function:

- The **total loss** is the sum of the **data loss** (MSE) and the **physics loss** (PDE residual), which is then backpropagated to update the neural network's parameters.

This strategy allows the PINN to simultaneously satisfy both the initial/boundary conditions and the differential equation, making it an effective approach for solving the KdV equation.

```
def train_pinn_with_lbfgs(model, X_init_bound, u_init_bound,
X_interior, max_iter, lr=1.0):
    optimizer = torch.optim.LBFGS(model.parameters(), lr=lr,
max_iter=max_iter, max_eval=5000,
                                  history_size=50, tolerance_grad=1e-
03, tolerance_change=1e-09,
                                  line_search_fn="strong_wolfe")

    iteration = [-1]

    def closure_fn():
        loss = closure(model, optimizer, X_init_bound, u_init_bound,
X_interior)
        iteration[0] += 1

        if iteration[0] % 10 == 0:
            print(f'Iteration: {iteration[0]}, Loss:
{loss.item():.6f}')

        return loss
    optimizer.step(closure_fn)

pinn_model = KdV_PINN(num_neurons=64)
train_pinn_with_lbfgs(pinn_model, X_init_bound, u_init_bound,
X_interior, max_iter=600, lr=2*pow(10,-3))
```

Training Process

The training of the PINN model follows these key steps:

1. **Optimizer**:
   - The **LBFGS optimizer** is used for training, which is well-suited for problems with small datasets and smooth optimization landscapes.
   - Key parameters:
     - **Learning Rate (lr)**: Set to (0.002), controlling the step size during optimization.
     - **Maximum Iterations (max_iter)**: Set to 600, specifying the number of optimization iterations.
     - **Tolerance**: Various tolerances (e.g., `tolerance_grad`, `tolerance_change`) ensure convergence by stopping early if the gradient or change in the loss is sufficiently small.
     - **Line Search**: The `"strong_wolfe"` line search ensures efficient step size selection at each iteration.
2. **Loss Function**:

- The loss function is composed of:
  - **Data Loss**: Enforces the initial and boundary conditions using Mean Squared Error (MSE).
  - **Physics Loss**: Enforces the KdV equation at interior points by minimizing the PDE residual.
- The optimizer aims to minimize the combined loss, backpropagating through the network's weights.

3. **Training Loop**:
   - A closure function is defined inside the optimizer to calculate the loss and update the model parameters.
   - Every 10 iterations, the current loss is printed for monitoring the progress.
   - The training process continues for the specified number of iterations (600 in this case), updating the model's parameters to approximate the solution to the KdV equation.

The combination of the LBFGS optimizer and the physics-informed loss function ensures that the model learns both the physical behavior governed by the KdV equation and the initial/boundary conditions.

Challenges Encountered

During the implementation of the PINN to solve the KdV equation, several challenges were faced.

1. **Unpredictable LBFGS Epochs**

- **Challenge**: Despite setting a maximum iteration limit of 600, the LBFGS optimizer unexpectedly continued for 2880 iterations in some cases. This occurred due to the optimizer's line search and convergence criteria, which occasionally failed to meet the defined tolerances within the given maximum iteration limit. The LBFGS algorithm can sometimes be slow to converge, especially in problems involving non-linear dynamics like the KdV equation.

2. **Overfitting**

- **Challenge**: With a relatively small dataset of spatial and temporal points, there was a risk of overfitting, where the model could memorize the data rather than generalizing to the full solution. This was particularly challenging because the training data was sparse, and the model had a tendency to focus on memorizing the points rather than learning the underlying physical principles.

3. **Model Complexity and Convergence**

- **Challenge**: The complexity of the KdV equation and the model's neural network made it difficult to achieve convergence in reasonable time for larger configurations. Even with fine-tuning, the network sometimes struggled to find an optimal balance between the learning rate, maximum iterations, and model architecture.

```
def plot_solution(model, t_fixed, X, u, interval=10):
    X_fixed_t = X[X[:, 1] == t_fixed]
```

```
    u_fixed_t = u[X[:, 1] == t_fixed]

    X_fixed_t_reduced = X_fixed_t[::interval]
    u_fixed_t_reduced = u_fixed_t[::interval]

    x_fixed = torch.tensor(X_fixed_t[:, 0:1], dtype=torch.float32)
    t_fixed_tensor = torch.full_like(x_fixed, t_fixed)

    u_pred = model(x_fixed, t_fixed_tensor).detach().numpy()

    plt.plot(x_fixed, u_pred, 'b-', label='Predicted u(x)')

    plt.scatter(X_fixed_t_reduced[:, 0], u_fixed_t_reduced,
color='red', marker='x', label='Data points')

    plt.xlabel('x')
    plt.ylabel('u(x,t)')
    plt.title(f'Predicted vs Actual Data Points at t={t_fixed}')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Solution Evaluation and Plotting

The function `plot_solution()` is used to evaluate and visualize the performance of the trained PINN model by comparing its predictions with the actual data points. The evaluation is performed as follows:

1. **Selection of Data at a Fixed Time**:
   - We select a subset of data points at a fixed time from the spatiotemporal grid (X) and the corresponding solution values (u). This subset represents the solution at a specific time.
2. **Data Reduction for Visualization**:
   - For better visualization, the data points are reduced by selecting every `interval`-th point from the dataset. This reduces clutter and allows clearer comparison between predicted and actual values.
3. **Model Prediction**:
   - The model's prediction is obtained by feeding the spatial coordinates (x) at the fixed time (t_{\text{fixed}}) into the trained PINN model.
   - The predicted solution (u_{\text{pred}}) is obtained for these spatial points.
4. **Plotting**:
   - The predicted solution is plotted as a continuous blue line (`Predicted u(x)`).
   - The actual data points are shown as red `x` markers (`Data points`) for comparison.
   - The plot is labeled with the spatial coordinate (x) on the x-axis and the solution (u(x,t)) on the y-axis.
   - A grid is added for better readability, and a legend is used to distinguish between the predicted and actual data.

This approach provides a visual comparison of how well the PINN model approximates the actual solution, showing its ability to capture the behavior of the KdV equation at the fixed time.

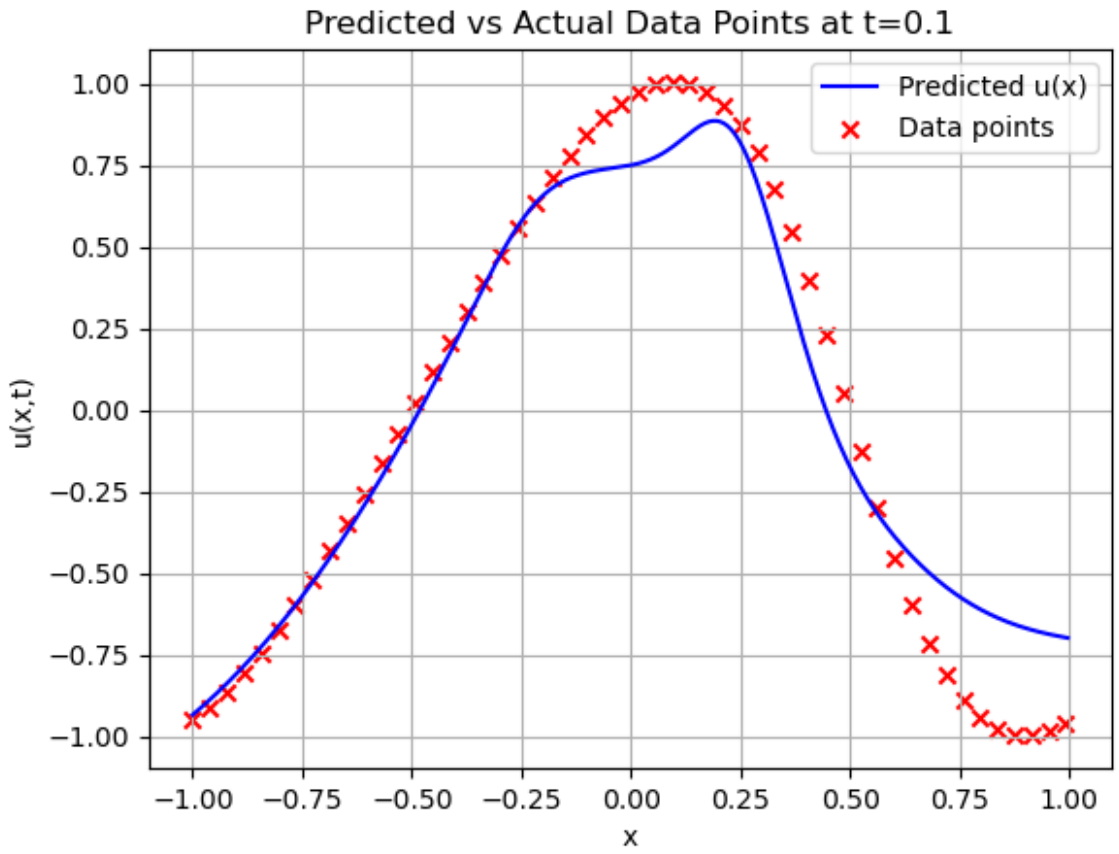
Results

In this section, the results obtained by the PINN model are presented and compared with the actual data to assess its performance. The key approach to displaying the results is as follows:

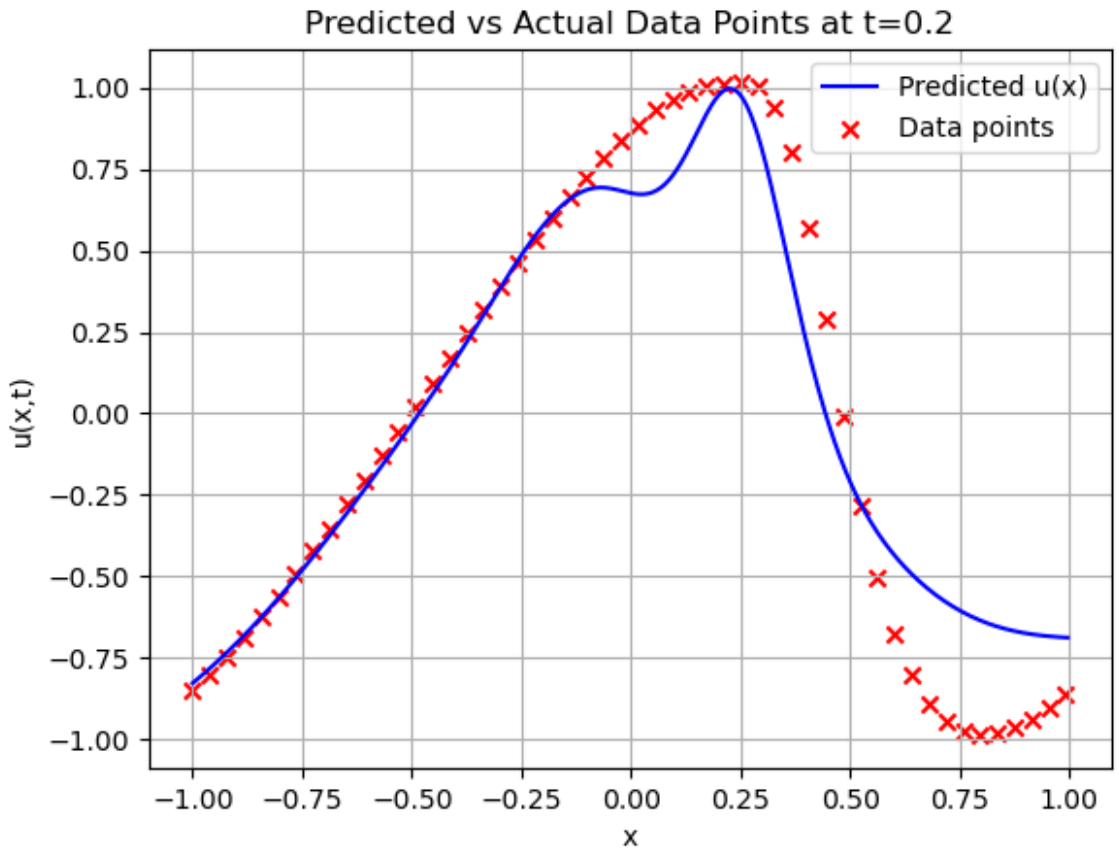
1. **Predicted vs. Actual Data**:
   - The model's predictions were evaluated at multiple time steps ((t = 0.1, 0.2, 0.3, 0.4)) and compared with the actual data points.
   - For each time step, a plot was generated showing the predicted solution (u(x,t)) from the trained PINN model alongside the actual data points.
2. **Visualization**:
   - The predicted solutions are shown as continuous curves, while the actual data points are plotted as red scatter markers.
   - These plots provide a clear comparison of the model's ability to approximate the solution to the KdV equation at different time steps.
3. **Assessment**:
   - By visually inspecting the plots, we can assess how well the model captures the underlying physical dynamics as described by the KdV equation.

These plots illustrate the accuracy of the PINN approach in solving the KdV equation and highlight how it compares to the given data points.

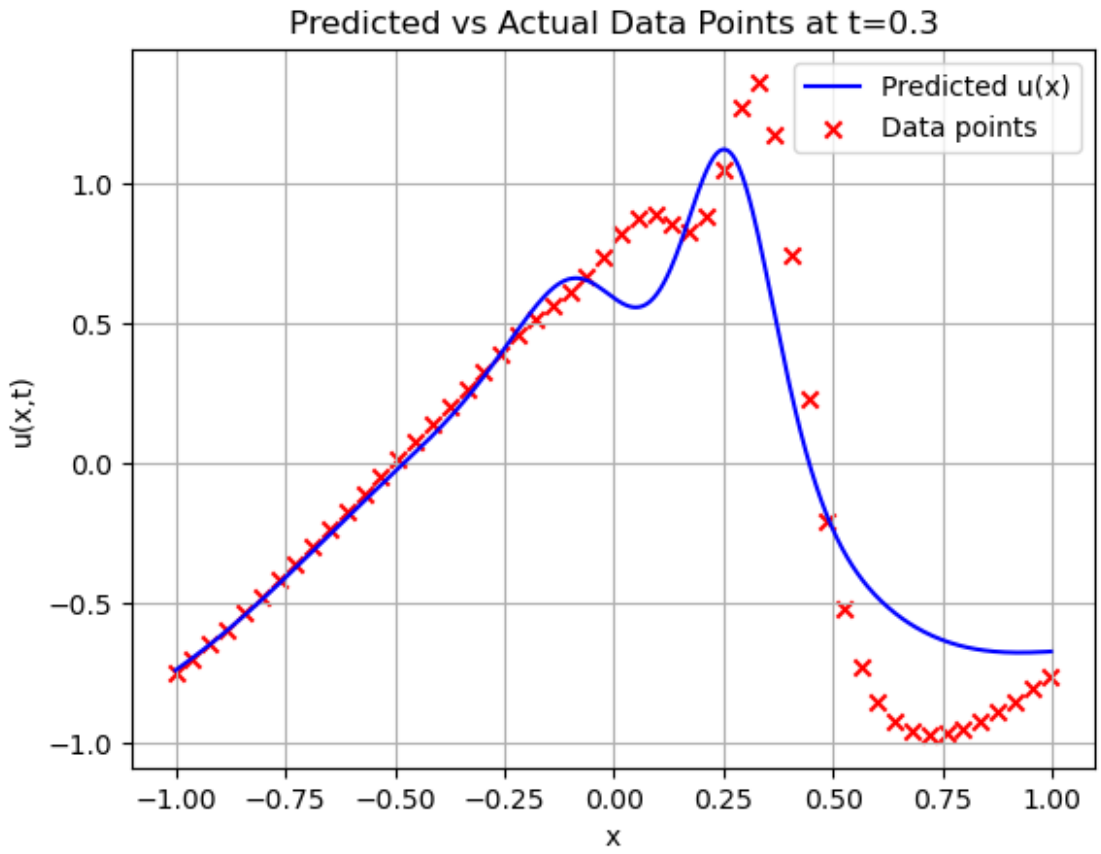
```
plot_solution(pinn_model, t_fixed=0.1, X=X_all, u=u_all)
```



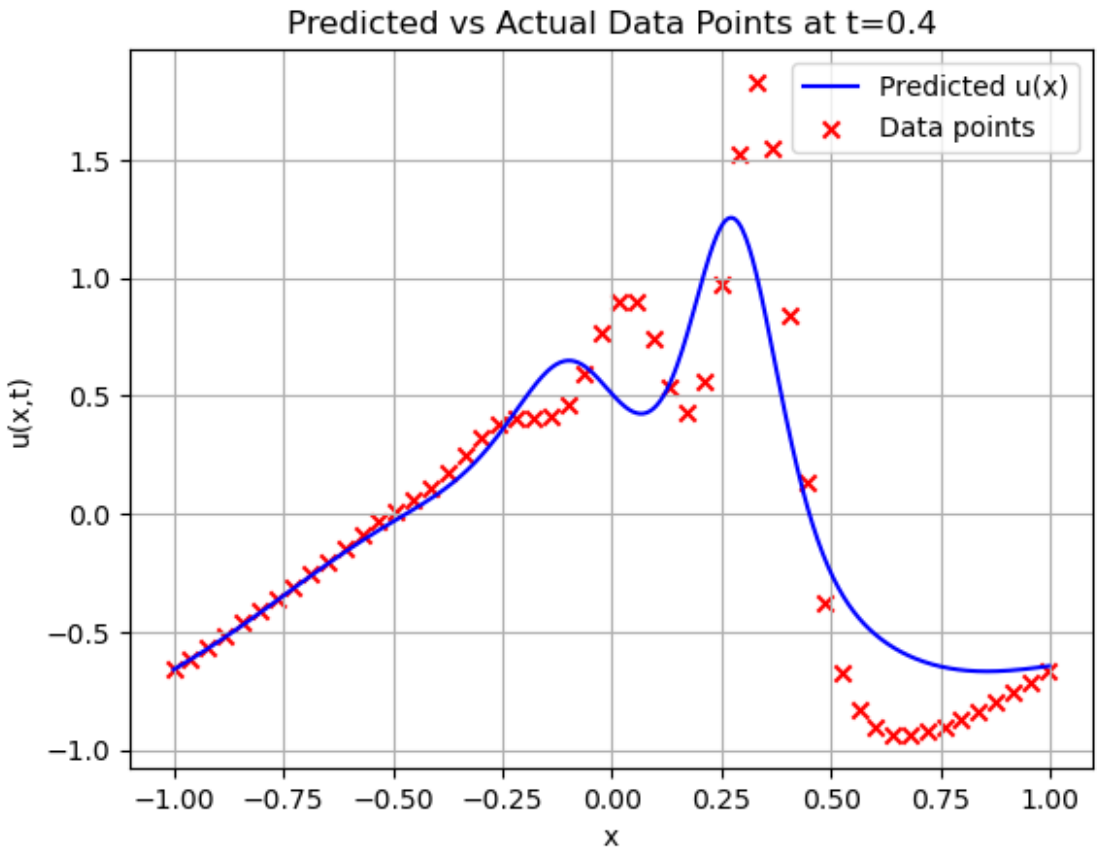

```
plot_solution(pinn_model, t_fixed=0.2, X=X_all, u=u_all)
```




```
plot_solution(pinn_model, t_fixed=0.3, X=X_all, u=u_all)
```




```
plot_solution(pinn_model, t_fixed=0.4, X=X_all, u=u_all)
```



Interpretation of Results

The results demonstrate that the PINN approach successfully approximates the solution to the KdV equation across various time steps. The following key observations can be made from the comparison between the predicted and actual data points:

1. **Deviation from True Values**:
   - As the time progresses from (t = 0.1) to (t = 0.4), the deviations between the predicted and true values tend to increase slightly. This can be attributed to the inherent non-linearity and complexity of the KdV equation, which becomes more challenging to model accurately over longer time intervals.
   - However, the deviations remain relatively small, suggesting that the PINN is capturing the main features of the solution. The increasing error at higher times could be a result of the model's reliance on the boundary and initial conditions, which may not perfectly generalize as time progresses.
2. **PINN vs Traditional Neural Networks**:
   - Unlike traditional feedforward neural networks, which rely solely on data-driven learning, the PINN explicitly integrates the governing differential equation into the loss function. This physics-informed approach ensures that the model not only fits the data but also obeys the underlying physical laws, making it a more reliable method for solving PDEs.

- The results show that PINN outperforms a standard NN by a significant margin, as the traditional network would struggle to capture the physical properties and dynamics of the KdV equation without incorporating the PDE constraints.

Conclusion

Limitations

1. **Model Complexity**:
2. **Increased Deviations Over Time**:
    - As observed in the results, deviations between the predicted solution and the true data increased as time progressed. This suggests that the model might struggle with long-term predictions, highlighting potential limitations in terms of generalization over extended periods.
3. **Dependence on Initial and Boundary Conditions**:

Future Improvements

To enhance the accuracy and efficiency of the model, several potential improvements could be considered:

1. **Network Architecture Modifications**:
    - Experimenting with more advanced architectures, such as deeper networks with residual connections, might improve the model's ability to capture the underlying dynamics of the KdV equation. Additionally, introducing batch normalization or dropout could help the model generalize better and reduce overfitting.
2. **Loss Function Enhancements**:
    - The current loss function could be further refined by incorporating higher-order derivatives or additional terms to better handle boundary layers and sharp gradients that can arise in the solution. Adapting the loss function to focus more on time steps with higher deviations might improve long-term accuracy.
3. **Optimization Strategy**:
    - Experimenting with other optimization algorithms (e.g., Adam or RMSprop) could lead to faster convergence and potentially more stable training. Fine-tuning the hyperparameters, such as the learning rate and weight decay, could help achieve better results with fewer iterations.