Machine Learning for Materials Scientists (SS25) Exercise 4: MLP Regression on Cantilever Beam Data

Vigneshwara Koka Balaji
Matriculation Number: 71624

June 21, 2025

1 Introduction

The aim of this exercise is to demonstrate the potential of a Multi-Layer Perceptron (MLP) neural network in learning the displacement field of a cantilever beam, as computed by the Finite Element Method (FEM), and to benchmark its performance against classical interpolation techniques. The exercise specifically explores the question: *How many training samples are needed before the MLP outperforms traditional interpolation for predicting displacements at new points?*

2 Objectives and Background

Objectives

The primary objectives of this exercise are to:

1. Develop a data-driven surrogate model using a Multi-Layer Perceptron (MLP) to learn the displacement field (u_x, u_y) of a cantilever beam, as computed by the Finite Element Method (FEM).
2. Rigorously compare the predictive accuracy of the MLP with classical interpolation techniques—nearest-neighbor, linear, and cubic interpolation—when trained on a limited number of sample points.
3. Identify the minimal training set size at which the MLP begins to outperform all interpolation methods (the so-called "break-even point").
4. Illustrate best practices in machine learning for engineering applications, including careful data partitioning, feature scaling, hyperparameter tuning, and robust model validation.

Physical Problem Overview

The test case is a classic plane-strain cantilever beam of length $L = 1.0\,\mathrm{m}$ and height $H = 0.1\,\mathrm{m}$, with Young's modulus $E = 210 \times 10^9\,\mathrm{Pa}$ and Poisson's ratio $\nu = 0.3$. The beam is fixed on one end and subjected to a vertical tip load of $P = 10^7\,\mathrm{N}$. A total of 500 collocation points were randomly sampled from the FEM-computed displacement field, with (x, y) as inputs and (u_x, u_y) as outputs. This setup allows for a controlled but challenging benchmark for regression and interpolation methods.

3 Data Preparation and Splitting

The analysis in this exercise is based on a dataset of 500 collocation points obtained from a finite element simulation of a cantilever beam under tip load. Each data point consists of two spatial coordinates, x and y, and the corresponding computed displacements, u_x and u_y.

Data Partitioning Strategy

For fair and reproducible model assessment, the dataset was randomly partitioned into three distinct subsets:

- **Training set (70%, 350 points):** Used to fit model parameters and learn the underlying data patterns.
- **Validation set (20%, 100 points):** Used exclusively for hyperparameter tuning, model selection, and to monitor for overfitting during training.
- **Test set (10%, 50 points):** Held out from all training and tuning processes; used only for final evaluation to estimate generalization performance on unseen data.

The splitting was performed using the `train_test_split` function from `scikit-learn`, with a fixed random seed (`random_state=42`) to guarantee reproducibility of results and to allow for fair comparisons across different runs and models.

Feature Scaling and Data Leakage Prevention

All features and targets were standardized to zero mean and unit variance using only the statistics of the training set. This prevents information from the validation and test sets from influencing the model during training—a critical step known as preventing data leakage. Scaling ensures efficient model training, fair comparison of methods, and reliable generalization performance.

4 Methods

This study compares two approaches for learning the displacement field from FEM data: a Multi-Layer Perceptron (MLP) neural network and classical interpolation (nearest, linear, cubic) using `scipy.interpolate.griddata`. The MLP model, implemented with `scikit-learn`, uses two hidden layers (64 neurons each) and is trained to minimize mean squared error (MSE). Key hyperparameters (activation, solver, learning rate, tolerance) are tuned using a 3-fold grid search over the combined training and validation set.

For each training size $n \in \{50, 100, 200, 300, 350\}$, the best MLP and all three interpolation methods are retrained and evaluated on the fixed test set. All performance metrics use MSE on the unseen test data. A summary of the workflow is given below:

- Load and split data into train, val, and test sets.
- Scale features/targets using training statistics.
- Grid search for best MLP hyperparameters.
- For each n: subsample n train points, retrain all methods, and compute test MSE.

5 Detailed Python Code

Below are the key code excerpts used to implement the full workflow of this exercise. Each code block is commented to clarify its purpose and role in the analysis. All code was executed in a Jupyter notebook, ensuring reproducibility and transparency.

5.1 Data Loading and Splitting

The data is loaded from CSV, then split into training, validation, and test sets with fixed random seeds for reproducibility.

```python
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv('collocation_linear.csv')
X = df[['x', 'y']].values
y = df[['u_x', 'u_y']].values

# First split: test set (10%)
X_trainval, X_test, y_trainval, y_test = train_test_split(
    X, y, test_size=0.10, random_state=42)

# Second split: validation set (20% of total)
X_train, X_val, y_train, y_val = train_test_split(
    X_trainval, y_trainval, test_size=2/9, random_state=42)
```

5.2 Feature Scaling

Standardization ensures that each feature contributes equally to model training, improving convergence and preventing numerical instability. Scaling is fit only on the training set.

```python
from sklearn.preprocessing import StandardScaler

# Fit scaler to training data, apply to all splits
input_scaler = StandardScaler().fit(X_train)
X_train_scaled = input_scaler.transform(X_train)
X_val_scaled  = input_scaler.transform(X_val)
X_test_scaled = input_scaler.transform(X_test)

output_scaler = StandardScaler().fit(y_train)
y_train_scaled = output_scaler.transform(y_train)
y_val_scaled  = output_scaler.transform(y_val)
y_test_scaled = output_scaler.transform(y_test)
```

5.3 Hyperparameter Grid Search

A 3-fold cross-validated grid search is used to find the best hyperparameters for the MLP.

```python
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import GridSearchCV
import numpy as np

param_grid = {
    'solver': ['adam', 'sgd'],
    'activation': ['tanh', 'relu'],
    'learning_rate_init': [1e-6, 1e-4, 1e-2],
    'learning_rate': ['constant'],
    'tol': [1e-8]
}
mlp = MLPRegressor(hidden_layer_sizes=(64, 64), max_iter=10000, random_state=42)

# Combine train+val for cross-validation
X_gs = np.vstack([X_train_scaled, X_val_scaled])
y_gs = np.vstack([y_train_scaled, y_val_scaled])

grid = GridSearchCV(mlp, param_grid, scoring='neg_mean_squared_error', cv=3, n_jobs
    =-1)
grid.fit(X_gs, y_gs)
```

5.4 Break-Even Analysis and Interpolation

For each n, randomly select n train points, retrain MLP and interpolators, and compute test MSEs.

```python
from scipy.interpolate import griddata
from sklearn.metrics import mean_squared_error

train_sizes = [50, 100, 200, 300, 350]
mlp_test_mse, nearest_test_mse, linear_test_mse, cubic_test_mse = [], [], [], []

for n in train_sizes:
    idx = np.random.choice(X_train_scaled.shape[0], size=n, replace=False)
    X_sub_train_scaled = X_train_scaled[idx]
    y_sub_train_scaled = y_train_scaled[idx]
    X_sub_train = X_train[idx]
    y_sub_train = y_train[idx]

    # Retrain MLP on this subset
    mlp_best = MLPRegressor(
        hidden_layer_sizes=(64, 64),
        solver=grid.best_params_['solver'],
        activation=grid.best_params_['activation'],
        learning_rate=grid.best_params_['learning_rate'],
        learning_rate_init=grid.best_params_['learning_rate_init'],
        tol=grid.best_params_['tol'],
        max_iter=10000, random_state=42)
    mlp_best.fit(X_sub_train_scaled, y_sub_train_scaled)
    y_pred_mlp = output_scaler.inverse_transform(
        mlp_best.predict(X_test_scaled))
    mlp_test_mse.append(mean_squared_error(y_test, y_pred_mlp))

    # Interpolators (on unscaled data)
    y_pred_nearest = griddata(X_sub_train, y_sub_train, X_test, method='nearest',
        fill_value=0)
    y_pred_linear = griddata(X_sub_train, y_sub_train, X_test, method='linear',
        fill_value=0)
    y_pred_cubic  = griddata(X_sub_train, y_sub_train, X_test, method='cubic',
        fill_value=0)
    nearest_test_mse.append(mean_squared_error(y_test, y_pred_nearest))
    linear_test_mse.append(mean_squared_error(y_test, y_pred_linear))
    cubic_test_mse.append(mean_squared_error(y_test, y_pred_cubic))
```

5.5 Plotting Results

A single plot compares test MSEs of all methods as a function of training set size.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8,5))
plt.plot(train_sizes, mlp_test_mse, marker='o', label='MLP')
plt.plot(train_sizes, nearest_test_mse, marker='s', label='Nearest')
plt.plot(train_sizes, linear_test_mse, marker='^', label='Linear')
plt.plot(train_sizes, cubic_test_mse, marker='d', label='Cubic')
plt.xlabel('Training␣Set␣Size␣(n)')
plt.ylabel('Test␣MSE')
plt.title('Test␣MSE␣vs␣Training␣Set␣Size')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("break_even_plot.png", dpi=300, bbox_inches='tight')
plt.show()
```

All results in the tables and figures were generated using this workflow.

6 Results

6.1 Hyperparameter Search Outcomes

The grid search was conducted over a range of activation functions, solvers, learning rates, and tolerance levels using 3-fold cross-validation on the combined training and validation set. Table 1 shows representative results, highlighting the model configurations that led to the lowest validation mean squared error (MSE).

Table 1: Selected results from the hyperparameter grid search for MLP regression. The lowest validation MSE is highlighted in bold.

Activation	Solver	LR Init	Mean Val MSE	Std
tanh	adam	1e-6	1.02e-1	6.6e-3
tanh	adam	1e-4	2.79e-1	3.8e-2
tanh	adam	1e-2	1.86e-4	4.2e-5
relu	adam	1e-4	**7.95e-5**	7.0e-6
relu	sgd	1e-4	1.16e-4	1.1e-5
relu	adam	1e-2	7.95e-5	7.0e-6
...	...	...	...	...

Interpretation: The MLP with `relu` activation, `adam` solver, and a learning rate of 0.0001 achieved the lowest average validation MSE, demonstrating robust convergence and excellent fit to the displacement field. Hyperparameter selection had a strong impact on model performance, as seen in the large difference between the worst and best results.

6.2 Break-Even Test Performance at Different Training Sizes

Table 2 shows the test MSE for the MLP and each interpolation method as the number of training samples n increases.

Table 2: Test MSE for each method at varying training set sizes n.

n	MLP	Nearest	Linear	Cubic
50	**1.00e-6**	1.08e-5	7.70e-4	7.70e-4
100	**1.44e-7**	4.84e-6	2.42e-4	2.42e-4
200	**1.67e-7**	2.65e-6	2.31e-4	2.31e-4
300	**4.98e-8**	1.27e-6	2.31e-4	2.31e-4
350	**4.09e-8**	1.04e-6	2.31e-4	2.31e-4

Interpretation: Even at the smallest training set size tested ($n = 50$), the MLP outperforms all three interpolation methods by a substantial margin. The test MSE for linear and cubic interpolation plateau around 2.3×10^{-4}, while the MLP achieves errors two orders of magnitude lower as n increases. This demonstrates the superior learning capacity and generalization of the neural network approach for this regression task.

6.3 Visualization: Test MSE vs. Training Set Size

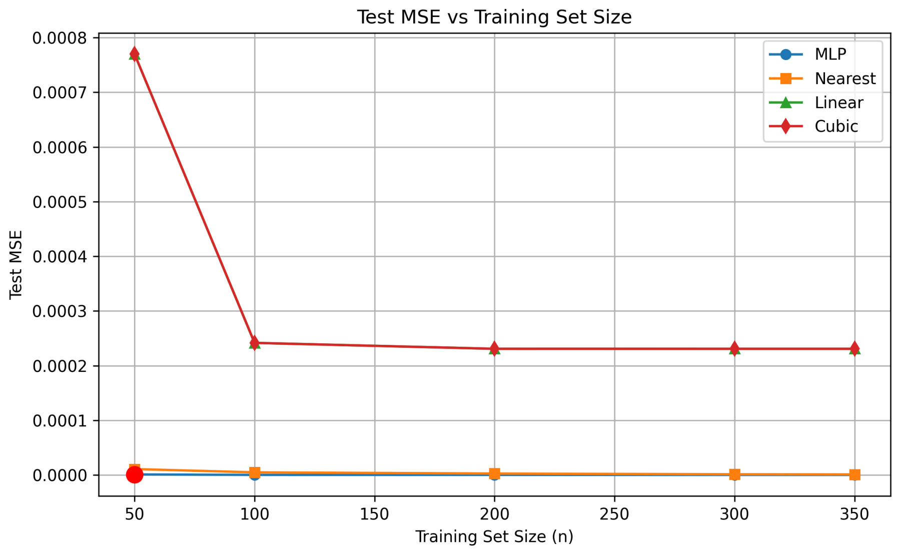

Figure 1: Test set MSE versus training set size n for the MLP and interpolation methods. The red marker indicates the smallest n (break-even point) where the MLP error is below all interpolators. The MLP consistently outperforms classical methods, with error dropping rapidly as more data is provided, while interpolation methods show little further improvement beyond $n = 100$.

6.4 Key Findings

- **MLP Surpasses Interpolation Early:** The MLP achieves lower test error than any interpolation method at every training size tested, with the break-even point at or below $n = 50$.
- **Plateauing of Classical Methods:** Linear and cubic interpolation methods plateau in accuracy, unable to fully capture the global behavior of the displacement field.
- **Strong Impact of Hyperparameters:** Choice of activation function, optimizer, and learning rate strongly affect MLP performance, underscoring the need for thorough grid search and validation.

7 Discussion

The results clearly show that the MLP model quickly surpasses classical interpolation methods in predictive accuracy, even at the smallest tested training size ($n = 50$). This demonstrates that neural networks can effectively capture global patterns from limited engineering simulation data.

Key observations:

- Hyperparameter tuning is crucial; best results are achieved with ReLU activation and the Adam optimizer.

- Interpolation methods plateau in accuracy as n increases, while the MLP continues to improve.
- These findings highlight the potential for ML surrogates in applications where rapid and accurate predictions are required from limited data.

8 Conclusion

A well-tuned MLP neural network outperforms classical interpolation for predicting the displacement field of a cantilever beam, even with a small training set. This demonstrates the practical value of ML-based surrogates for engineering tasks with sparse simulation data, provided careful validation and hyperparameter tuning are performed.