AI in Petroleum Industry
Final Project: Deep Neural Networks for Reservoir Production Forecasting

By Novin Nekuee (403134029) & Soroosh Danesh (810403045)
Dr. Emami & Eng. Nasiri

Note: This is a Side Project This notebook explores a Side Project as a supplemental experiment.
In this approach, statistical outliers were identified and removed from the dataset before the model training process began.

The main and final project, in which the model was trained on the complete dataset and achieved more accurate and robust results, can be found at the following link:

main-project(Trained on All Data)

1. Importing Libraries

This cell imports all the essential Python libraries required for the project.

In [51]:
```
"""
Data Handling and Numerical Operations: pandas for managing data in DataFrames, numpy for efficient numerical array operations, and os for file path mana

Deep Learning: tensorflow and the keras API are imported for building, training, and evaluating the neural network.
        This includes specific layers (Conv2D, Dense, Dropout), model components, and callbacks (EarlyStopping, ReduceLROnPlateau).

Data Preprocessing: scikit-learn is used for splitting the dataset into training and testing sets, and also MinMaxScaler for normalizing data.

Visualization: matplotlib.pyplot and seaborn are included for creating plots to visualize data and model results.

Image Processing: The PIL (Pillow) library is used to open and handle the TIFF image files.

Hyperparameter Tuning: optuna is imported to automate the hyperparameter optimization process.

Metrics: sklearn is used to calculate the metrics for the model.
"""

# Data Handling and Numerical Operations and Path Management
import pandas as pd
import numpy as np
import os

# Image Processing
from PIL import Image

# Deep Learning
import tensorflow as tf
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D, Flatten, concatenate, Dropout
from tensorflow.keras import regularizers

# Data Preprocessing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# Visualization
import seaborn as sns
import matplotlib.pyplot as plt

# Hyperparameter Tuning
import optuna

# Metrics
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Warning handler
import warnings
warnings.filterwarnings('ignore')
```

2. Loading the Dataset

This cell loads the tabular data from the `data.xlsx` file into a pandas DataFrame called data.

In [52]:
```
data = pd.read_excel('./assets/data.xlsx')
data.head()
```

Out[52]:

	sample number	Month (2026)	Initial Sw	Oil Rate (m3/day)	Cumulative Oil (M m3)
0	1	1	0.25	980.04	0.000681
1	1	3	0.25	410.07	25.471000
2	1	5	0.25	397.78	49.735000
3	1	7	0.25	388.08	73.408000
4	1	9	0.25	379.36	96.928000

3. Data Exploration and Validation

This cell performs an initial exploratory data analysis (EDA) to understand the dataset's structure, quality, and statistical properties.

`.isna().sum()` : This command counts the total number of missing or null values in each column, which is essential for identifying data quality issues that need to be addressed.

In [53]:
```
data.isna().sum()
```

Out[53]:
```
sample number            0
Month (2026)             0
Initial Sw              18
Oil Rate (m3/day)       18
Cumulative Oil (M m3)   19
dtype: int64
```

`.describe()` : This function generates a statistical summary for the numerical columns, including metrics like mean, standard deviation, and quartiles. It provides a quick overview of the data's distribution and scale.

In [54]:
```
data.describe()
```

Out[54]:

	sample number	Month (2026)	Initial Sw	Oil Rate (m3/day)	Cumulative Oil (M m3)
count	6300.000000	6300.000000	6282.000000	6282.000000	6281.000000
mean	525.500000	6.000000	0.214938	1423.594093	197.630083
std	303.132813	3.415921	0.028712	2813.436016	501.371992
min	1.000000	1.000000	0.170000	-145.740000	0.000000
25%	263.000000	3.000000	0.190000	345.830000	16.166000
50%	525.500000	6.000000	0.210000	809.145000	85.372000
75%	788.000000	9.000000	0.240000	1423.175000	214.100000
max	1050.000000	11.000000	0.260000	25000.000000	6256.200000

`.info()` : This method offers a concise summary of the DataFrame, showing the data type of each column, the number of non-null entries, and memory usage. This is useful for verifying that data has been loaded with the correct types.

In [55]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6300 entries, 0 to 6299
Data columns (total 5 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   sample number          6300 non-null   int64
 1   Month (2026)           6300 non-null   int64
 2   Initial Sw             6282 non-null   float64
 3   Oil Rate (m3/day)      6282 non-null   float64
 4   Cumulative Oil (M m3)  6281 non-null   float64
dtypes: float64(3), int64(2)
memory usage: 246.2 KB
```

4. Data Preprocessing and Reshaping (For Numerical Data)

This cell prepares the tabular data to align with the project's requirements and the available image data.

First, to ensure consistency between the datasets, the DataFrame is filtered to include only samples up to sample number 756. This is because the porosity image maps were only available for these first 756 samples.

The original dataset is in a `long` format, with six rows for each sample number, corresponding to six different months. This structure is not suitable for our model, as each sample (represented by one set of input images) should correspond to a single row of target values. Therefore, we reshape the data to `wide` format using a pivot_table. This operation transforms the six rows into a single row for each sample, creating 12 distinct columns: `six for the Oil Rate and six for the Cumulative Oil at each time step.`

Finally, the preprocessed data is cleaned by simplifying column names and handling any missing values through imputation (filling them with the `column's mean`). The .head() of the final DataFrame is then displayed for validation.

While more advanced methods like KNNImputer could be used, the simpler approach of mean imputation was chosen. This was done for simplicity and to avoid needing to normalize the data once for imputation and then again later for the model.

In [56]:
```
num_available_samples = 756
df_filtered = data[data['sample number'] <= num_available_samples].copy()
```

```
data_pivot = df_filtered.pivot_table(
    index='sample number',
    columns='Month (2026)',
    values=['Initial Sw', 'Oil Rate (m3/day)', 'Cumulative Oil (M m3)']
)

data_pivot.columns = [f'{val}_{month}' for val, month in data_pivot.columns]
data_pivot.reset_index(inplace=True)

sw_cols = [col for col in data_pivot.columns if 'Initial Sw' in col]
data_pivot['Initial Sw'] = data_pivot[sw_cols[0]]
data_pivot.drop(columns=sw_cols, inplace=True)

for col in data_pivot.columns:
    if data_pivot[col].isnull().any():
        mean_val = data_pivot[col].mean()
        data_pivot[col].fillna(mean_val, inplace=True)

data_pivot.head()
```

Out[56]:

	sample number	Cumulative Oil (M m3)_1	Cumulative Oil (M m3)_3	Cumulative Oil (M m3)_5	Cumulative Oil (M m3)_7	Cumulative Oil (M m3)_9	Cumulative Oil (M m3)_11	Oil Rate (m3/day)_1	Oil Rate (m3/day)_3	Oil Rate (m3/day)_5	Oil Rate (m3/day)_7	Oil Rate (m3/day)_9	Oil Rate (m3/day)_11	Ini
0	1	0.000681	25.4710	49.735	73.408	96.928	119.580	980.04	410.07	397.78	388.08	379.36	371.30	(
1	2	0.003307	118.8300	218.460	302.950	378.490	444.910	4762.30	1828.00	1578.20	1385.00	1218.40	1088.80	(
2	3	0.001104	45.5290	87.461	127.690	167.110	204.590	1590.00	725.02	687.42	659.53	635.76	614.49	(
3	4	0.003663	138.6700	259.880	369.820	470.760	558.220	5274.30	2199.30	1987.10	1802.20	1628.10	1433.70	(
4	5	0.000214	9.7412	19.257	28.656	38.117	47.345	307.56	159.06	156.00	154.09	152.59	151.28	(

Now lets save our processed dataset for easy access and easy compare

In [7]:
```
data_pivot.to_csv('processed_tabular_data.csv', index=False)
```

The `.shape` attribute is used to view the final dimensions (rows and columns) of our processed data_pivot DataFrame.

The output reveals that `our dataset has 753 rows, not the 756` we started with. This indicates that three samples were completely missing from the original tabular data. To ensure our image data and tabular data are perfectly aligned, these three corresponding image samples must also be removed, which is handled in a later step.

In [57]:
```
data_pivot.shape
```

Out[57]:
```
(753, 14)
```

Finally, another check for missing values confirms that the pivoted and imputed data is clean and ready for the next stage.

In [58]:
```
data_pivot.isna().sum()
```

Out[58]:
```
sample number               0
Cumulative Oil (M m3)_1     0
Cumulative Oil (M m3)_3     0
Cumulative Oil (M m3)_5     0
Cumulative Oil (M m3)_7     0
Cumulative Oil (M m3)_9     0
Cumulative Oil (M m3)_11    0
Oil Rate (m3/day)_1         0
Oil Rate (m3/day)_3         0
Oil Rate (m3/day)_5         0
Oil Rate (m3/day)_7         0
Oil Rate (m3/day)_9         0
Oil Rate (m3/day)_11        0
Initial Sw                  0
dtype: int64
```

5. Outlier Detection and Removal

This section details the process of identifying and handling statistical outliers within the dataset. This was performed as an experimental step to understand their impact on model performance.

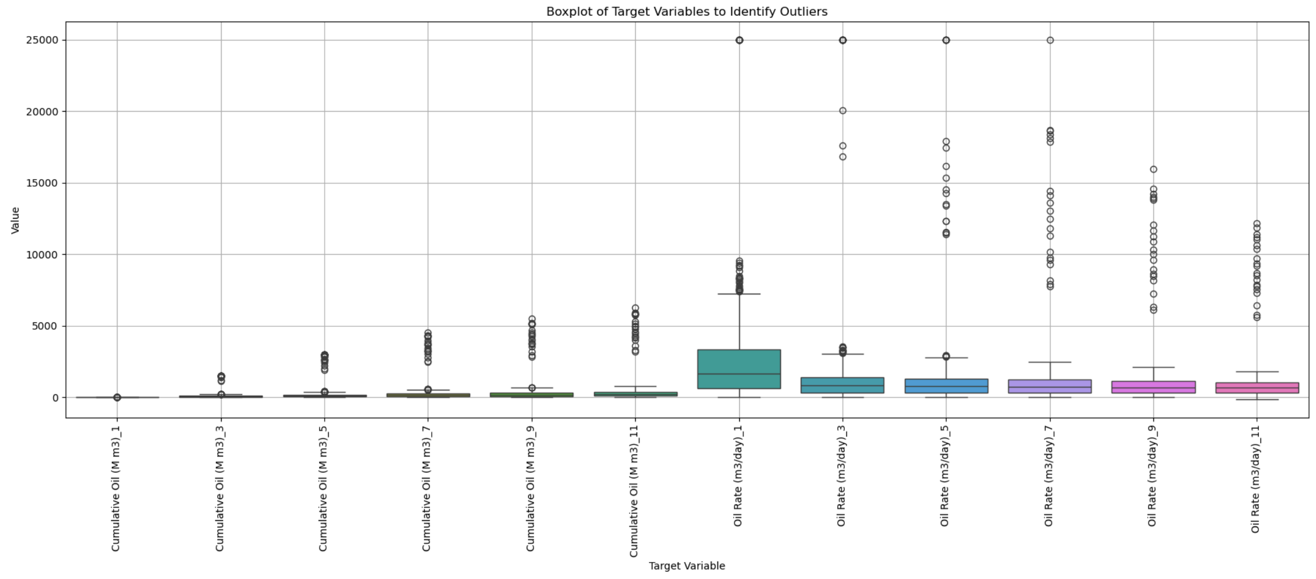
5-1. Visualizing Outliers with Boxplots

Before programmatically removing outliers, a boxplot is generated for all 12 target variables. This visualization helps in understanding the distribution of each variable and visually confirming the presence of outliers.

Analysis of the Plot

The boxplot displays the distribution for each target variable. The points that fall outside the whiskers of the boxes are considered statistical outliers. The plot clearly shows that several variables, particularly the Oil Rate and Cumulative Oil in later months, contain a number of these outlier points.

In [ ]:
```
target_cols = [col for col in data_pivot.columns if col not in ['sample number', 'Initial Sw']]
df_targets_for_plot = data_pivot[target_cols]

plt.figure(figsize=(18, 8))
sns.boxplot(data=df_targets_for_plot)

plt.xticks(rotation=90)
plt.title('Boxplot of Target Variables to Identify Outliers')
plt.xlabel('Target Variable')
plt.ylabel('Value')
plt.grid(True)
plt.tight_layout()

plt.show()
```



5-2. Programmatic Outlier Removal using the IQR Method

Based on the visual confirmation from the boxplots, the next step is to programmatically identify and remove these outliers to create a `cleaned` dataset for one of the training experiments.

The standard Interquartile Range (`IQR`) method is used for this purpose. The code iterates through each of the 12 target columns, calculates the IQR for each, and defines the upper and lower bounds (`typically Q1 - 1.5*IQR and Q3 + 1.5*IQR`). Any sample that falls outside these bounds in any of the target variables is flagged as an outlier. Finally, all unique outlier samples are removed to create a new, cleaned DataFrame.

```
In [ ]: target_cols = [col for col in data_pivot.columns if col not in ['sample number', 'Initial Sw']]

outlier_indices = set()

for col in target_cols:
    Q1 = data_pivot[col].quantile(0.25)
    Q3 = data_pivot[col].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    column_outliers = data_pivot[(data_pivot[col] < lower_bound) | (data_pivot[col] > upper_bound)].index

    outlier_indices.update(column_outliers)

outlier_list = list(outlier_indices)

print(f"Total number of unique outliers found across all 12 targets: {len(outlier_list)}")

print(f"\nOriginal DataFrame shape: {data_pivot.shape}")
data_pivot = data_pivot.drop(outlier_list)

print(f"DataFrame shape after removing all outliers: {data_pivot.shape}")
```

```
Total number of unique outliers found across all 12 targets: 45

Original DataFrame shape: (753, 14)
DataFrame shape after removing all outliers: (708, 14)
```

5-3. Verifying Outlier Removal

After programmatically removing the outliers using the IQR method, the boxplot is generated again on the new, cleaned DataFrame (`data_pivot`). The purpose of this step is to visually confirm the success of the outlier removal process.

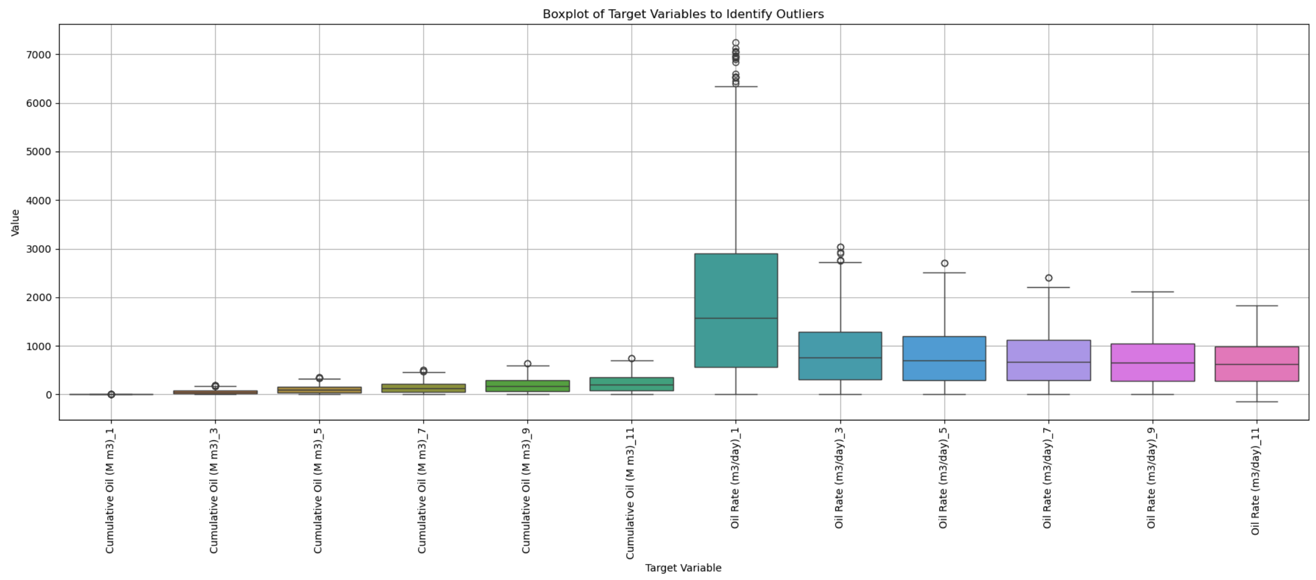
Analysis of the New Plot

As the chart above clearly demonstrates, the individual data points that were previously scattered far beyond the whiskers are now gone. The removal of these extreme values has allowed the plot's y-axis to rescale, providing a much clearer and more detailed view of the distribution of the remaining, `typical` data.

This visualization successfully confirms that the statistical outliers have been removed, resulting in a cleaner and more concentrated dataset for the subsequent training experiment.

```
In [ ]: target_cols = [col for col in data_pivot.columns if col not in ['sample number', 'Initial Sw']]
df_targets_for_plot = data_pivot[target_cols]
```

```
plt.figure(figsize=(18, 8))
sns.boxplot(data=df_targets_for_plot)

plt.xticks(rotation=90)
plt.title('Boxplot of Target Variables to Identify Outliers')
plt.xlabel('Target Variable')
plt.ylabel('Value')
plt.grid(True)
plt.tight_layout()

plt.show()
```



6. Loading and Stacking Image Data (Image Data)

This cell loads and processes the raw image data for the reservoir maps. The project requires two maps for each sample

The code iterates through the 756 sample numbers, loading each pair of permeability and porosity TIFF files. The two separate (64, 64) maps are then stacked along a new channel dimension using np.stack. This creates a single (64, 64, 2) array for each sample, where the first channel represents permeability and the second represents porosity.

Finally, all individual samples are combined into a single 4D NumPy array named image_data. The .shape attribute is called to verify the final dimensions of this array, which will be used as the image input for the model.

In [62]:
```
perm_folder = './assets/permeability/'
poro_folder = './assets/porosity/'

num_samples = 756
all_images_list = []

for i in range(1, num_samples + 1):
    sample_id = str(i).zfill(4)

    perm_path = os.path.join(perm_folder, f'perm_map_{sample_id}.tiff')
    poro_path = os.path.join(poro_folder, f'poro_map_{sample_id}.tiff')

    perm_img = Image.open(perm_path)
    poro_img = Image.open(poro_path)

    perm_array = np.array(perm_img, dtype=np.float32)
    poro_array = np.array(poro_img, dtype=np.float32)

    combined_image = np.stack([perm_array, poro_array], axis=-1)
    all_images_list.append(combined_image)

image_data = np.array(all_images_list)
image_data.shape
```

Out[62]:
```
(756, 64, 64, 2)
```

7. Aligning Tabular and Image Datasets

As identified in the data exploration step, our processed tabular DataFrame (data_pivot) contains 753 samples, while the initial image array was loaded with all 756 samples. This discrepancy means three samples present in the image folders were absent from the Excel file.

This cell resolves this issue to ensure the datasets are perfectly synchronized.

First, it extracts the list of `valid sample numbers` that exist in the final tabular data. These numbers are then converted to their corresponding zero-based array indices. Using this list of valid indices, the original `image_data` array is filtered, effectively removing the three images that do not have corresponding tabular data.

Finally, the synchronized datasets are separated into their final forms for the model: `X_image` (the filtered images), `X_numerical` (the 'Initial Sw' feature), and `y` (the 12 target variables). The shapes of these arrays are printed to confirm that they all now contain a consistent 753 samples.

In [63]:
```python
valid_sample_numbers = data_pivot['sample number'].values

valid_indices = valid_sample_numbers - 1

X_image_filtered = image_data[valid_indices]

X_image = X_image_filtered

X_numerical = data_pivot['Initial Sw'].values.reshape(-1, 1)

target_cols = [col for col in data_pivot.columns if col not in ['sample number', 'Initial Sw']]
y = data_pivot[target_cols].values

print("Data Shapes:")
print(f" images data shape: {X_image.shape}")
print(f" numerical data shapes: {X_numerical.shape}")
print(f" Output shapes: {y.shape}")
```

```
Data Shapes:
 images data shape: (708, 64, 64, 2)
 numerical data shapes: (708, 1)
 Output shapes: (708, 12)
```

8. Creating Training, Validation, and Test Sets

This cell splits the complete dataset into three essential subsets for training and evaluating the deep learning model

- Training Set: The largest portion of the data, used to train the model's parameters.
- Validation Set: A separate subset used during training to monitor the model's performance on unseen data, which helps in tuning hyperparameters and preventing overfitting.
- Test Set: A final, completely unseen subset that is used only once after all training and tuning is complete to provide an unbiased evaluation of the final model's performance.

While it's possible to have Keras create a validation set automatically using the `validation_split` argument within `model.fit()`, we have chosen to create an explicit validation set beforehand. This approach is preferred for process clarity, ensuring that all three datasets are explicitly defined before training begins.

The split is performed in two steps. First, 20% of the data is held back as the test set. The remaining 80% is then split again. To ensure the validation set is 20% of the original total data, we must allocate 25% of the remaining data block for it `(since 0.20 / 0.80 = 0.25)`. This results in a final data distribution of `60% for training, 20% for validation, and 20% for testing`.

The shapes of all three final datasets are printed to verify the dimensions.

In [ ]:
```python
X_train_img, X_test_img, X_train_num, X_test_num, y_train0, y_test = train_test_split(
    X_image, X_numerical, y,
    test_size = 0.15,
    random_state = 42
)

X_train_image, X_val_image, X_train_number, X_val_number, y_train, y_val = train_test_split(
    X_train_img , X_train_num, y_train0,
    test_size = 0.1765,  # 0.15 / 0.85 = 0.1765
    random_state = 42
)

print("\nTraining Data shape:")
print(f" images data shape: (X_train_image): {X_train_image.shape}")
print(f" numerical data shapes (X_train_number): {X_train_number.shape}")
print(f" Output shapes (y_train): {y_train.shape}")
print("-" * 50)

print("\nValidation Data shape:")
print(f" images data shape: (X_val_image): {X_val_image.shape}")
print(f" numerical data shapes (X_val_number): {X_val_number.shape}")
print(f" Output shapes (y_val): {y_val.shape}")
print("-" * 50)

print("\nTest Data shape:")
print(f" images data shape: (X_test_img): {X_test_img.shape}")
print(f" numerical data shapes (X_test_num): {X_test_num.shape}")
print(f" Output shapes (y_test): {y_test.shape}")
```

```
Training Data shape:
 images data shape: (X_train_image): (494, 64, 64, 2)
 numerical data shapes (X_train_number): (494, 1)
 Output shapes (y_train): (494, 12)
--------------------------------------------------

Validation Data shape:
 images data shape: (X_val_image): (107, 64, 64, 2)
 numerical data shapes (X_val_number): (107, 1)
 Output shapes (y_val): (107, 12)
--------------------------------------------------

Test Data shape:
 images data shape: (X_test_img): (107, 64, 64, 2)
 numerical data shapes (X_test_num): (107, 1)
 Output shapes (y_test): (107, 12)
```

This cell prints the minimum and maximum values for each of the unscaled train, validation, and test sets. The purpose is to observe and understand the different scales of our input (image, numerical) and output (target) data before applying normalization.

As the output demonstrates, the value ranges for the image data and the target variables are significantly larger than the range of the numerical input `(Initial Sw)`. This large discrepancy confirms the need for normalization. To ensure all features contribute effectively during model training and to improve numerical stability, we will apply a `MinMaxScaler` to all datasets.

In [65]:
```
print("Training Input (Image) before Scaling:")
print(X_train_image.max())
print(X_train_image.min())
print(X_test_img.max())
print(X_test_img.min())
print(X_val_image.max())
print(X_val_image.min())
print("-" * 50)
print("\nTraining Input (Numerical) before Scaling:")
print(X_train_number.max())
print(X_train_number.min())
print(X_test_num.max())
print(X_test_num.min())
print(X_val_number.max())
print(X_val_number.min())
print("-" * 50)
print("\nTarget Output before Scaling:")
print(y_train.max())
print(y_train.min())
print(y_test.max())
print(y_test.min())
print(y_val.max())
print(y_val.min())
```

```
Training Input (Image) before Scaling:
inf
0.0
277.761
0.0
nan
nan
--------------------------------------------------

Training Input (Numerical) before Scaling:
0.26
0.17
0.26
0.17
0.26
0.17
--------------------------------------------------

Target Output before Scaling:
7249.2
-145.74
7044.4
2.2312e-06
6900.7
2.91036e-07
```

9. Normalizing the Datasets

This cell normalizes all training, validation, and test sets to a consistent [0, 1] range. A crucial best practice is followed: the scaling parameters (e.g., min and max values) are learned only from the training data and then applied to transform all three subsets (train, validation, and test). This prevents any data leakage from the test and validation sets into the training process.

The correct methodology for scaling is to fit the scaler only on the training data and then use that same fitted scaler to transform the training, validation, and test sets. This is critical because the training process should have no knowledge of the test set, which simulates new, unseen data. Applying scaling separately to each set is incorrect as it would create an inconsistent transformation based on different min/max values. This method follows a crucial machine learning principle to prevent data leakage.

Scaling Tabular Data (Targets and Numerical Inputs)

For the target variables (y) and the numerical input (X_numerical), the standard MinMaxScaler from scikit-learn is used. Separate scaler objects are fit on the training data (y_train, X_train_number) and then used to transform the corresponding train, validation, and test sets.

Scaling Image Data

For the 4D image data, the min-max scaling logic is applied manually. This approach was chosen over using the MinMaxScaler object directly for two main reasons:

- Efficiency: For large, multi-dimensional NumPy arrays, direct vectorized operations are often more computationally efficient.
- Simplicity: It avoids the need to reshape the 4D image data into a 2D array to be compatible with the scikit-learn scaler and then reshape it back.

The scaling is performed per-channel, meaning the min/max for the permeability channel is calculated and applied separately from the porosity channel. This preserves the unique statistical distribution of each physical property. The cell concludes by printing the min/max values of all scaled datasets to verify that the normalization was successful.

$$X_{\text{scaled}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

You can see the scaled max and min data for each splitted data as a result:

In [66]:
```
y_scaler = MinMaxScaler()
y_train_scaled = y_scaler.fit_transform(y_train)
y_val_scaled = y_scaler.transform(y_val)
y_test_scaled = y_scaler.transform(y_test)

num_scaler = MinMaxScaler()
X_train_num_scaled = num_scaler.fit_transform(X_train_number)
X_val_num_scaled = num_scaler.transform(X_val_number)
X_test_num_scaled = num_scaler.transform(X_test_num)

perm_min = X_train_image[:, :, :, 0].min()
perm_max = X_train_image[:, :, :, 0].max()
if (perm_max - perm_min) != 0:
    X_train_image[:, :, :, 0] = (X_train_image[:, :, :, 0] - perm_min) / (perm_max - perm_min)
    X_val_image[:, :, :, 0] = (X_val_image[:, :, :, 0] - perm_min) / (perm_max - perm_min)
    X_test_img[:, :, :, 0] = (X_test_img[:, :, :, 0] - perm_min) / (perm_max - perm_min)

poro_min = X_train_image[:, :, :, 1].min()
poro_max = X_train_image[:, :, :, 1].max()
if (poro_max - poro_min) != 0:
    X_train_image[:, :, :, 1] = (X_train_image[:, :, :, 1] - poro_min) / (poro_max - poro_min)
    X_val_image[:, :, :, 1] = (X_val_image[:, :, :, 1] - poro_min) / (poro_max - poro_min)
    X_test_img[:, :, :, 1] = (X_test_img[:, :, :, 1] - poro_min) / (poro_max - poro_min)

X_train_image = np.nan_to_num(X_train_image)
X_val_image = np.nan_to_num(X_val_image)
X_test_img = np.nan_to_num(X_test_img)

X_train_img_scaled, X_val_img_scaled, X_test_img_scaled = X_train_image, X_val_image, X_test_img

print("Training Input (Image) after Scaling:")
print(f" maximum X_train image data: {X_train_img_scaled.max()}")
print(f" minimum X_train image data: {X_train_img_scaled.min()}")
print(f" maximum X_test image data: {X_test_img_scaled.max()}")
print(f" minimum X_test image data: {X_test_img_scaled.min()}")
print(f" maximum X_val image data: {X_val_img_scaled.max()}")
print(f" minimum X_val image data: {X_val_img_scaled.min()}")
print("-" * 50)
print("\nTraining Input (Numerical) after Scaling:")
print(f" maximum X_train numerical data: {X_train_num_scaled.max()}")
print(f" minimum X_train numerical data: {X_train_num_scaled.min()}")
print(f" maximum X_test numerical data: {X_test_num_scaled.max()}")
print(f" minimum X_test numerical data: {X_test_num_scaled.min()}")
print(f" maximum X_val numerical data: {X_val_num_scaled.max()}")
print(f" minimum X_val numerical data: {X_val_num_scaled.min()}")
```

```
Training Input (Image) after Scaling:
 maximum X_train image data: 1.0
 minimum X_train image data: 0.0
 maximum X_test image data: 1.0001295804977417
 minimum X_test image data: 0.0
 maximum X_val image data: 0.999765932559967
 minimum X_val image data: 0.0
--------------------------------------------------

Training Input (Numerical) after Scaling:
 maximum X_train numerical data: 0.9999999999999998
 minimum X_train numerical data: 0.0
 maximum X_test numerical data: 0.9999999999999998
 minimum X_test numerical data: 0.0
 maximum X_val numerical data: 0.9999999999999998
 minimum X_val numerical data: 0.0
```

Also you can see the scaled target values below:

In [67]:
```
print("Target output after Scaling:")
print(f" maximum y_train output data: {y_train_scaled.max()}")
print(f" minimum y_train output data: {y_train_scaled.min()}")
print(f" maximum y_test output data: {y_test_scaled.max()}")
print(f" minimum y_test output data: {y_test_scaled.min()}")
print(f" maximum y_val output data: {y_val_scaled.max()}")
print(f" minimum y_val output data: {y_val_scaled.min()}")
```

```
Target output after Scaling:
 maximum y_train output data: 1.0000000000000002
 minimum y_train output data: 0.0
 maximum y_test output data: 1.017809760781358
 minimum y_test output data: 0.00036133566906878546
 maximum y_val output data: 0.9519217920606939
 minimum y_val output data: -0.00011737324534753351
```

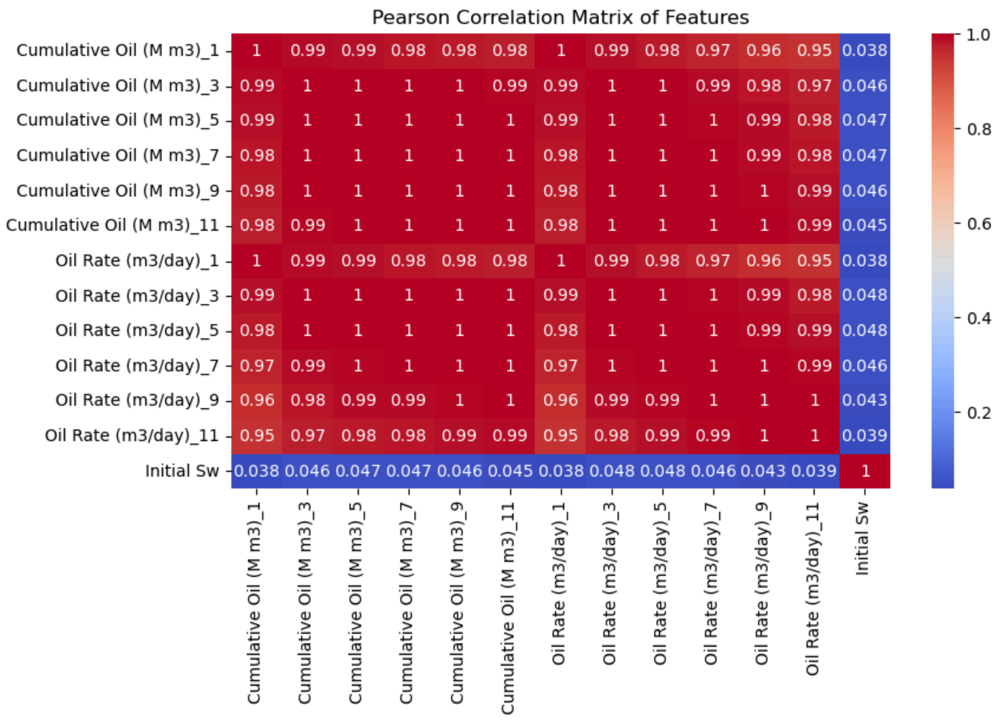
10. Visualizing Feature Correlations

This cell calculates and visualizes the Pearson correlation matrix for all numerical features in the dataset.

While this analysis is often a step towards feature selection, that is not the objective here, as the intention is to use all available features in the model. Instead, the primary purpose of this step is for exploratory data analysis. By creating a heatmap of the correlations, we can visually understand the strong intrinsic relationships between the input feature (Initial Sw) and the various output targets, as well as the relationships among the target variables themselves.

The heatmap displays these relationships, with warmer colors (red) indicating a strong positive correlation and cooler colors (blue) indicating a strong negative correlation. Finally, the specific correlation values for the `Initial Sw` (as only numerical input feature) feature are printed out for a more direct numerical analysis.

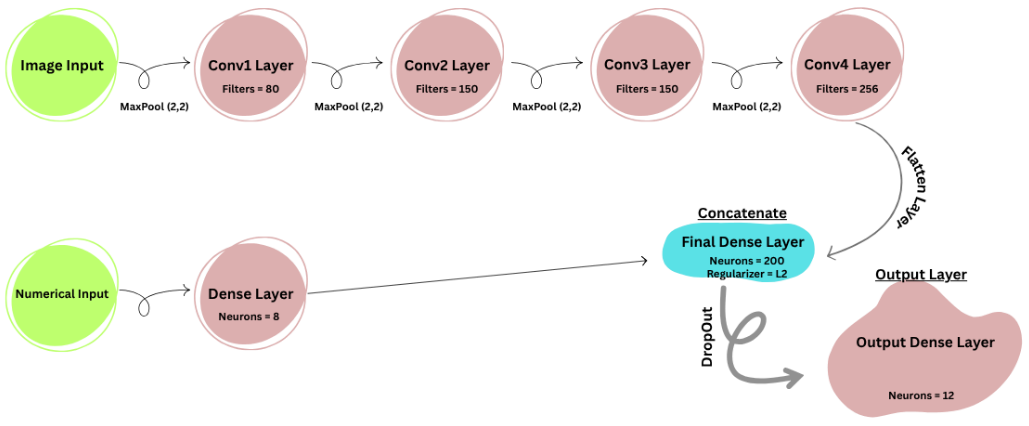
```
In [19]: correlation_matrix = data_pivot.drop('sample number', axis=1).corr(method='pearson')

         plt.figure(figsize=(9, 5))
         sns.heatmap(
             correlation_matrix,
             cmap='coolwarm',
             annot=True
         )
         plt.title('Pearson Correlation Matrix of Features')
         plt.show()

         print("\nCorrelation of 'Initial Sw' with Production Data")
         print(pd.DataFrame(correlation_matrix['Initial Sw'].sort_values(ascending=False)))
```



	Cumulative Oil (M m3)_1	Cumulative Oil (M m3)_3	Cumulative Oil (M m3)_5	Cumulative Oil (M m3)_7	Cumulative Oil (M m3)_9	Cumulative Oil (M m3)_11	Oil Rate (m3/day)_1	Oil Rate (m3/day)_3	Oil Rate (m3/day)_5	Oil Rate (m3/day)_7	Oil Rate (m3/day)_9	Oil Rate (m3/day)_11	Initial Sw
Cumulative Oil (M m3)_1	1	0.99	0.99	0.98	0.98	0.98	1	0.99	0.98	0.97	0.96	0.95	0.038
Cumulative Oil (M m3)_3	0.99	1	1	1	1	0.99	0.99	1	1	0.99	0.98	0.97	0.046
Cumulative Oil (M m3)_5	0.99	1	1	1	1	1	0.99	1	1	1	0.99	0.98	0.047
Cumulative Oil (M m3)_7	0.98	1	1	1	1	1	0.98	1	1	1	0.99	0.98	0.047
Cumulative Oil (M m3)_9	0.98	1	1	1	1	1	0.98	1	1	1	1	0.99	0.046
Cumulative Oil (M m3)_11	0.98	0.99	1	1	1	1	0.98	1	1	1	1	0.99	0.045
Oil Rate (m3/day)_1	1	0.99	0.99	0.98	0.98	0.98	1	0.99	0.98	0.97	0.96	0.95	0.038
Oil Rate (m3/day)_3	0.99	1	1	1	1	1	0.99	1	1	1	0.99	0.98	0.048
Oil Rate (m3/day)_5	0.98	1	1	1	1	1	0.98	1	1	1	0.99	0.99	0.048
Oil Rate (m3/day)_7	0.97	0.99	1	1	1	1	0.97	1	1	1	1	0.99	0.046
Oil Rate (m3/day)_9	0.96	0.98	0.99	0.99	1	1	0.96	0.99	0.99	1	1	1	0.043
Oil Rate (m3/day)_11	0.95	0.97	0.98	0.98	0.99	0.99	0.95	0.98	0.99	0.99	1	1	0.039
Initial Sw	0.038	0.046	0.047	0.047	0.046	0.045	0.038	0.048	0.048	0.046	0.043	0.039	1

```
Correlation of 'Initial Sw' with Production Data
                          Initial Sw
Initial Sw                  1.000000
Oil Rate (m3/day)_5         0.048151
Oil Rate (m3/day)_3         0.047877
Cumulative Oil (M m3)_7     0.046707
Cumulative Oil (M m3)_5     0.046706
Oil Rate (m3/day)_7         0.046189
Cumulative Oil (M m3)_9     0.046135
Cumulative Oil (M m3)_3     0.045793
Cumulative Oil (M m3)_11    0.045343
Oil Rate (m3/day)_9         0.042900
Oil Rate (m3/day)_11        0.039098
Oil Rate (m3/day)_1         0.038003
Cumulative Oil (M m3)_1     0.038002
```

Neural Network Schema



11. Building the Final Optimized Model

This cell defines the final, optimized Keras model architecture using the best hyperparameters discovered by the Optuna search in the previous step. The optimal parameters from the best trial are hardcoded into the model's layers for this definitive version.

The architecture consists of:

- A deep, four-layer Convolutional Neural Network (CNN) branch to extract complex spatial features from the 2-channel input maps.
- A small Dense branch to process the single numerical input feature.
- A concatenation layer that merges the outputs from the CNN and Dense branches.
- A final Dense block with strong regularization (both L2 and Dropout) to interpret the combined features.
- An output layer with 12 linear units to produce the final regression predictions for the 12 target variables.

Finally, model.summary() is called to print a detailed summary of the final architecture.

We used Optuna HyperParameter Optimization to find the best HyperParameters. We find this parameters metioned below:

Best trial:
Value (minimized val_loss): 0.004182
Best Parameters:
filters_c1: 32
filters_c2: 64
filters_c3: 150
filters_c4: 200
dense_units: 128
dropout_rate: 0.22900252867496643
l2_factor: 0.007089910795610233
learning_rate: 0.00035324361573002924
optimizer: Nadam

In [ ]:
```python
# Optuna Hyper Parameter Tuning Best Result in 100 trials
"""
[I 2025-08-03 10:56:41,979] Trial 47 finished with value: 0.004181728232651949 and parameters: {'filters_c1': 32, 'filters_c2': 64,
'filters_c3': 150, 'filters_c4': 200, 'dense_units': 128, 'dropout_rate': 0.22900252867496643, 'l2_factor': 0.007089910795610233,
'learning_rate': 0.00035324361573002924, 'optimizer': 'Nadam'}. Best is trial 47 with value: 0.004181728232651949.
"""

image_input = Input(shape=(64, 64, 2), name='image_input')
cnn = Conv2D(filters=32, kernel_size=(3, 3), activation='selu', kernel_initializer='lecun_normal')(image_input)
cnn = MaxPooling2D(pool_size=(2, 2))(cnn)
cnn = Conv2D(filters=64, kernel_size=(3, 3), activation='selu', kernel_initializer='lecun_normal')(cnn)
cnn = MaxPooling2D(pool_size=(2, 2))(cnn)
cnn = Conv2D(filters=150, kernel_size=(3, 3), activation='selu', kernel_initializer='lecun_normal')(cnn)
cnn = MaxPooling2D(pool_size=(2, 2))(cnn)
cnn = Conv2D(filters=200, kernel_size=(3, 3), activation='selu', kernel_initializer='lecun_normal')(cnn)
cnn_flatten = Flatten()(cnn)

numerical_input = Input(shape=(1,), name='numerical_input')
dense_num = Dense(units=8, activation='selu', kernel_initializer='lecun_normal')(numerical_input)
```

```
combined_features = concatenate([cnn_flatten, dense_num])
final_dense = Dense(units=128, activation='selu', kernel_initializer='lecun_normal', kernel_regularizer=regularizers.l2(0.007089910795610233))(combined_f
final_dense = Dropout(0.22900252867496643)(final_dense)

output = Dense(units=12, activation='linear', name='output')(final_dense)

model = Model(inputs=[image_input, numerical_input], outputs=output)
model.summary()
```

```
Model: "functional"

┏━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)        ┃ Output Shape      ┃    Param # ┃ Connected to      ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
│ image_input         │ (None, 64, 64, 2) │          0 │ -                 │
│ (InputLayer)        │                   │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ conv2d (Conv2D)     │ (None, 62, 62,    │        608 │ image_input[0][0] │
│                     │ 32)               │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ max_pooling2d       │ (None, 31, 31,    │          0 │ conv2d[0][0]      │
│ (MaxPooling2D)      │ 32)               │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ conv2d_1 (Conv2D)   │ (None, 29, 29,    │     18,496 │ max_pooling2d[0]… │
│                     │ 64)               │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ max_pooling2d_1     │ (None, 14, 14,    │          0 │ conv2d_1[0][0]    │
│ (MaxPooling2D)      │ 64)               │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ conv2d_2 (Conv2D)   │ (None, 12, 12,    │     86,550 │ max_pooling2d_1[… │
│                     │ 150)              │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ max_pooling2d_2     │ (None, 6, 6, 150) │          0 │ conv2d_2[0][0]    │
│ (MaxPooling2D)      │                   │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ conv2d_3 (Conv2D)   │ (None, 4, 4, 200) │    270,200 │ max_pooling2d_2[… │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ numerical_input     │ (None, 1)         │          0 │ -                 │
│ (InputLayer)        │                   │            │                   │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ flatten (Flatten)   │ (None, 3200)      │          0 │ conv2d_3[0][0]    │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ dense (Dense)       │ (None, 8)         │         16 │ numerical_input[… │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ concatenate         │ (None, 3208)      │          0 │ flatten[0][0],    │
│ (Concatenate)       │                   │            │ dense[0][0]       │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ dense_1 (Dense)     │ (None, 128)       │    410,752 │ concatenate[0][0] │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ dropout (Dropout)   │ (None, 128)       │          0 │ dense_1[0][0]     │
├─────────────────────┼───────────────────┼────────────┼───────────────────┤
│ output (Dense)      │ (None, 12)        │      1,548 │ dropout[0][0]     │
└─────────────────────┴───────────────────┴────────────┴───────────────────┘

 Total params: 788,170 (3.01 MB)
 Trainable params: 788,170 (3.01 MB)
 Non-trainable params: 0 (0.00 B)
```

12. Defining Callbacks and Compiling the Model

This cell prepares the model for training by defining essential callbacks and compiling it with the optimal learning configuration.

Callbacks

Two key Keras callbacks are defined to monitor and control the training process:

- **EarlyStopping**: This callback stops the training automatically if the validation loss (val_loss) does not improve for a specified number of epochs (patience=15). Crucially, restore_best_weights=True ensures that the model's final weights are reverted to those from the epoch with the lowest validation loss, preventing overfitting.
- **ReduceLROnPlateau**: This acts as an adaptive learning rate scheduler (Performance learning schedule). It reduces the learning rate by a factor of 10 if the validation loss stagnates for 10 epochs, allowing the model to make finer adjustments as it approaches a solution.

Model Compilation

The `model.compile()` method configures the model for training with the following settings:

- **Optimizer**: The Adam optimizer is selected, using the optimal learning_rate discovered by the Optuna search.
- **Loss Function**: mean_squared_error is chosen, as it is a standard loss function for regression tasks.
- **Metrics**: In addition to the loss, the model will also track and report the Mean Absolute Error (mae) and Root Mean Squared Error (rmse) during training.

```
In [21]: early_stopper = EarlyStopping(
             monitor='val_loss',
             patience=15,
             restore_best_weights=True,
             verbose=1
         )

         lr_reducer = ReduceLROnPlateau(
             monitor='val_loss',
    factor=0.1,
    patience=10,
    min_lr=0.00001,
    verbose=1
)

model.compile(
    optimizer=tf.keras.optimizers.Nadam(learning_rate=0.00035324361573002924), # 1e-3
    loss='mean_squared_error',
    metrics=[
        tf.keras.metrics.MeanAbsoluteError(name='mae'),
        tf.keras.metrics.RootMeanSquaredError(name='rmse')
    ]
)
```

13. Training the Model

This cell executes the main training loop for the compiled Keras model.

The `model.fit()` function trains the model on the scaled training data (`X_train_..._scaled` and `y_train_scaled`). The `validation_data` is also provided, allowing the model to evaluate its performance on unseen data at the end of each epoch.

The training is set to run for a maximum of 150 epochs with a batch size of 32. The `EarlyStopping` and `ReduceLROnPlateau` callbacks, defined in the previous cell, are passed to the training process to monitor performance and prevent overfitting. The results of the training, such as the loss and metrics for each epoch, are stored in the `history` object for later analysis and visualization.

In [22]:
```
epochs = 200
batch_size = 32

history = model.fit(
    x={'image_input': X_train_img_scaled, 'numerical_input': X_train_num_scaled},
    y=y_train_scaled,
    validation_data=(
        {'image_input': X_val_img_scaled, 'numerical_input': X_val_num_scaled},
        y_val_scaled
    ),
    epochs=epochs,
    batch_size=batch_size,
    callbacks=[lr_reducer, early_stopper]
)
```

```
Epoch 1/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 94ms/step - loss: 1.7603 - mae: 0.6037 - rmse: 0.8961 - val_loss: 0.9361 - val_mae: 0.1543 - val_rmse: 0.2009 - learning_rat
e: 3.5324e-04
Epoch 2/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 73ms/step - loss: 0.9547 - mae: 0.1978 - rmse: 0.2535 - val_loss: 0.8957 - val_mae: 0.1209 - val_rmse: 0.1580 - learning_rat
e: 3.5324e-04
Epoch 3/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 71ms/step - loss: 0.9020 - mae: 0.1540 - rmse: 0.1964 - val_loss: 0.8607 - val_mae: 0.1127 - val_rmse: 0.1461 - learning_rat
e: 3.5324e-04
Epoch 4/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 69ms/step - loss: 0.8621 - mae: 0.1371 - rmse: 0.1766 - val_loss: 0.8212 - val_mae: 0.0982 - val_rmse: 0.1293 - learning_rat
e: 3.5324e-04
Epoch 5/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 67ms/step - loss: 0.8211 - mae: 0.1240 - rmse: 0.1596 - val_loss: 0.7823 - val_mae: 0.0914 - val_rmse: 0.1200 - learning_rat
e: 3.5324e-04
Epoch 6/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.7789 - mae: 0.1095 - rmse: 0.1418 - val_loss: 0.7439 - val_mae: 0.0868 - val_rmse: 0.1151 - learning_rat
e: 3.5324e-04
Epoch 7/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.7402 - mae: 0.1048 - rmse: 0.1366 - val_loss: 0.7057 - val_mae: 0.0839 - val_rmse: 0.1108 - learning_rat
e: 3.5324e-04
Epoch 8/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.7003 - mae: 0.0967 - rmse: 0.1263 - val_loss: 0.6680 - val_mae: 0.0805 - val_rmse: 0.1066 - learning_rat
e: 3.5324e-04
Epoch 9/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.6635 - mae: 0.0958 - rmse: 0.1254 - val_loss: 0.6314 - val_mae: 0.0774 - val_rmse: 0.1032 - learning_rat
e: 3.5324e-04
Epoch 10/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.6251 - mae: 0.0881 - rmse: 0.1139 - val_loss: 0.5956 - val_mae: 0.0746 - val_rmse: 0.0982 - learning_rat
e: 3.5324e-04
Epoch 11/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.5900 - mae: 0.0858 - rmse: 0.1112 - val_loss: 0.5621 - val_mae: 0.0732 - val_rmse: 0.0978 - learning_rat
e: 3.5324e-04
Epoch 12/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.5561 - mae: 0.0825 - rmse: 0.1076 - val_loss: 0.5293 - val_mae: 0.0699 - val_rmse: 0.0937 - learning_rat
e: 3.5324e-04
Epoch 13/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.5243 - mae: 0.0806 - rmse: 0.1066 - val_loss: 0.4984 - val_mae: 0.0688 - val_rmse: 0.0912 - learning_rat
e: 3.5324e-04
Epoch 14/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.4933 - mae: 0.0764 - rmse: 0.1019 - val_loss: 0.4689 - val_mae: 0.0665 - val_rmse: 0.0881 - learning_rat
e: 3.5324e-04
Epoch 15/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.4637 - mae: 0.0741 - rmse: 0.0969 - val_loss: 0.4415 - val_mae: 0.0660 - val_rmse: 0.0883 - learning_rat
e: 3.5324e-04
Epoch 16/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.4362 - mae: 0.0717 - rmse: 0.0946 - val_loss: 0.4159 - val_mae: 0.0671 - val_rmse: 0.0898 - learning_rat
e: 3.5324e-04
Epoch 17/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.4104 - mae: 0.0707 - rmse: 0.0931 - val_loss: 0.3909 - val_mae: 0.0640 - val_rmse: 0.0864 - learning_rat
e: 3.5324e-04
Epoch 18/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.3858 - mae: 0.0681 - rmse: 0.0899 - val_loss: 0.3672 - val_mae: 0.0605 - val_rmse: 0.0823 - learning_rat
e: 3.5324e-04
Epoch 19/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.3633 - mae: 0.0673 - rmse: 0.0908 - val_loss: 0.3453 - val_mae: 0.0587 - val_rmse: 0.0801 - learning_rat
e: 3.5324e-04
Epoch 20/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.3412 - mae: 0.0638 - rmse: 0.0860 - val_loss: 0.3247 - val_mae: 0.0569 - val_rmse: 0.0784 - learning_rat
e: 3.5324e-04
Epoch 21/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.3211 - mae: 0.0636 - rmse: 0.0854 - val_loss: 0.3054 - val_mae: 0.0564 - val_rmse: 0.0771 - learning_rat
e: 3.5324e-04
Epoch 22/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.3021 - mae: 0.0627 - rmse: 0.0844 - val_loss: 0.2875 - val_mae: 0.0558 - val_rmse: 0.0773 - learning_rat
e: 3.5324e-04
Epoch 23/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.2840 - mae: 0.0602 - rmse: 0.0812 - val_loss: 0.2706 - val_mae: 0.0556 - val_rmse: 0.0764 - learning_rat
e: 3.5324e-04
Epoch 24/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.2677 - mae: 0.0609 - rmse: 0.0827 - val_loss: 0.2543 - val_mae: 0.0534 - val_rmse: 0.0726 - learning_rat
e: 3.5324e-04
Epoch 25/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.2515 - mae: 0.0582 - rmse: 0.0785 - val_loss: 0.2402 - val_mae: 0.0566 - val_rmse: 0.0774 - learning_rat
e: 3.5324e-04
Epoch 26/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.2367 - mae: 0.0572 - rmse: 0.0772 - val_loss: 0.2256 - val_mae: 0.0523 - val_rmse: 0.0719 - learning_rat
e: 3.5324e-04
Epoch 27/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.2233 - mae: 0.0571 - rmse: 0.0781 - val_loss: 0.2126 - val_mae: 0.0519 - val_rmse: 0.0718 - learning_rat
e: 3.5324e-04
Epoch 28/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.2106 - mae: 0.0573 - rmse: 0.0790 - val_loss: 0.2004 - val_mae: 0.0523 - val_rmse: 0.0720 - learning_rat
e: 3.5324e-04
Epoch 29/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.1977 - mae: 0.0538 - rmse: 0.0728 - val_loss: 0.1886 - val_mae: 0.0511 - val_rmse: 0.0691 - learning_rat
e: 3.5324e-04
Epoch 30/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.1865 - mae: 0.0532 - rmse: 0.0732 - val_loss: 0.1787 - val_mae: 0.0544 - val_rmse: 0.0749 - learning_rat
e: 3.5324e-04
Epoch 31/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.1762 - mae: 0.0554 - rmse: 0.0749 - val_loss: 0.1678 - val_mae: 0.0515 - val_rmse: 0.0690 - learning_rat
e: 3.5324e-04
Epoch 32/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.1660 - mae: 0.0527 - rmse: 0.0728 - val_loss: 0.1579 - val_mae: 0.0494 - val_rmse: 0.0660 - learning_rat
```

```
e: 3.5324e-04
Epoch 33/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.1567 - mae: 0.0524 - rmse: 0.0729 - val_loss: 0.1496 - val_mae: 0.0505 - val_rmse: 0.0700 - learning_rat
e: 3.5324e-04
Epoch 34/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.1474 - mae: 0.0508 - rmse: 0.0691 - val_loss: 0.1412 - val_mae: 0.0512 - val_rmse: 0.0697 - learning_rat
e: 3.5324e-04
Epoch 35/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.1397 - mae: 0.0531 - rmse: 0.0728 - val_loss: 0.1329 - val_mae: 0.0473 - val_rmse: 0.0656 - learning_rat
e: 3.5324e-04
Epoch 36/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.1313 - mae: 0.0493 - rmse: 0.0675 - val_loss: 0.1253 - val_mae: 0.0479 - val_rmse: 0.0640 - learning_rat
e: 3.5324e-04
Epoch 37/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.1242 - mae: 0.0502 - rmse: 0.0686 - val_loss: 0.1189 - val_mae: 0.0499 - val_rmse: 0.0683 - learning_rat
e: 3.5324e-04
Epoch 38/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 58ms/step - loss: 0.1182 - mae: 0.0535 - rmse: 0.0742 - val_loss: 0.1123 - val_mae: 0.0491 - val_rmse: 0.0668 - learning_rat
e: 3.5324e-04
Epoch 39/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.1104 - mae: 0.0474 - rmse: 0.0641 - val_loss: 0.1060 - val_mae: 0.0478 - val_rmse: 0.0654 - learning_rat
e: 3.5324e-04
Epoch 40/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 55ms/step - loss: 0.1054 - mae: 0.0503 - rmse: 0.0714 - val_loss: 0.1002 - val_mae: 0.0476 - val_rmse: 0.0651 - learning_rat
e: 3.5324e-04
Epoch 41/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0987 - mae: 0.0463 - rmse: 0.0642 - val_loss: 0.0948 - val_mae: 0.0472 - val_rmse: 0.0654 - learning_rat
e: 3.5324e-04
Epoch 42/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0934 - mae: 0.0457 - rmse: 0.0644 - val_loss: 0.0894 - val_mae: 0.0455 - val_rmse: 0.0631 - learning_rat
e: 3.5324e-04
Epoch 43/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.0882 - mae: 0.0460 - rmse: 0.0634 - val_loss: 0.0849 - val_mae: 0.0470 - val_rmse: 0.0655 - learning_rat
e: 3.5324e-04
Epoch 44/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.0835 - mae: 0.0450 - rmse: 0.0634 - val_loss: 0.0806 - val_mae: 0.0491 - val_rmse: 0.0675 - learning_rat
e: 3.5324e-04
Epoch 45/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0790 - mae: 0.0457 - rmse: 0.0638 - val_loss: 0.0767 - val_mae: 0.0514 - val_rmse: 0.0700 - learning_rat
e: 3.5324e-04
Epoch 46/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0750 - mae: 0.0470 - rmse: 0.0651 - val_loss: 0.0721 - val_mae: 0.0483 - val_rmse: 0.0660 - learning_rat
e: 3.5324e-04
Epoch 47/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - loss: 0.0719 - mae: 0.0496 - rmse: 0.0713 - val_loss: 0.0679 - val_mae: 0.0444 - val_rmse: 0.0622 - learning_rat
e: 3.5324e-04
Epoch 48/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0668 - mae: 0.0429 - rmse: 0.0605 - val_loss: 0.0651 - val_mae: 0.0503 - val_rmse: 0.0684 - learning_rat
e: 3.5324e-04
Epoch 49/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.0637 - mae: 0.0457 - rmse: 0.0642 - val_loss: 0.0615 - val_mae: 0.0472 - val_rmse: 0.0665 - learning_rat
e: 3.5324e-04
Epoch 50/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0604 - mae: 0.0454 - rmse: 0.0640 - val_loss: 0.0576 - val_mae: 0.0449 - val_rmse: 0.0612 - learning_rat
e: 3.5324e-04
Epoch 51/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0571 - mae: 0.0445 - rmse: 0.0627 - val_loss: 0.0547 - val_mae: 0.0451 - val_rmse: 0.0612 - learning_rat
e: 3.5324e-04
Epoch 52/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0536 - mae: 0.0423 - rmse: 0.0586 - val_loss: 0.0521 - val_mae: 0.0474 - val_rmse: 0.0632 - learning_rat
e: 3.5324e-04
Epoch 53/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0513 - mae: 0.0438 - rmse: 0.0625 - val_loss: 0.0491 - val_mae: 0.0436 - val_rmse: 0.0602 - learning_rat
e: 3.5324e-04
Epoch 54/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0488 - mae: 0.0444 - rmse: 0.0628 - val_loss: 0.0466 - val_mae: 0.0436 - val_rmse: 0.0610 - learning_rat
e: 3.5324e-04
Epoch 55/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0457 - mae: 0.0413 - rmse: 0.0581 - val_loss: 0.0443 - val_mae: 0.0446 - val_rmse: 0.0613 - learning_rat
e: 3.5324e-04
Epoch 56/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0437 - mae: 0.0426 - rmse: 0.0611 - val_loss: 0.0427 - val_mae: 0.0484 - val_rmse: 0.0660 - learning_rat
e: 3.5324e-04
Epoch 57/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0414 - mae: 0.0426 - rmse: 0.0601 - val_loss: 0.0399 - val_mae: 0.0444 - val_rmse: 0.0608 - learning_rat
e: 3.5324e-04
Epoch 58/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.0392 - mae: 0.0420 - rmse: 0.0589 - val_loss: 0.0380 - val_mae: 0.0442 - val_rmse: 0.0614 - learning_rat
e: 3.5324e-04
Epoch 59/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0369 - mae: 0.0406 - rmse: 0.0566 - val_loss: 0.0362 - val_mae: 0.0454 - val_rmse: 0.0622 - learning_rat
e: 3.5324e-04
Epoch 60/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0350 - mae: 0.0390 - rmse: 0.0555 - val_loss: 0.0343 - val_mae: 0.0434 - val_rmse: 0.0616 - learning_rat
e: 3.5324e-04
Epoch 61/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0337 - mae: 0.0420 - rmse: 0.0594 - val_loss: 0.0334 - val_mae: 0.0498 - val_rmse: 0.0671 - learning_rat
e: 3.5324e-04
Epoch 62/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0323 - mae: 0.0435 - rmse: 0.0615 - val_loss: 0.0306 - val_mae: 0.0416 - val_rmse: 0.0577 - learning_rat
e: 3.5324e-04
Epoch 63/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.0301 - mae: 0.0399 - rmse: 0.0563 - val_loss: 0.0296 - val_mae: 0.0447 - val_rmse: 0.0618 - learning_rat
e: 3.5324e-04
Epoch 64/200
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0291 - mae: 0.0429 - rmse: 0.0602 - val_loss: 0.0289 - val_mae: 0.0491 - val_rmse: 0.0667 - learning_rat
e: 3.5324e-04
Epoch 65/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0278 - mae: 0.0418 - rmse: 0.0607 - val_loss: 0.0267 - val_mae: 0.0440 - val_rmse: 0.0605 - learning_rat
e: 3.5324e-04
Epoch 66/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0263 - mae: 0.0418 - rmse: 0.0593 - val_loss: 0.0255 - val_mae: 0.0427 - val_rmse: 0.0609 - learning_rat
e: 3.5324e-04
Epoch 67/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.0247 - mae: 0.0389 - rmse: 0.0560 - val_loss: 0.0242 - val_mae: 0.0427 - val_rmse: 0.0602 - learning_rat
e: 3.5324e-04
Epoch 68/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0233 - mae: 0.0384 - rmse: 0.0549 - val_loss: 0.0246 - val_mae: 0.0518 - val_rmse: 0.0715 - learning_rat
e: 3.5324e-04
Epoch 69/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0226 - mae: 0.0416 - rmse: 0.0580 - val_loss: 0.0220 - val_mae: 0.0433 - val_rmse: 0.0599 - learning_rat
e: 3.5324e-04
Epoch 70/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0217 - mae: 0.0416 - rmse: 0.0591 - val_loss: 0.0207 - val_mae: 0.0415 - val_rmse: 0.0568 - learning_rat
e: 3.5324e-04
Epoch 71/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - loss: 0.0206 - mae: 0.0409 - rmse: 0.0580 - val_loss: 0.0199 - val_mae: 0.0421 - val_rmse: 0.0585 - learning_rat
e: 3.5324e-04
Epoch 72/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0191 - mae: 0.0369 - rmse: 0.0531 - val_loss: 0.0191 - val_mae: 0.0423 - val_rmse: 0.0590 - learning_rat
e: 3.5324e-04
Epoch 73/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0185 - mae: 0.0382 - rmse: 0.0558 - val_loss: 0.0184 - val_mae: 0.0433 - val_rmse: 0.0605 - learning_rat
e: 3.5324e-04
Epoch 74/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0173 - mae: 0.0369 - rmse: 0.0523 - val_loss: 0.0175 - val_mae: 0.0430 - val_rmse: 0.0593 - learning_rat
e: 3.5324e-04
Epoch 75/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0167 - mae: 0.0374 - rmse: 0.0543 - val_loss: 0.0174 - val_mae: 0.0473 - val_rmse: 0.0648 - learning_rat
e: 3.5324e-04
Epoch 76/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.0161 - mae: 0.0397 - rmse: 0.0555 - val_loss: 0.0161 - val_mae: 0.0434 - val_rmse: 0.0599 - learning_rat
e: 3.5324e-04
Epoch 77/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0153 - mae: 0.0377 - rmse: 0.0541 - val_loss: 0.0155 - val_mae: 0.0438 - val_rmse: 0.0607 - learning_rat
e: 3.5324e-04
Epoch 78/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0144 - mae: 0.0370 - rmse: 0.0525 - val_loss: 0.0147 - val_mae: 0.0416 - val_rmse: 0.0589 - learning_rat
e: 3.5324e-04
Epoch 79/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0141 - mae: 0.0386 - rmse: 0.0547 - val_loss: 0.0146 - val_mae: 0.0450 - val_rmse: 0.0633 - learning_rat
e: 3.5324e-04
Epoch 80/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0130 - mae: 0.0360 - rmse: 0.0508 - val_loss: 0.0134 - val_mae: 0.0420 - val_rmse: 0.0583 - learning_rat
e: 3.5324e-04
Epoch 81/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0128 - mae: 0.0379 - rmse: 0.0536 - val_loss: 0.0134 - val_mae: 0.0456 - val_rmse: 0.0618 - learning_rat
e: 3.5324e-04
Epoch 82/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0122 - mae: 0.0372 - rmse: 0.0532 - val_loss: 0.0149 - val_mae: 0.0586 - val_rmse: 0.0765 - learning_rat
e: 3.5324e-04
Epoch 83/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0129 - mae: 0.0464 - rmse: 0.0631 - val_loss: 0.0125 - val_mae: 0.0449 - val_rmse: 0.0624 - learning_rat
e: 3.5324e-04
Epoch 84/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0113 - mae: 0.0372 - rmse: 0.0532 - val_loss: 0.0127 - val_mae: 0.0492 - val_rmse: 0.0678 - learning_rat
e: 3.5324e-04
Epoch 85/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0108 - mae: 0.0376 - rmse: 0.0528 - val_loss: 0.0115 - val_mae: 0.0443 - val_rmse: 0.0617 - learning_rat
e: 3.5324e-04
Epoch 86/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - loss: 0.0104 - mae: 0.0367 - rmse: 0.0526 - val_loss: 0.0108 - val_mae: 0.0427 - val_rmse: 0.0592 - learning_rat
e: 3.5324e-04
Epoch 87/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0101 - mae: 0.0372 - rmse: 0.0540 - val_loss: 0.0104 - val_mae: 0.0416 - val_rmse: 0.0592 - learning_rat
e: 3.5324e-04
Epoch 88/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0095 - mae: 0.0358 - rmse: 0.0519 - val_loss: 0.0099 - val_mae: 0.0414 - val_rmse: 0.0580 - learning_rat
e: 3.5324e-04
Epoch 89/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - loss: 0.0092 - mae: 0.0360 - rmse: 0.0520 - val_loss: 0.0096 - val_mae: 0.0428 - val_rmse: 0.0579 - learning_rat
e: 3.5324e-04
Epoch 90/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0090 - mae: 0.0377 - rmse: 0.0536 - val_loss: 0.0103 - val_mae: 0.0495 - val_rmse: 0.0665 - learning_rat
e: 3.5324e-04
Epoch 91/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0092 - mae: 0.0396 - rmse: 0.0577 - val_loss: 0.0097 - val_mae: 0.0464 - val_rmse: 0.0639 - learning_rat
e: 3.5324e-04
Epoch 92/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0080 - mae: 0.0354 - rmse: 0.0495 - val_loss: 0.0088 - val_mae: 0.0424 - val_rmse: 0.0591 - learning_rat
e: 3.5324e-04
Epoch 93/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 55ms/step - loss: 0.0080 - mae: 0.0364 - rmse: 0.0526 - val_loss: 0.0096 - val_mae: 0.0483 - val_rmse: 0.0676 - learning_rat
e: 3.5324e-04
Epoch 94/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0080 - mae: 0.0381 - rmse: 0.0553 - val_loss: 0.0081 - val_mae: 0.0413 - val_rmse: 0.0572 - learning_rat
e: 3.5324e-04
Epoch 95/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0076 - mae: 0.0374 - rmse: 0.0531 - val_loss: 0.0083 - val_mae: 0.0441 - val_rmse: 0.0609 - learning_rat
e: 3.5324e-04
```

```
Epoch 96/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0077 - mae: 0.0387 - rmse: 0.0559 - val_loss: 0.0082 - val_mae: 0.0456 - val_rmse: 0.0623 - learning_rat
e: 3.5324e-04
Epoch 97/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0070 - mae: 0.0361 - rmse: 0.0515 - val_loss: 0.0080 - val_mae: 0.0453 - val_rmse: 0.0626 - learning_rat
e: 3.5324e-04
Epoch 98/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0070 - mae: 0.0368 - rmse: 0.0540 - val_loss: 0.0074 - val_mae: 0.0420 - val_rmse: 0.0584 - learning_rat
e: 3.5324e-04
Epoch 99/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 56ms/step - loss: 0.0065 - mae: 0.0366 - rmse: 0.0513 - val_loss: 0.0071 - val_mae: 0.0426 - val_rmse: 0.0582 - learning_rat
e: 3.5324e-04
Epoch 100/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0066 - mae: 0.0375 - rmse: 0.0538 - val_loss: 0.0077 - val_mae: 0.0469 - val_rmse: 0.0647 - learning_rat
e: 3.5324e-04
Epoch 101/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - loss: 0.0062 - mae: 0.0367 - rmse: 0.0517 - val_loss: 0.0067 - val_mae: 0.0412 - val_rmse: 0.0567 - learning_rat
e: 3.5324e-04
Epoch 102/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0059 - mae: 0.0350 - rmse: 0.0500 - val_loss: 0.0066 - val_mae: 0.0417 - val_rmse: 0.0574 - learning_rat
e: 3.5324e-04
Epoch 103/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0055 - mae: 0.0333 - rmse: 0.0479 - val_loss: 0.0069 - val_mae: 0.0441 - val_rmse: 0.0614 - learning_rat
e: 3.5324e-04
Epoch 104/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0056 - mae: 0.0357 - rmse: 0.0504 - val_loss: 0.0066 - val_mae: 0.0426 - val_rmse: 0.0600 - learning_rat
e: 3.5324e-04
Epoch 105/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0055 - mae: 0.0343 - rmse: 0.0506 - val_loss: 0.0067 - val_mae: 0.0452 - val_rmse: 0.0625 - learning_rat
e: 3.5324e-04
Epoch 106/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0055 - mae: 0.0359 - rmse: 0.0517 - val_loss: 0.0068 - val_mae: 0.0466 - val_rmse: 0.0639 - learning_rat
e: 3.5324e-04
Epoch 107/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0052 - mae: 0.0358 - rmse: 0.0503 - val_loss: 0.0058 - val_mae: 0.0423 - val_rmse: 0.0571 - learning_rat
e: 3.5324e-04
Epoch 108/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0052 - mae: 0.0355 - rmse: 0.0517 - val_loss: 0.0061 - val_mae: 0.0426 - val_rmse: 0.0601 - learning_rat
e: 3.5324e-04
Epoch 109/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0054 - mae: 0.0375 - rmse: 0.0544 - val_loss: 0.0061 - val_mae: 0.0443 - val_rmse: 0.0608 - learning_rat
e: 3.5324e-04
Epoch 110/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.0047 - mae: 0.0334 - rmse: 0.0483 - val_loss: 0.0057 - val_mae: 0.0417 - val_rmse: 0.0583 - learning_rat
e: 3.5324e-04
Epoch 111/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.0046 - mae: 0.0334 - rmse: 0.0487 - val_loss: 0.0058 - val_mae: 0.0430 - val_rmse: 0.0603 - learning_rat
e: 3.5324e-04
Epoch 112/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0044 - mae: 0.0327 - rmse: 0.0468 - val_loss: 0.0056 - val_mae: 0.0423 - val_rmse: 0.0592 - learning_rat
e: 3.5324e-04
Epoch 113/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0042 - mae: 0.0331 - rmse: 0.0467 - val_loss: 0.0054 - val_mae: 0.0410 - val_rmse: 0.0581 - learning_rat
e: 3.5324e-04
Epoch 114/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 56ms/step - loss: 0.0045 - mae: 0.0341 - rmse: 0.0498 - val_loss: 0.0056 - val_mae: 0.0436 - val_rmse: 0.0605 - learning_rat
e: 3.5324e-04
Epoch 115/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0048 - mae: 0.0368 - rmse: 0.0539 - val_loss: 0.0066 - val_mae: 0.0508 - val_rmse: 0.0687 - learning_rat
e: 3.5324e-04
Epoch 116/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0049 - mae: 0.0391 - rmse: 0.0552 - val_loss: 0.0057 - val_mae: 0.0448 - val_rmse: 0.0620 - learning_rat
e: 3.5324e-04
Epoch 117/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0052 - mae: 0.0395 - rmse: 0.0574 - val_loss: 0.0055 - val_mae: 0.0434 - val_rmse: 0.0611 - learning_rat
e: 3.5324e-04
Epoch 118/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0042 - mae: 0.0346 - rmse: 0.0495 - val_loss: 0.0052 - val_mae: 0.0398 - val_rmse: 0.0584 - learning_rat
e: 3.5324e-04
Epoch 119/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - loss: 0.0040 - mae: 0.0332 - rmse: 0.0478 - val_loss: 0.0052 - val_mae: 0.0436 - val_rmse: 0.0595 - learning_rat
e: 3.5324e-04
Epoch 120/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0043 - mae: 0.0353 - rmse: 0.0512 - val_loss: 0.0052 - val_mae: 0.0440 - val_rmse: 0.0598 - learning_rat
e: 3.5324e-04
Epoch 121/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 54ms/step - loss: 0.0040 - mae: 0.0340 - rmse: 0.0496 - val_loss: 0.0050 - val_mae: 0.0427 - val_rmse: 0.0592 - learning_rat
e: 3.5324e-04
Epoch 122/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0038 - mae: 0.0335 - rmse: 0.0476 - val_loss: 0.0051 - val_mae: 0.0428 - val_rmse: 0.0603 - learning_rat
e: 3.5324e-04
Epoch 123/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0041 - mae: 0.0352 - rmse: 0.0514 - val_loss: 0.0062 - val_mae: 0.0504 - val_rmse: 0.0693 - learning_rat
e: 3.5324e-04
Epoch 124/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0039 - mae: 0.0353 - rmse: 0.0497 - val_loss: 0.0051 - val_mae: 0.0436 - val_rmse: 0.0610 - learning_rat
e: 3.5324e-04
Epoch 125/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - loss: 0.0037 - mae: 0.0329 - rmse: 0.0482 - val_loss: 0.0053 - val_mae: 0.0459 - val_rmse: 0.0628 - learning_rat
e: 3.5324e-04
Epoch 126/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0038 - mae: 0.0345 - rmse: 0.0499 - val_loss: 0.0047 - val_mae: 0.0420 - val_rmse: 0.0580 - learning_rat
e: 3.5324e-04
Epoch 127/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0037 - mae: 0.0343 - rmse: 0.0486 - val_loss: 0.0058 - val_mae: 0.0487 - val_rmse: 0.0667 - learning_rat
```

```
e: 3.5324e-04
Epoch 128/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.0041 - mae: 0.0361 - rmse: 0.0526 - val_loss: 0.0051 - val_mae: 0.0434 - val_rmse: 0.0616 - learning_rat
e: 3.5324e-04
Epoch 129/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0036 - mae: 0.0338 - rmse: 0.0488 - val_loss: 0.0055 - val_mae: 0.0466 - val_rmse: 0.0651 - learning_rat
e: 3.5324e-04
Epoch 130/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 48ms/step - loss: 0.0035 - mae: 0.0344 - rmse: 0.0481 - val_loss: 0.0048 - val_mae: 0.0434 - val_rmse: 0.0599 - learning_rat
e: 3.5324e-04
Epoch 131/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0034 - mae: 0.0330 - rmse: 0.0469 - val_loss: 0.0046 - val_mae: 0.0431 - val_rmse: 0.0592 - learning_rat
e: 3.5324e-04
Epoch 132/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0039 - mae: 0.0357 - rmse: 0.0523 - val_loss: 0.0054 - val_mae: 0.0453 - val_rmse: 0.0652 - learning_rat
e: 3.5324e-04
Epoch 133/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0035 - mae: 0.0337 - rmse: 0.0479 - val_loss: 0.0057 - val_mae: 0.0501 - val_rmse: 0.0676 - learning_rat
e: 3.5324e-04
Epoch 134/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0037 - mae: 0.0348 - rmse: 0.0503 - val_loss: 0.0069 - val_mae: 0.0540 - val_rmse: 0.0760 - learning_rat
e: 3.5324e-04
Epoch 135/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 60ms/step - loss: 0.0036 - mae: 0.0347 - rmse: 0.0493 - val_loss: 0.0071 - val_mae: 0.0547 - val_rmse: 0.0776 - learning_rat
e: 3.5324e-04
Epoch 136/200
15/16 ━━━━━━━━━━━━━━━━━━━━ 0s 44ms/step - loss: 0.0041 - mae: 0.0382 - rmse: 0.0547
Epoch 136: ReduceLROnPlateau reducing learning rate to 3.532436094246805e-05.
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0041 - mae: 0.0380 - rmse: 0.0544 - val_loss: 0.0048 - val_mae: 0.0431 - val_rmse: 0.0603 - learning_rat
e: 3.5324e-04
Epoch 137/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0034 - mae: 0.0326 - rmse: 0.0478 - val_loss: 0.0043 - val_mae: 0.0397 - val_rmse: 0.0564 - learning_rat
e: 3.5324e-05
Epoch 138/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0029 - mae: 0.0282 - rmse: 0.0422 - val_loss: 0.0042 - val_mae: 0.0392 - val_rmse: 0.0559 - learning_rat
e: 3.5324e-05
Epoch 139/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 61ms/step - loss: 0.0031 - mae: 0.0306 - rmse: 0.0446 - val_loss: 0.0043 - val_mae: 0.0402 - val_rmse: 0.0571 - learning_rat
e: 3.5324e-05
Epoch 140/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 61ms/step - loss: 0.0027 - mae: 0.0274 - rmse: 0.0408 - val_loss: 0.0042 - val_mae: 0.0399 - val_rmse: 0.0565 - learning_rat
e: 3.5324e-05
Epoch 141/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 92ms/step - loss: 0.0026 - mae: 0.0272 - rmse: 0.0401 - val_loss: 0.0042 - val_mae: 0.0400 - val_rmse: 0.0566 - learning_rat
e: 3.5324e-05
Epoch 142/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 85ms/step - loss: 0.0027 - mae: 0.0275 - rmse: 0.0411 - val_loss: 0.0042 - val_mae: 0.0399 - val_rmse: 0.0568 - learning_rat
e: 3.5324e-05
Epoch 143/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 68ms/step - loss: 0.0024 - mae: 0.0250 - rmse: 0.0376 - val_loss: 0.0041 - val_mae: 0.0395 - val_rmse: 0.0563 - learning_rat
e: 3.5324e-05
Epoch 144/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 70ms/step - loss: 0.0027 - mae: 0.0270 - rmse: 0.0411 - val_loss: 0.0041 - val_mae: 0.0397 - val_rmse: 0.0562 - learning_rat
e: 3.5324e-05
Epoch 145/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 71ms/step - loss: 0.0026 - mae: 0.0271 - rmse: 0.0412 - val_loss: 0.0041 - val_mae: 0.0402 - val_rmse: 0.0567 - learning_rat
e: 3.5324e-05
Epoch 146/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 66ms/step - loss: 0.0022 - mae: 0.0240 - rmse: 0.0359 - val_loss: 0.0042 - val_mae: 0.0402 - val_rmse: 0.0570 - learning_rat
e: 3.5324e-05
Epoch 147/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 66ms/step - loss: 0.0024 - mae: 0.0257 - rmse: 0.0389 - val_loss: 0.0041 - val_mae: 0.0399 - val_rmse: 0.0563 - learning_rat
e: 3.5324e-05
Epoch 148/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 65ms/step - loss: 0.0025 - mae: 0.0263 - rmse: 0.0398 - val_loss: 0.0042 - val_mae: 0.0407 - val_rmse: 0.0575 - learning_rat
e: 3.5324e-05
Epoch 149/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 75ms/step - loss: 0.0025 - mae: 0.0260 - rmse: 0.0399 - val_loss: 0.0041 - val_mae: 0.0398 - val_rmse: 0.0569 - learning_rat
e: 3.5324e-05
Epoch 150/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 69ms/step - loss: 0.0022 - mae: 0.0241 - rmse: 0.0367 - val_loss: 0.0041 - val_mae: 0.0399 - val_rmse: 0.0569 - learning_rat
e: 3.5324e-05
Epoch 151/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 68ms/step - loss: 0.0023 - mae: 0.0255 - rmse: 0.0385 - val_loss: 0.0041 - val_mae: 0.0404 - val_rmse: 0.0573 - learning_rat
e: 3.5324e-05
Epoch 152/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 65ms/step - loss: 0.0024 - mae: 0.0256 - rmse: 0.0394 - val_loss: 0.0041 - val_mae: 0.0404 - val_rmse: 0.0572 - learning_rat
e: 3.5324e-05
Epoch 153/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 71ms/step - loss: 0.0025 - mae: 0.0264 - rmse: 0.0404 - val_loss: 0.0041 - val_mae: 0.0403 - val_rmse: 0.0568 - learning_rat
e: 3.5324e-05
Epoch 154/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 63ms/step - loss: 0.0024 - mae: 0.0258 - rmse: 0.0401 - val_loss: 0.0039 - val_mae: 0.0389 - val_rmse: 0.0556 - learning_rat
e: 3.5324e-05
Epoch 155/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 65ms/step - loss: 0.0022 - mae: 0.0245 - rmse: 0.0371 - val_loss: 0.0039 - val_mae: 0.0390 - val_rmse: 0.0554 - learning_rat
e: 3.5324e-05
Epoch 156/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 67ms/step - loss: 0.0024 - mae: 0.0263 - rmse: 0.0402 - val_loss: 0.0039 - val_mae: 0.0391 - val_rmse: 0.0560 - learning_rat
e: 3.5324e-05
Epoch 157/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 71ms/step - loss: 0.0024 - mae: 0.0257 - rmse: 0.0392 - val_loss: 0.0039 - val_mae: 0.0391 - val_rmse: 0.0560 - learning_rat
e: 3.5324e-05
Epoch 158/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 65ms/step - loss: 0.0023 - mae: 0.0259 - rmse: 0.0392 - val_loss: 0.0041 - val_mae: 0.0401 - val_rmse: 0.0578 - learning_rat
```

```
e: 3.5324e-05
Epoch 159/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0022 - mae: 0.0241 - rmse: 0.0372 - val_loss: 0.0039 - val_mae: 0.0397 - val_rmse: 0.0561 - learning_rat
e: 3.5324e-05
Epoch 160/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0023 - mae: 0.0254 - rmse: 0.0383 - val_loss: 0.0041 - val_mae: 0.0406 - val_rmse: 0.0573 - learning_rat
e: 3.5324e-05
Epoch 161/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0023 - mae: 0.0255 - rmse: 0.0389 - val_loss: 0.0040 - val_mae: 0.0403 - val_rmse: 0.0569 - learning_rat
e: 3.5324e-05
Epoch 162/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0023 - mae: 0.0255 - rmse: 0.0390 - val_loss: 0.0040 - val_mae: 0.0407 - val_rmse: 0.0572 - learning_rat
e: 3.5324e-05
Epoch 163/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 55ms/step - loss: 0.0022 - mae: 0.0247 - rmse: 0.0383 - val_loss: 0.0039 - val_mae: 0.0400 - val_rmse: 0.0564 - learning_rat
e: 3.5324e-05
Epoch 164/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - loss: 0.0022 - mae: 0.0243 - rmse: 0.0374
Epoch 164: ReduceLROnPlateau reducing learning rate to 1e-05.
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0022 - mae: 0.0244 - rmse: 0.0375 - val_loss: 0.0040 - val_mae: 0.0404 - val_rmse: 0.0573 - learning_rat
e: 3.5324e-05
Epoch 165/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0021 - mae: 0.0241 - rmse: 0.0367 - val_loss: 0.0039 - val_mae: 0.0395 - val_rmse: 0.0563 - learning_rat
e: 1.0000e-05
Epoch 166/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0020 - mae: 0.0236 - rmse: 0.0359 - val_loss: 0.0039 - val_mae: 0.0397 - val_rmse: 0.0566 - learning_rat
e: 1.0000e-05
Epoch 167/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0024 - mae: 0.0260 - rmse: 0.0411 - val_loss: 0.0040 - val_mae: 0.0396 - val_rmse: 0.0567 - learning_rat
e: 1.0000e-05
Epoch 168/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0021 - mae: 0.0231 - rmse: 0.0362 - val_loss: 0.0039 - val_mae: 0.0393 - val_rmse: 0.0562 - learning_rat
e: 1.0000e-05
Epoch 169/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 56ms/step - loss: 0.0023 - mae: 0.0251 - rmse: 0.0397 - val_loss: 0.0040 - val_mae: 0.0398 - val_rmse: 0.0569 - learning_rat
e: 1.0000e-05
Epoch 170/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.0025 - mae: 0.0270 - rmse: 0.0417 - val_loss: 0.0039 - val_mae: 0.0394 - val_rmse: 0.0561 - learning_rat
e: 1.0000e-05
Epoch 171/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0023 - mae: 0.0253 - rmse: 0.0393 - val_loss: 0.0039 - val_mae: 0.0398 - val_rmse: 0.0566 - learning_rat
e: 1.0000e-05
Epoch 172/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0022 - mae: 0.0241 - rmse: 0.0379 - val_loss: 0.0039 - val_mae: 0.0393 - val_rmse: 0.0560 - learning_rat
e: 1.0000e-05
Epoch 173/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 57ms/step - loss: 0.0022 - mae: 0.0246 - rmse: 0.0380 - val_loss: 0.0039 - val_mae: 0.0394 - val_rmse: 0.0563 - learning_rat
e: 1.0000e-05
Epoch 174/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0023 - mae: 0.0250 - rmse: 0.0395 - val_loss: 0.0038 - val_mae: 0.0392 - val_rmse: 0.0557 - learning_rat
e: 1.0000e-05
Epoch 175/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0022 - mae: 0.0248 - rmse: 0.0382 - val_loss: 0.0039 - val_mae: 0.0394 - val_rmse: 0.0561 - learning_rat
e: 1.0000e-05
Epoch 176/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.0022 - mae: 0.0250 - rmse: 0.0388 - val_loss: 0.0039 - val_mae: 0.0393 - val_rmse: 0.0562 - learning_rat
e: 1.0000e-05
Epoch 177/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0021 - mae: 0.0240 - rmse: 0.0372 - val_loss: 0.0039 - val_mae: 0.0394 - val_rmse: 0.0565 - learning_rat
e: 1.0000e-05
Epoch 178/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - loss: 0.0020 - mae: 0.0237 - rmse: 0.0365 - val_loss: 0.0040 - val_mae: 0.0397 - val_rmse: 0.0571 - learning_rat
e: 1.0000e-05
Epoch 179/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 57ms/step - loss: 0.0021 - mae: 0.0243 - rmse: 0.0377 - val_loss: 0.0039 - val_mae: 0.0395 - val_rmse: 0.0568 - learning_rat
e: 1.0000e-05
Epoch 180/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0020 - mae: 0.0231 - rmse: 0.0356 - val_loss: 0.0039 - val_mae: 0.0395 - val_rmse: 0.0566 - learning_rat
e: 1.0000e-05
Epoch 181/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0022 - mae: 0.0247 - rmse: 0.0380 - val_loss: 0.0039 - val_mae: 0.0396 - val_rmse: 0.0565 - learning_rat
e: 1.0000e-05
Epoch 182/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.0021 - mae: 0.0238 - rmse: 0.0371 - val_loss: 0.0039 - val_mae: 0.0397 - val_rmse: 0.0568 - learning_rat
e: 1.0000e-05
Epoch 183/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0023 - mae: 0.0255 - rmse: 0.0395 - val_loss: 0.0039 - val_mae: 0.0395 - val_rmse: 0.0566 - learning_rat
e: 1.0000e-05
Epoch 184/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0022 - mae: 0.0250 - rmse: 0.0389 - val_loss: 0.0039 - val_mae: 0.0398 - val_rmse: 0.0567 - learning_rat
e: 1.0000e-05
Epoch 185/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 57ms/step - loss: 0.0022 - mae: 0.0246 - rmse: 0.0384 - val_loss: 0.0039 - val_mae: 0.0399 - val_rmse: 0.0568 - learning_rat
e: 1.0000e-05
Epoch 186/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step - loss: 0.0024 - mae: 0.0261 - rmse: 0.0413 - val_loss: 0.0039 - val_mae: 0.0397 - val_rmse: 0.0567 - learning_rat
e: 1.0000e-05
Epoch 187/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - loss: 0.0020 - mae: 0.0240 - rmse: 0.0366 - val_loss: 0.0039 - val_mae: 0.0395 - val_rmse: 0.0564 - learning_rat
e: 1.0000e-05
Epoch 188/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 52ms/step - loss: 0.0022 - mae: 0.0247 - rmse: 0.0392 - val_loss: 0.0039 - val_mae: 0.0394 - val_rmse: 0.0563 - learning_rat
e: 1.0000e-05
Epoch 189/200
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - loss: 0.0022 - mae: 0.0252 - rmse: 0.0385 - val_loss: 0.0039 - val_mae: 0.0397 - val_rmse: 0.0566 - learning_rat
```

```
e: 1.0000e-05
Epoch 189: early stopping
Restoring model weights from the end of the best epoch: 174.
```

14. Final Model Evaluation on the Test Set

This cell evaluates the performance of the final trained model on the completely unseen test set.

- **Prediction**: The `model.predict()` method is called on the scaled test inputs (`X_test_..._scaled`) to generate predictions. These initial predictions are in the normalized [0, 1] range.
- **Inverse Scaling**: To make the results interpretable, the `y_scaler` object (which was fit on the original training data) is used. Its `.inverse_transform()` method converts both the model's scaled predictions and the scaled true target values (`y_test_scaled`) back to their original physical units.
- **Metric Calculation**: Standard regression metrics—Mean Absolute Error (`MAE`), Mean Squared Error (`MSE`), Root Mean Squared Error (`RMSE`), and R-squared (`R²`)—are calculated by comparing the un-scaled predictions against the un-scaled true values.

In [23]:
```
y_pred_scaled = model.predict({
    'image_input': X_test_img_scaled,
    'numerical_input': X_test_num_scaled
})

y_test = y_scaler.inverse_transform(y_test_scaled)
y_pred = y_scaler.inverse_transform(y_pred_scaled)

mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print(f"MAE  = {mae:.4f}")
print(f"MSE  = {mse:.4f}")
print(f"RMSE = {rmse:.4f}")
print(f"R²   = {r2:.4f}")
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step
MAE  = 82.1035
MSE  = 35818.6940
RMSE = 189.2583
R²   = 0.9330
```

15. Visualizing Training History

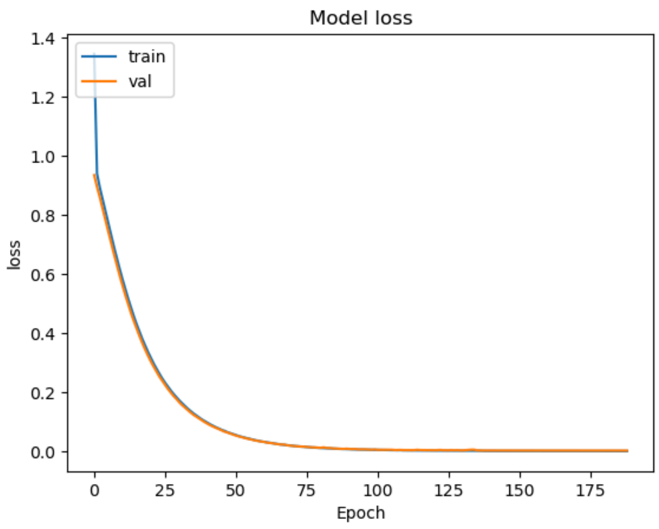
This cell uses matplotlib to create a plot of the model's training and validation loss over each epoch.

The loss values are extracted from the history object, which was returned by the model.fit() function. Plotting these two curves on the same graph is a critical step for diagnosing the model's behavior. It allows for a visual inspection of how the model learned over time and helps confirm that the training was successful and free from significant overfitting or underfitting. The plot is given a title, axis labels, and a legend for clarity.

The key observations from this chart are:

- Healthy Learning Trend: Both the training loss (loss) and the validation loss (val_loss) show a strong, consistent downward trend. The most significant improvements occur in the initial epochs, followed by a period of steady fine-tuning.
- Excellent Convergence: Both the training loss (loss) and the validation loss (val_loss) decrease sharply and smoothly, eventually converging to a very low error value. This indicates the model learned the data efficiently and effectively.
- No Signs of Overfitting: Crucially, the validation loss consistently tracks the training loss without diverging or increasing. The small, stable gap between the two curves is ideal and indicates that the model is generalizing well to new, unseen data. If the model were overfitting, the validation loss would start to increase while the training loss continued to decrease.
- Well-Balanced Model: The minimal and stable gap between the two curves is the hallmark of a well-balanced model that generalizes very well to unseen data.
- Conclusion: This plot demonstrates a well-balanced and successful training process. The model has learned effectively without memorizing the training data, resulting in a final model that is neither overfit nor underfit.

In [24]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])

plt.title('Model loss')
plt.ylabel ('loss')
plt.xlabel('Epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```



16. Model Validation and Visualization

After training the final model, this section focuses on a deep visual analysis of its performance on the unseen test set.

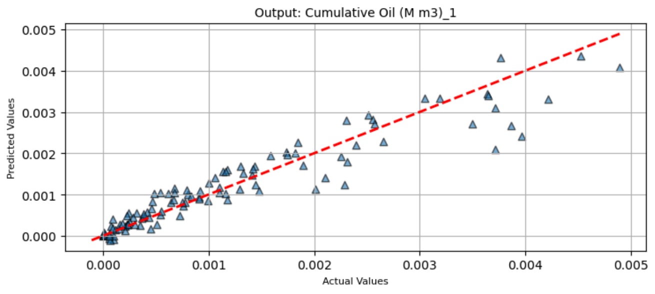
16-1. Visual Analysis: `Actual vs. Predicted` Plots

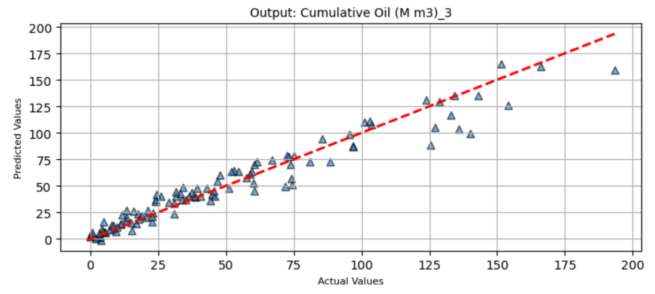
The charts below show the `Actual vs. Predicted` results for all 12 target variables from this experimental model.

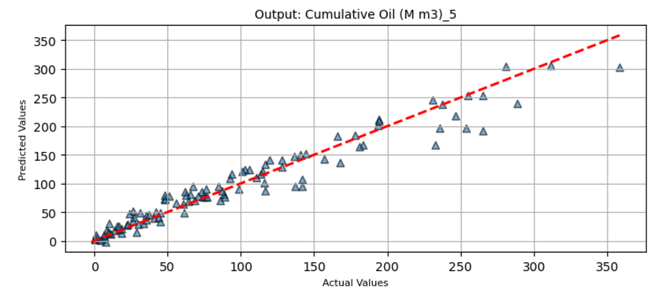
At first glance, these plots appear to have less scatter compared to the plots from the main model (which was trained on all data). The data points seem more concentrated. However, a closer inspection, particularly in the plots corresponding to larger production values, reveals that the model's predictions tend to be lower than the actual values (the points often fall below the red line).

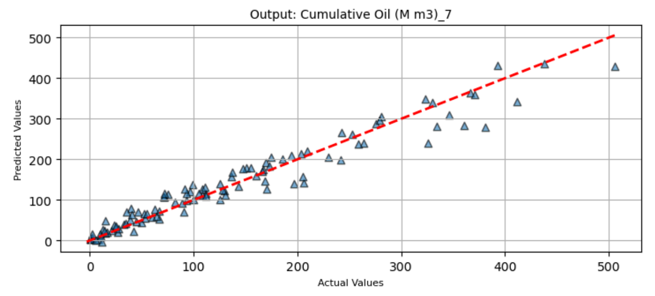
This suggests that because the model was not trained on the high-value data that was flagged as outliers, its accuracy is lower when predicting these larger values.

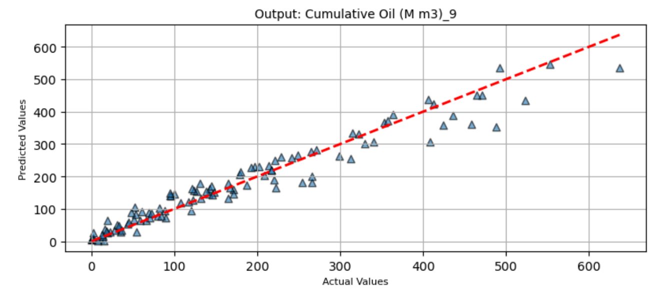
In [76]:
```
fig, axes = plt.subplots(6, 2, figsize=(15, 20))

axes = axes.flatten()

for i in range(12):
    ax = axes[i]

    actual_data = y_test[:, i]
    predicted_data = y_pred[:, i]

    ax.scatter(actual_data, predicted_data, alpha=0.6, edgecolors='k', marker="^")

    if 'target_cols' in locals():
        ax.set_title(f'Output: {target_cols[i]}', fontsize=10)
    else:
        ax.set_title(f'Output Variable #{i+1}', fontsize=10)

    ax.set_xlabel('Actual Values', fontsize=8)
    ax.set_ylabel('Predicted Values', fontsize=8)
    ax.grid(True)

    min_val = min(actual_data.min(), predicted_data.min())
    max_val = max(actual_data.max(), predicted_data.max())
    ax.plot([min_val, max_val], [min_val, max_val], 'r--', lw=2)

plt.tight_layout()

plt.show()
```

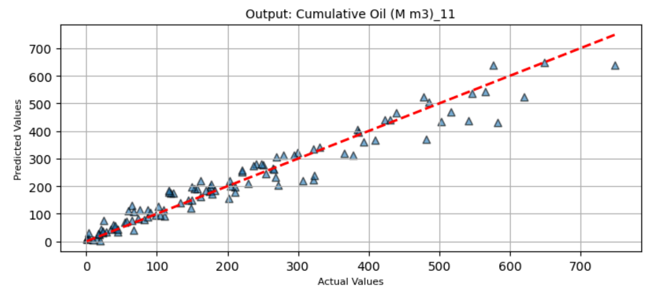


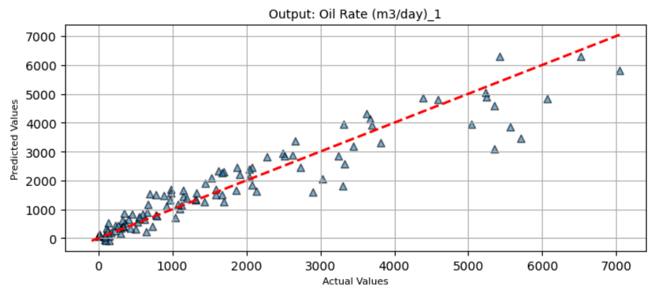


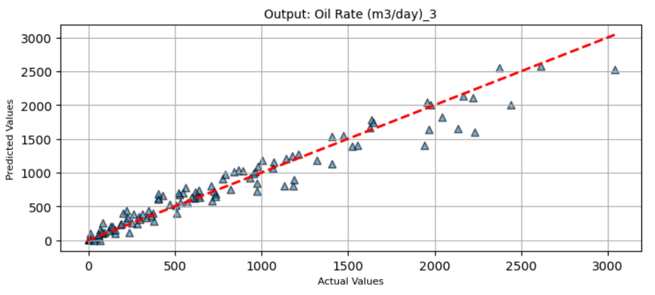


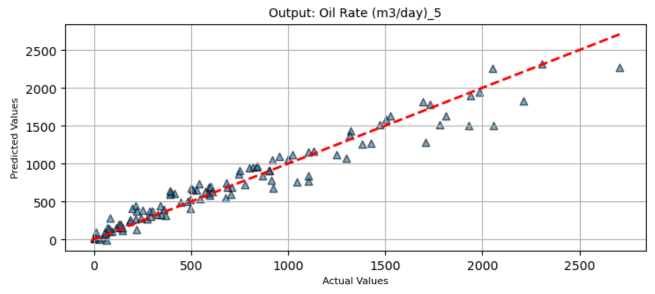


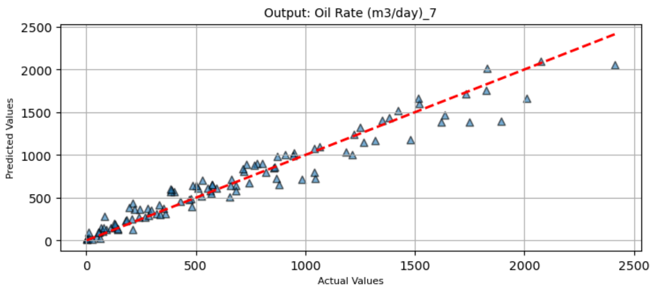


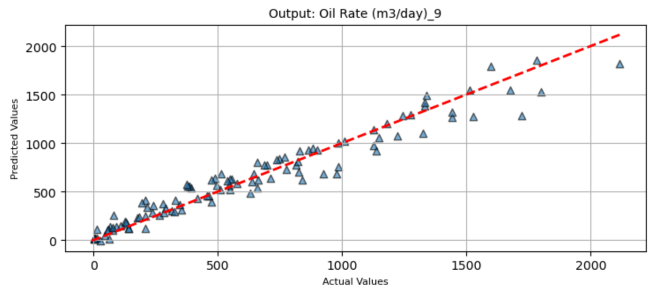


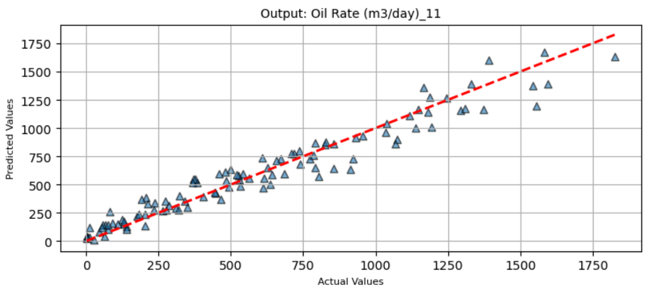












16-2. Analyzing Residual Plots to Identify Outlier Impact

These charts show the residuals for the experimental model (the one trained on data after outliers were removed). This analysis further confirms the previous conclusion that the main model, trained on all data, is the more robust and reliable choice.

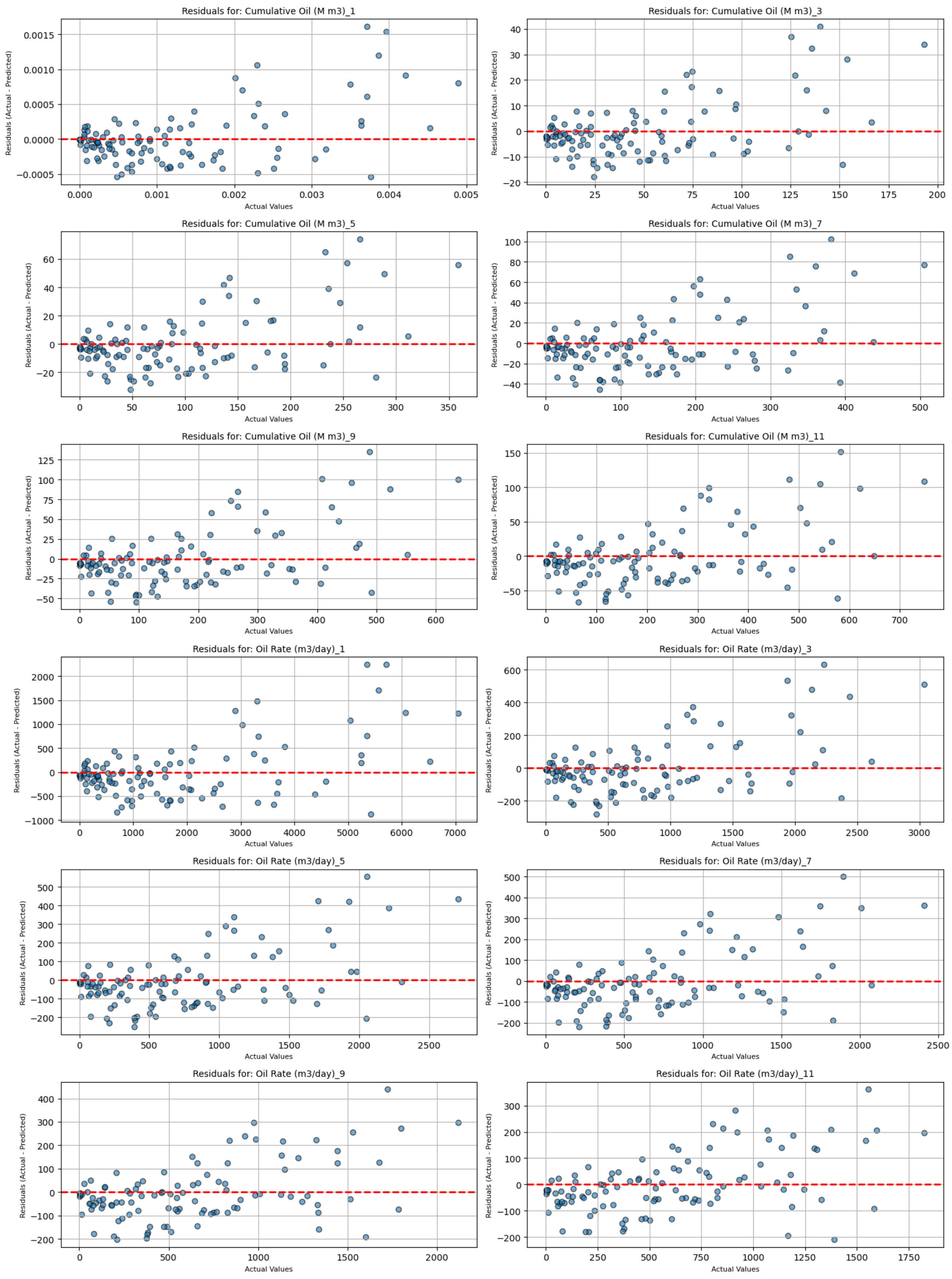
Analysis of the Plots

At first glance, because the extreme outlier samples have been removed, the vertical scatter of the errors in these plots appears smaller, and most points are contained within a tighter band around the zero-error line.

However, the distribution of the remaining residuals is not as perfectly random and patternless as it was for the main model. The subtle patterns in the scatter suggest that this experimental model, by not being exposed to the challenging outlier data during training, has not learned the underlying physical relationships as robustly as the main model.

Therefore, this final analysis reinforces the decision that the model trained on the complete dataset is the superior and more trustworthy model for this project.

In [77]:
```python
fig, axes = plt.subplots(6, 2, figsize=(15, 20))

axes = axes.flatten()

for i in range(12):
    ax = axes[i]

    actual_data = y_test[:, i]
    predicted_data = y_pred[:, i]

    residuals = actual_data - predicted_data

    ax.scatter(actual_data, residuals, alpha=0.6, edgecolors='k')

    ax.axhline(0, color='red', linestyle='--', linewidth=2)

    if 'target_cols' in locals():
        ax.set_title(f'Residuals for: {target_cols[i]}', fontsize=10)
    else:
        ax.set_title(f'Residuals for Output #{i+1}', fontsize=10)

    ax.set_xlabel('Actual Values', fontsize=8)
    ax.set_ylabel('Residuals (Actual - Predicted)', fontsize=8)
    ax.grid(True)

plt.tight_layout()

plt.show()
```



16-3. Analysis of the Error Distribution

The histogram shows that the majority of the errors are correctly centered around zero, which is desirable. However, it is also clear that the distribution has `long tails`, indicating that even after removing the initial outliers, a number of predictions still have a significantly high error. This confirms that the inherent complexity of the problem leads to challenging predictions for certain samples, regardless of the initial data cleaning.

Final Strategic Decision

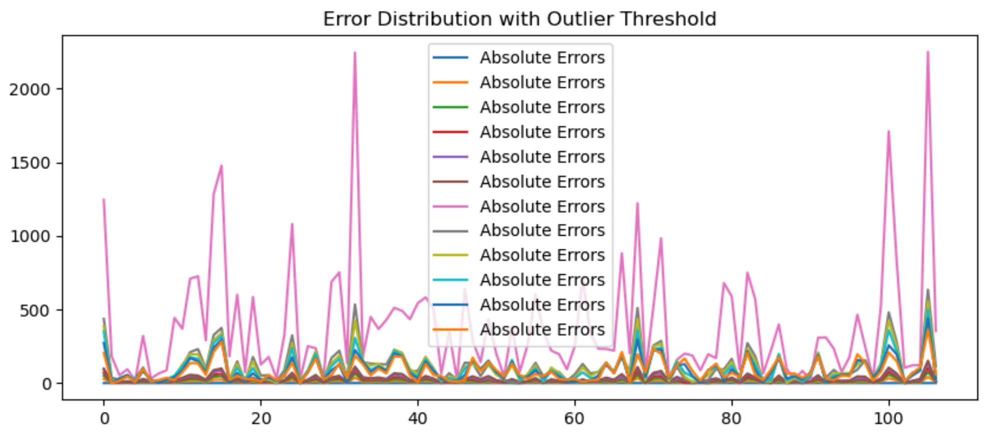
Based on the comprehensive analysis comparing the two modeling approaches, a final strategic decision was made:

1. Both approaches (the model trained on all data and the model trained on cleaned data) will be made available in the final `Streamlit` dashboard to allow for a complete comparison.

2. However, the model that was trained on the complete dataset, including all outliers, will be considered the primary reference model for this project.

The reason for this choice is that the primary model, despite being trained on more challenging data, ultimately achieved a higher overall accuracy (`97.5% (98.2%) vs 93.3%`). This demonstrates its superior ability to generalize and makes it a more robust and reliable tool for real-world scenarios where rare and unusual cases are expected.

In [79]:
```
errors = np.abs(y_test - y_pred)

plt.figure(figsize=(10, 4))
plt.plot(errors, label="Absolute Errors")
plt.title("Error Distribution with Outlier Threshold")
plt.legend()
plt.show()
```



16-4. Case Study: Analyzing Predictions for a Single Sample

While aggregate metrics like `MAE` and `R²` provide a high-level view of performance, it's also insightful to inspect individual predictions to get a more concrete understanding of the model's behavior.

To achieve this, a single sample (`sample_index = 35`) was randomly selected from the test set. The code below iterates through all 12 target variables for this sample and prints a side-by-side comparison of the Actual (true) values and the Predicted values generated by the model.

This tabular view provides a tangible example of the model's performance on a case-by-case basis. It allows us to see the magnitude of the prediction error for each specific time step in its original, physical units, offering a more granular perspective on the errors that are summarized by the overall `MAE` and `RMSE` scores.

In [78]:
```
sample_index = 35

print("Output Name".ljust(30), "Actual".ljust(15), "Predicted")
print("-" * 60)

for i, name in enumerate(target_cols):
    actual_val = y_test[sample_index, i]
    predicted_val = y_pred[sample_index, i]
    print(f"{name.ljust(30)} {str(round(actual_val, 2)).ljust(15)} {round(predicted_val, 2)}")
```

```
Output Name                    Actual          Predicted
------------------------------------------------------------
Cumulative Oil (M m3)_1        0.0             0.0
Cumulative Oil (M m3)_3        15.91           25.809999465942383
Cumulative Oil (M m3)_5        31.28           48.689998626708984
Cumulative Oil (M m3)_7        46.26           70.45999908447266
Cumulative Oil (M m3)_9        61.17           92.33999633789062
Cumulative Oil (M m3)_11       75.58           113.9000015258789
Oil Rate (m3/day)_1            459.04          826.1500244140625
Oil Rate (m3/day)_3            261.16          390.95001220703125
Oil Rate (m3/day)_5            251.95          384.8699951171875
Oil Rate (m3/day)_7            245.52          360.1199951171875
Oil Rate (m3/day)_9            240.45          351.8399963378906
Oil Rate (m3/day)_11           236.2           335.54998779296875
```

16-5. Performance Comparison and Conclusion

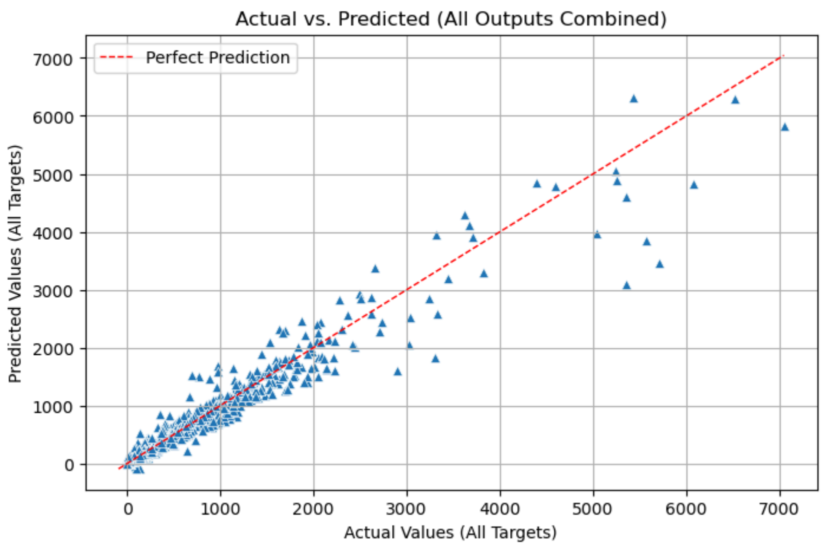
From above visuals the results are confirmed by the numerical results. This model achieved a final R^2 of `93.3%`, which is significantly lower than the `97.5%` accuracy of the main model trained on the complete dataset.

The reason for this is that by removing the outliers beforehand, the model was deprived of learning from realistic but rare high-production scenarios. As a result, its ability to generalize to these important edge cases was diminished.

Therefore, although this model's plots may appear `cleaner` at first glance, the results of the main model (trained on all data) are more reliable, more robust, and ultimately more accurate.

Even so, to complete the analysis of this experimental model, the corresponding residual and error distribution plots were also generated to gain a full understanding of its error behavior.

In [ ]:
```
y_test_flat = y_test.flatten()
y_pred_flat = y_pred.flatten()

plt.figure(figsize=(8, 5))
sns.scatterplot(x=y_test_flat, y=y_pred_flat, marker="^")

min_val = min(y_test_flat.min(), y_pred_flat.min())
max_val = max(y_test_flat.max(), y_pred_flat.max())
plt.plot([min_val, max_val], [min_val, max_val], 'r--', lw=1, label='Perfect Prediction')

plt.title('Actual vs. Predicted (All Outputs Combined)')
plt.xlabel('Actual Values (All Targets)')
plt.ylabel('Predicted Values (All Targets)')
plt.legend()
plt.grid(True)
plt.show()
```



Now lets save our model for using it in our `Streamlit` dashboard

In [32]:
```
model.save("model_outlier.keras")
```

Attachements

This cell will show you the attachements and helper section to find the best results.

1. Automated Hyperparameter Optimization with Optuna

To find the optimal combination of hyperparameters and achieve the highest possible model accuracy, the Optuna optimization framework was employed. This approach automates the complex and time-consuming process of manual tuning by performing an intelligent, guided search.

The mechanism is implemented as follows:

- Objective Function: The entire process of building, compiling, and training the model is encapsulated within an objective function. This function takes a trial object as an argument.
- Search Space Definition: Inside the objective function, a search space is defined for each hyperparameter (e.g., learning rate, dropout rate, number of filters in CNN layers, etc.). For each trial, the trial object suggests a new value for each parameter from its defined range.
- Training and Evaluation: The model is built and trained using the hyperparameters suggested by the current trial. At the end of training, the best validation loss (val_loss) is returned as the objective value.
- Optimization: Optuna runs the objective function for a predefined number of trials. It uses intelligent search algorithms to learn from the results of past trials, allowing it to focus on more promising regions of the hyperparameter space and efficiently converge toward the best possible combination.

This systematic approach allowed for the discovery of the optimal architecture and training parameters required to maximize the model's predictive accuracy.

A sample run of this optimization process is available in a Google Colab environment at the following link:

Link to Colab

In [ ]:
```
import optuna
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D, Flatten, concatenate, Dropout
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
```

```
def objective(trial):
    filters_c1 = trial.suggest_categorical('filters_c1', [32, 50, 64])
    filters_c2 = trial.suggest_categorical('filters_c2', [64, 100, 128])
    filters_c3 = trial.suggest_categorical('filters_c3', [128, 150, 200])
    filters_c4 = trial.suggest_categorical('filters_c4', [200, 256, 300])

    dense_units = trial.suggest_categorical('dense_units', [64, 128, 150])
    dropout_rate = trial.suggest_float('dropout_rate', 0.15, 0.4)
    l2_factor = trial.suggest_float('l2_factor', 1e-4, 1e-2, log=True)
    learning_rate = trial.suggest_float('learning_rate', 1e-4, 1e-2, log=True)
    optimizer_name = trial.suggest_categorical('optimizer', ['Adam', 'RMSprop', 'Nadam'])

    image_input = Input(shape=(64, 64, 2), name='image_input')
    cnn = Conv2D(filters=filters_c1, kernel_size=(3, 3), activation='selu', kernel_initializer='lecun_normal')(image_input)
    cnn = MaxPooling2D(pool_size=(2, 2))(cnn)
    cnn = Conv2D(filters=filters_c2, kernel_size=(3, 3), activation='selu', kernel_initializer='lecun_normal')(cnn)
    cnn = MaxPooling2D(pool_size=(2, 2))(cnn)
    cnn = Conv2D(filters=filters_c3, kernel_size=(3, 3), activation='selu', kernel_initializer='lecun_normal')(cnn)
    cnn = MaxPooling2D(pool_size=(2, 2))(cnn)
    cnn = Conv2D(filters=filters_c4, kernel_size=(3, 3), activation='selu', kernel_initializer='lecun_normal')(cnn)

    cnn_flatten = Flatten()(cnn)

    numerical_input = Input(shape=(1,), name='numerical_input')
    dense_num = Dense(units=8, activation='selu', kernel_initializer='lecun_normal')(numerical_input)

    combined_features = concatenate([cnn_flatten, dense_num])
    final_dense = Dense(units=dense_units, activation='selu', kernel_initializer='lecun_normal', kernel_regularizer=regularizers.l2(l2_factor))(combined_
    final_dense = Dropout(dropout_rate)(final_dense)

    output = Dense(units=12, activation='linear', name='output')(final_dense)
    model = Model(inputs=[image_input, numerical_input], outputs=output)

    if optimizer_name == 'Adam':
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif optimizer_name == 'RMSprop':
        optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate)
    else:
        optimizer = tf.keras.optimizers.Nadam(learning_rate=learning_rate)

    model.compile(optimizer=optimizer, loss='mean_squared_error')
    early_stopper = EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True, verbose=0)

    history = model.fit(
        x={'image_input': X_train_img_scaled, 'numerical_input': X_train_num_scaled},
        y=y_train_scaled,
        validation_data=({'image_input': X_val_img_scaled, 'numerical_input': X_val_num_scaled}, y_val_scaled),
        epochs=150,
        batch_size=32,
        callbacks=[early_stopper],
        verbose=0
    )

    best_val_loss = min(history.history['val_loss'])
    return best_val_loss

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=50)

print("\n" + "="*50)
print("Optuna Hyperparameter Tuning Finished.")
print("="*50)
print(f"Number of finished trials: {len(study.trials)}")
print("\nBest trial:")
best_trial = study.best_trial
print(f"  Value (minimized val_loss): {best_trial.value:.6f}")
print("  Best Parameters: ")
for key, value in best_trial.params.items():
    print(f"    {key}: {value}")
```

2. Saving the Training History

This code uses the pickle library to save the `history.history` dictionary, which contains the `loss` and `metrics` from each training epoch, into a file named `training_history_outlier.pkl`. This file is saved so that the training history can be loaded and visualized later in the `Streamlit` dashboard.

In [ ]:
```
import pickle

history_filename = 'training_history_outlier.pkl'

with open(history_filename, 'wb') as file:
    pickle.dump(history.history, file)

print(f"Training history successfully saved to: {history_filename}")
```

```
Training history successfully saved to: training_history_outlier.pkl
```

3. Generating Predictions for the Entire Dataset

As a final step, this cell runs the fully trained model on the entire dataset (`combining all training, validation, and test samples`). The goal is to generate a complete prediction set for every sample, which can be used for a comprehensive final review or for direct comparison with the original ground truth data.

The process involves the following steps:

- The saved Keras model and the fitted scaler objects are loaded from disk.

- The complete set of image and numerical inputs are scaled using these loaded objects to ensure consistent preprocessing.
- The `model.predict()` method is called on the entire prepared dataset.
- The scaled predictions are inverse-transformed back to their original, physical units.
- A final post-processing step clips any physically impossible negative predictions to zero.

The resulting predictions for all 12 target variables are then compiled into a pandas DataFrame and saved to an Excel file named `out_dataset_predictions.xlsx` for future analysis.

In [ ]:
```python
X_numerical_all = data_pivot['Initial Sw'].values.reshape(-1, 1)
sample_numbers = data_pivot['sample number'].values

model = keras.models.load_model('model_outlier.keras')

X_numerical_scaled_all = num_scaler.transform(X_numerical_all)

X_image_scaled_all = X_image.copy()
if (perm_max - perm_min) != 0:
    X_image_scaled_all[:, :, :, 0] = (X_image_scaled_all[:, :, :, 0] - perm_min) / (perm_max - perm_min)
if (poro_max - poro_min) != 0:
    X_image_scaled_all[:, :, :, 1] = (X_image_scaled_all[:, :, :, 1] - poro_min) / (poro_max - poro_min)
X_image_scaled_all = np.nan_to_num(X_image_scaled_all)

y_pred_s = model.predict({
    'image_input': X_image_scaled_all,
    'numerical_input': X_numerical_scaled_all
})

predictions_original = y_scaler.inverse_transform(y_pred_s)

predictions_original[predictions_original < 0] = 0

target_cols = [col for col in data_pivot.columns if col not in ['sample number', 'Initial Sw']]
df_predictions = pd.DataFrame(predictions_original, columns=target_cols)
df_predictions.insert(0, 'sample number', sample_numbers)

pd.DataFrame(df_predictions).head()
df_predictions.to_excel('out_dataset_predictions.xlsx', index=False)
print("Predictions saved to 'out_dataset_predictions.xlsx'")
```

```
23/23 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step
Predictions saved to 'out_dataset_predictions.xlsx'
```

You can see the result of predicting for entire dataset below:

In [85]:
```python
pd.DataFrame(df_predictions).head()
```

Out[85]:

	sample number	Cumulative Oil (M m3)_1	Cumulative Oil (M m3)_3	Cumulative Oil (M m3)_5	Cumulative Oil (M m3)_7	Cumulative Oil (M m3)_9	Cumulative Oil (M m3)_11	Oil Rate (m3/day)_1	Oil Rate (m3/day)_3	Oil Rate (m3/day)_5	Oil Rate (m3/day)_7	Oil Rate (m3/day)_9	Oil Rate (m3/day)_11
0	1	0.000458	21.761353	43.277096	66.350441	86.987038	109.675888	666.768127	357.678406	371.771240	355.176971	349.267975	341.194244
1	2	0.002857	110.394875	205.864746	292.679260	363.907715	435.864349	4223.372070	1717.343750	1524.597778	1375.816528	1244.735962	1122.145020
2	3	0.001163	45.193005	87.518379	123.997398	164.040939	199.954071	1686.332520	728.713013	689.183411	640.797607	604.829895	552.322571
3	4	0.003638	136.712463	263.539703	371.856598	464.365417	557.227966	5173.840332	2195.350342	1967.209839	1808.294434	1618.734131	1481.748535
4	5	0.000202	9.909275	19.406796	30.010529	40.593971	48.246876	301.837097	162.653900	173.060043	171.330658	161.463577	154.872009

Checking shape of dataset due to the errors which has been solved

In [69]:
```python
print(X_numerical_all.shape)
print(sample_numbers.shape)
print(X_image_scaled_all.shape)
```

```
(708, 1)
(708,)
(708, 64, 64, 2)
```

4. Saving Preprocessing Objects for the Dashboard

This cell uses the joblib library to save the trained scalers and calculated image scaling parameters to disk.

These saved files are essential for the `Live Prediction` feature of the Streamlit dashboard. The dashboard will load these objects to:

- Scale new user inputs (the uploaded images and the Initial Sw value) using the exact same parameters that were learned from the original training data.
- Inverse-transform the model's scaled predictions back into their real-world, interpretable units.
- The scale factors are not same to each other (First approach)

In [75]:
```python
import joblib
image_scaling_params = {
    'perm_min_out': perm_min,
    'perm_max_out': perm_max,
    'poro_min_out': poro_min,
    'poro_max_out': poro_max
}
joblib.dump(y_scaler, 'y_scaler_out.gz')
joblib.dump(num_scaler, 'num_scaler_out.gz')
joblib.dump(image_scaling_params, 'image_param_out.gz')
```

```
Out[75]: ['image_param_out.gz']
```