User Manual: `Deep Learning Models` and `Qt-Software`

Author: Fatimah Ahmadi Godini

September 2025

Contents

Part I
Qt-Based Software Interface

1 Overview

The Qt software provides a graphical user interface (GUI) for using the Deep Learning models and Physics-Informed neural Networks without writing Python code. It allows users to select datasets, configure model parameters, train models, and visualize results interactively.

2 Installation

The Qt-based software requires both the core scientific Python libraries and the PyQt framework for the graphical interface. It is recommended to install dependencies inside a virtual environment to avoid version conflicts.

Step 1: Create and Activate a Virtual Environment

```
# Windows
python -m venv venv
venv\Scripts\activate

# Linux/MacOS
python3 -m venv venv
source venv/bin/activate
```

Step 2: Install Required Packages

Install the essential dependencies using pip:

```
pip install numpy pandas scikit-learn scipy matplotlib seaborn
    PyQt5 tensorflow tensorboard
```

Alternatively, install everything from the provided `requirements.txt`:

```
pip install -r requirements.txt
```

Step 3: Launch the Interface

Once installed, run the application with:

```
python software_IntelligenceModel.py
```

This will open the graphical user interface for model configuration, training, and visualization.

3 GUI Layout

The graphical user interface (GUI) is organized into several functional tabs, each corresponding to a stage of the analysis workflow:

- **ReadData:** Browse and select input CSV datasets. The selected datasets are displayed in a table. This tab also shows the maximum pressure/stress values with their corresponding locations, and allows visualization of both individual slices and the entire pipe (including the center line).

- **Outlier:** Detect and remove outliers from the dataset. Users can select the outliers and visualize them before processing.
- **FeatureAnalysis:** Perform correlation analysis, dimensionality reduction, and normalization. Supports both *Standard* and *MaxMin* normalization methods.
- **Model:** Configure and train machine learning models. Users can select the prediction target (pressure or stress), choose the model type (MLP, LSTM, or CNN), optionally enable a Physics-Informed Neural Network (PINN) extension, and adjust hyperparameters. Models can be initialized and trained directly from this tab.
- **Test:** Evaluate trained models on unseen datasets. Users can browse and select test data, load saved normalization parameters, and select a trained model. Once testing is complete, results and evaluation metrics are displayed, and slice visualizations can be generated.
- **Regression:** Provides a polynomial-based regression interface to fit analytical equations for maximum pressure (or other variables) as functions of geometric parameters such as radius and angle.

4 Workflow

The recommended workflow for using the interface is as follows:

1. **Load Data:** In the *ReadData* tab, enter the folder name and specify the radius and angle of the dataset you want to view. Click the *LoadData* button, then select all input datasets from the file dialog and press *Open*. The loading progress is displayed in the terminal, and once completed successfully, the GUI shows the message *"Open all files Successful"*. The table displays the values of the selected dataset (based on the specified radius and angle), and the GUI also shows the maximum pressure/stress values along with their locations.
2. **Outlier Handling:** In the *Outlier* tab, review the data, identify outliers, and remove them if necessary. Selected outliers can be visualized before processing.
3. **Feature Normalization:** If normalized input data are required, open the *FeatureAnalysis* tab. Choose between *Standard* or *MaxMin* normalization, then press the *Normalization* button. After normalization is completed, the button is disabled to prevent duplicate operations.
4. **Model Training:** In the *Model* tab:
   - Select the target variable: *Pressure* or *Stress*.
   - Choose the model type: *MLP*, *LSTM*, or *CNN*.
   - Optionally enable the *PINN* checkbox to add physics constraints.

   Press *Initialize* to configure the model, then set the hyperparameters. Finally, press *Compute* to start training.

- If the dataset has not been normalized, the software prompts: *"Are you sure you want to continue without normalization?"* Select *Yes* to continue without normalization, or *No* to return to the *FeatureAnalysis* tab.
- During training, progress and metrics are displayed in the terminal. When training finishes, the GUI shows the loss curve, and the terminal prints evaluation metrics along with maximum predicted values and their coordinates.

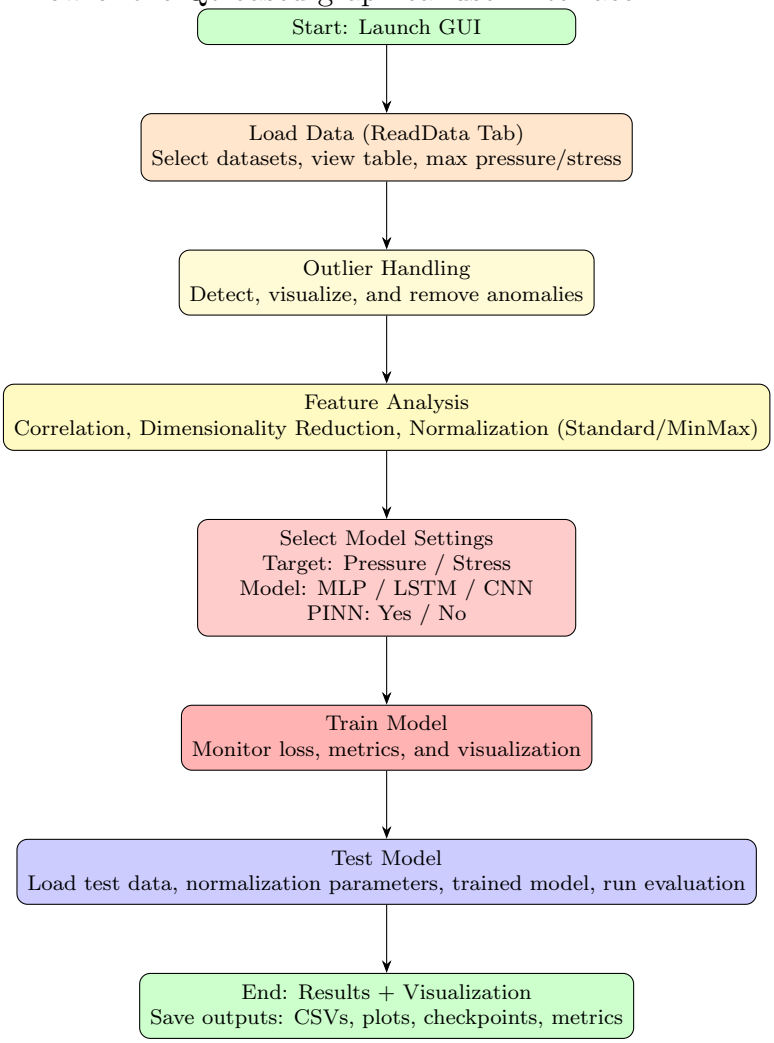
5. **Model Testing:** In the *Test* tab:

   - Press *LoadTestData* to browse and select test datasets.
   - Press *LoadPara* to import the normalization parameters used during training.
   - Press *SelectModel* to load the trained model.

   Once these are loaded, the *Test* button becomes active. Press it to run testing. Results (metrics, maximum predicted values, and locations) are printed in the terminal. Additionally, users can draw slices by setting the parameters and pressing *DrawSlice*.

5 Qt Workflow Flowchart

Workflow of the Qt-based graphical user interface.



Part II
Python Module: `models_builder.py`

1 Overview

The Python module `models_builder.py` provides a set of functions for building, training, and evaluating deep learning models for predicting wall pressure, normal stress, and shear stress. It supports standard deep neural networks including MLP, LSTM, and CNN architectures, as well as their physics-informed counterparts (PINNs) for integrating known physical constraints into the learning process.

It also includes utilities for:

- Data normalization/denormalization.
- Time series dataset generation
- Visualization (3D scatter, loss curves).
- Gradient computation with VTK.
- Automatic training/evaluation pipelines.

2 Installation

To use this module, ensure you have Python 3.x installed along with the following packages:

```
numpy
pandas
scikit-learn
tensorflow
vtk
joblib
matplotlib
```

You can install missing packages using pip:

```
pip install packages_name
```

Note: This module can also be executed inside the virtual environment of the Qt-based software, which eliminates the need for a separate installation of these dependencies.

3 Functions

Below is a description of the key functions in the module.

3.1 Data Preprocessing Functions

- `normalize_data` Normalizes selected features in a DataFrame using either *Min-Max* scaling or *standardization*. It appends the normalized values as new columns with the suffix *_norm*, leaving the original columns intact. If a pre-fitted scaler is provided, the function applies it to transform the data. Otherwise, it fits a new scaler on the input dataset. The fitted scaler can optionally be saved to disk using *joblib*, and the normalized DataFrame can be exported as a CSV file if a *normalized_path* is specified. The function returns both the normalized DataFrame and the scaler object for reuse in consistent transformations.

- `denormalize_data` Reverts normalized predictions or targets back to their original physical scale using the fitted scaler object. The function reconstructs the full feature space expected by the scaler by inserting the provided normalized values into the correct feature positions (depending on whether pressure or wall shear stress is being processed). It then applies the scaler's inverse transformation to obtain values in the original units. For pressure (*style=0*), only the pressure column is denormalized and returned. For wall shear stress (*style=1*), the corresponding six stress components are denormalized together and returned as a matrix. This ensures that predictions can be compared directly with the ground-truth physical quantities.

- `sequences_from_indices` Constructs a TensorFlow dataset of subsequences extracted from an input array using precomputed index windows. The function first slices the dataset between *start_index* and *end_index* to restrict the range of interest. It then pairs this restricted dataset with a dataset of index sequences (*indices_ds*), where each index sequence corresponds to the positions of timesteps that form one window. By applying *tf.gather*, the function collects the elements at the specified indices for each window, effectively reconstructing subsequences of fixed length from the original array. This utility is primarily used inside *timeseries_dataset_from_array* to efficiently generate overlapping or strided windows of series data that can be fed into sequence models (e.g., LSTMs or CNNs). The resulting dataset can be batched, shuffled, and prefetched for efficient training workflows.

- `timeseries_dataset_from_array` Converts raw sequential data into a TensorFlow *Dataset* suitable for training recurrent or convolutional sequence models (e.g., LSTM, CNN). The function slices the input array into overlapping or strided subsequences of fixed length (*sequence_length*), where each subsequence is defined by the stride (*sequence_stride*) and sampling interval (*sampling_rate*). These subsequences form the input samples, while the corresponding target values (*targets*) are aligned either with all timesteps in the sequence (*return_all_targets=True*) or only with the final timestep (*return_all_targets$\bar{F}$alse*). Internally, the function relies on *sequences_from_indices* to map start positions into index windows and efficiently gather the required timesteps. It also validates the stride and sampling parameters to prevent invalid configurations. Additional options include dataset shuffling (with reproducible *seed*), batching (*batch_size*), and prefetching for performance optimization. The resulting dataset yields tuples of the form *(input_sequence, target)* if targets are provided, or only *input_sequence* if not. This makes it a flexible utility for transforming non-time series data into supervised learning sequences directly compatible with TensorFlow training workflows.

3.2 Utility Functions

- `get_optimizer` Returns a Keras optimizer object based on the provided string identifier. Supported optimizers include *adam*, *adadelta*, *adagrad*, *rmsprop*, and *sgd*. The function also accepts a learning rate (*lr*) parameter, which is passed to the optimizer constructor. It simplifies experiment management where hyperparameter tuning involves trying different optimization algorithms.

- `get_loss` Computes a loss value given the ground-truth targets (*y_true*) and predicted outputs (*y_pre*) according to the chosen loss function name. Supported loss functions include:

  - Mean Squared Error (*mse*)
  - Mean Absolute Error (*mae*)
  - Mean Absolute Percentage Error (*mape*)
  - Mean Squared Logarithmic Error (*msle*)
  - Binary Crossentropy (*binary_crossentropy* or *bce*)

- `draw_LossFig` Plots and saves line charts of training and validation loss curves across epochs, with an optional third curve for physics-informed (PINN) loss. The plot is formatted with clear labels, Times New Roman font, and a high-resolution style suitable for publications. The resulting figure is saved to the directory `./res/result` with the filename pattern `[name]_lossfig.png`. It is called at the end of every training loop in the MLP, LSTM, CNN, and PINN functions.

- `compute_max_ind` Identifies and compares the maximum values of target variables (pressure or wall shear stresses) between ground-truth and model predictions. For each variable, the function finds the row in the dataset where the value is maximized, extracts its 3D spatial coordinates, and logs both the actual and predicted maxima. It also calculates the absolute difference between the two maxima and records the comparison along with their respective locations. Results are printed to the console and saved as CSV files (`[name]_max_results.csv`) inside `./res/result`.

3.3 Visualization

- `draw_3d_dataframe` Generates a 3D scatter plot from a given *pandas.DataFrame*. The user specifies which columns represent the x, y, and z coordinates (default: *Points_0*, *Points_1*, *Points_2*). An optional *color_col* parameter allows mapping point colors to a variable such as pressure or stress, providing physical meaning to the visualization. A colorbar is automatically added if *color_col* is used. This function is useful for inspecting spatial distributions of pressure or stresses across the fluid domain.

- `draw_3d_points` Plots a 3D point cloud from a NumPy array containing coordinates in the format *(n_samples, 3)*. The points are displayed as a scatter plot with user-defined color and transparency. This visualization is especially suited for inspecting raw spatial geometry, grid points, or particle distributions before and after preprocessing.

- `draw_input_dataset` Visualizes 3D points extracted from a batch of time-series inputs, assuming the first three features of each sequence correspond to spatial coordinates (x, y, z). From each sample, the point at the *last timestep* is selected, resulting in one spatial coordinate per sequence. This function is primarily used to verify the spatial distribution of batched data fed into LSTM or CNN models during training.

3.4 Physics-Informed Loss

- `compute_gradient_vtk` Computes the spatial gradients of a scalar field defined on a set of 3D points using the Visualization Toolkit (VTK). The function accepts arrays of spatial coordinates (x, y, z) and a scalar variable (e.g., pressure) defined at those coordinates. Internally, the points are inserted into a VTK *UnstructuredGrid*, where each point is treated as a vertex cell. The scalar variable is added as point data, and VTK's *GradientFilter* is applied to compute the derivatives with respect to x, y, and z. The result is three NumPy arrays: $\partial variable/\partial x$, $\partial variable/\partial y$, and $\partial variable/\partial z$. This function is critical for Physics-Informed Neural Networks (PINNs), where the residuals of governing equations (e.g., Navier-Stokes or stress tensor formulations) require accurate computation of spatial derivatives of predicted fields.

- `Navier_Stokes_Residual` Implements the residual form of the incompressible Navier-Stokes equations for pressure-driven flows. The function accepts the predicted pressure field along with velocity gradient information (first and second derivatives of velocity components). Using *compute_gradient_vtk*, it calculates spatial derivatives of pressure, which are then compared against viscous diffusion terms $(\mu\nabla^2 u, \mu\nabla^2 v, \mu\nabla^2 w)$. In addition, the continuity constraint (mass conservation) is enforced. The total loss returned is a weighted sum of Momentum residuals in x, y, z directions and Continuity residual.

- `Stress_Tensor_Residual` Computes the physics-informed residuals for the stress tensor in incompressible flows. The function accepts:

  - *spatial_gradu_p_data*: Array containing velocity gradients, second derivatives, and optionally pressure.
  - *predicted_stress*: Neural network predictions of stress components.
  - *spatial_data* (optional): 3D coordinates of points for gradient computation.
  - *mu*: Dynamic viscosity of the fluid (default 0.001).

  The function extracts velocity gradients and second derivatives from the input data, computes spatial derivatives of pressure and each predicted stress component using *compute_gradient_vtk*. Then evaluates the Navier-Stokes, normal stress, and shear stress residuals. Finally, it converts all residuals to TensorFlow tensors and computes a weighted sum of their mean-squared errors.

3.5 Neural Network Models

- `My_PINN_MLP` Implements a fully connected feedforward neural network (Multilayer Perceptron) with optional Physics-Informed Neural Network (PINN) capabilities.

The network accepts input features including spatial coordinates and optionally velocity gradients, and predicts either scalar fields (pressure) or tensor fields (stress components). Key features include:

  - Configurable hidden layers and neurons per layer.
  - Dropout regularization to prevent overfitting.
  - Flexible activation functions and output layer size (1 for pressure, 6 for stress components).
  - Physics-informed training: if *pinn=True*, the loss includes residuals from *Navier_Stokes_Resid* or *Stress_Tensor_Residual*.
  - Model evaluation on validation and test datasets, with metrics including MSE, MAE, and RMSE.
  - Early stopping based on validation loss, automatic saving of best-performing models and loss history plotting.

- `My_PINN_LSTM` Implements a Long Short-Term Memory (LSTM) network suitable for sequential or temporal data, with optional PINN functionality. The network is designed to handle input sequences of spatial and velocity gradient data, producing predictions for pressure or stress components. Key features include:

  - Stacked LSTM layers with configurable units and dropout.
  - Return sequences for intermediate layers to enable multi-layer temporal learning.
  - Optional physics-informed loss using *Navier_Stokes_Residual* or *Stress_Tensor_Residual*.
  - Batch-wise training with gradient accumulation using TensorFlow *GradientTape*.
  - Early stopping based on validation loss, automatic model saving, and loss history recording.
  - Test evaluation with detailed per-component metrics and prediction outputs.

- `My_PINN_CNN` Implements a one-dimensional Convolutional Neural Network (CNN) with optional PINN capabilities, suitable for spatially structured input data. The network extracts local patterns from input features and predicts scalar (pressure) or tensor (stress) fields. Key features include:

  - Configurable convolutional blocks with filters, kernel size, stride, and pooling layers.
  - Dropout regularization applied after each convolutional block and fully connected layers.
  - Flattening followed by dense layers for final prediction output.
  - Physics-informed training using *Navier_Stokes_Residual* or *Stress_Tensor_Residual* when *pinn=True*.
  - Model training with early stopping, batch-wise loss computation, and history tracking.
  - Comprehensive evaluation on test data with output predictions and component-wise error metrics.

3.6 Training Pipeline

- `Compute` Orchestrates the entire training workflow for neural network models (MLP, LSTM, CNN) with optional Physics-Informed Neural Network (PINN) integration. The function performs the following steps:

  1. **Data Splitting:**
     - Automatically divides input CSV data into training, validation, and test sets (default ratio 80:12:8).
     - Supports shuffling and loading of pre-split datasets.
     - Handles both directories and ZIP files containing CSVs.
     - Saves split file names for reproducibility.
  2. **Data Loading:**
     - Reads CSV files into Pandas DataFrames.
     - Concatenates multiple files into a single dataset for each split.
     - Supports optional feature normalization or scaling using user-defined methods.
  3. **Feature and Target Preparation:**
     - Selects input features, optionally including velocity gradients and higher-order derivatives for PINN training.
     - Prepares target variables: scalar pressure or tensorial stress components, optionally computing normal stresses for physics-informed models.
     - Converts all datasets into NumPy arrays suitable for TensorFlow.
  4. **Model Selection and Training:**
     - Supports three model types:
       (a) `MLP` - fully connected feedforward network.
       (b) `LSTM` - sequential model for time- or sequence-based data.
       (c) `CNN` - convolutional model for spatially structured inputs.
     - Automatically prepares datasets according to model type (e.g., time series sequences for LSTM/CNN).
     - Trains the selected model with configurable hyperparameters including units/filters, kernel size, dropout, activation, optimizer, learning rate, and batch size.
     - Supports optional PINN loss integration with physics weight parameter to enforce Navier-Stokes or stress tensor residuals.
  5. **Output and Evaluation:**
     - Monitors training and validation performance across epochs.
     - Evaluates the trained model on the test set.
     - Provides logging and debugging information, including dataset shapes, batch numbers, and exceptions.

  The *Compute* function provides a complete and flexible pipeline for training physics-informed and conventional neural networks, streamlining dataset preparation, model building, and evaluation.

4 How to Use

This section demonstrates a simple workflow for using the module.

4.1 Set Compute Function Parameters

- **datapath**: Path to the folder or *.zip* file containing input CSV files (default: *./wall_csv/*). Each file should include spatial coordinates (*Points_0*, *Points_1*, *Points_2*, *Points_Magnitude*) and target variables:
  - Pressure (p)
  - Stress (*NormalStress_i*, *wallShearStress_i*)

  For PINN:
  - Pressure prediction requires first and second gradients of U
  - Stress prediction requires first and second gradients of U and p
- **units**: List defining the number of neurons per layer for MLP/LSTM, or feature maps for CNN.
- **epochs**: Number of training epochs.
- **style**: *0* for pressure prediction, *1* for shear stress prediction.
- **pinn**: Boolean; *True* to include physics-informed loss, *False* for standard deep learning models.
- **model_ind**: Model type: *0* $\rightarrow$ MLP, *1* $\rightarrow$ LSTM, *2* $\rightarrow$ CNN.
- **kernel_size**, **pool_size**, **strides**: Applicable only for CNN models.
- **batch_size**: Number of samples per training batch.
- **method**: Normalization method (*'minmax'*, *'standard'*, or *None*).
- **shuffledata**: Boolean; *True* to shuffle data before training.
- **splitdata**: Boolean; *True* to automatically split data into training/validation sets.
- **dropout**: Dropout rate for regularization.
- **optimizer**: Optimizer for training (*'adam'*, *'sgd'*, etc.).
- **loss**: Loss function (*'mse'*, etc.).
- **lr**: Learning rate.
- **mu**: Coefficient for weight decay (optional).
- **physics_weight**: Weight of physics-informed loss in PINN training.
- **activation**: Activation function (mainly for MLP and LSTM; e.g., *'tanh'*).

4.2 Example Usages

4.2.1 Pressure Prediction – LSTM

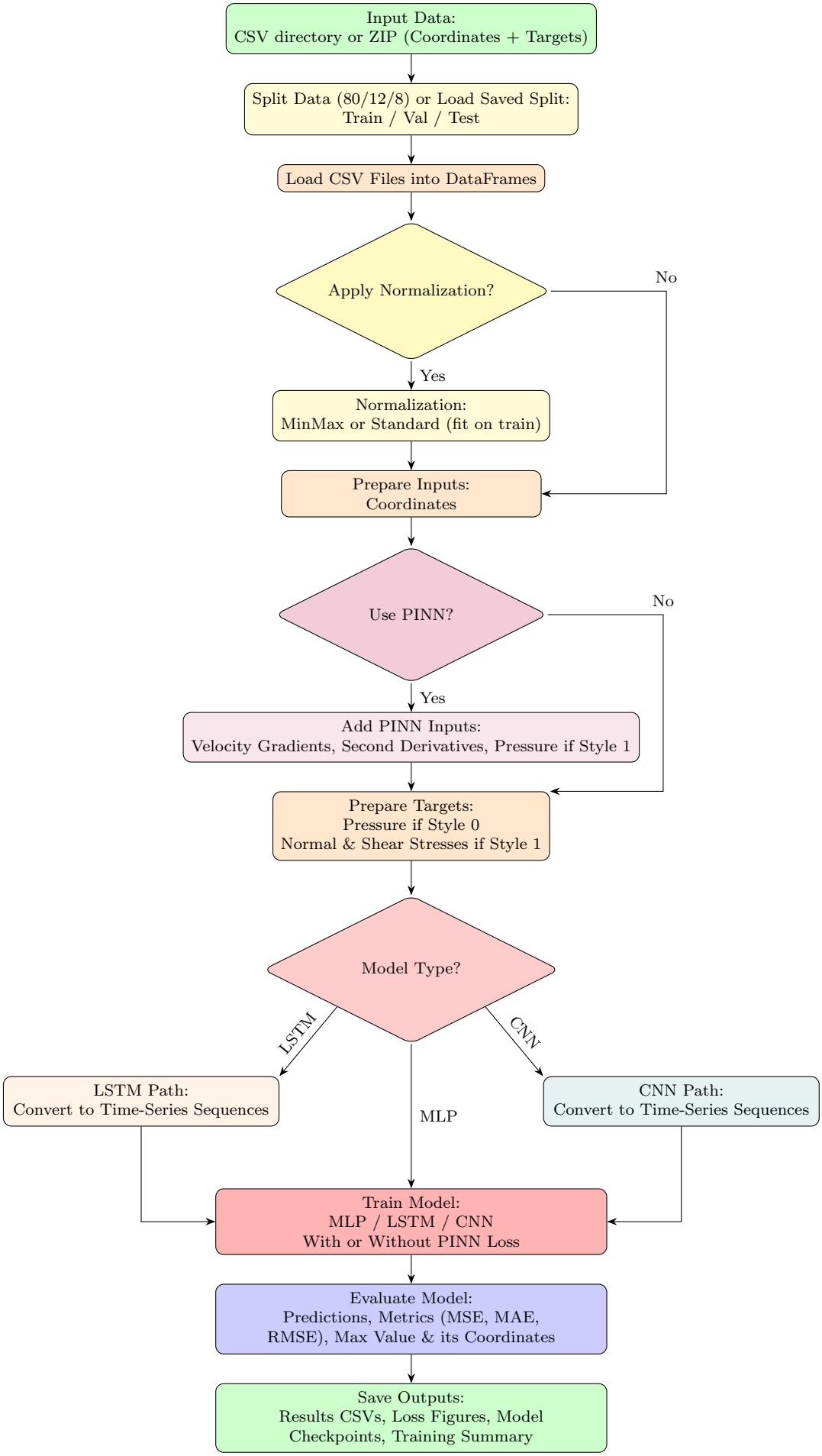
```
Compute(datapath="./wall_csv/",
    units=[32, 64], epochs=200,
    style=0, pinn=False, model_ind=1,
    activation='tanh', batch_size=256,
    method=None, shuffledata=True, splitdata=False,
    dropout=0.3, optimizer='adam', loss='mse', lr=0.001)
```

4.2.2 Shear Stress Prediction – PINN-CNN

```
Compute(datapath="./wall_csv/",
    units=[32, 64, 32], epochs=200,
    style=1, pinn=True, model_ind=2,
    kernel_size=3, pool_size=2, strides=1,
    batch_size=512,
    method=None, shuffledata=True, splitdata=False,
    dropout=0.3, optimizer='adam', loss='mse', lr=0.005,
    mu=0.001, physics_weight=1)
```

5 Module Flowchart

This module follows the workflow below.


Input Data:
CSV directory or ZIP (Coordinates + Targets)
Split Data (80/12/8) or Load Saved Split:
Train / Val / Test
Load CSV Files into DataFrames
Apply Normalization?
No
Yes
Normalization:
MinMax or Standard (fit on train)
Prepare Inputs:
Coordinates
Use PINN?
No
Yes
Add PINN Inputs:
Velocity Gradients, Second Derivatives, Pressure if Style 1
Prepare Targets:
Pressure if Style 0
Normal & Shear Stresses if Style 1
Model Type?
LSTM
MLP
CNN
LSTM Path:
Convert to Time-Series Sequences
CNN Path:
Convert to Time-Series Sequences
Train Model:
MLP / LSTM / CNN
With or Without PINN Loss
Evaluate Model:
Predictions, Metrics (MSE, MAE,
RMSE), Max Value & its Coordinates
Save Outputs:
Results CSVs, Loss Figures, Model
Checkpoints, Training Summary