STP 598 Project: Machine Learning techniques for opposition control based drag predictions

Anushka Subedi

ASU ID: 1225812200

Contents

1 Introduction

Drag reduction is essential in many engineering applications due to the energy losses associated with wall friction. Opposition control, where wall-normal velocity at a certain off-wall distance is used to generate a counteracting wall response, has proven effective in reducing drag in turbulent flows. However, direct implementation is often impractical due to off-wall sensing limitations in experiments.

In this project, we use machine learning to predict wall shear stress (as a measure of drag) from off-wall velocity measurements (taken from simulation), drawing inspiration from the opposition control strategy. This lays the groundwork for future drag and wall pressure prediction, which is particularly valuable in high-fidelity CFD solvers like Nek5000 that are highly sensitive to wall pressure boundary conditions. Moreover, since pressure sensors are often expensive and difficult to deploy in experiments, a robust ML model could benefit both simulation and experimental workflows by serving as a virtual pressure sensor.

At this stage, the focus is on evaluating the prediction capability of various learning methods. We assess model performance using RMSE values and scatter plots comparing predicted vs. true shear stress values (y-pred vs. y-test). While active control or real-time deployment is not implemented in this phase, the predictive models will serve as a foundation for future work in pressure estimation and passive control integration.

2 Data

The dataset is extracted from a turbulent channel flow simulation with opposition control applied at a friction Reynolds number of $Re_\tau \approx 180$. Vertical velocity values (u_y) were sampled from an off-wall location of $y^+ = 15$, which has been shown to be near-optimal for control effectiveness. For each sampling instance, 16, 384 spatial grid points were recorded, corresponding to the full set of off-wall velocity inputs. These were collected every 500 time steps, up to a total simulation time of $t = 400$, yielding 137 total samples.

Each input array (X) consists of 16, 384 velocity values, while each target (Y) is the corresponding wall-averaged wall shear stress. This results in a one-to-one mapping of high-dimensional off-wall velocity snapshots to scalar wall shear stress values, forming the basis of our regression problem. The same dataset is used in our associated publication (Subedi and Peet, AIAA Aviation 2024), though only a subset is used here due to storage and feasibility constraints.

3 Methods

This study formulates the wall shear stress prediction as a supervised regression task, where the input X consists of 16, 384 vertical velocity (u_y) values sampled from an off-wall location at $y^+ = 15$, and the output Y is a scalar representing the wall-averaged wall shear stress corresponding to that snapshot. To evaluate different learning strategies, we implement the following regression models:

1. Linear Regression: A baseline model that assumes a linear relationship between off-wall velocity and wall shear stress.
2. K-Nearest Neighbors (KNN): A non-parametric method that predicts based on the average of k nearest training examples in the feature space.
3. Decision Tree: A tree-based model with controlled maximum leaf nodes to prevent overfitting.
4. Random Forest: An ensemble of decision trees trained with bagging to improve generalization.
5. Gradient Boosting: A sequential ensemble that corrects errors from previous models, using a learning rate and depth control.
6. Neural Network (PyTorch): A fully connected feedforward network with hidden layers and ReLU activations, trained using Huber and Mean Squared Error (MSE) loss and the Adam optimizer.

The dataset is split into training and testing subsets. All input features are standardized using `StandardScaler` from scikit-learn, and the target variable is normalized prior to model training. Performance is evaluated using Root Mean Squared Error (RMSE) on the test set. Additionally, scatter plots of predicted vs. true wall shear stress values ($\hat{y}$ vs. y) are used to visually assess model accuracy. The red line in all plots represents $\hat{y} = y$ line. All models are implemented in Python using scikit-learn and PyTorch libraries.

4 Results

4.1 Linear Regression

Linear regression resulted in a relatively high RMSE of 0.3, significantly worse than all other methods tested. The model was unable to capture the nonlinear trends present in the data, particularly failing to predict the initial sharp increase (or "bump") in wall shear stress. This proves the inadequacy of simple linear models for learning complex mappings between high-dimensional off-wall velocity fields and scalar wall quantities. The predicted vs. actual wall shear stress comparison is shown in Figure 1.

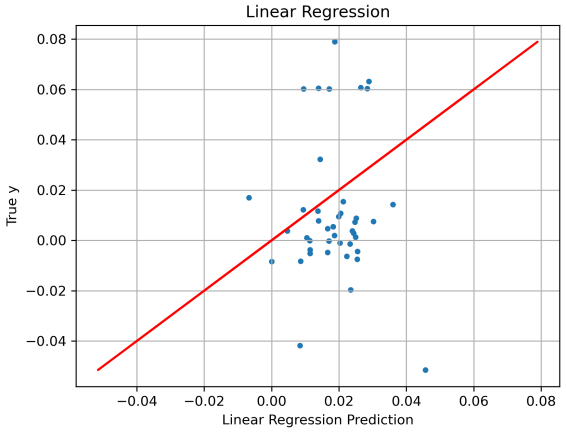

Figure 1: Predicted vs. actual wall shear stress using linear regression

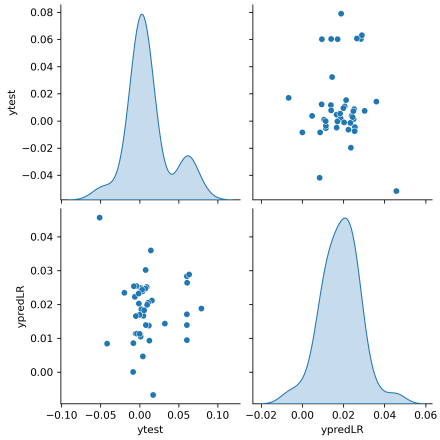

Figure 2: Comparison with actual wall-shear stress

4.2 K-Nearest Neighbors

The K-Nearest Neighbors (KNN) regressor was evaluated across a range of neighbor counts (N), as summarized in Table 1. The data was split 70%/30% into training and testing sets, resulting in only 96 training samples. This makes the choice of N especially critical, as large values begin to approach the total number of training points.

Number of Neighbors (N)	**RMSE**
5	0.064
30	0.032
40	0.029
50	0.028
95	0.027

Table 1: RMSE of K-Nearest Neighbors regressor for different values of N

As shown, the RMSE improves as N increases, with the lowest error at $N = 95$ (RMSE = 0.027). However, since $N = 95$ uses almost the entire training set for each prediction, the model effectively averages over

nearly all available data points. This reduces variance but significantly increases bias, leading to a loss of sensitivity to local variations i.e. underfitting. The prediction behavior across different N values is visualized in Figures 3-5. At low N (e.g., $N = 5$), the model overfits, showing high variance and poor generalization. At intermediate values like $N = 50$, the model achieves a good balance between bias and variance, better capturing compared to other Ns.

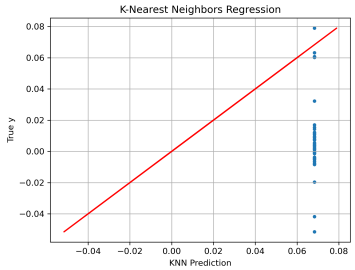

Figure 3: KNN for $N = 5$ (overfitting)

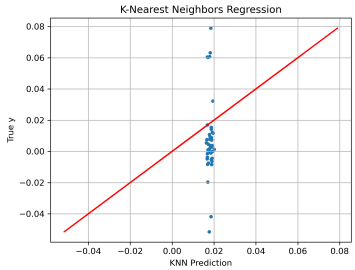

Figure 4: KNN for $N = 50$ (best trade-off)

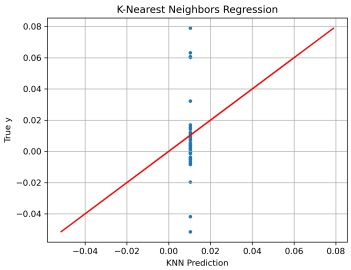

Figure 5: KNN for $N = 95$ (underfitting)

4.3 Decision Tree

The Decision Tree Regressor was evaluated with varying limits on the number of leaf nodes to understand the effect of model complexity on prediction quality. Table 2 summarizes the RMSE values obtained for different numbers of leaf nodes. Although the minimum RMSE of 0.014 occurred at $N = 2$, this configuration significantly underfits the data, as the model can only produce two unique output values. This causes flat predictions and makes the model miss smaller variations in the data.

Number of Leaf Nodes (N)	**RMSE**
2	0.014
4	0.016
15	0.015
25	0.017
50	0.017
100	0.017
200	0.017

Table 2: RMSE of Decision Tree Regressor for different maximum leaf node values

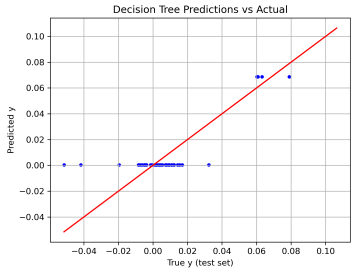

Figure 6: Tree for $N = 2$ (underfitting)

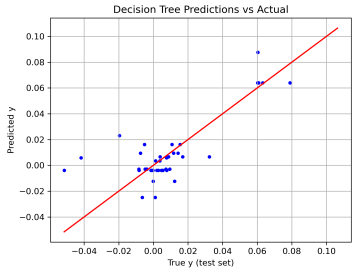

Figure 7: Tree for $N = 15$ (best balance)

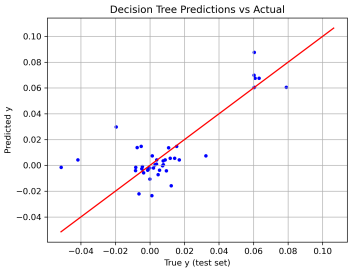

Figure 8: Tree for $N = 200$ (no further gain)

Among all tested models, the tree with $N = 15$ leaf nodes produced the most balanced results. As shown in Figure 7, it follows the ideal $\hat{y} = y$ line closely, captures the nonlinear trend effectively, and avoids

both underfitting and overfitting. Trees with more than 50–100 leaf nodes (e.g., $N = 200$) show no visible improvement, suggesting a saturation point in performance likely due to limited training data.

4.4 Random Forest

Random Forest regressors were trained with varying numbers of trees n-estimators = 100, 500, and 2000, while all other hyperparameters were kept at their default scikit-learn values (max depth=None, max features=auto, and bootstrap=True), allowing each decision tree in the ensemble to grow fully and use random subsets of data and features during training.

As shown in Table 3, the best RMSE of 0.014 was achieved with 100 trees, but increasing the number of estimators to 500 or 2000 did not improve performance. This is likely due to the limited training data because adding more trees does not introduce meaningful diversity in the model ensemble, and the averaging effect smooths out the predictions even further.

Number of Estimators	**RMSE**
100	0.014
500	0.014
2000	0.014

Table 3: RMSE of Random Forest Regressor for different numbers of trees

The scatter plots in Figures 9-11 show that predictions from all three models are tightly clustered around the mean, forming a narrow vertical band. While Random Forests achieved RMSE values similar to simpler decision trees, they produced overly smoothed predictions and showed limited generalization across the full range of shear stress values.

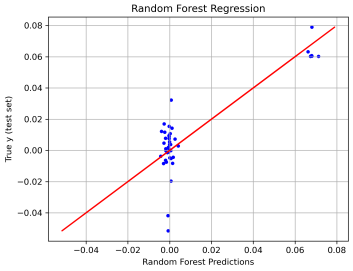

Figure 9: RF, $n = 100$

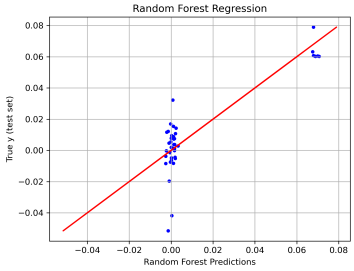

Figure 10: RF, $n = 500$

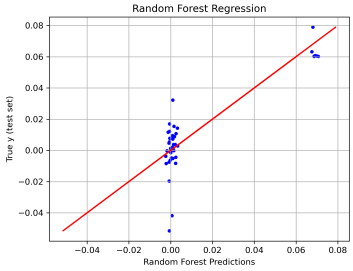

Figure 11: RF, $n = 2000$

4.5 Boosting

Gradient Boosting Regressors were evaluated using combinations of n-estimators, tree depth, and learning rate to understand their impact on prediction quality. As shown in Table 4, all RMSE values fell in a narrow range between 0.014 and 0.016, suggesting similar numerical performance across models. But the predicted vs. actual plots showed more significant differences.

The model with $n = 500$, depth=10, and learning rate=0.1 (Figure 12) produced predictions that followed the ideal diagonal well, capturing a wide range of wall shear stress values with minimal bias or flattening. Whereas, the shallow and conservative model with $n = 100$, depth=2, and learning rate=0.01 (Figure 13) clustered predictions tightly near the mean, indicating underfitting, even though its RMSE was only slightly higher. This shows that RMSE alone may not fully reflect a model's predictive quality, especially when evaluating regression with less samples. In such cases, visual diagnostics help identifying models that generalize well across the physical range of interest. Boosting with deeper trees and a moderately high learning rate offered the best balance between learning capacity and generalization.

n-estimators	max-depth	learning-rate	RMSE
100	2	0.1	0.014
100	10	0.1	0.015
100	2	0.01	0.015
100	10	0.01	0.016
500	2	0.1	0.014
500	10	0.1	0.015
500	2	0.01	0.014
500	10	0.01	0.015

Table 4: RMSE of Gradient Boosting Regressor for various hyperparameters

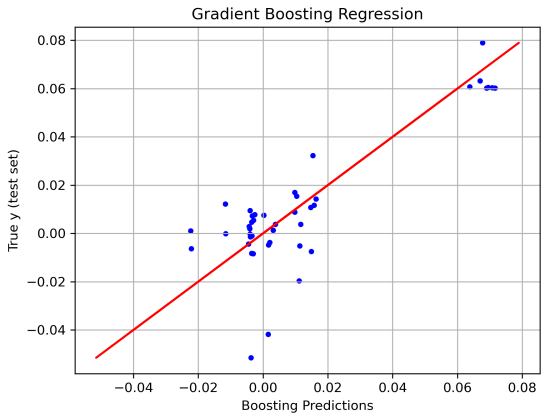

Figure 12: Best visual result: $n = 500$, depth=10, $\eta = 0.1$

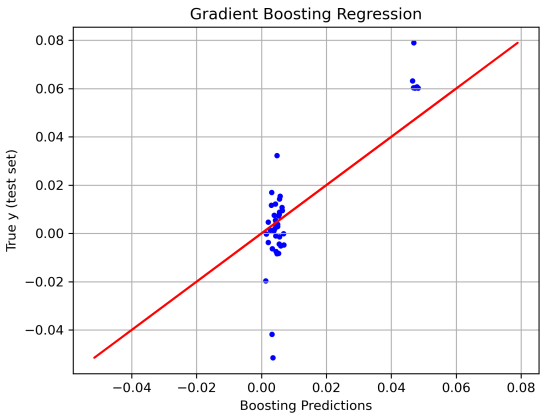

Figure 13: Worst visual result: $n = 100$, depth=2, $\eta = 0.01$

4.6 Neural Nets

For neural network, many hyperparameters were tested by varying the learning rate, number of epochs, L2 regularization strength, number of hidden layers, and the choice of loss function (Huber and MSE). The model still consistently failed to produce accurate predictions. Tested parameters are listed in Table 5.

Epoch	Learning Rate	Reg. Strength	Loss Function	Hidden Layers	Neurons	PCA Comp.	RMSE
1000	1e-3	1e-4	Huber	2	64	-	0.029
1000	1e-4	1e-4	Huber	2	64	-	0.03
1000	1e-2	1e-4	Huber	2	64	-	0.029
1000	1e-2	1e-4	Huber	3	64	-	0.046
1000	1e-3	1e-4	Huber	3	512	-	0.028
1000	1e-4	-	Huber	3	1024	-	0.03
2000	1e-3	1e-4	Huber	3	1024	-	0.03
500	1e-3	1e-4	Huber	3	1024	50	0.046
500	1e-3	1e-4	MSE	3	1024	50	0.040
5000	1e-3	1e-4	MSE	3	1024	50	0.048
500	1e-3	1e-4	MSE	3	1024	20	0.040
500	1e-3	1e-4	MSE	3	1024	10	0.036
500	1e-3	1e-4	MSE	3	1024	5	0.035
500	1e-3	1e-4	MSE	3	128	5	0.038
500	1e-3	1e-4	MSE	3	1024	1	0.037
500	1e-3	1e-4	MSE	3	128	1	0.028

Table 5: Hyperparameter combinations and PCA settings tested for the neural network model

To reduce input dimensionality, Principal Component Analysis (PCA) was applied to compress the 16,384-dimensional velocity field into a smaller number of components. While PCA improved training stability and reduced overfitting risk, it did not lead to meaningful performance gains. The underlying problem was

that the model was trying to learn from a very small dataset (only 96 training samples) with extremely high-dimensional input, even after reduction. The limitation of fully connected networks in this setting is

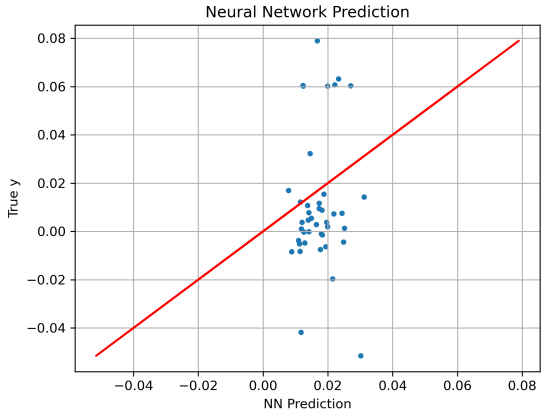

Figure 14: Best performing neural net

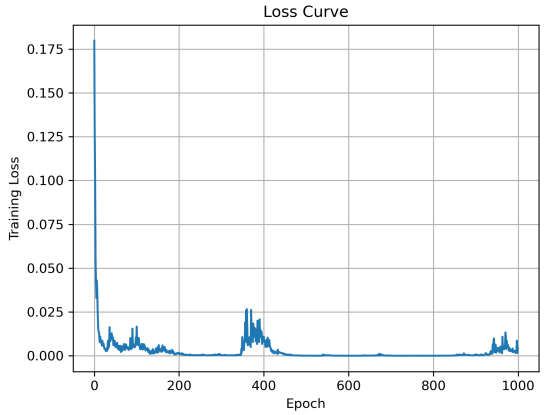

Figure 15: Loss curve, Huber

that they do not leverage the spatial structure present in the velocity field. Each velocity array corresponds to values at fixed grid points, meaning the data contains local spatial patterns like gradients, shear layers, or flow structures that dense layers could not use efficiently. Maybe Convolutional Neural Networks (CNNs), which are well-suited for learning from spatial data could fix this issue. CNNs apply local filters across the input, allowing them to detect features like localized velocity fluctuations and their spatial correlations. This drastically reduces the number of learnable parameters and makes the network more robust to overfitting, especially in low-data regimes.

5 Conclusion

In this study, various machine learning methods were tested to predict wall shear stress from off-wall vertical velocity fields. Among the models, random forests and gradient boosting yielded the best results, likely due to their ensemble nature and robustness to small datasets. Even with extensive tuning, the neural network performed poorly, showing high correlation with linear regression but not with the true target values.

	ytest	**ypredLR**	**ypredKNN**	**ypredT**	**ypredRF**	**ypredB**	**ypredN**
ytest	1.000	-0.030	-0.183	0.814	0.861	0.850	0.056
ypredLR	-0.030	1.000	0.222	0.134	0.109	0.094	0.900
ypredKNN	-0.183	0.222	1.000	-0.101	-0.130	-0.141	0.094
ypredT	0.814	0.134	-0.101	1.000	0.933	0.984	0.218
ypredRF	0.861	0.109	-0.130	0.933	1.000	0.944	0.172
ypredB	0.850	0.094	-0.141	0.984	0.944	1.000	0.170
ypredN	0.056	0.900	0.094	0.218	0.172	0.170	1.000

Table 6: Correlation matrix between true values and predictions from various regression models

This shows overfitting to noise and highlights the challenge of learning from high-dimensional inputs with very limited training data. Tree-based ensemble models were most effective for this simplified setup, and future improvements should include either dimensionality-aware architectures like Convolutional Neural Networks or larger, more structured datasets.

Appendix

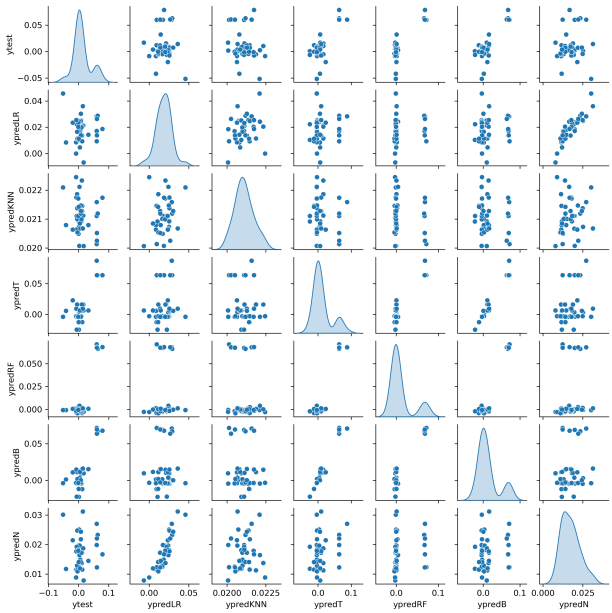

Figure 16: Pairplot comparing true wall shear stress values (y_{test}) with predictions from different regression models. Diagonal plots show kernel density estimates, while off-diagonal scatter plots illustrate correlation between model outputs.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ML tools
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
#nn
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from sklearn.decomposition import PCA

# RMSE function
def myrmse(y, yhat):
    rmse = np.sqrt(np.sum((y - yhat) ** 2) / len(y))
    return np.round(rmse, 3)
#*****************************************************************************
#Data Extraction
# Constants
num_velocity_lines = 16384
block_size = num_velocity_lines + 5  # total lines per time step

filename = 'Alldata.txt'
```

```
X_list = []
y_list = []

with open(filename, 'r') as f:
    lines = f.readlines()

num_blocks = len(lines) // block_size
print(f"Detected {num_blocks} time steps")

for i in range(num_blocks):
    start = i * block_size
    vel_lines = lines[start : start + num_velocity_lines]
    y_line = lines[start + block_size - 1]  # line N+5 = last line of the block

    try:
        velocities = [float(line.strip().split()[-1]) for line in vel_lines]

        if len(velocities) != num_velocity_lines:
            raise ValueError(f"Incomplete velocity data in block {i}")

        # Extract Y (wall shear stress)     last value in final line
        y_value = float(y_line.strip().split()[-1])

        X_list.append(velocities)
        y_list.append(y_value)

    except Exception as e:
        print(f"Skipping time step {i} due to error: {e}")
        continue

# Convert to arrays
X = np.array(X_list)
y = np.array(y_list)
#*****************************************************************************
#Train-test split
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.3, random_state=88)

#Scale features
scaler = StandardScaler()
Xtrain = scaler.fit_transform(Xtrain)
Xtest = scaler.transform(Xtest)

#*****************************************************************************
#Linear Regression
linreg = LinearRegression()
linreg.fit(Xtrain, ytrain)
ypredLR = linreg.predict(Xtest)

#*****************************************************************************
#K-Nearest Neighbors Regressor
knn = KNeighborsRegressor(n_neighbors=40)
knn.fit(Xtrain, ytrain)
ypredKNN = knn.predict(Xtest)
```

```
#*****************************************************************************
#Decision Tree Regressor
tmod = DecisionTreeRegressor(max_leaf_nodes=15,random_state=88)
tmod.fit(Xtrain, ytrain)
ypredT = tmod.predict(Xtest)
#*****************************************************************************
#Random Forest Regressor
rfm = RandomForestRegressor(
    n_estimators=100,
    random_state=88,
    n_jobs=-1
)
rfm.fit(Xtrain, ytrain)
ypredRF = rfm.predict(Xtest)
#*****************************************************************************
#Boosting Model
gbm = GradientBoostingRegressor(
    learning_rate=0.1,
    n_estimators=500,
    max_depth=10,
    random_state=88
)

gbm.fit(Xtrain, ytrain)
ypredB = gbm.predict(Xtest)
#*****************************************************************************
#Neural Nets
#Normalize X and y
sc = StandardScaler()
xtr = sc.fit_transform(Xtrain)
xte = sc.transform(Xtest)

scy = StandardScaler()
ytr = scy.fit_transform(ytrain.reshape(-1, 1))
yte = scy.transform(ytest.reshape(-1, 1))

#Convert to Tensors
xtr = torch.from_numpy(xtr.astype('float32'))
xte = torch.from_numpy(xte.astype('float32'))
ytr = torch.from_numpy(ytr.astype('float32'))
yte = torch.from_numpy(yte.astype('float32'))

#Dataset Class
class DF(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __len__(self):
        return len(self.y)
    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

#DataLoader
```

```
train_ds = DF(xtr, ytr)
train_dl = DataLoader(train_ds, batch_size=64, shuffle=True)

#Neural Network Model
class SLNN(nn.Module):
    def __init__(self, ninputs, nunits=64):
        super(SLNN, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(ninputs, nunits),
            nn.ReLU(),
            nn.Linear(nunits, nunits),
            nn.ReLU(),
            nn.Linear(nunits, 1)
        )
    def forward(self, x):
        return self.net(x)

#Initialize model
model = SLNN(ninputs=X.shape[1], nunits=64)

#Loss and Optimizer
learning_rate = 1e-3
l2par = 1e-4
#l2par = 0
loss_fn = nn.HuberLoss() #nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=l2par)

#Training Loop
epochs = 1000
loss_vals = np.zeros(epochs)

def train_loop(dataloader, model, loss_fn, optimizer):
    model.train()
    for X, y in dataloader:
        pred = model(X)
        loss = loss_fn(pred, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return loss_fn(model(dataloader.dataset.x), dataloader.dataset.y).item()

#Prediction on Test Set
model.eval()
with torch.no_grad():
    ypredN = model(xte).numpy()
    ypredN = scy.inverse_transform(ypredN).flatten()
```