Regression using ANN (function approximation)

Case study

using synthetic data

(numerical and categorical features)

Content

1. **Data Generation and Preprocessing**:
    - Create synthetic data and standardizing it for training.
    - Add categorical data and use one-hot encoder
    - Understanding data – data correlation
2. **Building a Deep Neural Network**:
    - How to design a regression-focused DNN using Keras.
3. **Training the Model**:
    - Training with validation splits and visualizing training progress.
    - Use regularization (dropout) and Early stopping
4. **Model Evaluation**:
    - Calculating Mean Squared Error and understanding model performance.
5. **Making Predictions**:
    - Converting scaled predictions back to the original scale for interpretation.

Dataset generation

Dataset Description

The dataset used is a synthetic dataset with the next characteristics:

1. **Numerical Features**:
   - The dataset contains *num_samples* = 3000 examples with *num_features* = 8 numerical features
   - These features are generated randomly using np.random.rand, which creates values between 0 and 1, filling an array with the specified shape (num_samples, num_features).
   - 3000 rows (samples) and 8 columns (features).
   - This random data is then transformed in various non-linear ways using functions like sin, cos, exp, powers, and sums, to create complex relationships within the features and contribute to the target variable's value
2. **Categorical Features**:
   - Two categorical features (xcat_1, xcat_2) are added using np.random.choice.
   - These are generated by randomly selecting values from the categories ['A', 'B', 'C'].

Dataset generation

3. Combining into a DataFrame:

- All these features (numerical and categorical) are combined into a pandas DataFrame called x for easier handling.
- The target variable (y) is also included in the final DataFrame called combined_df.

4. Target Variable (y):

- The target variable is calculated using a combination of these non-linear interactions between the numerical features.
- Gaussian noise (randomness) is then added to the target variable, controlled by noise_level, to make the prediction task more challenging.

Shape and Format:

- The final dataset has 3000 samples (rows) and 10 features (columns): 8 numerical and 2 categorical.
- The target variable (y) is a separate array or Series with 3000 values.

Purpose:

- The dataset is designed to be "difficult" in the context of regression problems.
- The non-linear relationships and noise introduced in the target variable make it challenging for simple models to make accurate predictions. This is a common practice to evaluate the performance of more complex models like deep neural networks.

Dataset generation

```python
def generate_difficult_dataset(n_samples=10000, n_features=20, noise_level=0.5):
    """
    Generates a difficult dataset for deep learning regression.
    Args:
        n_samples: Number of data points to generate.
        n_features: Number of features.
        noise_level: Standard deviation of the Gaussian noise added to the target.
    Returns:
        X: Feature matrix (numpy array).
        y: Target variable (numpy array).
    """
    # Generate random features
    X = np.random.rand(n_samples, n_features)
    # Create non-linear interactions between features
    y = (
        np.sin(X[:, 0] * X[:, 1]) +  # Interaction between feature 0 and 1
        np.cos(X[:, 2] ** 2) +      # Non-linear transformation of feature 2
        X[:, 3] * X[:, 4] -         # Interaction between feature 3 and 4
        X[:, 5] ** 3 +              # Non-linear transformation of feature 5
        np.exp(X[:, 6]) +           # Exponential transformation of feature 6
        np.sum(X[:, 7:10], axis=1) +  # Sum of features 7, 8, and 9
        np.random.randn(n_samples) * noise_level  # Add Gaussian noise
    )
    return X, y
```

Dataset generation

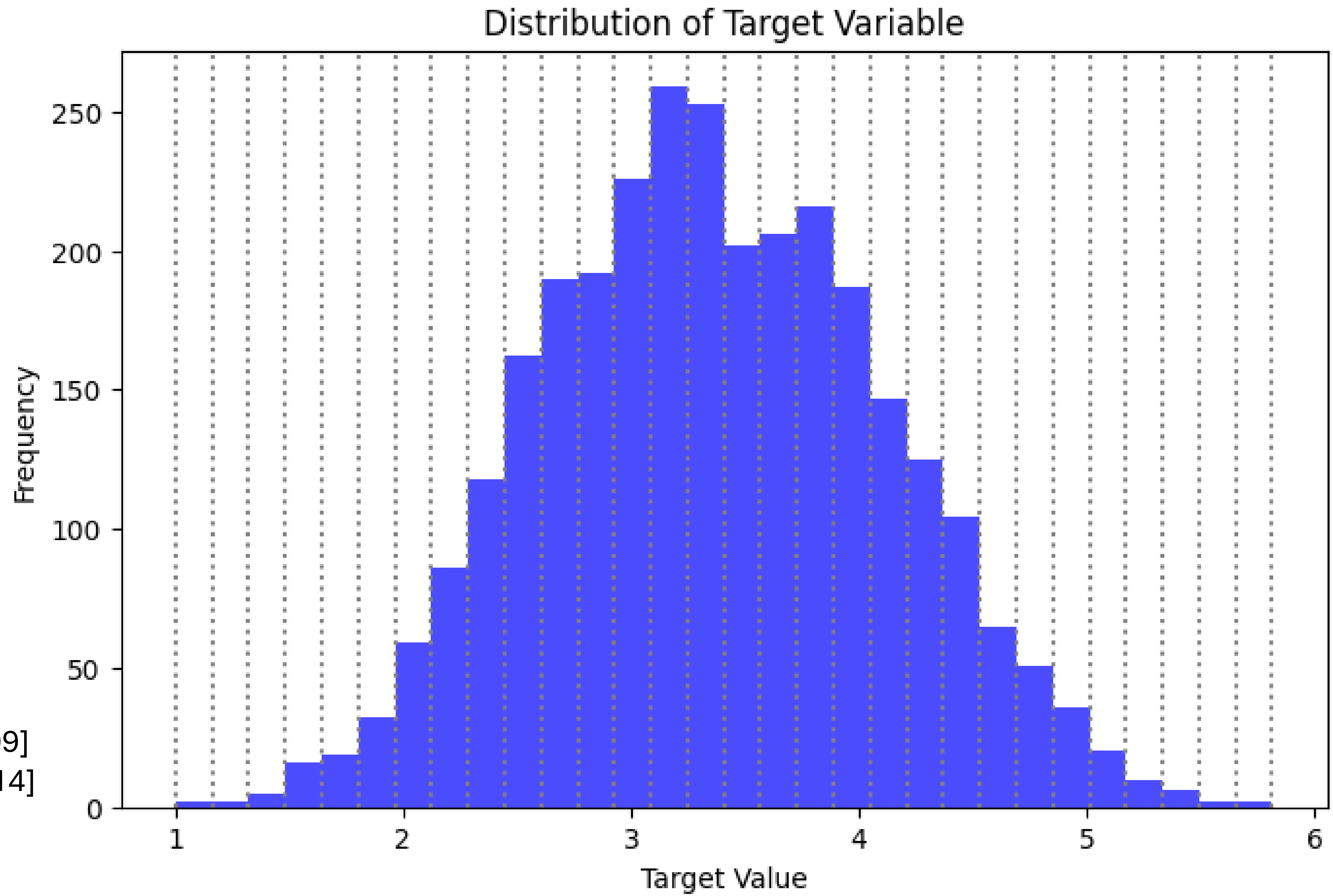
```
# Generate the numeric dataset
num_samples = 3000
num_features = 8
X_numeric, y = generate_difficult_dataset(n_samples=num_samples,
                                          n_features=num_features, noise_level=0.25)
# Add synthetic categorical data
np.random.seed(42)
X_categorical = np.random.choice(['A', 'B', 'C'], size=(num_samples, 2))

# Combine numerical and categorical data into a DataFrame
X = pd.DataFrame(X_numeric, columns=[f"xnum_{i}" for i in range(8)])
X['xcat_1'] = X_categorical[:, 0]
X['xcat_2'] = X_categorical[:, 1]

y = y.reshape(-1, 1)  # Reshape y to match expected input format

print("Shape of X (features):", X.shape)
print("Shape of y (target):", y.shape)

# Create a DataFrame for y
y_df = pd.DataFrame(y, columns=['target'])  # Give y a column name
# Concatenate X and y_df horizontally
combined_df = pd.concat([X, y_df], axis=1)
# Print the combined DataFrame
print(combined_df.to_string())
```

Dataset structure

Shape of X (features): (3000, 10)
Shape of y (target): (3000, 1)

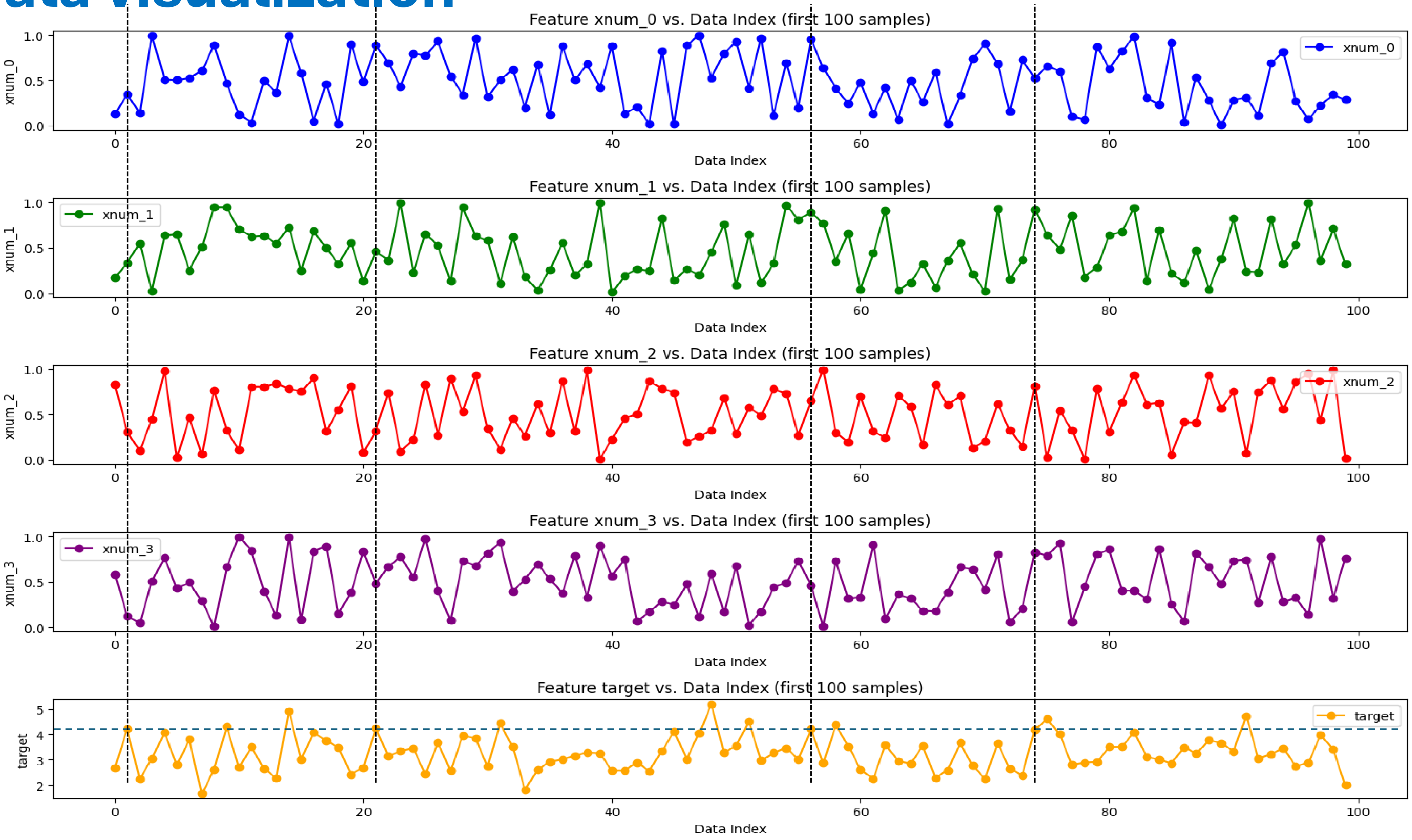
	xnum_0	xnum_1	xnum_2	xnum_3	xnum_4	xnum_5	xnum_6	xnum_7	xcat_1	xcat_2	target
0	0.125566	0.171295	0.827357	0.584839	0.337532	0.802851	0.246520	0.976033	C	A	2.678303
1	0.344818	0.337311	0.311912	0.122112	0.111137	0.648105	0.875118	0.790521	C	C	4.228160
2	0.141108	0.545582	0.105592	0.045295	0.656386	0.535152	0.082024	0.034939	A	A	2.250429
3	0.989756	0.030021	0.444528	0.504619	0.117280	0.169062	0.351805	0.761676	C	B	3.043173
4	0.507082	0.637902	0.985175	0.768005	0.434361	0.446442	0.618772	0.665365	C	C	4.083991
5	0.500711	0.644861	0.023532	0.429623	0.278938	0.129174	0.099547	0.603192	C	C	2.796914
6	0.523601	0.253584	0.470950	0.492232	0.195009	0.652093	0.780919	0.791464	A	C	3.814045
7	0.602470	0.511639	0.061007	0.287831	0.093271	0.963823	0.135080	0.151775	B	A	1.653826
8	0.889577	0.943928	0.765829	0.005405	0.001346	0.378101	0.143395	0.018386	B	B	2.615890
9	0.464067	0.943062	0.323820	0.661141	0.652235	0.251080	0.650425	0.212334	B	B	4.302319
10	0.122007	0.701383	0.113441	0.996909	0.183452	0.594142	0.411375	0.126386	A	A	2.697805

Data visualization



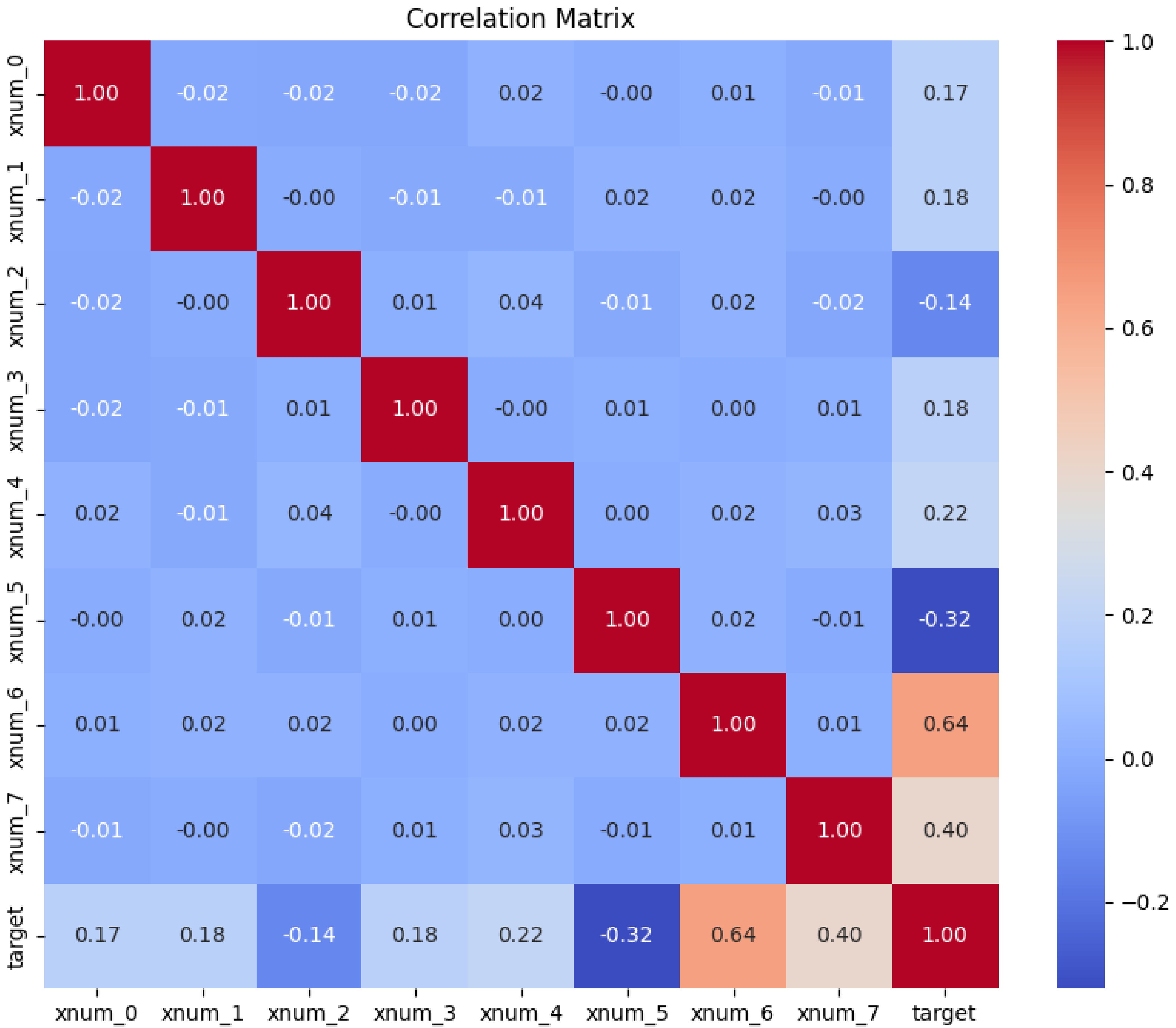
minimum: [1.00076199]
maximum: [5.81300314]
bin size: [0.16040804]

Data visualization



Correlation matrix for numerical features

```
# Step 3: Calculate and visualize the correlation matrix for numerical features
# Add the target variable to the DataFrame for correlation analysis
data = X.copy()
data['target'] = y

# Select only numerical features for correlation analysis
numerical_features = data.select_dtypes(include=np.number).columns

# Calculate the correlation matrix for numerical features only
correlation_matrix = data[numerical_features].corr()

# Visualize the correlation matrix using a heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix')
plt.show()
```

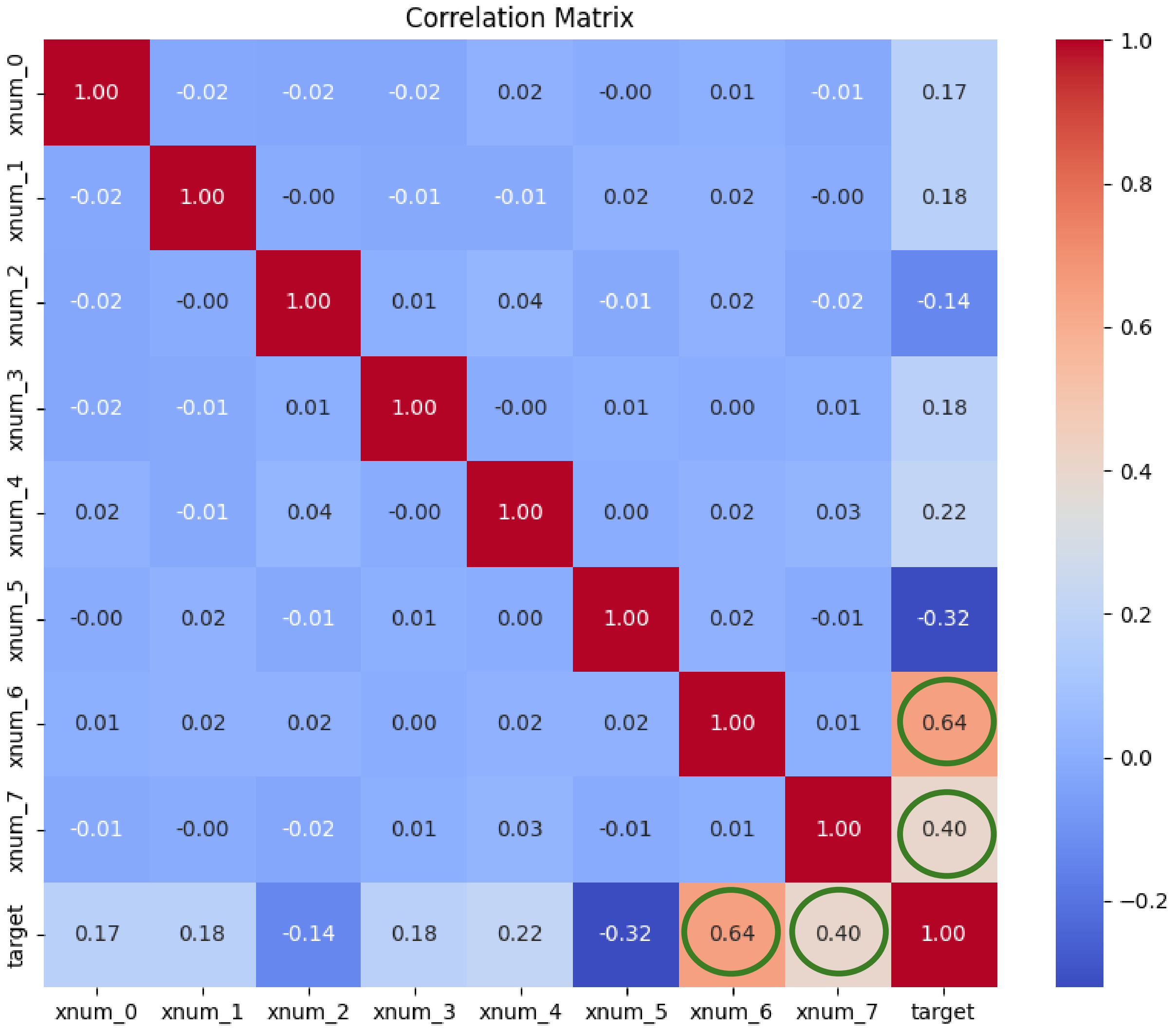


Understand how each feature is related to the target variable and other features.

The correlation matrix is a table that shows the correlation coefficients between multiple variables.

Each cell in the table represents the correlation between two variables.

The correlation coefficient, typically denoted by "r", ranges from -1 to +1.



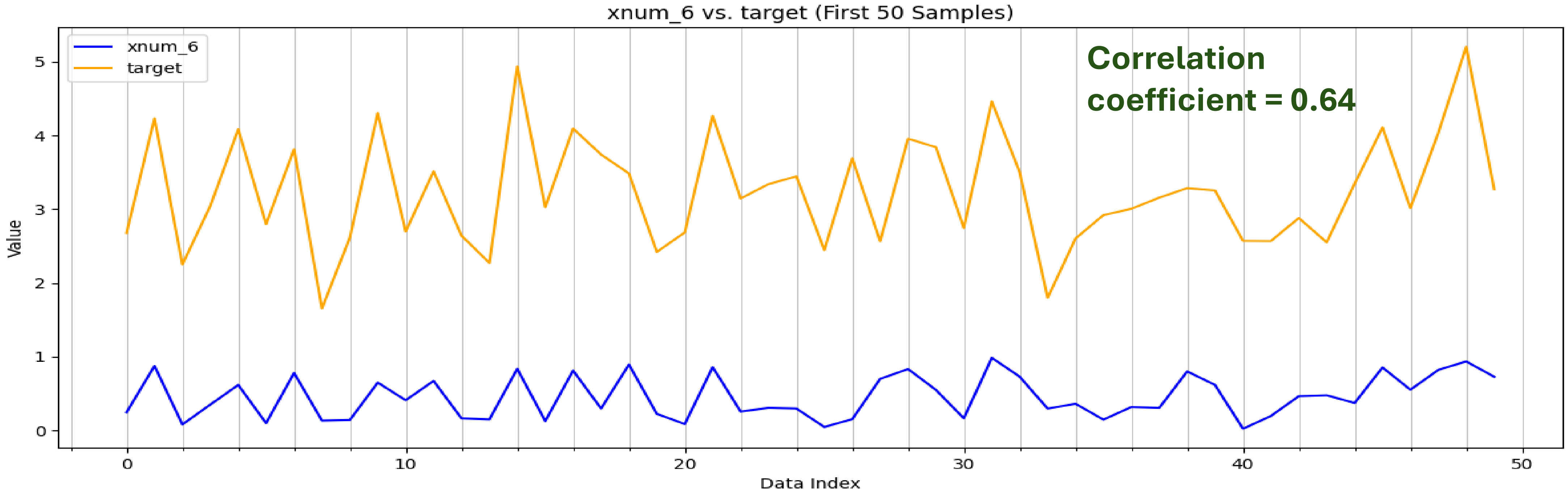
- **Positive Correlation (r > 0):** Indicates that as one variable increases, the other variable tends to increase as well. A value **closer to +1** indicates a **stronger positive correlation**
  - ✓ **important predictor**

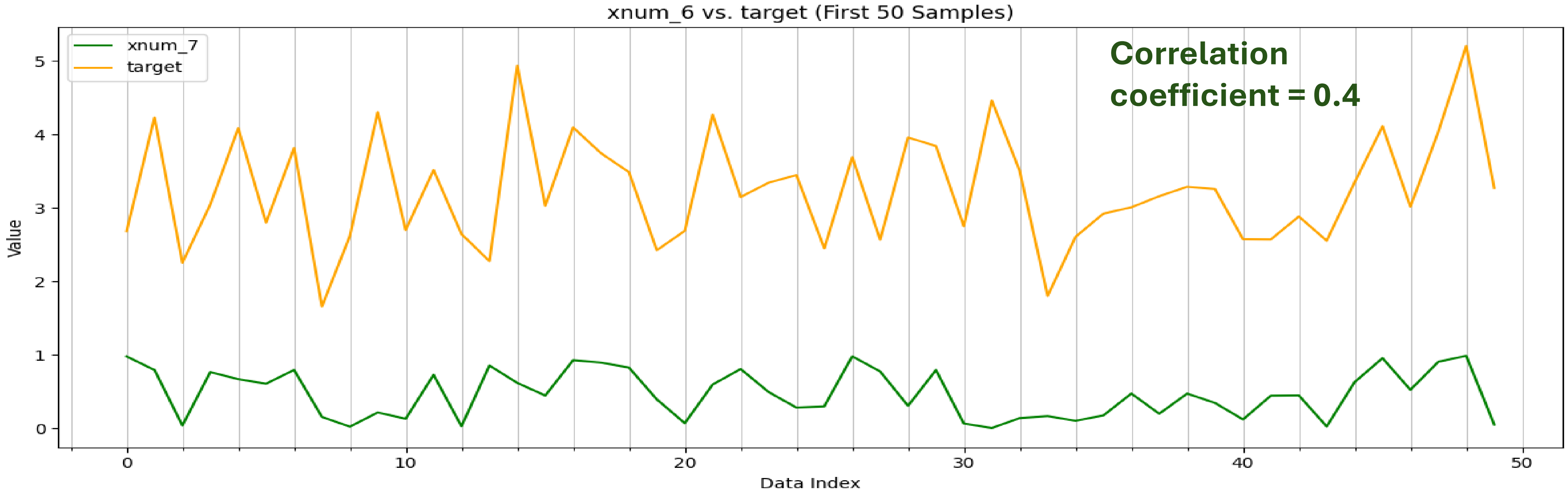
- **Negative Correlation (r < 0):** Indicates that as one variable increases, the other variable tends to decrease. A value **closer to -1** indicates a **stronger negative correlation**
  - ✓ **important predictor**

- **No Correlation (r ≈ 0):** Indicates that there is **no linear relationship** between the two variables. Changes in one variable do not predictably affect the other
  - ➢ **may be relevant, but their influence is less direct**

Check for multicollinearity:

If **two features have a very high correlation with each other** (e.g., above 0.8 or 0.9), it could indicate multicollinearity. This means they **provide similar information, and one of them might be redundant** in your model. - not the case here


xnum_6 vs. target (First 50 Samples)
xnum_6
target
Correlation coefficient = 0.64
Value
Data Index
0
1
2
3
4
5
0
10
20
30
40
50


xnum_6 vs. target (First 50 Samples)
xnum_7
target
Correlation coefficient = 0.4
Value
Data Index
0
1
2
3
4
5
0
10
20
30
40
50

Convert categorical data into a numerical format

Categorical data refers to variables that represent categories or labels rather than numerical values.

- ➢ "color" could have categories like "red" "blue" and "green"
- ➢ or in general one could have categories like "A" "B" and "C"

❑ Machine learning algorithms, however, typically require numerical input to perform computations.

- ✓ Converting categorical data into numerical form allows these algorithms to process and learn from the data.

	xcat_1	xcat_2	target
0	C	A	2.678303
1	C	C	4.228160
2	A	A	2.250429
3	C	B	3.043173
4	C	C	4.083991
5	C	C	2.796914
6	A	C	3.814045
7	B	A	1.653826
8	B	B	2.615890
9	B	B	4.302319

Convert categorical data into a numerical format using **label encoder**

Label Encoding is a technique used to convert categorical data (data that is represented by categories or labels) into numerical data.

This is often necessary because many machine learning algorithms work best with numerical input.

	xcat_1	xcat_2	target
0	C	A	2.678303
1	C	C	4.228160
2	A	A	2.250429
3	C	B	3.043173
4	C	C	4.083991
5	C	C	2.796914
6	A	C	3.814045
7	B	A	1.653826
8	B	B	2.615890
9	B	B	4.302319

A → 0
B → 1
C → 2

	xcat_1	xcat_2	target
0	2	0	2.678303
1	2	2	4.228160
2	0	0	2.250429
3	2	1	3.043173
4	2	2	4.083991
5	2	2	2.796914
6	0	2	3.814045
7	1	0	1.653826
8	1	1	2.615890
9	1	1	4.302319

How it works:

1. Fit: The Label Encoder analyzes the categorical feature (column) to identify all the unique categories or labels present. This is done using the fit method.

2. Transform: Once it has learned the categories, it assigns a unique numerical label to each category.

It **starts from 0 and assigns consecutive integers to each distinct category**. This is done using the transform method.

3. Fit_transform: The fit_transform method is a convenient combination. It performs both steps in a single call.

Convert categorical data into a numerical format using **label encoder**

	xcat_1	xcat_2	target
0	C	A	2.678303
1	C	C	4.228160
2	A	A	2.250429
3	C	B	3.043173
4	C	C	4.083991
5	C	C	2.796914
6	A	C	3.814045
7	B	A	1.653826
8	B	B	2.615890
9	B	B	4.302319

	xcat_1	xcat_2	target
0	2	0	2.678303
1	2	2	4.228160
2	0	0	2.250429
3	2	1	3.043173
4	2	2	4.083991
5	2	2	2.796914
6	0	2	3.814045
7	1	0	1.653826
8	1	1	2.615890
9	1	1	4.302319

- **Simplicity:** It's a simple and straightforward technique to apply.
- **Preserves information:** It preserves the order of categories if there is a natural ordering.

Limitations:

- **Ordinality assumption:** Label Encoding might introduce an ordinal relationship between categories where none exists.
  - For example, assigning 0 to A, 1 to B, 2 to C might imply that B is somehow between A and C which might not be true.
- **Impact on models:** This implied ordinality can mislead some algorithms, especially distance-based algorithms like KNN.

Convert categorical data into a numerical format using **One-Hot Encoding**

- For each unique category in a categorical feature, One-Hot Encoding creates a new binary feature (column).
- If a data point belongs to that category, the corresponding binary feature is set to 1; otherwise, it's set to 0.

	xcat_1	xcat_2	target
0	C	A	2.678303
1	C	C	4.228160
2	A	A	2.250429
3	C	B	3.043173
4	C	C	4.083991
5	C	C	2.796914
6	A	C	3.814045
7	B	A	1.653826
8	B	B	2.615890
9	B	B	4.302319

	xcat_1_A	xcat_1_B	xcat_1_C	xcat_2_A	xcat_2_B	xcat_2_C	target
0	0	0	1	1	0	0	2.678303
1	0	0	1	0	0	1	4.228160
2	1	0	0	1	0	0	2.250429
3	0	0	1	0	1	0	3.043173
4	0	0	1	0	0	1	4.083991
5	0	0	1	0	0	1	2.796914
6	1	0	0	0	0	1	3.814045
7	0	1	0	1	0	0	1.653826
8	0	1	0	0	1	0	2.615890
9	0	1	0	0	1	0	4.302319

Convert categorical data into a numerical format using **One-Hot Encoding**

Benefits of One-Hot Encoding:

- Avoids ordinality: It doesn't introduce any ordinal relationship between categories.
- Suitable for most algorithms: Works well with a wide range of machine learning algorithms.

Considerations:

- Increased dimensionality: Can significantly increase the number of features, especially with high-cardinality categorical features (features with many unique categories). This can lead to increased computational cost and potential overfitting.
- Sparsity: The resulting data can be sparse (lots of zeros), which might require specialized data structures or algorithms.

Convert categorical data into a numerical format using **One-Hot Encoding**

Collinearity issue

Collinearity, or multicollinearity, occurs when two or more predictor variables (features) in a regression model are highly correlated with each other.

This can cause problems in the model, such as:

- Unstable coefficients: The estimated coefficients of the collinear variables can become unstable and vary significantly with small changes in the data.
- Reduced interpretability: It becomes difficult to interpret the individual effects of collinear variables on the target variable.
- Inflated standard errors: The standard errors of the coefficients can increase, making it harder to determine statistical significance.

Convert categorical data into a numerical format using **One-Hot Encoding**

Addressing Collinearity in One-Hot Encoded Data

One-Hot encoding can introduce perfect collinearity, also known as the "dummy variable trap."

This happens because the created dummy variables are perfectly linearly dependent.

If you have three categories (A, B, C) and you create three dummy variables (A, B, C), then knowing the values of two of the dummies automatically determines the value of the third.

To avoid this, you can **drop one of the dummy variables for each categorical feature**. This is called **dummy variable dropping** and is a common practice to address collinearity in One-Hot encoded data.

Convert categorical data into a numerical format using **One-Hot Encoding**

```python
# Perform One-Hot Encoding without colinearity
onehot_df = pd.get_dummies(
    subset_df, columns=['xcat_1', 'xcat_2'],
    drop_first=True,
    prefix=['xcat_1', 'xcat_2'],
    dtype=int)
```

	xcat_1	xcat_2	target
0	C	A	2.678303
1	C	C	4.228160
2	A	A	2.250429
3	C	B	3.043173
4	C	C	4.083991
5	C	C	2.796914
6	A	C	3.814045
7	B	A	1.653826
8	B	B	2.615890
9	B	B	4.302319

	xcat_1_B	xcat_1_C	xcat_2_B	xcat_2_C	target
0	0	1	0	0	2.678303
1	0	1	0	1	4.228160
2	0	0	0	0	2.250429
3	0	1	1	0	3.043173
4	0	1	0	1	4.083991
5	0	1	0	1	2.796914
6	0	0	0	1	3.814045
7	1	0	0	0	1.653826
8	1	0	1	0	2.615890
9	1	0	1	0	4.302319

Encoded Dataset

Shape of X (features): (3000, 12)
Shape of y (target): (3000, 1)

	xnum_0	xnum_1	xnum_2	xnum_3	xnum_4	xnum_5	xnum_6	xnum_7	xcat_1_B	xcat_1_C	xcat_2_B	xcat_2_C	target
0	0.125566	0.171295	0.827357	0.584839	0.337532	0.802851	0.246520	0.976033	0	1	0	0	2.678303
1	0.344818	0.337311	0.311912	0.122112	0.111137	0.648105	0.875118	0.790521	0	1	0	1	4.228160
2	0.141108	0.545582	0.105592	0.045295	0.656386	0.535152	0.082024	0.034939	0	0	0	0	2.250429
3	0.989756	0.030021	0.444528	0.504619	0.117280	0.169062	0.351805	0.761676	0	1	1	0	3.043173
4	0.507082	0.637902	0.985175	0.768005	0.434361	0.446442	0.618772	0.665365	0	1	0	1	4.083991
5	0.500711	0.644861	0.023532	0.429623	0.278938	0.129174	0.099547	0.603192	0	1	0	1	2.796914
6	0.523601	0.253584	0.470950	0.492232	0.195009	0.652093	0.780919	0.791464	0	0	0	1	3.814045
7	0.602470	0.511639	0.061007	0.287831	0.093271	0.963823	0.135080	0.151775	1	0	0	0	1.653826
8	0.889577	0.943928	0.765829	0.005405	0.001346	0.378101	0.143395	0.018386	1	0	1	0	2.615890
9	0.464067	0.943062	0.323820	0.661141	0.652235	0.251080	0.650425	0.212334	1	0	1	0	4.302319
10	0.122007	0.701383	0.113441	0.996909	0.183452	0.594142	0.411375	0.126386	0	0	0	0	2.697805

Data set split

Train set: used to learn the parameters of the model

Val set (validation set): supervises the learning generality (identify overfitting);

Test set: used as a proxy for unseen data and evaluate our model on test-set (brand-new data set)

Now we will **split** the full data set in 2 subsets:

- **test_set 20%** of the entire data set (**600** examples)
- train_set 80% of the entire data set
  - Later on, when we will use .fit method to train our ANN model; from the train_set
    - 22.22% (**30** examples) will be used for **validation**
    - 77.78% (**105** examples) for real model **trainig**

Data set split

```
# Step 5: Split data into training and test sets
from sklearn.model_selection import train_test_split # Import the train_test_split function
X_train, X_test, y_train, y_test = train_test_split(X_encoded, y, test_size=0.2, random_state=42)
```

Shape of X_train: (2400, 12)
Shape of X_test: (600, 12)
Shape of y_train: (2400, 1)
Shape of y_test: (600, 1)

Normalize dataset

Both sets
Train set
Test set

- Numerical features
- Targets
- Encoded features are not normalized

```python
# Step 6: Standardize the numerical data
from sklearn.preprocessing import StandardScaler # Import the StandardScaler class

scaler_X = StandardScaler()
scaler_y = StandardScaler()

# Standardize only numerical columns for train set
X_train.iloc[:, :8] = scaler_X.fit_transform(X_train.iloc[:, :8])
# Standardize only numerical columns for test set
X_test.iloc[:, :8] = scaler_X.transform(X_test.iloc[:, :8])

y_train = scaler_y.fit_transform(y_train)  # Standardize target for train test
y_test = scaler_y.transform(y_test)  # Standardize target for test test

# In pandas DataFrames, iloc is primarily used for integer-location based indexing.
# It allows to select rows and columns from a DataFrame using their numerical positions (indices)
```

no fit !

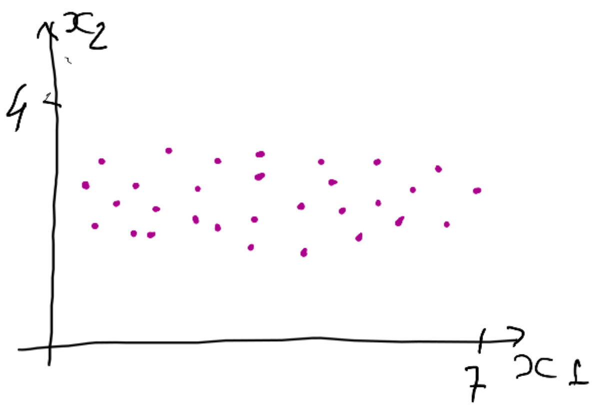
Standard Scaler

$$x\_scaled = \frac{x - x\_mean}{standard\_deviation}$$

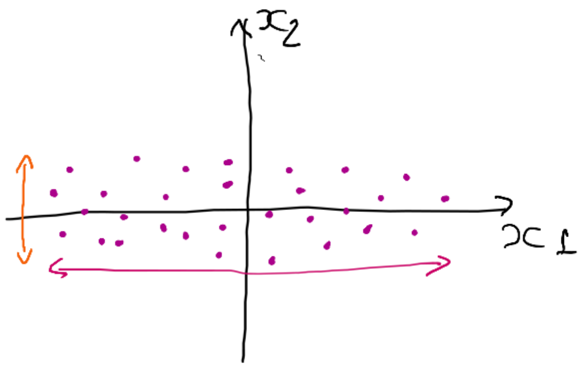
Applied separately on each data feature

Use the same μ, σ to normalize all data sets

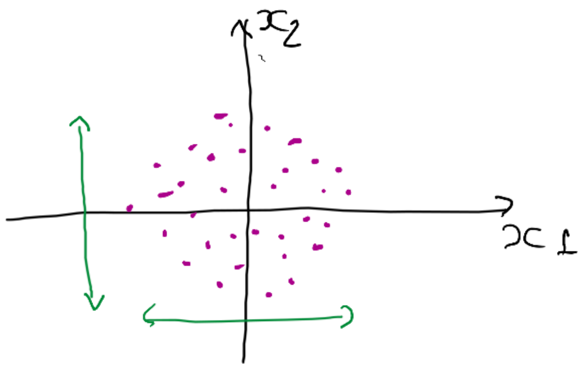
- ✓ **Training**
- ✓ **Validation**
- ✓ **Test**



Initial dataset



Subtract mean (zero out the mean)



Normalize the variance

Standardizes features by removing the mean and scaling to unit variance.

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} ; \quad \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$$

$$x := x - \mu$$

μ – mean

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)}\right)^2$$

σ^2 – variance

$$x := \frac{x}{\sigma}$$

σ – standard deviation $\quad \sigma = \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix}$

1. **Centering:** The mean of the feature is subtracted from each feature value (x). This shifts the distribution of the feature so that its mean becomes 0.
2. **Scaling:** Each centered feature value is then divided by the standard deviation. This scales the distribution so that its variance becomes 1.

ANN model Secvential

```python
# Step 7: Build the deep neural network model - no Dropout here
import tensorflow
from tensorflow import keras

model = keras.Sequential([
    keras.layers.Input(shape=(X_train.shape[1],), name="Input"),
    keras.layers.Dense(units=64, activation='relu', name="hidden_layer_1"),
    # keras.layers.Dropout(rate =0.1, name = "dropout_1"),
            # Drop 10% of the neurons in this layer during training
    keras.layers.Dense(units=128, activation='relu', name="hidden_layer_2"),
    # keras.layers.Dropout(rate =0.1, name = "dropout_2"),
            # Drop 10% of the neurons in this layer during training
    keras.layers.Dense(64,'relu', name="hidden_layer_3"),
    # keras.layers.Dropout(0.2, name = "dropout_3"),
            # Drop 20% of the neurons in this layer during training
    keras.layers.Dense(1, activation=None, name = 'output')
            # Output layer for regression (1 node, no activation function)
], name = "Regression_ANN")
```

ANN model summary and diagram

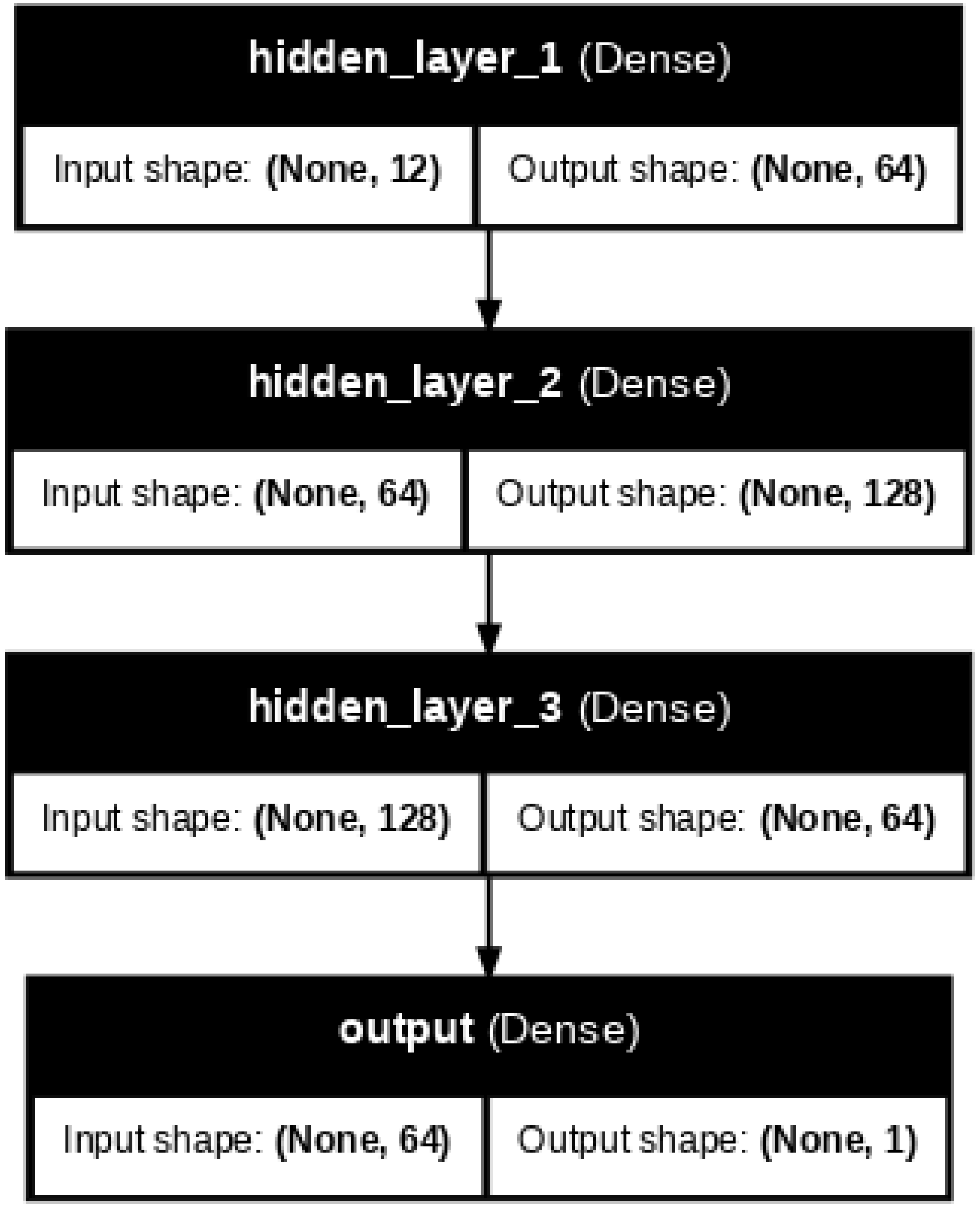
```python
model.summary()
keras.utils.plot_model(model, to_file='model_diagram.png',
                       show_shapes=True, show_layer_names=True,
                       dpi=64, rankdir='TB')  # Adjust dpi and rankdir
```

Model: "Regression_ANN"

Layer (type)	Output Shape	Param #
hidden_layer_1 (Dense)	(None, 64)	832
hidden_layer_2 (Dense)	(None, 128)	8,320
hidden_layer_3 (Dense)	(None, 64)	8,256
output (Dense)	(None, 1)	65

Total params: 17,473 (68.25 KB)
Trainable params: 17,473 (68.25 KB)
Non-trainable params: 0 (0.00 B)

ANN model summary and diagram



Early stopping

A regularization technique used in deep learning to prevent overfitting. Overfitting occurs when a model learns the training data too well, including its noise and random fluctuations, resulting in poor performance on unseen data.

How it Works:

1. Validation Set: A portion of the training data is set aside as a validation set. This set is not used to train the model directly.

2. Monitoring: During training, the model's performance (e.g., loss or accuracy) is evaluated on both the training set and the validation set.

3. Stopping Criteria: If the model's performance on the validation set starts to worsen (e.g., validation loss increases or validation accuracy decreases) while the performance on the training set continues to improve, it indicates that the model is starting to overfit.

4. Early Stop: Training is stopped before the model has a chance to fully overfit the training data. The model parameters from the epoch with the best performance on the validation set are saved and used as the final model.

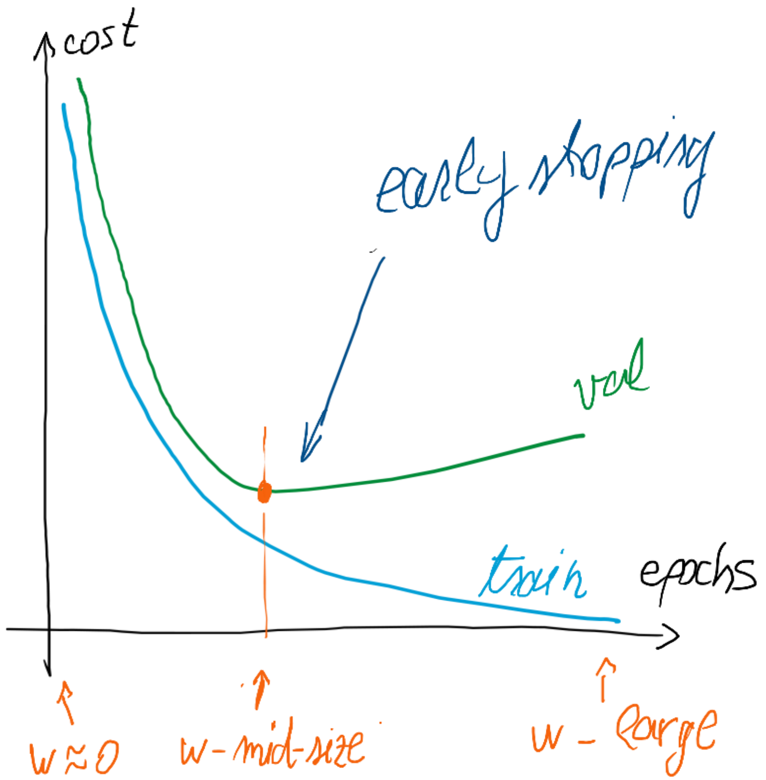
Early stopping

Benefits of Early Stopping:

•**Prevents Overfitting:** Stops training before the model overfits, leading to better generalization on unseen data.

•**Saves Time and Resources:** Reduces unnecessary training time and computational resources by stopping training when further improvements are unlikely.

•**Improves Model Performance:** Can lead to a more robust and accurate model by selecting the best performing model during training.




```python
# Step 8: Introduce Early Stopping
import tensorflow
from tensorflow import keras

early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',  # Monitor validation loss
    patience=10,         # Stop if no improvement for 10 epochs
    restore_best_weights=True  # Restore weights from the best epoch
)
```

Configure (compile) the ANN - loss function

```
# Configure (compile) the model
import tensorflow
from tensorflow import keras

model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.0002),
              loss=keras.losses.MeanSquaredError(),
              metrics=['mean_squared_error'])
```

$$\text{mean\_squared\_error} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

Train the ANN model

```
# Step 9: Train the model
history = model.fit(
    X_train, y_train,
    validation_split=0.25, # used as data validation set
    epochs=500,
    batch_size=1024,
    # callbacks=[early_stopping],  # Use EarlyStopping callback
    verbose=1
)
```

Early stopping is not used for now

$$2400 \cdot 0.25 = 600 \text{ examples for validation}$$
$$2400 - 600 = 1800\text{examples for training}$$

Train the ANN model

Beginning of training

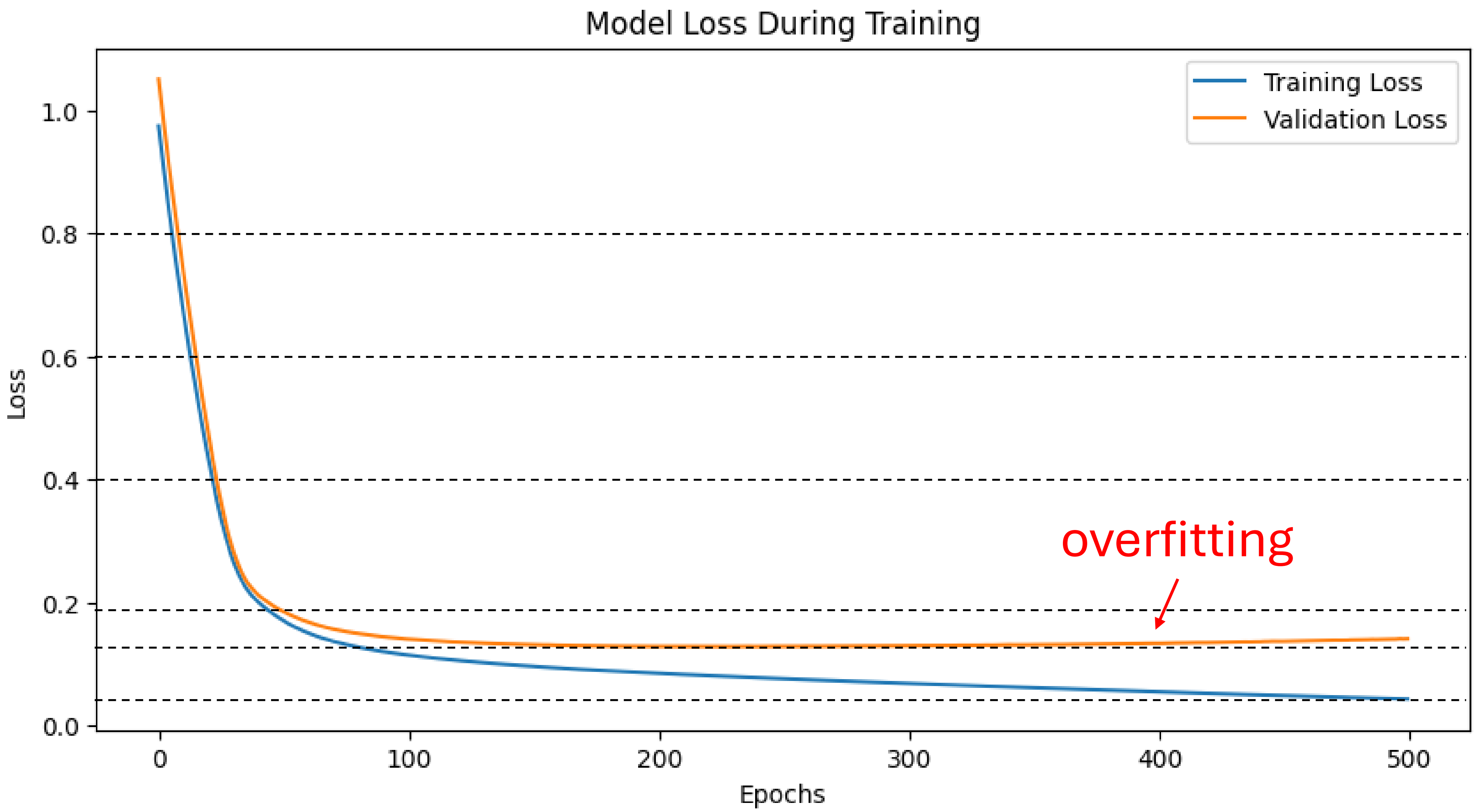
```
Epoch 1/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 266ms/step - loss: 0.9804 - mean_squared_error: 0.9804 - val_loss: 1.0506 - val_mean_squared_error: 1.0506
Epoch 2/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - loss: 0.9509 - mean_squared_error: 0.9509 - val_loss: 1.0148 - val_mean_squared_error: 1.0148
Epoch 3/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - loss: 0.9113 - mean_squared_error: 0.9113 - val_loss: 0.9799 - val_mean_squared_error: 0.9799
Epoch 4/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - loss: 0.8678 - mean_squared_error: 0.8678 - val_loss: 0.9462 - val_mean_squared_error: 0.9462
Epoch 5/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - loss: 0.8449 - mean_squared_error: 0.8449 - val_loss: 0.9136 - val_mean_squared_error: 0.9136
Epoch 6/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - loss: 0.8248 - mean_squared_error: 0.8248 - val_loss: 0.8819 - val_mean_squared_error: 0.8819
Epoch 7/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - loss: 0.7687 - mean_squared_error: 0.7687 - val_loss: 0.8511 - val_mean_squared_error: 0.8511
```

Train the ANN model

End of training

```
Epoch 491/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - loss: 0.0435 - mean_squared_error: 0.0435 - val_loss: 0.1398 - val_mean_squared_error: 0.1398
Epoch 492/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step - loss: 0.0442 - mean_squared_error: 0.0442 - val_loss: 0.1399 - val_mean_squared_error: 0.1399
Epoch 493/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step - loss: 0.0442 - mean_squared_error: 0.0442 - val_loss: 0.1403 - val_mean_squared_error: 0.1403
Epoch 494/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 44ms/step - loss: 0.0424 - mean_squared_error: 0.0424 - val_loss: 0.1401 - val_mean_squared_error: 0.1401
Epoch 495/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - loss: 0.0437 - mean_squared_error: 0.0437 - val_loss: 0.1401 - val_mean_squared_error: 0.1401
Epoch 496/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - loss: 0.0424 - mean_squared_error: 0.0424 - val_loss: 0.1404 - val_mean_squared_error: 0.1404
Epoch 497/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step - loss: 0.0434 - mean_squared_error: 0.0434 - val_loss: 0.1409 - val_mean_squared_error: 0.1409
Epoch 498/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step - loss: 0.0434 - mean_squared_error: 0.0434 - val_loss: 0.1405 - val_mean_squared_error: 0.1405
Epoch 499/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step - loss: 0.0426 - mean_squared_error: 0.0426 - val_loss: 0.1406 - val_mean_squared_error: 0.1406
Epoch 500/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step - loss: 0.0423 - mean_squared_error: 0.0423 - val_loss: 0.1408 - val_mean_squared_error: 0.1408
```

Analyze the learning progress



Evaluate the ANN model

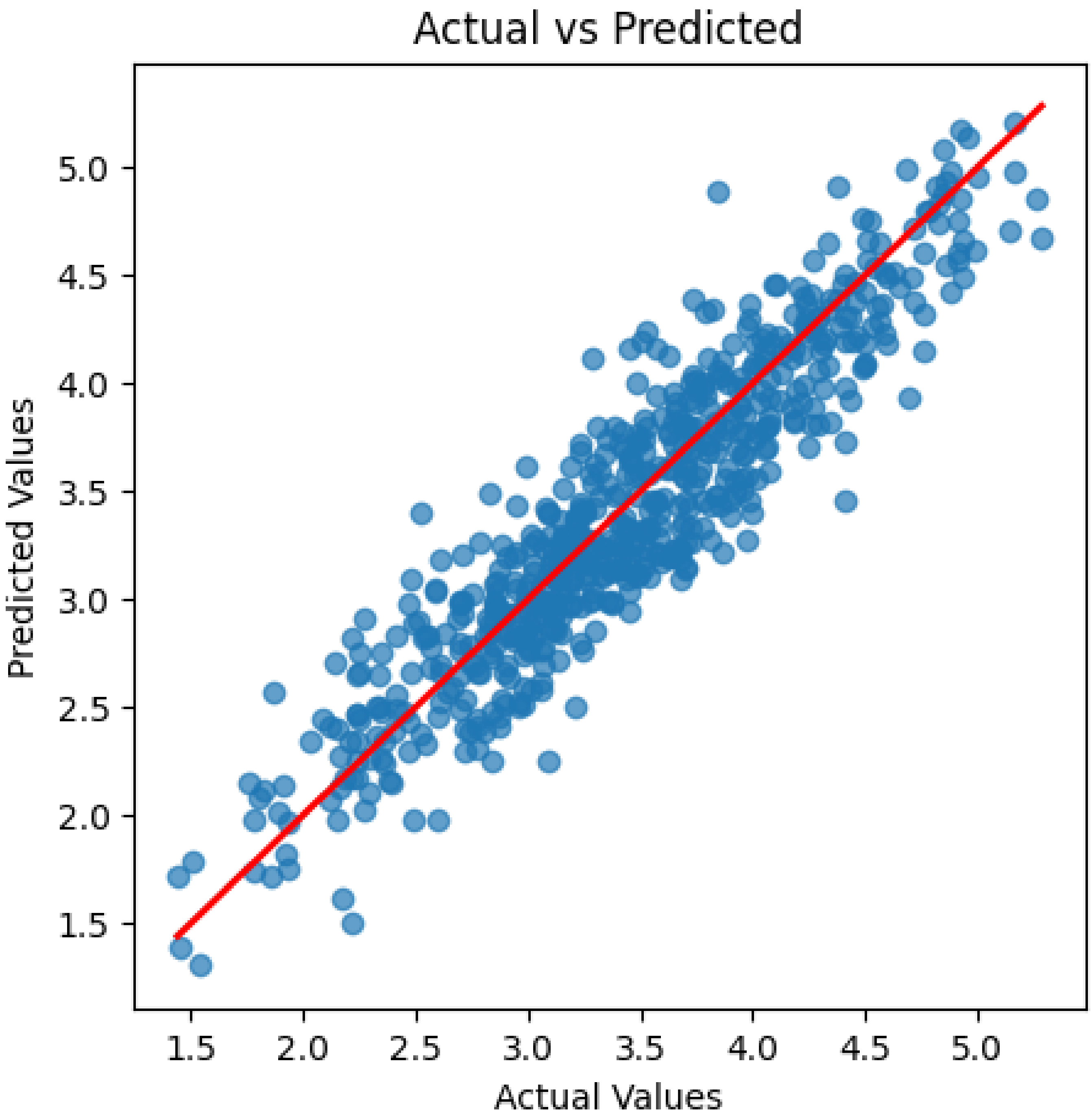
```
------------------------------
Model Evaluation Metrics:
------------------------------
Metric          Train      Test
------------------------------
MSE             0.0382     0.0811
R-squared       0.9329     0.8679
------------------------------
```

```python
from sklearn.metrics import r2_score, mean_squared_error

def display_scores(model, X_train, X_test, y_train, y_test, scaler_y):
    """
    Calculates and displays MSE and R-squared scores for train and test sets.
    Args:
        model: The trained Keras model.
        X_train: Training data features;  X_test: Test data features.
        y_train: Training data target;    y_test: Test data target.
        scaler_y: The StandardScaler object used for target variable scaling.
    """
    # Get predictions for train and test sets
    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)
    # Inverse transform to get actual values
    y_train_actual = scaler_y.inverse_transform(y_train)
    y_test_actual = scaler_y.inverse_transform(y_test)
    y_train_pred_actual = scaler_y.inverse_transform(y_train_pred)
    y_test_pred_actual = scaler_y.inverse_transform(y_test_pred)
    # Calculate MSE and R-squared
    train_mse = mean_squared_error(y_train_actual, y_train_pred_actual)
    test_mse = mean_squared_error(y_test_actual, y_test_pred_actual)
    train_r2 = r2_score(y_train_actual, y_train_pred_actual)
    test_r2 = r2_score(y_test_actual, y_test_pred_actual)
    # Display the scores in a formatted table
    print("-" * 30)
    print("Model Evaluation Metrics:")
    print("-" * 30)
    print(f"{'Metric':<15} {'Train':<10} {'Test':<10}")
    print("-" * 30)
    print(f"{'MSE':<15} {train_mse:<10.4f} {test_mse:<10.4f}")
    print(f"{'R-squared':<15} {train_r2:<10.4f} {test_r2:<10.4f}")
    print("-" * 30)

display_scores(model, X_train, X_test, y_train, y_test, scaler_y)
```

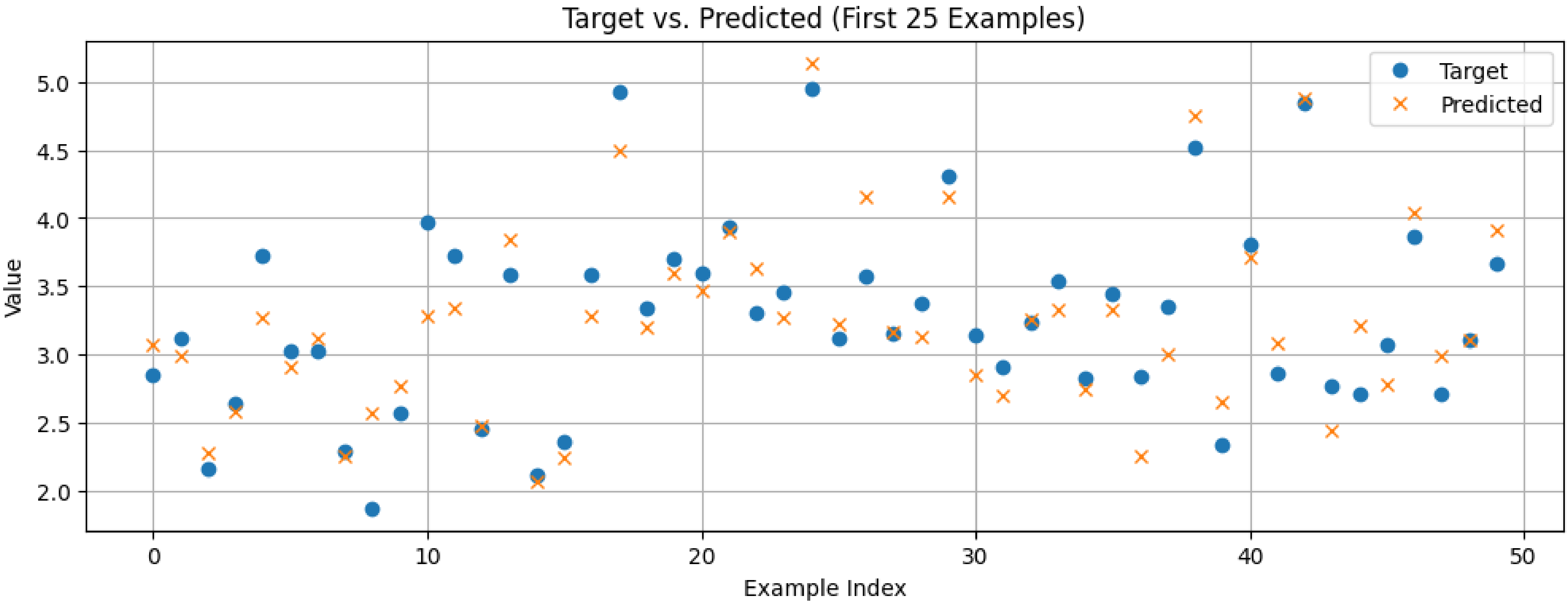
Target vs predicted for test set

```
# Step 12: Make predictions
y_pred = model.predict(X_test)

# Convert predictions back to the original scale
y_pred_original = scaler_y.inverse_transform(y_pred)
y_test_original = scaler_y.inverse_transform(y_test)

# Plot actual vs predicted values
plt.figure(figsize=(10, 5))
plt.scatter(y_test_original, y_pred_original, alpha=0.7)
plt.title('Actual vs Predicted')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.plot([min(y_test_original), max(y_test_original)],
         [min(y_test_original), max(y_test_original)],
         color='red', linewidth=2)  # Reference line
plt.show()
```

Target vs predicted for test set



Target vs predicted for test set



ANN model with Dropout regularization

```
# Step 7: Build the deep neural network model - with Dropout
import tensorflow
from tensorflow import keras

model = keras.Sequential([
    keras.layers.Input(shape=(X_train.shape[1],), name="Input"),
    keras.layers.Dense(units=64, activation='relu', name="hidden_layer_1"),
    keras.layers.Dropout(rate =0.1, name = "dropout_1"),
            # Drop 10% of the neurons in this layer during training
    keras.layers.Dense(units=128, activation='relu', name="hidden_layer_2"),
    keras.layers.Dropout(rate =0.1, name = "dropout_2"),
            # Drop 10% of the neurons in this layer during training
    keras.layers.Dense(64,'relu', name="hidden_layer_3"),
    keras.layers.Dropout(0.2, name = "dropout_3"),
            # Drop 20% of the neurons in this layer during training
    keras.layers.Dense(1, activation=None, name = 'output')
            # Output layer for regression (1 node, no activation function)
], name = "Regression_ANN")
```

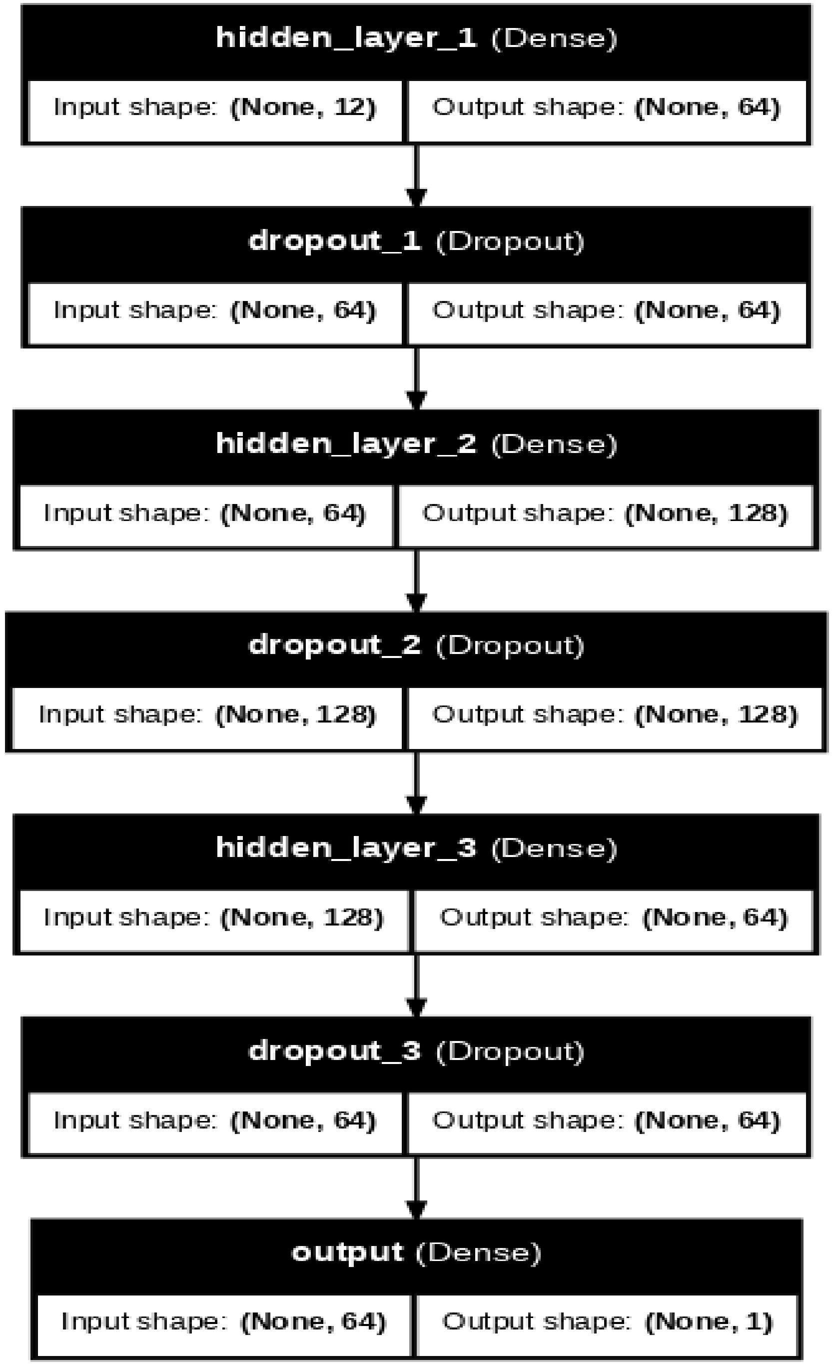
ANN model with Dropout regularization

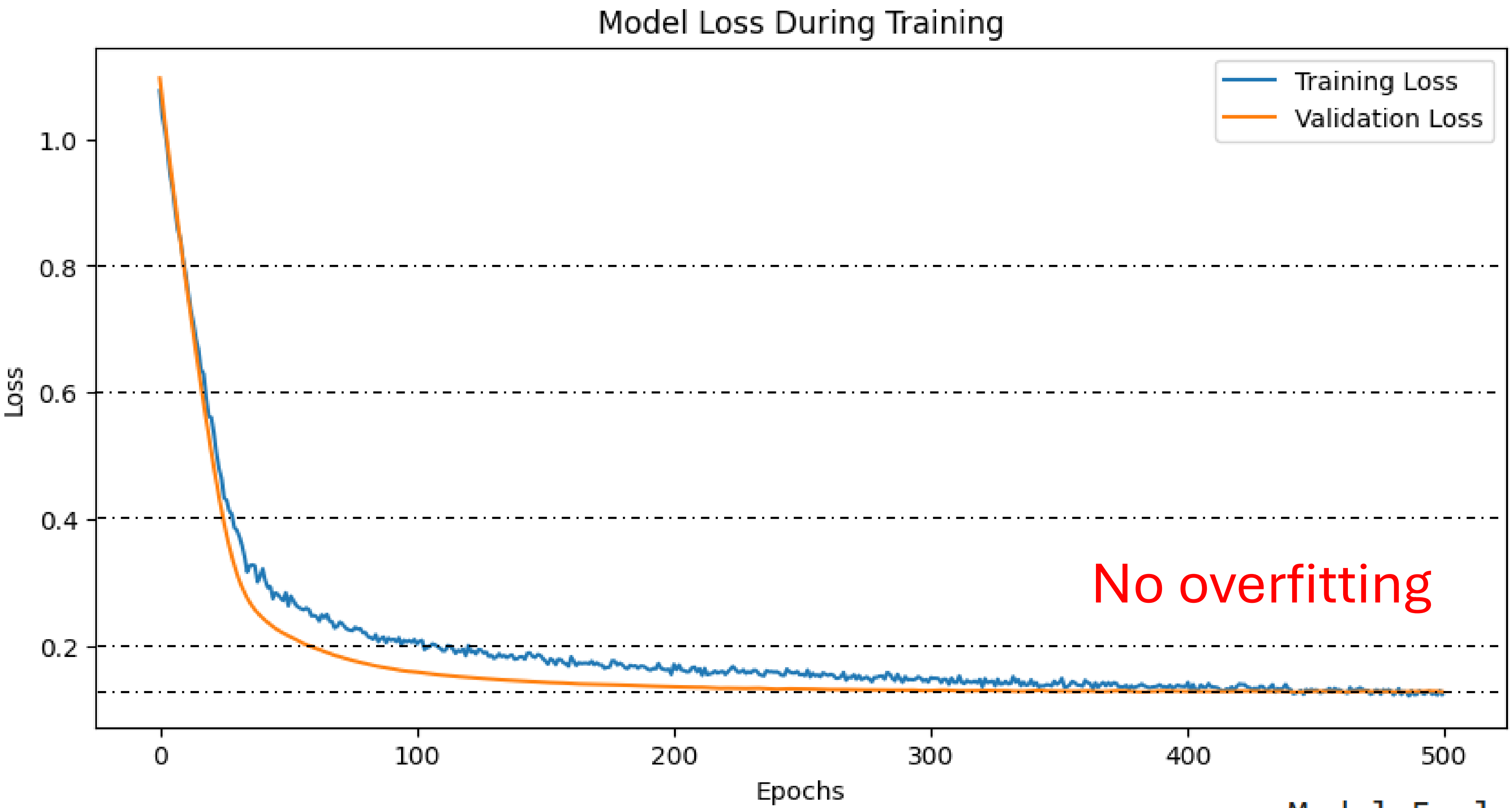
Model: "Regression_ANN"

Layer (type)	Output Shape	Param #
hidden_layer_1 (Dense)	(None, 64)	832
dropout_1 (Dropout)	(None, 64)	0
hidden_layer_2 (Dense)	(None, 128)	8,320
dropout_2 (Dropout)	(None, 128)	0
hidden_layer_3 (Dense)	(None, 64)	8,256
dropout_3 (Dropout)	(None, 64)	0
output (Dense)	(None, 1)	65

Total params: 17,473 (68.25 KB)
Trainable params: 17,473 (68.25 KB)
Non-trainable params: 0 (0.00 B)

ANN model with Dropout regularization





ANN model with Dropout regularization

```
Model Evaluation Metrics:
------------------------------
Metric          Train     Test
------------------------------
MSE             0.0558    0.0721
R-squared       0.9017    0.8765
------------------------------
```

ANN model with Early stopping regularization

```
# Step 8: Introduce Early Stopping
import tensorflow
from tensorflow import keras

early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',   # Monitor validation loss
    patience=10,          # Stop if no improvement for 10 epochs
    restore_best_weights=True  # Restore weights from the best epoch
)

```

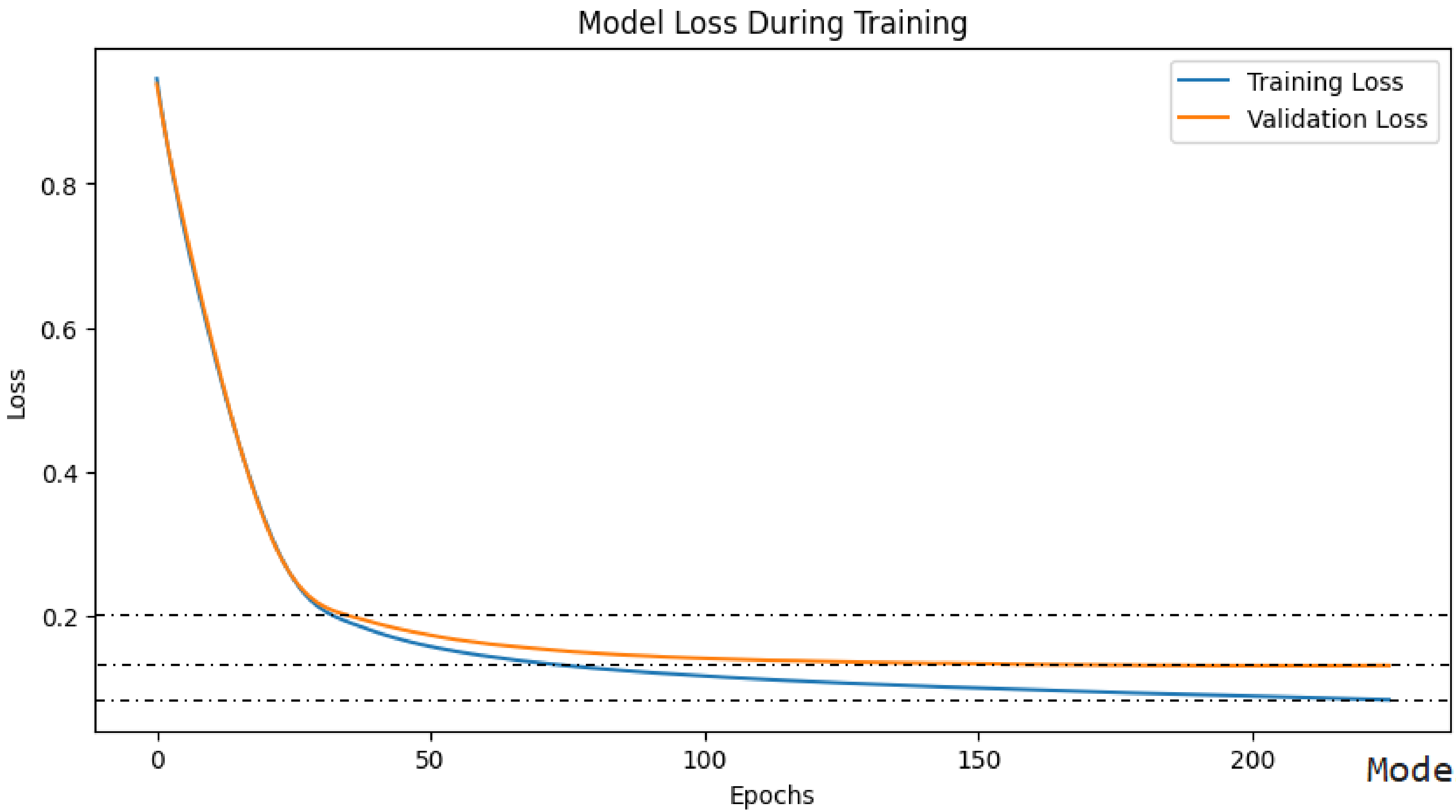
```

# Step 9: Train the model
history = model.fit(
    X_train, y_train,
    validation_split=0.25, # used as data validation set
    epochs=500,
    batch_size=1024,
    callbacks=[early_stopping],  # Use EarlyStopping callback
    verbose=1
)
```

ANN model with Early stopping regularization

```
Epoch 216/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 45ms/step - loss: 0.0870 - mean_squared_error: 0.0870 - val_loss: 0.1317 - val_mean_squared_error: 0.1317
Epoch 217/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 45ms/step - loss: 0.0846 - mean_squared_error: 0.0846 - val_loss: 0.1318 - val_mean_squared_error: 0.1318
Epoch 218/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - loss: 0.0867 - mean_squared_error: 0.0867 - val_loss: 0.1319 - val_mean_squared_error: 0.1319
Epoch 219/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - loss: 0.0857 - mean_squared_error: 0.0857 - val_loss: 0.1320 - val_mean_squared_error: 0.1320
Epoch 220/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step - loss: 0.0858 - mean_squared_error: 0.0858 - val_loss: 0.1319 - val_mean_squared_error: 0.1319
Epoch 221/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - loss: 0.0837 - mean_squared_error: 0.0837 - val_loss: 0.1319 - val_mean_squared_error: 0.1319
Epoch 222/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step - loss: 0.0856 - mean_squared_error: 0.0856 - val_loss: 0.1319 - val_mean_squared_error: 0.1319
Epoch 223/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - loss: 0.0839 - mean_squared_error: 0.0839 - val_loss: 0.1321 - val_mean_squared_error: 0.1321
Epoch 224/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - loss: 0.0851 - mean_squared_error: 0.0851 - val_loss: 0.1321 - val_mean_squared_error: 0.1321
Epoch 225/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step - loss: 0.0844 - mean_squared_error: 0.0844 - val_loss: 0.1320 - val_mean_squared_error: 0.1320
Epoch 226/500
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step - loss: 0.0843 - mean_squared_error: 0.0843 - val_loss: 0.1319 - val_mean_squared_error: 0.1319
```

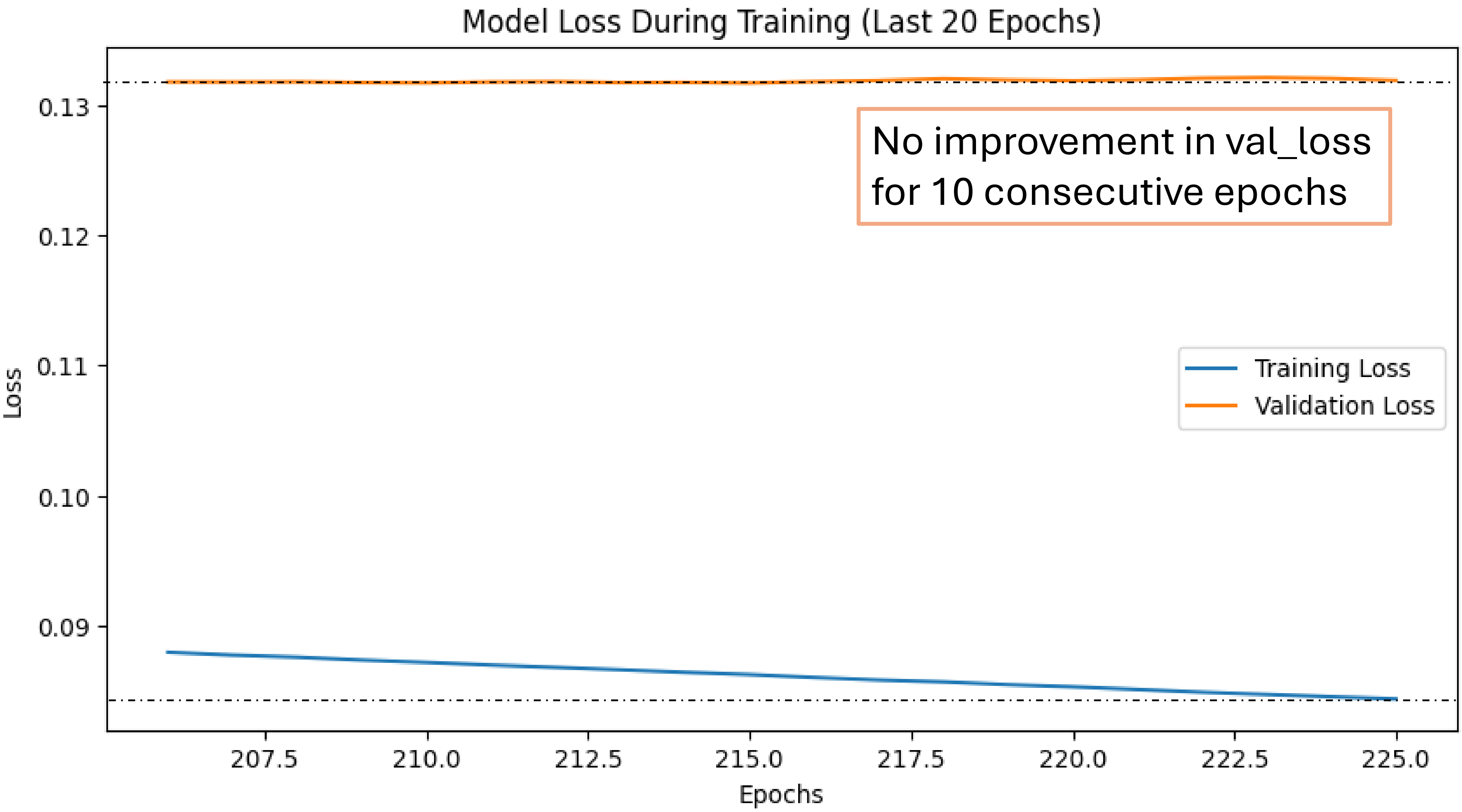
No improvement in val_loss for 10 consecutive epochs

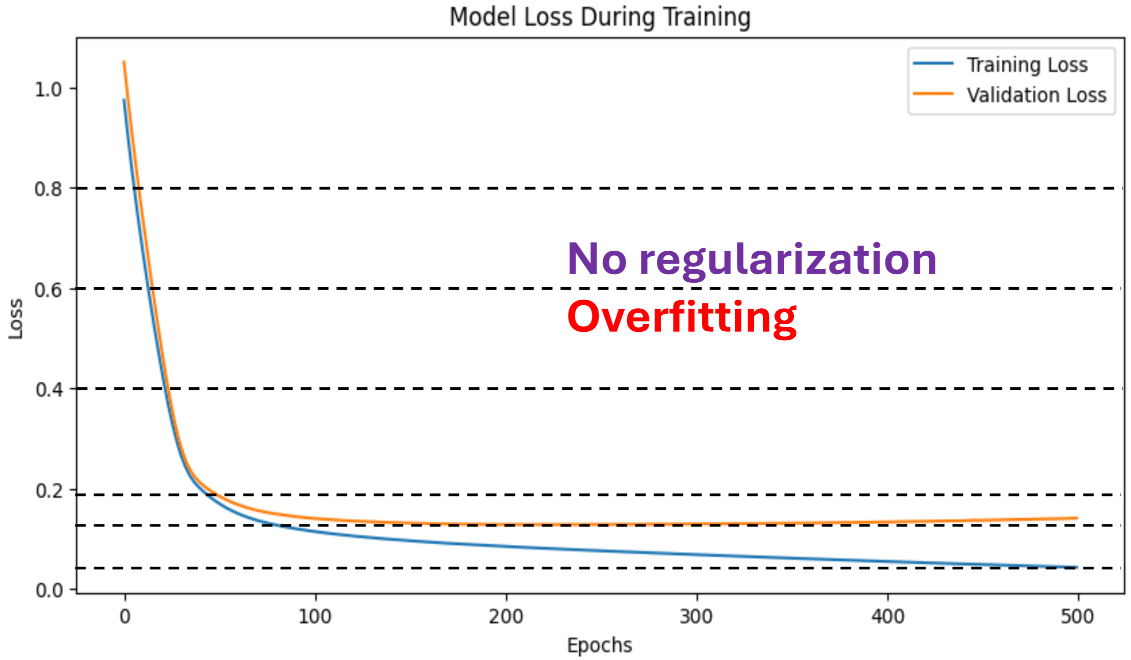
ANN model with Early stopping regularization




```
Model Evaluation Metrics:
------------------------------
Metric              Train      Test
------------------------------
MSE                 0.0553     0.0760
R-squared           0.9026     0.8697
------------------------------
```

ANN model with Early stopping regularization



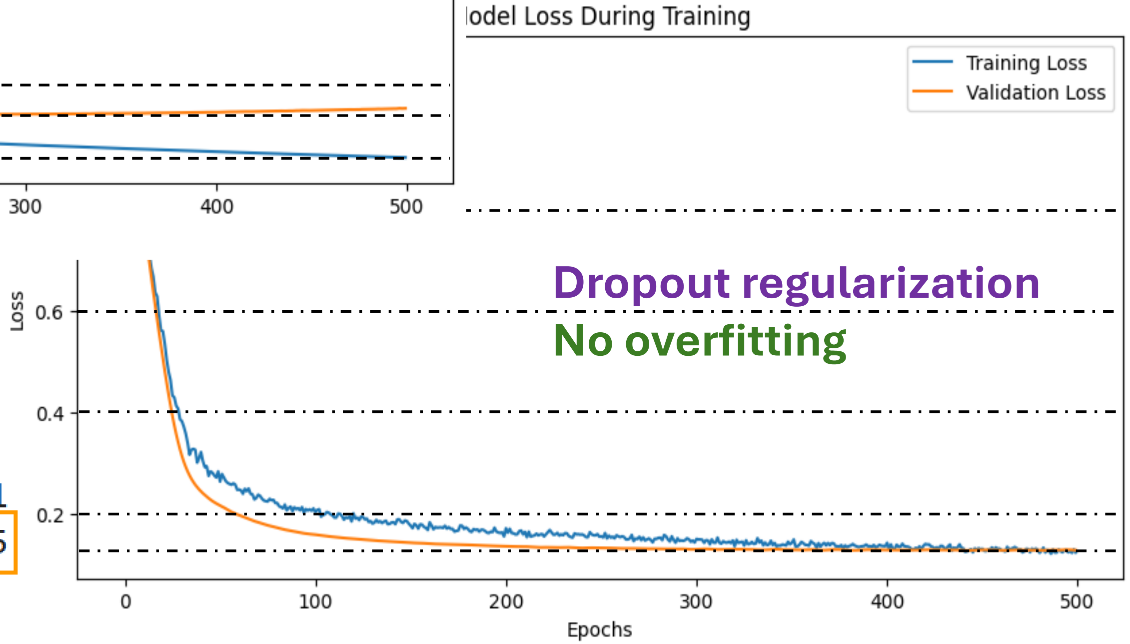


Model Evaluation Metrics:

Metric	Train	Test
MSE	0.0382	0.0811
R-squared	0.9329	0.8679

Model Evaluation Metrics:

Metric	Train	Test
MSE	0.0558	0.0721
R-squared	0.9017	0.8765



Model Evaluation Metrics:

Metric	Train	Test
MSE	0.0553	0.0760
R-squared	0.9026	0.8697