Predicting Hyperelastic Parameters using Artificial Neural Networks

Nathan Chiu

1. Introduction

Curve fitting is the process of finding a mathematical function (curve) that best represents the relationship between a set of data points. To curve fit a given dataset, such as stress-strain data collected from experiments, the conventional approach begins by selecting a parameterized function, such as the Ogden model. Curve fitting software then applies traditional mathematical methods, such as least-squares regression or nonlinear optimization algorithms, to estimate the function parameters that best align the chosen function with the experimental data. As an alternative, the neural networks developed in this work provide a machine learning–based approach to curve fitting. Unlike traditional curve-fitting software, which predicts parameters within a predefined equation, neural networks learn the stress–strain relationship directly from training data and generate predictions without requiring an explicit equation.

2. Synthetic Training data

Synthetic stress–strain data generated using MATLAB was used to train the neural networks, as experimental data for hyperelastic materials are often costly and time-consuming to obtain. Generated data enables networks to observe diverse stress-strain behaviours, thereby improving their ability to generalize. The dataset consists of 9 models: Ogden, Reduced Polynomial, and Polynomial (each of orders 1, 2, and 3). The dataset has a shape of [9000 × 210], containing 9000 rows (all unique samples, 1000 per model) and 210 columns: 1 for the model label, 100 for stress values, 100 for strain values, and 9 for model parameters (with unused parameter columns filled with NaN for models requiring fewer than 9). In each sample, the strain values were generated as 100 random numbers between 0 and 3, sorted in ascending order. Random noise between 3–5% was added to 50% of the stress values to mimic experimental uncertainty. The remaining noise-free samples allowed the model to learn the true stress-strain relationship.

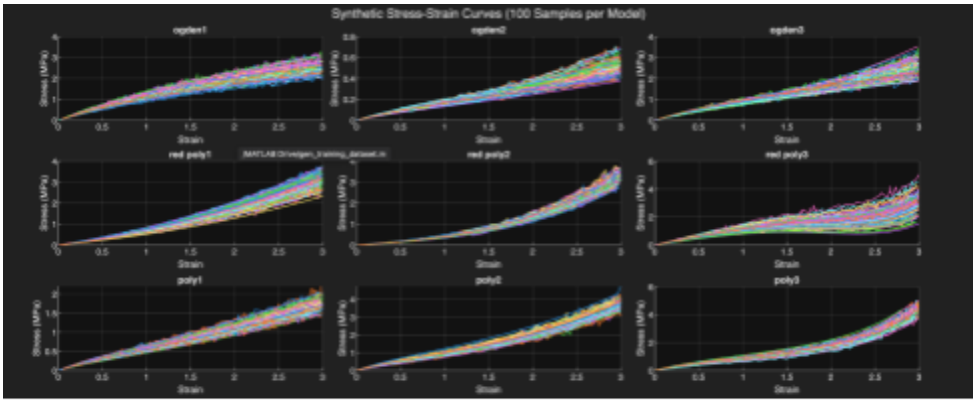


Figure 1: 100 Sample Curves for Each Model in the Synthetic Stress-Strain Training Dataset

3. Data Preprocessing

The dataset was split into 80% training and 20% testing, where the input features (X) are stress and strain, and the target features (y) are the model parameters. All values were normalized to the range [0,1] using the equation in Figure 2, with strain scaled using fixed bounds of 0.0 (min) and 3.0 (max). Stress values were normalized at every point, where each stress point was normalized using the minimum and maximum values of that point across all training samples of the corresponding model. For example, to normalize the value of stress_32 in the Ogden1 model, the minimum and maximum are found by comparing all training samples at that same column (stress_32) within Ogden1. This process is repeated for every stress column, meaning each model has 100 distinct min/max values that correspond to its 100 stress points. The same normalization approach was applied to the target features (parameters). Since each model can have up to nine parameters, this results in a maximum of nine distinct min/max values per model, with fewer if the model uses fewer parameters. We **only** use the min/max from the **training set** to ensure consistent scaling. For example, if a feature in the training set has a min of 0 and a max of 100, scaling maps 100→1.0 and 50→0.5. However, if the test set were scaled with its own min/max (from the test set) and the maximum was 50, then 50 would incorrectly be mapped to 1.0, thus leading to normalization inconsistencies.

$$X_{NORMALIZED} = \frac{(X - X_{MINIMUM})}{(X_{MAXIMUM} - X_{MINIMUM})}$$

Figure 2: Normalization Formula used to Scale Data

4. Network Architecture

The proposed model is a fully connected feedforward neural network consisting of six layers: an input layer, a flatten layer, three hidden dense layers with 128, 64, and 32 neurons, respectively, each using ReLu activation functions, and a dense output layer where the number of neurons corresponds to the number of parameters in the selected model.

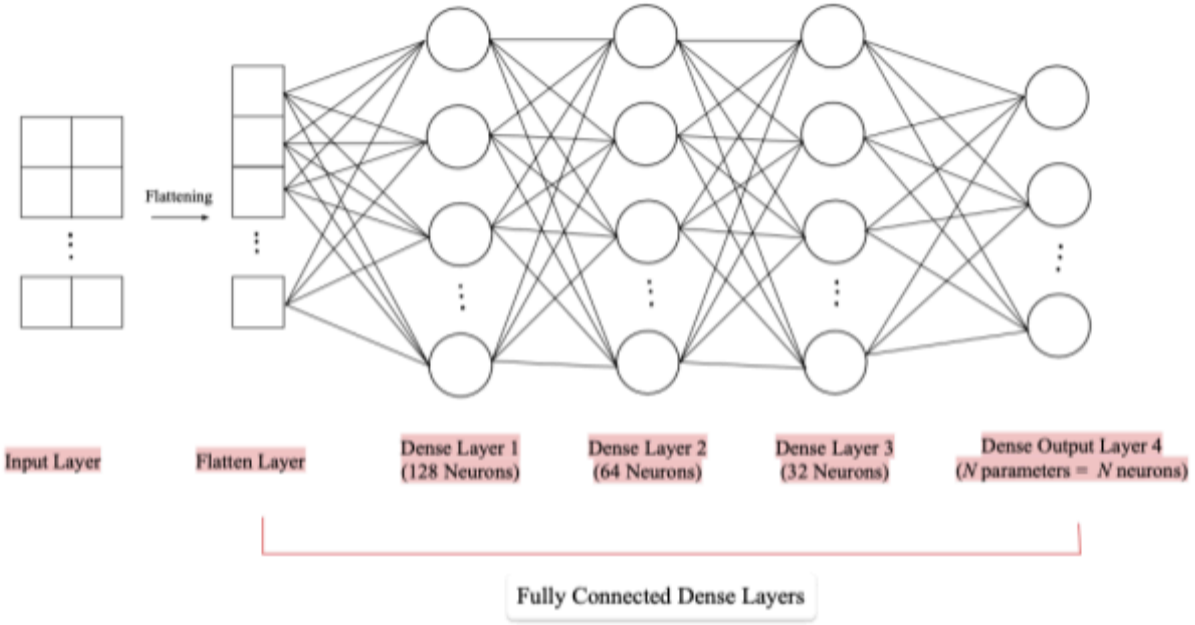


Figure 3: Schematic of Proposed Neural Network

4.1 Flattening the Input

The input layer accepts data with shape (100, 2), where each sample corresponds to a stress–strain curve with 100 points, and each point contains two features: stress and strain. This forms a 2D array of dimension 100×2. Since dense layers require 1D vectorized input, a flatten layer is applied to convert the input into a single vector of length 200.

```
layers.Flatten()
   [[stress_1, strain1], [stress_2,strain_2], … , [stress_100, strain_100]]
→ [stress_1, strain_1, stress_2, strain_2, …, stress_100, strain_100]
```

Figure 4: Flattening Input to Match Required 1D vectorized input

4.2 Dense Layers

The dense layers are fully connected layers, meaning that every neuron in a layer receives input from every neuron in the preceding layer, as seen in Figure 5. The dense layers function as a hierarchy of pattern recognizers. The early layers capture local patterns in the stress-strain data while the deeper layers combine these patterns into higher-level representations that describe the overall behaviour of the material. In the hidden layers, each neuron combines the inputs through

a weighted summation and then applies an activation function, which allows the network to capture the complex relationships between hyperelastic stress-strain [2].

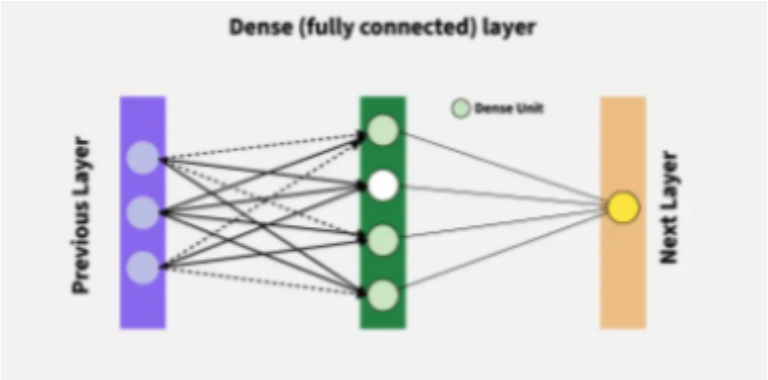


Figure 5: Every neuron in one layer is connected to every neuron in the previous[1]

5. Training the Neural Networks

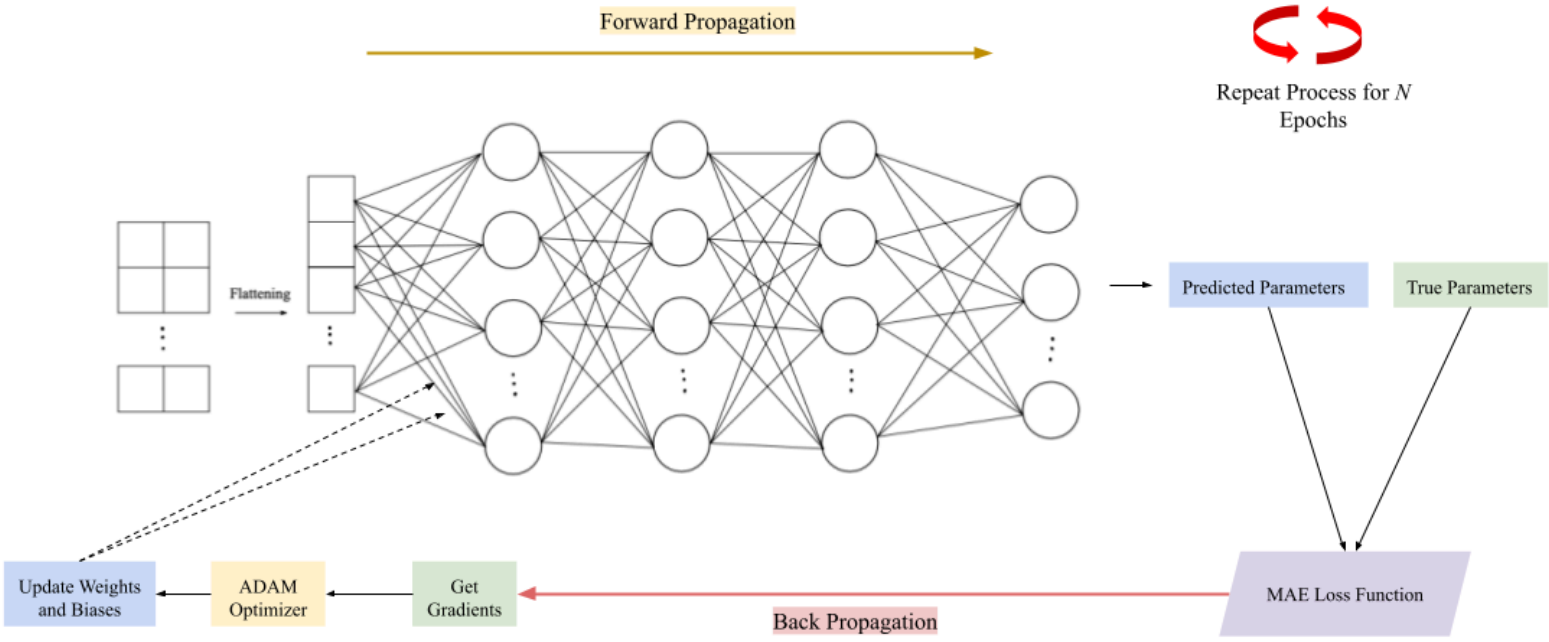


Figure 6: Neural Network training process

5.1 Forward Propagation

Forward propagation initiates the training process, in which the training data is passed through the network's layers to produce a predicted output. The equations in Figure 7 describe the forward propagation process through the proposed network. The input $A^{[0]}$ represents the stress–strain curve sampled at 100 points with two features per point. The flattened layer reshapes this into a single 200-dimensional vector, $A^{[1]}$, which is then passed into the first dense layer. In each dense layer, the neuron outputs are computed by taking a weighted sum of the previous layer's activations, $Z^{[n\ layer]} = W^{[n\ layer]} * A^{[n-1\ layer]} + b^{[n\ layer]}$, followed by the application of a non-linear activation function $A^{[n\ layer]} = f(Z^{[n\ layer]}) = \text{ReLU}(Z^{[n\ layer]})$ [3]. The ReLU activation function was chosen for the hidden layers because it has been shown to perform effectively in capturing non-linear relationships in regression tasks [4]. Finally, the output layer produces the network predictions. Training was carried out for 100 epochs with a batch size of 8, meaning the network updated its weights and biases after every 8 samples, and completed 100 full passes through the training dataset.

Layer 1, Input layer $\rightarrow A^{[0]} = X$, Shape [100x2]

Layer 2, Flatten Layer $\rightarrow A^{[1]} = \text{Flatten}(A^{[0]})$, Shape [200]

Layer 3, Dense layer $\rightarrow Z^{[2]} = W^{[2]} * A^{[1]} + b^{[2]}$, $A^{[2]} = f(Z^{[2]}) = \text{ReLU}(Z^{[2]})$

Layer 4, Dense layer $\rightarrow Z^{[3]} = W^{[3]} * A^{[2]} + b^{[3]}$, $A^{[3]} = f(Z^{[3]}) = \text{ReLU}(Z^{[3]})$

Layer 5, Dense layer $\rightarrow Z^{[4]} = W^{[4]} * A^{[3]} + b^{[4]}$, $A^{[4]} = f(Z^{[4]}) = \text{ReLU}(Z^{[4]})$

Layer 6, Dense Output Layer $\rightarrow Z^{[5]} = W^{[5]} * A^{[4]} + b^{[5]}$, $A^{[5]} = f_{out}(Z^{[5]}) = Z^{[5]}$

X = input data
$A^{[n\ layer]}$ = Activation (output) of layer *n*
$Z^{[n\ layer]}$ = Weighted input to layer *n*/linear combination of layer *n* before activation
$b^{[n\ layer]}$ = Bias for layer *n*
$f(Z^{[n\ layer]})$ = Activation function applied to $Z^{[n\ layer]}$

Figure 7: Forward Propagation Equations of the Proposed Neural Network [3]

5.2 Loss Function

Once parameter predictions are made during forward propagation, these predictions are passed through the mean average error (MAE) loss function. The Mean Absolute Error (MAE) loss function computes the average of the absolute differences between the predicted and true parameter values. Mean Squared Error (MSE) is more sensitive to outliers since it squares the errors, whereas MAE applies a uniform penalty to all errors. This is important for stress–strain data, which often contains noise. Using MAE helps the model remain less influenced by outliers, focus on the general trends in the data, and produce more stable parameter predictions [5].

$$\mathrm{MAE} = \frac{1}{n} \sum_{i=1}^{n} | x_i - x |$$

Figure 8: MAE Formula

5.3 Back Propagation

After the model compares its predictions to the true values using the loss function, it must determine how much to adjust the weights and biases, as well as whether these changes should increase or decrease. To do this, the network performs backpropagation, a technique that moves backward through the layers to calculate how each weight and bias contributed to the final error. These calculations, known as gradients, serve as a guide for optimization algorithms, such as the Adam (Adaptive Moment Estimation) optimizer used in the proposed network. The optimizer then uses these gradients to determine exactly which weights and biases to update, and in what direction, to minimize the prediction error [6].

6. Evaluating the Neural Networks

6.1 Interpolation

Before the models can predict parameters, the evaluation data must be interpolated to match the input shape the model was trained on. The model expects an input of shape (100, 2), meaning 100 samples with 2 features: stress and strain. In practice, evaluation datasets don't often have exactly 100 samples, so interpolation is required. First, the strain is interpolated by taking the minimum and maximum values and generating 100 evenly spaced points between them. Next, the stress is interpolated by estimating values at these new strain points. For each new strain point, the algorithm takes the two nearest original strain values and linearly interpolates the corresponding stress.

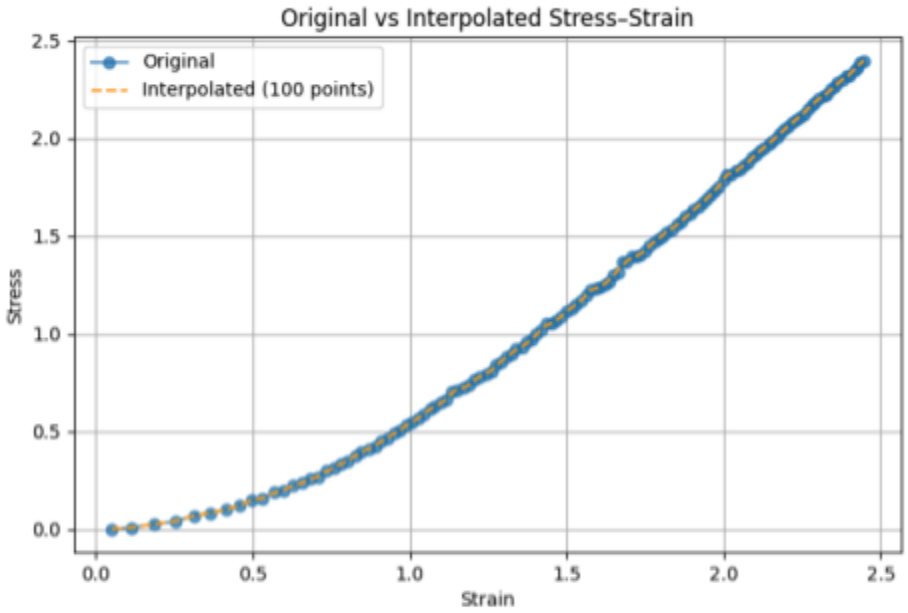


Figure 9: 130 stress-strain points interpolated to 100 stress-strain points

6.2 Selection of the Best-Fit Hyperelastic Model

To determine the best-fit hyperelastic model for a given dataset, each neural network first predicts the material parameters. These parameters are then used to calculate stress using the corresponding stress equations. The predicted stresses from every model are compared with the true experimental stresses to evaluate accuracy. Although it may seem that the model is tested on data it had already seen, the network does not memorize these stress values, but instead, generates predictions based on the stress-strain relationship learned during training. The R^2-score is used as the evaluation metric, as it measures how well a predicted curve matches the overall shape of the experimental stress–strain data. The neural network with the highest R^2-score is then selected as the best-fit model.

```
ogden1
r^2: 0.8015783799141691
ogden2
r^2: 0.7491145117008093
ogden3
r^2: 0.9358090958278388
poly1
r^2: 0.6452298732862182
poly2
r^2: 0.7828570475090516
poly3
r^2: 0.7462507723530426
red_poly1
r^2: 0.8293432401197821
red_poly2
r^2: 0.6491720723453802
red_poly3
r^2: 0.25714491426113195

Chosen Model: ogden3
R^2 Score: 0.9358090958278388

Predicted Parameters[ 2.08968896e-01  8.53548926e-03  8.13131442e-02
-6.29442490e+00  4.88549475e-03  9.13958261e-02]
```

Figure 10: Terminal output that displays the results from each neural network and displaying the best performing one based on R^2

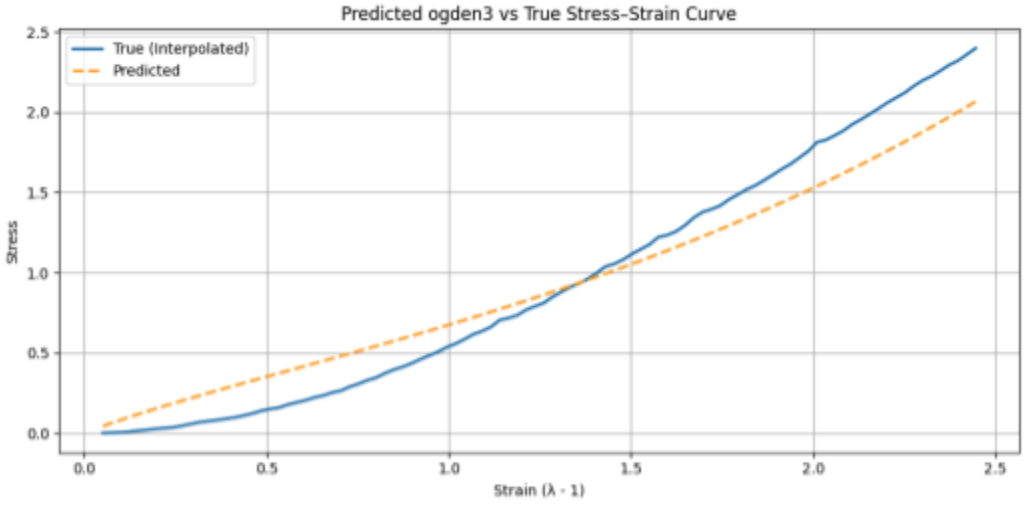


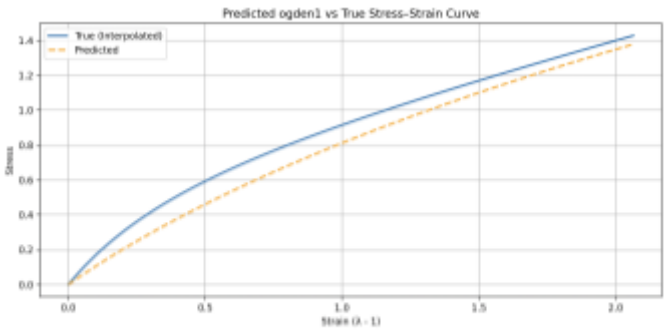
Figure 11: Plotting the Best-Fit Model

6.3 Mapping Parameter Columns to Parameter Type

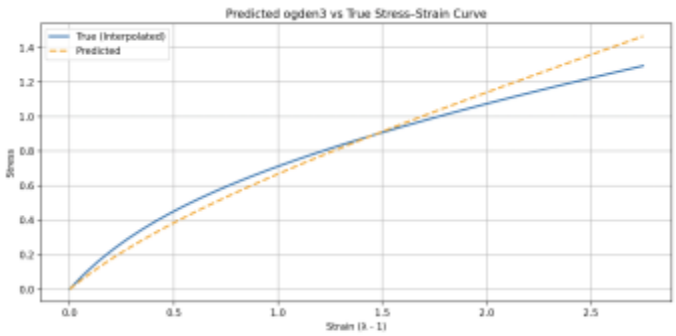
In the training data, the nine parameters correspond to different material constants depending on the chosen model. When a model outputs parameters, they are listed as an array in the format [*param_1, param_2, ..., param_N*], where N is the number of constants in that model. The following table provides the mapping between these generic parameters and their corresponding hyperelastic constants.

	Ogden1	Ogden2	Ogden3	Reduced Poly1	Reduced Poly2	Reduced Poly3	Poly1	Poly2	Poly3
Param1	mu1	mu1	mu1	C10	C10	C10	C10	C10	C10
Param2	alpha1	alpha1	alpha1		C20	C20	C01	C01	C01
Param3		mu2	mu2			C30		C20	C20
Param4		alpha2	alpha2					C11	C11
Param5			mu3						C02
Param6			alpha3						C30
Param7									C21
Param8									C12
Param9									C03

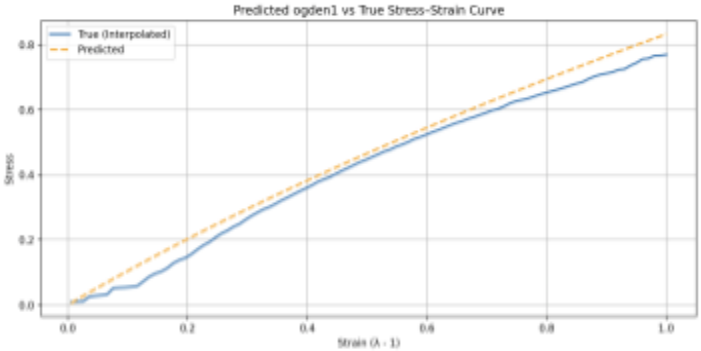
Sample Results



Chosen Model: ogden1
R^2 Score: 0.9746661385881633



Chosen Model: ogden3
R^2 Score: 0.9392594543657758



Chosen Model: ogden1
R^2 Score: 0.9394882356709204

Conclusions

The proposed neural networks demonstrate the use of machine learning as an approach to stress–strain curve-fitting. The model achieved an average R^2 score of 0.93, demonstrating strong predictive performance. Future improvements could focus on diversifying the training parameters to enhance generalizability. The findings highlight the potential of neural networks to complement or extend traditional curve-fitting methods, offering greater flexibility and adaptability for modelling complex material behavior.

[1] GeeksforGeeks. (2025, June 14). *What is fully connected layer in deep learning?*. GeeksforGeeks. https://www.geeksforgeeks.org/deep-learning/what-is-fully-connected-layer-in-deep-learning/

[2]Montufar, G., Pascanu, R., Cho, K., & Bengio, Y. (n.d.). On the Number of Linear Regions of Deep Neural Networks. https://arxiv.org/pdf/1402.1869

[3]Chapter 8 Neural Networks. (n.d.). https://openlearninglibrary.mit.edu/assets/courseware/v1/9c36c444e5df10eef7ce4d052e4a2ed1/asset-v1%3AMITx%2B6.036%2B1T2019%2Btype%40asset%2Bblock/notes_chapter_Neural_Networks.pdf

[4]Brownlee, J. (2020, August 20). A Gentle Introduction to the Rectified Linear Unit (ReLU). https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

[5]Google. (n.d.). *Linear regression: Loss*. Google. https://developers.google.com/machine-learning/crash-course/linear-regression/loss

[6]Bergmann, D., & Stryker, C. (2025, July 2). *What is backpropagation?*. IBM. https://www.ibm.com/think/topics/backpropagation