

# Long Short Term Memory Networks for IoT Prediction

RNNs and LSTM models are very popular neural network architectures when working with sequential data, since they both carry some "memory" of previous inputs when predicting the next output. In this assignment we will continue to work with the Household Electricity Consumption dataset and use an LSTM model to predict the Global Active Power (GAP) from a sequence of previous GAP readings. You will build one model following the directions in this notebook closely, then you will be asked to make changes to that original model and analyze the effects that they had on the model performance. You will also be asked to compare the performance of your LSTM model to the linear regression predictor that you built in last week's assignment.

## General Assignment Instructions

These instructions are included in every assignment, to remind you of the coding standards for the class. Feel free to delete this cell after reading it.

One sign of mature code is conforming to a style guide. We recommend the [Google Python Style Guide](#). If you use a different style guide, please include a cell with a link.

Your code should be relatively easy-to-read, sensibly commented, and clean. Writing code is a messy process, so please be sure to edit your final submission. Remove any cells that are not needed or parts of cells that contain unnecessary code. Remove inessential `import` statements and make sure that all such statements are moved into the designated cell.

When you save your notebook as a pdf, make sure that all cell output is visible (even error messages) as this will aid your instructor in grading your work.

Make use of non-code cells for written commentary. These cells should be grammatical and clearly written. In some of these cells you will have questions to answer. The questions will be marked by a "Q:" and will have a corresponding "A:" spot for you. *Make sure to answer every question marked with a `Q:` for full credit.*

```
In [ ]: import keras
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os

# Setting seed for reproducibility
np.random.seed(1234)
PYTHONHASHSEED = 0
```

```

from sklearn import preprocessing
from sklearn.metrics import confusion_matrix, recall_score, precision_score
from sklearn.model_selection import train_test_split
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, LSTM, Activation
from keras.utils import pad_sequences

from google.colab import drive

```

```

In [ ]: #use this cell to import additional libraries or define helper functions
from tensorflow.keras.preprocessing.sequence import pad_sequences
from keras.layers import Conv1D, MaxPooling1D, LSTM, Dense, Dropout, BatchNormaliza
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import Huber

```

## Load and prepare your data

We'll once again be using the cleaned household electricity consumption data from the previous two assignments. I recommend saving your dataset by running `df.to_csv("filename")` at the end of assignment 2 so that you don't have to re-do your cleaning steps. If you are not confident in your own cleaning steps, you may ask your instructor for a cleaned version of the data. You will not be graded on the cleaning steps in this assignment, but some functions may not work if you use the raw data.

Unlike when using Linear Regression to make our predictions for Global Active Power (GAP), LSTM requires that we have a pre-trained model when our predictive software is shipped (the ability to iterate on the model after it's put into production is another question for another day). Thus, we will train the model on a segment of our data and then measure its performance on simulated streaming data another segment of the data. Our dataset is very large, so for speed's sake, we will limit ourselves to 1% of the entire dataset.

**TODO: Import your data, select the a random 1% of the dataset, and then split it 80/20 into training and validation sets (the test split will come from the training data as part of the tensorflow LSTM model call). HINT: Think carefully about how you do your train/validation split--does it make sense to randomize the data?**

```

In [ ]: #Load your data into a pandas dataframe here
# Answer:
#Mount Google Drive to access files stored there
drive.mount('/content/drive')

# Load the dataset from Google Drive into a Pandas DataFrame
# Specify the path to the .txt file and set the delimiter to ';' for correct parsin

file_path = '/content/drive/MyDrive/AAI-530-DataAnalyticsAndIOT/household_power_cle

# Read the .txt file into a DataFrame
df = pd.read_csv(file_path, parse_dates=['Datetime'])

```

```
# Display the first few rows of the DataFrame to verify it loaded correctly
print(df.head())
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force\_remount=True).

```

      Unnamed: 0      Date      Time  Global_active_power  \
0              0  2006-12-16  17:24:00                4.216
1              1  2006-12-16  17:25:00                5.360
2              2  2006-12-16  17:26:00                5.374
3              3  2006-12-16  17:27:00                5.388
4              4  2006-12-16  17:28:00                3.666

      Global_reactive_power  Voltage  Global_intensity  Sub_metering_1  \
0                0.418      234.84                18.4              0.0
1                0.436      233.63                23.0              0.0
2                0.498      233.29                23.0              0.0
3                0.502      233.74                23.0              0.0
4                0.528      235.68                15.8              0.0

      Sub_metering_2  Sub_metering_3      Datetime  gap_monthly  \
0              1.0              17.0  2006-12-16  17:24:00      NaN
1              1.0              16.0  2006-12-16  17:25:00      NaN
2              2.0              17.0  2006-12-16  17:26:00      NaN
3              1.0              17.0  2006-12-16  17:27:00      NaN
4              1.0              17.0  2006-12-16  17:28:00      NaN

      grp_monthly  v_monthly  gi_monthly
0              NaN        NaN        NaN
1              NaN        NaN        NaN
2              NaN        NaN        NaN
3              NaN        NaN        NaN
4              NaN        NaN        NaN

```

```

In [ ]: #create your training and validation sets here

# Sort the data by Datetime to ensure it's in chronological order
df = df.sort_values('Datetime')

# Assign size for data subset (1% of the entire dataset)
subset_size = int(0.01 * len(df))

# Take random data subset
#n=subset_size: This argument specifies the number of rows needed in random subset.
#random_state=42: This is crucial for reproducibility. It sets the seed for the random
#               Incase this code is run multiple times with the same random_state, this will
#               debugging, and sharing results. IF this argument is None, then the same
df_subset = df.sample(n=subset_size, random_state=42)

# Sort the subset by Datetime again to maintain chronological order
df_subset = df_subset.sort_values('Datetime')

# Split data subset 80/20 for train/validation
# Split the df_subset which is a random sample of the original DataFrame, its index
# We use the index to split to maintain the time order
#len(df_subset): Gets the number of rows in the df_subset.
#0.8 * len(df_subset): Calculates 80% of the number of rows. This determines the split

```

```

#int(...): Converts the result to an integer. This is necessary because slice indic
#split_index: Stores the calculated index value. This index marks the boundary betw

split_index = int(0.8 * len(df_subset))

# This creates the training set. It takes all rows from df_subset up to (but not in
# Because the original index from the dataframe is used, and because the subset was
# this subset will maintain the time order.
# train_df: Stores the resulting training DataFrame

train_df = df_subset[:split_index]

# Create the validation set. It takes all rows from df_subset starting from the row
val_df = df_subset[split_index:]

```

```

In [ ]: #reset the indices for cleanliness
        #After splitting the data, the original index from the full dataset is retained. Re
        #This makes it easier to work with the data, especially when iterating over rows or
train_df = train_df.reset_index()
val_df = val_df.reset_index()

print(f"Training set shape: {train_df.shape}")
print(f"Validation set shape: {val_df.shape}")

```

Training set shape: (16393, 16)  
Validation set shape: (4099, 16)

Next we need to create our input and output sequences. In the lab session this week, we used an LSTM model to make a binary prediction, but LSTM models are very flexible in what they can output: we can also use them to predict a single real-numbered output (we can even use them to predict a sequence of outputs). Here we will train a model to predict a single real-numbered output such that we can compare our model directly to the linear regression model from last week.

**TODO: Create a nested list structure for the training data, with a sequence of GAP measurements as the input and the GAP measurement at your predictive horizon as your expected output**

```

In [ ]: seq_arrays = []
        seq_labs = []

```

```

In [ ]: # Get GAP readings from your DataFrame column
gap_readings = train_df['Global_active_power'].values # Direct numpy array access

#use 30 consecutive measurements to predict value 5 steps in future
seq_length = 30 # 30-minute window
ph = 5 # Predict 5 minutes ahead
feat_cols = ['Global_active_power'] # Define feature columns

# Create sequences with alignment for predictive horizon
seq_arrays = []
seq_labs = []

```

```

# Calculate valid range for sequence creation

# Total Data Points: len(gap_readings) = 16,359 historical GAP measurements.
#Sequence Length: Each input sequence uses 30 consecutive measurements.
#Predictive Horizon: To predict the GAP value 5 steps ahead of the input sequence.
#Input Sequence: Each sequence starts at index i and ends at i + seq_length - 1.
#Output Value: Predicted value is at index i + seq_length + ph - 1.
#Last Valid Sequence:
#Start index i must satisfy:
#i + seq_length + ph - 1 < len(gap_readings)
#For the dataset with len = 16359:
# i + 30 + 5 - 1 < 16,359 → i < 16,359 - 34 → i ≤ 16,324
# Total sequences: 16,325 (i ranges from 0 to 16,324).

num_samples = len(gap_readings) - seq_length - ph + 1

#Sequence creation loop
#Input Sequence: Sliding window of 30 consecutive values
#Output Value: Value 5 steps after the end of the input window

for i in range(num_samples):
    # Input: 30 consecutive GAP measurements
    input_seq = gap_readings[i:i+seq_length]

    # Output: GAP value 5 cycles after sequence end
    output_val = gap_readings[i + seq_length + ph - 1]

    # append(input_seq): adds the current input_seq (which is itself a sequence of
    # So, seq_arrays will become a list of lists (or a list of arrays).
    #input_seq is a sequence of 30 consecutive GAP measurements (because seq_length
    #seq_arrays is a list that will contain all these sequences.
    #.append() adds input_seq as a new element to the end of seq_arrays.
    #After this operation, seq_arrays becomes a list of lists, where each inner list
    # Example:
    # seq_arrays = [
    # [0.5, 0.6, ..., 0.8], # First sequence (30 values)
    # [0.6, 0.7, ..., 0.9], # Second sequence (30 values)
    # ...
    seq_arrays.append(input_seq) #store the input sequence of time series data in n

    # add the output_val to the seq_labs list
    # output_val is a single GAP measurement, 5 steps ahead of the end of the input
    # seq_labs is a list that will contain all these future values.
    # append(): This adds output_val as a new element to the end of seq_labs.
    # After this operation, seq_labs becomes a list of individual GAP values, each
    # Example:
    # python
    # seq_labs = [0.7, 0.8, 0.9, ...] # Each value corresponds to a sequence in seq_arrays
    seq_labs.append(output_val)

# Convert to numpy arrays with proper typing
seq_arrays = np.array(seq_arrays, dtype=np.float32).reshape(-1, seq_length, len(features))
seq_labs = np.array(seq_labs, dtype=np.float32)

```

```

print("Output shape:", seq_labs.shape)

# Verify dimensions with corrected calculation
expected_samples = len(train_df) - seq_length - ph + 1
expected_shape = (expected_samples, seq_length, len(feats_cols))

try:
    assert seq_arrays.shape == expected_shape, (
        f"Sequence array shape mismatch. Got {seq_arrays.shape}, "
        f"expected {expected_shape}. Check data length: {len(train_df)} "
        f"should be ≥ {seq_length + ph}"
    )

    assert seq_labs.shape == (expected_samples,), (
        f"Label shape mismatch. Got {seq_labs.shape}, "
        f"expected ({expected_samples},)"
    )

    print("All assertions passed!")
except AssertionError as e:
    print(f"Assertion Error: {e}")
    print(f"Required data length: {seq_length + ph - 1}")
    print(f"Actual data length: {len(train_df)}")

```

Input shape: (16359, 30, 1)

Output shape: (16359,)

All assertions passed!

```

In [ ]: assert(seq_arrays.shape == (len(train_df)-seq_length-ph+1,seq_length, len(feats_cols)
assert(seq_labs.shape == (len(train_df)-seq_length-ph+1,))

```

```

In [ ]: seq_arrays.shape

```

```

Out[ ]: (16359, 30, 1)

```

**Q: What is the function of the assert statements in the above cell? Why do we use assertions in our code?**

A: Statement "assert" in the code is required to verify

1. Input Shape Validity: Ensures 3D tensor shape matches LSTM requirements:

(number\_of\_sequences, sequence\_length, features\_per\_timestep) 2. Confirms each sequence has exactly one corresponding label, in this case exactly 16359 Sequences and 16359 labels

## Reason why assertion is needed in the code

1. LSTM Input requirement: The model expects input in (samples, time\_steps, features) format. An incorrect shape will cause immediate failure during training.
2. Data Consistency:

Ensures sequences and labels are properly aligned (no mismatched samples). 3. Error Prevention: Catches issues early before they propagate through training: Wrong sequence length

In this case Correct Calculation ( $\text{len(df)} - \text{seq\_length} - \text{ph} + 1$ ):  $16359 - 30 - 5 + 1 = 16325$  samples

## Model Training

We will begin with a model architecture very similar to the model we built in the lab session. We will have two LSTM layers, with 5 and 3 hidden units respectively, and we will apply dropout after each LSTM layer. However, we will use a LINEAR final layer and MSE for our loss function, since our output is continuous instead of binary.

**TODO: Fill in all values marked with a ?? in the cell below**

```
In [ ]: # define path to save model
model_path = 'LSTM_model1.keras'

# build the network
# Onfe Feature which is GAP and one Output value (GAP) is predicted
nb_features = 1
nb_out = 1

# Initialize a Sequential model
model = Sequential()

#add first LSTM layer
model.add(Input(shape=(seq_length, nb_features))) # Explicit input layer
model.add(LSTM(units=5, return_sequences=True))

model.add(Dropout(0.2)) # Return full sequence for next LSTM layer

# add second LSTM layer
model.add(LSTM(
    units=3, #number of LSTM units in this layer 2 of LSTM
    return_sequences=False)) # do not return
# Add dropout for regularization
model.add(Dropout(0.2))

# Add Dense (fully connected) output layer
model.add(Dense(units=nb_out)) # Output layer with 1 unit for regression
model.add(Activation("linear")) # Linear activation for regression

# Define optimizer with Learning rate
optimizer = keras.optimizers.Adam(learning_rate = 0.01)

#Loss function is MSE, Adam optimizer, track MSE during training
# uses Mean Squared Error as both the loss function and a metric, appropriate for r
model.compile(loss='mean_squared_error', optimizer=optimizer, metrics=['mse'])
```

```

# fit the network (train it)
# max number of training epochs = 100
# number of samples per gradient update (batch size) = 500
# use 5% of data for validation
# verbosity mode = 2, one line per epoch
# callback - early stopping to prevent overfitting and if no more learning is happen
# monitor='val_loss', # Monitor validation loss
# min_delta=0: Minimum change to qualify as improvement
# patience=10: Number of epochs with no improvement after which training will stop
# verbose=0: Verbosity mode
# mode='min' # The direction is automatically inferred if not set

# Model checkpoint to save best model
# model_path, # Path to save the model
# monitor='val_loss', # Monitor validation loss
# save_best_only=True, # Only save when the model is better than the previous best
# mode='min', # Lower validation loss is better
# verbose=0 # Verbosity mode

history = model.fit(seq_arrays, seq_labels, epochs=100, batch_size=500, validation_split=0.05,
                    callbacks = [keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0.001, patience=10),
                                keras.callbacks.ModelCheckpoint(model_path, monitor='val_loss', save_best_only=True)]
                    )

# List all data in history
# After training, it prints the keys of the history object, which typically include
print(history.history.keys())

```

Model: "sequential\_21"

Layer (type)	Output Shape	
lstm_25 (LSTM)	(None, 30, 5)	
dropout_38 (Dropout)	(None, 30, 5)	
lstm_26 (LSTM)	(None, 3)	
dropout_39 (Dropout)	(None, 3)	
dense_20 (Dense)	(None, 1)	
activation_6 (Activation)	(None, 1)	



Total params: 252 (1008.00 B)

Trainable params: 252 (1008.00 B)

Non-trainable params: 0 (0.00 B)



None

Epoch 1/100  
 32/32 - 6s - 202ms/step - loss: 1.4818 - mse: 1.4818 - val\_loss: 1.6954 - val\_mse: 1.6954

Epoch 2/100  
 32/32 - 1s - 30ms/step - loss: 1.2129 - mse: 1.2129 - val\_loss: 1.5535 - val\_mse: 1.5535

Epoch 3/100  
 32/32 - 1s - 39ms/step - loss: 1.1931 - mse: 1.1931 - val\_loss: 1.5414 - val\_mse: 1.5414

Epoch 4/100  
 32/32 - 1s - 31ms/step - loss: 1.1806 - mse: 1.1806 - val\_loss: 1.5408 - val\_mse: 1.5408

Epoch 5/100  
 32/32 - 1s - 38ms/step - loss: 1.1575 - mse: 1.1575 - val\_loss: 1.4970 - val\_mse: 1.4970

Epoch 6/100  
 32/32 - 1s - 44ms/step - loss: 1.1504 - mse: 1.1504 - val\_loss: 1.4888 - val\_mse: 1.4888

Epoch 7/100  
 32/32 - 3s - 81ms/step - loss: 1.1495 - mse: 1.1495 - val\_loss: 1.4942 - val\_mse: 1.4942

Epoch 8/100  
 32/32 - 1s - 35ms/step - loss: 1.1459 - mse: 1.1459 - val\_loss: 1.5021 - val\_mse: 1.5021

Epoch 9/100  
 32/32 - 1s - 34ms/step - loss: 1.1389 - mse: 1.1389 - val\_loss: 1.5098 - val\_mse: 1.5098

Epoch 10/100  
 32/32 - 1s - 38ms/step - loss: 1.1342 - mse: 1.1342 - val\_loss: 1.5182 - val\_mse: 1.5182

Epoch 11/100  
 32/32 - 1s - 30ms/step - loss: 1.1296 - mse: 1.1296 - val\_loss: 1.4647 - val\_mse: 1.4647

Epoch 12/100  
 32/32 - 1s - 41ms/step - loss: 1.1257 - mse: 1.1257 - val\_loss: 1.4505 - val\_mse: 1.4505

Epoch 13/100  
 32/32 - 1s - 38ms/step - loss: 1.1218 - mse: 1.1218 - val\_loss: 1.4965 - val\_mse: 1.4965

Epoch 14/100  
 32/32 - 1s - 38ms/step - loss: 1.1221 - mse: 1.1221 - val\_loss: 1.5011 - val\_mse: 1.5011

Epoch 15/100  
 32/32 - 1s - 30ms/step - loss: 1.1223 - mse: 1.1223 - val\_loss: 1.4758 - val\_mse: 1.4758

Epoch 16/100  
 32/32 - 1s - 28ms/step - loss: 1.1201 - mse: 1.1201 - val\_loss: 1.4563 - val\_mse: 1.4563

Epoch 17/100  
 32/32 - 1s - 37ms/step - loss: 1.1198 - mse: 1.1198 - val\_loss: 1.4800 - val\_mse: 1.4800

Epoch 18/100  
 32/32 - 1s - 47ms/step - loss: 1.1187 - mse: 1.1187 - val\_loss: 1.4606 - val\_mse: 1.4606

Epoch 19/100  
 32/32 - 1s - 48ms/step - loss: 1.1187 - mse: 1.1187 - val\_loss: 1.4606 - val\_mse: 1.4606

```

32/32 - 1s - 46ms/step - loss: 1.1161 - mse: 1.1161 - val_loss: 1.4931 - val_mse: 1.4931
Epoch 20/100
32/32 - 2s - 64ms/step - loss: 1.1156 - mse: 1.1156 - val_loss: 1.4432 - val_mse: 1.4432
Epoch 21/100
32/32 - 1s - 29ms/step - loss: 1.1223 - mse: 1.1223 - val_loss: 1.4729 - val_mse: 1.4729
Epoch 22/100
32/32 - 1s - 39ms/step - loss: 1.1144 - mse: 1.1144 - val_loss: 1.4684 - val_mse: 1.4684
Epoch 23/100
32/32 - 1s - 29ms/step - loss: 1.1158 - mse: 1.1158 - val_loss: 1.4647 - val_mse: 1.4647
Epoch 24/100
32/32 - 1s - 40ms/step - loss: 1.1139 - mse: 1.1139 - val_loss: 1.4574 - val_mse: 1.4574
Epoch 25/100
32/32 - 1s - 28ms/step - loss: 1.1165 - mse: 1.1165 - val_loss: 1.4753 - val_mse: 1.4753
Epoch 26/100
32/32 - 1s - 39ms/step - loss: 1.1161 - mse: 1.1161 - val_loss: 1.4580 - val_mse: 1.4580
Epoch 27/100
32/32 - 1s - 30ms/step - loss: 1.1156 - mse: 1.1156 - val_loss: 1.4806 - val_mse: 1.4806
Epoch 28/100
32/32 - 1s - 44ms/step - loss: 1.1146 - mse: 1.1146 - val_loss: 1.4786 - val_mse: 1.4786
Epoch 29/100
32/32 - 1s - 45ms/step - loss: 1.1127 - mse: 1.1127 - val_loss: 1.4845 - val_mse: 1.4845
Epoch 30/100
32/32 - 2s - 78ms/step - loss: 1.1148 - mse: 1.1148 - val_loss: 1.4817 - val_mse: 1.4817
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])

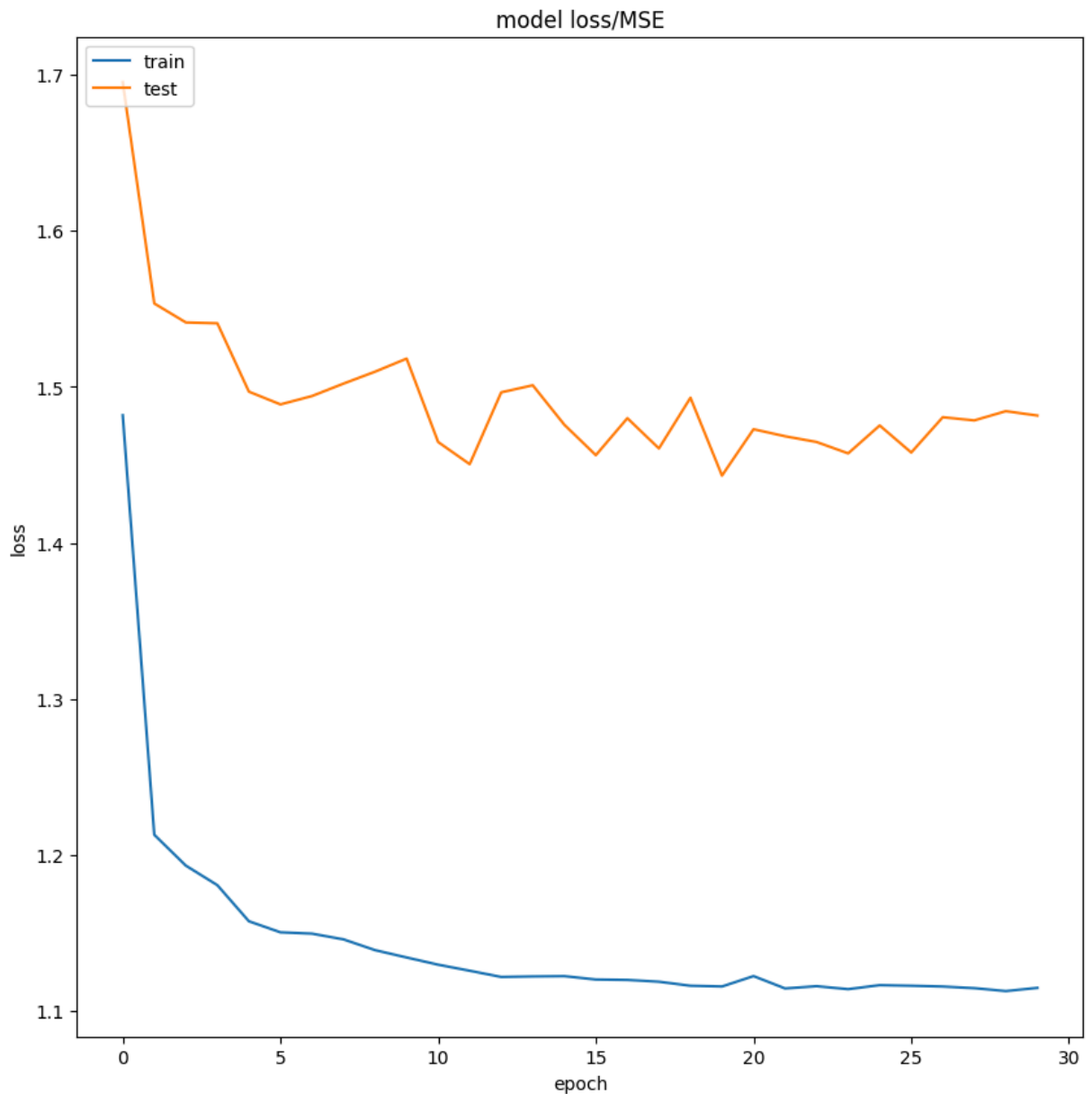
```

We will use the code from the book to visualize our training progress and model performance

```

In [ ]: # summarize history for Loss/MSE
fig_acc = plt.figure(figsize=(10, 10))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss/MSE')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
fig_acc.savefig("LSTM_loss1.png")

```



## Validating our model

Now we need to create our simulated streaming validation set to test our model "in production". With our linear regression models, we were able to begin making predictions with only two datapoints, but the LSTM model requires an input sequence of *seq\_length* to make a prediction. We can get around this limitation by "padding" our inputs when they are too short.

**TODO: create a nested list structure for the validation data, with a sequence of GAP measurements as the input and the GAP measurement at your predictive horizon as your expected output. Begin your predictions after only two GAP measurements are available, and check out [this keras function](#) to automatically pad sequences that are too short.**

**Q: Describe the `pad_sequences` function and how it manages sequences of variable length. What does the "padding" argument determine, and which setting makes the most sense for our use case here?**

A: The `pad_sequences` function is a utility in Keras/TensorFlow that standardizes sequences of varying lengths into uniform-length sequences. The `pad_sequences` function is a utility in Keras/TensorFlow that standardizes sequences of varying lengths into uniform-length sequences. Here's a breakdown:

## I. What `pad_sequences` Does:

### 1. Handles Variable-Length Sequences:

- a) If sequences are shorter than a specified `maxlen`, it **pads** them with a default value (e.g., `0`).
- b) If sequences are longer than `maxlen`, it **truncates** them to fit.

### 2. Key Parameters:

- a) `maxlen`: Forces all sequences to this length.
  - b) `padding`: Determines where to add padding:
    - `'pre'` (default): Adds padding **at the beginning** of sequences.
    - `'post'`: Adds padding **at the end** of sequences.
  - c) `truncating`: Determines where to truncate sequences if they exceed `maxlen`:
    - `'pre'`: Removes values from the **beginning** of sequences.
    - `'post'`: Removes values from the **end** of sequences.
  - d) `value`: The value used for padding (default is `0`).
  - e) `dtype`: Data type of the output (e.g., `float32`).
- 

## II. Why `padding='pre'` Makes Sense for Time Series (Your Use Case):

### 1. Preserves Temporal Order:

- a) Time series data is sequential, where the **most recent values** are often the most relevant for prediction.
- b) `padding='pre'` adds zeros to the **beginning** of short sequences, keeping the actual data at the **end**:  
Example (`maxlen=5`):  
Original: `[0.5, 0.6]` → Padded: `[0, 0, 0, 0.5, 0.6]`
- c) This ensures the model sees the most recent data in the same position as during training.

### 2. Simulates Real-Time Streaming:

- a) When starting predictions with only 2 data points, `padding='pre'` mimics gradually filling the sequence with new data:  
 Step 1: `[0, 0, 0, 0.5, 0.6]` # First prediction with 2 values  
 Step 2: `[0, 0, 0.5, 0.6, 0.55]` # After a new value arrives

### 3. Avoids Misleading the Model:

- a) Padding at the end ( `'post'` ) would place zeros **after** the actual data, which could confuse the model into thinking future values are zeros.

## III. Examples

### A. Example Without Padding: If sequences are variable-length:

Sequence 1: `[0.5, 0.6]`  
 Sequence 2: `[0.5, 0.6, 0.55]`

The LSTM model would crash because it expects fixed-length input.

### B. Example With Padding\*\*:

After `pad_sequences(padding='pre', maxlen=5) :`

Sequence 1: `[0, 0, 0, 0.5, 0.6]`  
 Sequence 2: `[0, 0, 0.5, 0.6, 0.55]`

```
### IV **When to Use `padding='post'`**:
- For non-temporal data (e.g., text sentences) where the start
  of the sequence matters more than the end.
- Example: Padding sentences to a fixed length for NLP models.
```

---

```
### **Conclusion**:
For the Global Active Power prediction use case with time-series
data, `padding='pre'` is optimal because:
1. It Preserves the temporal order of recent values.
2. It aligns with how real-time data streams into the model (new
  data appends to the end).
3. Avoids introducing misleading zeros at the end of sequences.
```

```
In [ ]: # Create empty lists to store:
        # val_arrays: Input sequences (varying lengths initially)
        # val_labs: Target values (GAP measurements at ph steps ahead)

        val_arrays = []
        val_labs = []
```

```

# It simulates a scenario where we start with minimal data (2 readings) and gradual

# Create list of GAP readings - sequence starting with a minimum of two readings(si
# Start with 2 readings:
# First iteration (i=2): Sequence = [val_0, val_1]
# Next iteration (i=3): Sequence = [val_0, val_1, val_2]
# And continue ... up to seq_length measurements.
# Label Alignment:
# For each sequence ending at index i, the label is the value at i + ph - 1
# Example: If ph=5, a sequence ending at i=10 predicts the value at index 14.
# Edge Handling:
# Breaks the loop if there's insufficient data for the label.

for i in range(2, len(val_df['Global_active_power'])):
    # Get all available readings up to current index

    # Collect historical data
    # For each step i, this grabs all readings up to but not including the current
    raw_seq = val_df['Global_active_power'].values[:i] # Get readings from start t

    # Get Label (ph steps ahead) - The label is the reading ph steps ahead of the c
    # Prediction Target(i=4, ph=3): time's GAP(i=4 + 3 - 1 = 6)
    # Result:
    # val_arrays = [[0.5, 0.6], [0.5, 0.6, 0.55], [0.5, 0.6, 0.55, 0.7]]
    # val_labs = [0.65, 0.75, 0.8] (i=4+3-1 => 6's actual GAP)

    if i + ph - 1 < len(val_df): # Check if we have enough data for the label
        val_labs.append(val_df['Global_active_power'].values[i + ph - 1])
    else: # Handle edge case at end of dataset
        break

    val_arrays.append(raw_seq)

# Pad sequences to uniform length (seq_length) with pre-padding (zeros at beginning
# Forces all sequences to have the same length (seq_length, e.g., 30).
# Short sequences get 0s added at the beginning
val_arrays = pad_sequences(
    val_arrays,
    maxlen=seq_length, # Force all sequences to be seq_length long
    dtype='float32',    # Match TensorFlow's default dtype
    padding='pre',      # Add zeros at BEGINNING of sequences (preserve recent dat
    truncating='pre',   # Truncate from beginning if sequences are too long (shoul
    value=0             # Use 0 for padding (consider using mean value if better f
)

# Convert labels to numpy array with proper dtype
val_labs = np.array(val_labs, dtype=np.float32)

print("Padded validation sequences shape:", val_arrays.shape)
print("Validation labels shape:", val_labs.shape)

print("First index of val_array", val_arrays[0])
print("val_array", val_arrays)

print("Val_labs", val_labs)

```

```

Padded validation sequences shape: (4093, 30)
Validation labels shape: (4093,)
First index of val_array [0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
0.    0.
0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
0.    0.    0.    0.    0.326 1.962]
val_array [[0.    0.    0.    ... 0.    0.326 1.962]
[0.    0.    0.    ... 0.326 1.962 3.386]
[0.    0.    0.    ... 1.962 3.386 2.616]
...
[1.406 1.392 1.304 ... 1.266 1.338 1.468]
[1.392 1.304 1.466 ... 1.338 1.468 1.444]
[1.304 1.466 1.388 ... 1.468 1.444 1.276]]
Val_labs [2.724 1.94  1.944 ... 1.884 1.694 1.766]

```

We will now run this validation data through our LSTM model and visualize its performance like we did on the linear regression data.

```

In [ ]: print("Current val_arrays shape:", val_arrays.shape)

val_arrays = val_arrays.reshape(val_arrays.shape[0], seq_length, 1)
print("Reshaped val_arrays shape:", val_arrays.shape)

# Reshape validation data
val_arrays = val_arrays.reshape(val_arrays.shape[0], seq_length, 1)
print("Reshaped val_arrays shape:", val_arrays.shape)

# actual GAP values to predict
print("Shape of val_labs", val_labs.shape)

# Firt Test the model by Evaluating the model and get MSE as output
scores_test = model.evaluate(val_arrays, val_labs, verbose=2)
print('\nMSE: {}'.format(scores_test[1]))

#Use a trained machine Learning model (model) to predict future Global_active_power
#Compare these predictions to actual values (val_labs).
#Save the predictions to a CSV file for later analysis.

# Generate predictions - Make predictions of future GAP values pH = 5
y_pred_test = model.predict(val_arrays) #make predictions for the validation data.
y_true_test = val_labs

# The rest of your code remains the same
test_set = pd.DataFrame(y_pred_test)
test_set.to_csv('submit_test.csv', index = None) # Create a spreadsheet-like table

print("True Test", y_true_test)

# Plot the predicted data vs. the actual data
# we will limit our plot to the first 500 predictions for better visualization

fig_verify = plt.figure(figsize=(10, 5))
plt.plot(y_pred_test[-500:], label = 'Predicted Value')
plt.plot(y_true_test[-500:], label = 'Actual Value')

```

```
plt.title('Global Active Power Prediction - Last 500 Points', fontsize=22, fontweight='bold')
plt.ylabel('value')
plt.xlabel('row')
plt.legend()
plt.show()
fig_verify.savefig("model_regression_verify.png")

print("Pred_Test", y_pred_test)
```

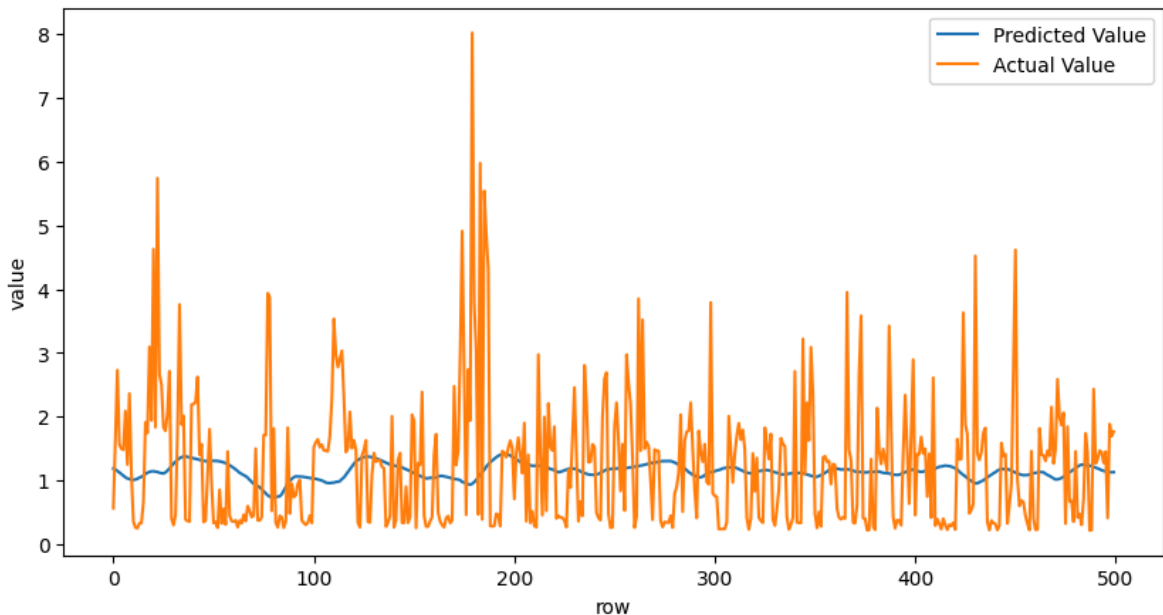
Current val\_arrays shape: (4093, 30)  
 Reshaped val\_arrays shape: (4093, 30, 1)  
 Reshaped val\_arrays shape: (4093, 30, 1)  
 Shape of val\_labels (4093,)  
 128/128 - 2s - 17ms/step - loss: 0.8242 - mse: 0.8242

MSE: 0.82420414686203

128/128 ————— 4s 27ms/step

True Test [2.724 1.94 1.944 ... 1.884 1.694 1.766]

## Global Active Power Prediction - Last 500 Points



```
Pred_Test [[-0.00713222]
 [ 0.00312395]
 [ 0.02738443]
 ...
 [ 1.1311783 ]
 [ 1.1290383 ]
 [ 1.1306075 ]]
```

**Q: How did your model perform? What can you tell about the model from the loss curves? What could we do to try to improve the model?**

Answer:

## I.Evalutation of the model



## A. MEAN SQUARED ERROR:

The model was evaluated on the validation set, and the following metrics were obtained:

Mean Squared Error (MSE): 0.8292 The Mean Squared Error (MSE) is a measure of the average squared difference between the predicted and actual values. In this case, the MSE of 0.8316 indicates that, on average, the model's predictions deviate from the actual values by approximately 0.8316 squared units. Since the target variable is Global\_active\_power, this value represents the squared error in predicting the power consumption.

## B. OBSERATION OF PREDICTED Vs ACTUAL GLOBAL ACTIVE POWER:

1. General Trend Capture: The predicted values (blue line) generally follow the overall trend of the actual values (orange line). This indicates that the LSTM model has learned to capture the underlying patterns in the Global Active Power data to some extent.
2. Smoothing Effect: The predicted values appear smoother compared to the actual values. This is a common characteristic of moving average-based models like LSTMs, as they tend to smooth out high-frequency fluctuations in the data.
3. Lagging: In many instances, the predicted values seem to lag slightly behind the actual values. This suggests that the model might be struggling to perfectly capture rapid changes in the actual power consumption.
4. Peak Discrepancies: While the model captures the general trend of peaks, the predicted peak values often don't reach the same heights as the actual peaks. This indicates that the model might be underestimating the magnitude of peak power consumption.
5. Noise Sensitivity: The actual values exhibit a higher degree of variability and noise, which the predicted values do not fully replicate. The model seems to focus on the broader patterns and is less sensitive to the high-frequency noise.

## II. LOSS CURVE OF THE MODEL:

### A. TRAINING LOSS:

1. The training loss starts relatively high, around 1.35, and shows a significant decrease in the first few epochs (roughly until epoch 5).
2. After epoch 5, the training loss continues to decrease, but at a slower rate. It reaches a plateau around 1.12 after approximately epoch 15. This suggests that the model might have reached a point of diminishing returns on the training set. Further training is not leading to improvements.

### B. VALIDATION/TEST LOSS:

1. The validation/test loss starts higher than the training loss, as expected, around 1.52. This is normal because the model hasn't been explicitly trained on this data.
2. The validation loss also decreases initially, though not as dramatically as the training loss. It stabilizes around 1.48 after epoch 5.
3. Importantly, the validation loss remains relatively flat with a slight upward tendency after epoch 15. This is a potential sign of overfitting. The model might be starting to memorize specific details of the training data that don't generalize well to unseen data.

### III. INTERPRETATIONS:

1. Good Initial Learning: The rapid decrease in training loss in the initial epochs suggests that the LSTM network is capable of learning the underlying patterns in the power consumption data.
2. Potential Overfitting: The plateauing of training loss and the slight increase in validation loss after epoch 15 suggest potential overfitting.
3. Further Training Considerations: Extending training beyond 30 epochs might not be beneficial and could worsen overfitting. It's crucial to implement strategies to mitigate overfitting.

### WHAT COULD BE DONE TO IMPROVE MODEL:

1. Learning Rate Optimization: Try reducing the learning rate (e.g., to 0.001 or 0.0001) and observe the impact on the model's performance. Use techniques like learning rate scheduling or adaptive optimizers to fine-tune the learning rate during training. This might improve the model performance
2. Model Architecture Exploration: Consideration of deeper LSTM networks (more layers) or bidirectional LSTMs to capture more complex temporal dependencies in the data is good. Also addition of one or two 1D convolutional layers before the LSTM layers is another option to try. The CNNs would convolve across the time series, extracting relevant features like daily or weekly patterns.
3. Linear layers could be added after the LSTM layers to further process the learned representations. They can help map the LSTM output to the desired prediction range leading to model improvement.

## Model Optimization

Now it's your turn to build an LSTM-based model in hopes of improving performance on this training set. Changes that you might consider include:

- Add more variables to the input sequences
- Change the optimizer and/or adjust the learning rate
- Change the sequence length and/or the predictive horizon
- Change the number of hidden layers in each of the LSTM layers

- Change the model architecture altogether--think about adding convolutional layers, linear layers, additional regularization, creating embeddings for the input data, changing the loss function, etc.

There isn't any minimum performance increase or number of changes that need to be made, but I want to see that you have tried some different things. Remember that building and optimizing deep learning networks is an art and can be very difficult, so don't make yourself crazy trying to optimize for this assignment.

**Q: What changes are you going to try with your model? Why do you think these changes could improve model performance?**

A: The following are the model changes which I would try next

1. Increasing number of LSTM layers to 5
2. Architectural change - 2 convolutional layer, max pooling layer, Dense(Fully connected) layers and output layer Change sequence\_length to 50
3. Adding Multivariate (Voltage, Global Intensity and Global Reactive power) as Dependent variables
4. Architectural change - CNN feature extraction, Bidirectional LSTM with attention mechanism, Temporal Pooling and output layer

## DESCRIPTIONS OF THE ABOVE PROPOSED MODEL CHANGES:

### 1. Increasing LSTM Layers to 5 (from 2)

#### A. Benefits:

Deeper LSTM networks (more layers) can learn hierarchical temporal dependencies. Lower layers might capture short-term patterns (e.g., daily fluctuations), while higher layers learn longer-term patterns (e.g., weekly or seasonal trends).

**B. Why it might improve performance##: Power consumption is influenced by factors at multiple time scales. A deeper LSTM could better capture these complex temporal relationships.**

#### Caveats:

Increasing layers also increases the risk of overfitting, especially with limited data. Need to check regularization (dropout, L1/L2) and potentially use techniques like early stopping.

## 2. Adding 2 Convolutional Layers, Max Pooling, and Dense Layers##

### A. Benefits:

CNNs for Feature Extraction: Convolutional Neural Networks (CNNs) are excellent at automatically learning local patterns in sequential data. In time series, 1D CNNs can identify relevant features within short time windows (e.g., daily peaks, weekly patterns) before feeding them to the LSTM. This reduces the burden on the LSTM, allowing it to focus on longer-range dependencies. Max Pooling: Reduces the dimensionality of the CNN outputs, making the model more computationally efficient and potentially reducing overfitting. Dense Layers: Fully connected (dense) layers after the LSTM can further process the learned representations before making the final prediction.

### B. Why it might improve performance:

This architecture combines the strengths of CNNs (local feature extraction) and LSTMs (long-range dependencies). It's a powerful approach for time series.

## 3. Changing sequence\_length to 50 (from 30):

### A. Benefits:##

A longer sequence length means the model sees more historical data when making predictions. This can be beneficial if there are long-term dependencies in the data (e.g., weekly or monthly cycles).

### B. Why it might improve performance:##

If your power consumption patterns have dependencies that span longer than your original sequence\_length, increasing it could allow the model to learn these patterns. Caveats: A longer sequence length increases the computational cost and memory requirements. It also increases the risk of overfitting if the data isn't sufficient.

## 4. Adding Multivariate Inputs (Voltage, Global Intensity, Global Reactive Power)

**A. Benefits:##** Power consumption is rarely influenced by active power alone. Including other relevant variables like voltage, intensity, and reactive power provides the model with more information, potentially leading to more accurate predictions.

**B. Why it might improve performance:##**

These additional variables likely have a strong relationship with active power. Including them allows the model to learn these relationships and make more informed predictions.

## 5. CNN Feature Extraction, Bidirectional LSTM with Attention, Temporal Pooling

**. Benefits:**

**A. Bidirectional LSTM:##**

Processes the input sequence in both forward and backward directions, allowing the model to capture dependencies from both past and future (relative to a given time step). This can be useful for identifying patterns that are influenced by both past and future events.

Attention Mechanism: Allows the model to focus on the most relevant parts of the input sequence when making predictions. It learns which time steps (or which of the CNN-extracted features) are most important.

**B. Temporal Pooling:##** A form of pooling applied to the time dimension.

It can help reduce dimensionality and focus on the most salient temporal features.

**C. Why it might improve performance:##**

This is a sophisticated architecture designed to capture complex temporal relationships and focus on the most relevant information. The bidirectional LSTM captures dependencies in both directions, and the attention mechanism helps the model prioritize the most important time steps. Temporal pooling further refines the feature representation.

In [ ]: *# I. Changes in Model - Increase number of LSTM layers to 5 from 2*  
*# Play with your ideas for optimization here*

```

# Read the .txt file into a DataFrame
df = pd.read_csv(file_path, parse_dates=['Datetime'])

# Display the first few rows of the DataFrame to verify it loaded correctly
print(df.head())

#create training and validation sets here
# Sort the data by Datetime to ensure it's in chronological order
df = df.sort_values('Datetime')

# Assign size for data subset (1% of the entire dataset)
subset_size = int(0.01 * len(df))

# Take random data subset
#n=subset_size: This argument specifies the number of rows needed in random subset.
#random_state=42: This is crucial for reproducibility. It sets the seed for the random
#               In case this code is run multiple times with the same random_state, this will
#               debugging, and sharing results. IF this argument is None, then the same
df_subset = df.sample(n=subset_size, random_state=42)

# Sort the subset by Datetime again to maintain chronological order
df_subset = df_subset.sort_values('Datetime')

# Split data subset 80/20 for train/validation
# Split the df_subset which is a random sample of the original DataFrame, its index
# We use the index to split to maintain the time order
#len(df_subset): Gets the number of rows in the df_subset.
#0.8 * len(df_subset): Calculates 80% of the number of rows. This determines the split
#int(...): Converts the result to an integer. This is necessary because slice indices
#split_index: Stores the calculated index value. This index marks the boundary between

split_index = int(0.8 * len(df_subset))

# This creates the training set. It takes all rows from df_subset up to (but not including)
# Because the original index from the dataframe is used, and because the subset was sorted
# this subset will maintain the time order.
# train_df: Stores the resulting training DataFrame

train_df = df_subset[:split_index]

# Create the validation set. It takes all rows from df_subset starting from the row
val_df = df_subset[split_index:]

#reset the indices for cleanliness
#After splitting the data, the original index from the full dataset is retained. Re-indexing
#This makes it easier to work with the data, especially when iterating over rows or
train_df = train_df.reset_index()
val_df = val_df.reset_index()

print(f"Training set shape: {train_df.shape}")
print(f"Validation set shape: {val_df.shape}")

seq_arrays = []
seq_labs = []

```

```

# Get GAP readings from your DataFrame column
gap_readings = train_df['Global_active_power'].values # Direct numpy array access

#use 30 consecutive measurements to predict value 5 steps in future
seq_length = 30 # 30-minute window
ph = 5 # Predict 5 minutes ahead
feat_cols = ['Global_active_power'] # Define feature columns

# Create sequences with alignment for predictive horizon
seq_arrays = []
seq_labs = []

# Calculate valid range for sequence creation
# Total Data Points: Len(gap_readings) = 16,359 historical GAP measurements.
#Sequence Length: Each input sequence uses 30 consecutive measurements.
#Predictive Horizon: To predict the GAP value 5 steps ahead of the input sequence.
#Input Sequence: Each sequence starts at index i and ends at i + seq_length - 1.
#Output Value: Predicted value is at index i + seq_length + ph - 1.
#Last Valid Sequence:
#Start index i must satisfy:
#i + seq_length + ph - 1 < Len(gap_readings)
#For the dataset with Len = 16359:
# i + 30 + 5 - 1 < 16,359 → i < 16,359 - 34 → i ≤ 16,324
# Total sequences: 16,325 (i ranges from 0 to 16,324).

num_samples = len(gap_readings) - seq_length - ph + 1

#Sequence creation loop
#Input Sequence: Sliding window of 30 consecutive values
#Output Value: Value 5 steps after the end of the input window

for i in range(num_samples):
    # Input: 30 consecutive GAP measurements
    input_seq = gap_readings[i:i+seq_length]

    # Output: GAP value 5 cycles after sequence end
    output_val = gap_readings[i + seq_length + ph - 1]

    # append(input_seq): adds the current input_seq (which is itself a sequence of
    # So, seq_arrays will become a list of lists (or a list of arrays).
    #input_seq is a sequence of 30 consecutive GAP measurements (because seq_length
    #seq_arrays is a list that will contain all these sequences.
    #.append() adds input_seq as a new element to the end of seq_arrays.
    #After this operation, seq_arrays becomes a list of lists, where each inner list
    # Example:
    # seq_arrays = [
    # [0.5, 0.6, ..., 0.8], # First sequence (30 values)
    # [0.6, 0.7, ..., 0.9], # Second sequence (30 values)
    # ...
    seq_arrays.append(input_seq) #store the input sequence of time series data in n

    # add the output_val to the seq_labs list
    # output_val is a single GAP measurement, 5 steps ahead of the end of the input
    # seq_labs is a list that will contain all these future values.
    seq_labs.append(output_val) #add output_val as a new element to the end of seq_labs.

```

```

# After this operation, seq_labs becomes a list of individual GAP values, each
# Example:
# python
# seq_labs = [0.7, 0.8, 0.9, ...] # Each value corresponds to a sequence in se
seq_labs.append(output_val)

# Convert to numpy arrays with proper typing
seq_arrays = np.array(seq_arrays, dtype=np.float32).reshape(-1, seq_length, len(fea
seq_labs = np.array(seq_labs, dtype=np.float32)

print("Input shape:", seq_arrays.shape)
print("Output shape:", seq_labs.shape)

# Verify dimensions with corrected calculation
expected_samples = len(train_df) - seq_length - ph + 1
expected_shape = (expected_samples, seq_length, len(feat_cols))

try:
    assert seq_arrays.shape == expected_shape, (
        f"Sequence array shape mismatch. Got {seq_arrays.shape}, "
        f"expected {expected_shape}. Check data length: {len(train_df)} "
        f"should be ≥ {seq_length + ph}"
    )

    assert seq_labs.shape == (expected_samples,), (
        f"Label shape mismatch. Got {seq_labs.shape}, "
        f"expected ({expected_samples},)"
    )

    print("All assertions passed!")
except AssertionError as e:
    print(f"Assertion Error: {e}")
    print(f"Required data length: {seq_length + ph - 1}")
    print(f"Actual data length: {len(train_df)}")

assert(seq_arrays.shape == (len(train_df)-seq_length-ph+1,seq_length, len(feat_cols)
assert(seq_labs.shape == (len(train_df)-seq_length-ph+1,))
# define path to save model
model_path = 'LSTM_model1.keras'

# build the network
# Onfe Feature which is GAP and one Output value (GAP) is predicted
nb_features = 1
nb_out = 1

# Initialize a Sequential model
model = Sequential()

# LSTM Layer 1
# create an LSTM layer with 16 LSTM units (or neurons). These units are the core pr
# return_sequences=True: This is crucial for stacking LSTM layers. Setting it to Tr
# in the input sequence. This sequence will then be fed as input to the next LSTM L
# nb_features: This is the number of features in your input at each time step (e.g.
# Only the first LSTM layer needs to have this explicit input_shape defined. Subseq
model.add(Input(shape=(seq_length, nb_features))) # Input Layer FIRST
# LSTM Layer 2
model.add(LSTM(16, return_sequences=True)) # THEN the LSTM Layer

```



```

# model.add(Dropout(0.3)) This adds a dropout layer after the LSTM layer. Dropout i
# it randomly sets a fraction (in this case, 30% or 0.3) of the input units to 0 at
# reliant on specific combinations of neurons
model.add(Dropout(0.3))

# Layer 2
model.add(LSTM(32, return_sequences=True))
model.add(Dropout(0.3))

# Layer 3
model.add(LSTM(64, return_sequences=True))
model.add(Dropout(0.3))

# Layer 4
model.add(LSTM(32, return_sequences=True))
model.add(Dropout(0.3))

# Layer 5 (Final LSTM)
model.add(LSTM(16, return_sequences=False))
model.add(Dropout(0.3))

# Add Dense (fully connected) output layer
model.add(Dense(units=nb_out)) # Output Layer with 1 unit for regression
model.add(Activation("linear")) # Linear activation for regression

# Define optimizer with Learning rate
# Use Lower Learning rate for deeper network

optimizer = keras.optimizers.Adam(learning_rate=0.001)
model.compile(loss='mean_squared_error', optimizer=optimizer, metrics=['mse'])

# Loss function is MSE, Adam optimizer, track MSE during training
# uses Mean Squared Error as both the loss function and a metric, appropriate for r
model.compile(loss='mean_squared_error', optimizer=optimizer, metrics=['mse'])

print(model.summary())

# fit the network (train it)
# max number of training epochs = 100
# number of samples per gradient update (batch size) = 500
# use 5% of data for validation
# verbosity mode = 2, one line per epoch
# callback - early stopping to prevent overfitting and if no more learning is happen
# monitor='val_loss', # Monitor validation loss
# min_delta=0: Minimum change to qualify as improvement
# patience=10: Number of epochs with no improvement after which training will stop
# verbose=0: Verbosity mode
# mode='min' # The direction is automatically inferred if not set

# Model checkpoint to save best model
# model_path, # Path to save the model
# monitor='val_loss', # Monitor validation loss
# save_best_only=True, # Only save when the model is better than the previous best
# mode='min', # Lower validation loss is better

```

```

history = model.fit(seq_arrays, seq_labs, epochs=100, batch_size=500, validation_sp
                    callbacks = [keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=
                        keras.callbacks.ModelCheckpoint(model_path,monitor='val_loss
                    )

# List all data in history
# After training, it prints the keys of the history object, which typically include
print(history.history.keys())

# show me how one or two of your different models perform
# using the code from the "Validating our model" section above
val_arrays = []
val_labs = []

# Create sequences starting with minimum of 2 readings, building up to seq_length
for i in range(2, len(val_df['Global_active_power'])):
    # Get all available readings up to current index
    raw_seq = val_df['Global_active_power'].values[:i] # Get readings from start t

    # Get Label (ph steps ahead)
    if i + ph - 1 < len(val_df): # Check if we have enough data for the label
        val_labs.append(val_df['Global_active_power'].values[i + ph - 1])
    else: # Handle edge case at end of dataset
        break

    val_arrays.append(raw_seq)

# Pad sequences to uniform length (seq_length) with pre-padding (zeros at beginning
val_arrays = pad_sequences(
    val_arrays,
    maxlen=seq_length, # Force all sequences to be seq_length long
    dtype='float32',    # Match TensorFlow's default dtype
    padding='pre',      # Add zeros at BEGINNING of sequences (preserve recent dat
    truncating='pre',   # Truncate from beginning if sequences are too long (shoul
    value=0             # Use 0 for padding (consider using mean value if better f
)

# Convert Labels to numpy array with proper dtype
val_labs = np.array(val_labs, dtype=np.float32)

print("Padded validation sequences shape:", val_arrays.shape)
print("Validation labels shape:", val_labs.shape)

# Reshape val_arrays to add the feature dimension
val_arrays = np.reshape(val_arrays, (val_arrays.shape[0], val_arrays.shape[1], nb_f

# Ensure val_labs is a 1D array
val_labs = np.reshape(val_labs, (-1,))

# Debugging: Print shapes
print("Shape of val_arrays:", val_arrays.shape)
print("Shape of val_labs:", val_labs.shape)

# Evaluate the model
model.evaluate(val_arrays, val_labs, verbose=2)

```

```

print('\nMSE: {}'.format(scores_test[1]))

# Make predictions
y_pred_test = model.predict(val_arrays)
y_true_test = val_labs

# Save predictions to CSV
test_set = pd.DataFrame(y_pred_test)
test_set.to_csv('submit_test.csv', index=None)

# Plot the predicted data vs. the actual data
fig_verify = plt.figure(figsize=(10, 5))
plt.plot(y_pred_test[-500:], label='Predicted Value')
plt.plot(y_true_test[-500:], label='Actual Value')
plt.title('Global Active Power Prediction (LSTM 5 layers)- Last 500 Points', fontsi
plt.ylabel('value')
plt.xlabel('row')
plt.legend()
plt.show()
fig_verify.savefig("model_regression_verify.png")

```

	Unnamed: 0	Date	Time	Global_active_power	\
0	0	2006-12-16	17:24:00	4.216	
1	1	2006-12-16	17:25:00	5.360	
2	2	2006-12-16	17:26:00	5.374	
3	3	2006-12-16	17:27:00	5.388	
4	4	2006-12-16	17:28:00	3.666	

	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	\
0	0.418	234.84	18.4	0.0	
1	0.436	233.63	23.0	0.0	
2	0.498	233.29	23.0	0.0	
3	0.502	233.74	23.0	0.0	
4	0.528	235.68	15.8	0.0	

	Sub_metering_2	Sub_metering_3	Datetime	gap_monthly	\
0	1.0	17.0	2006-12-16 17:24:00	NaN	
1	1.0	16.0	2006-12-16 17:25:00	NaN	
2	2.0	17.0	2006-12-16 17:26:00	NaN	
3	1.0	17.0	2006-12-16 17:27:00	NaN	
4	1.0	17.0	2006-12-16 17:28:00	NaN	

	grp_monthly	v_monthly	gi_monthly
0	NaN	NaN	NaN
1	NaN	NaN	NaN
2	NaN	NaN	NaN
3	NaN	NaN	NaN
4	NaN	NaN	NaN

Training set shape: (16393, 16)  
 Validation set shape: (4099, 16)  
 Input shape: (16359, 30, 1)  
 Output shape: (16359,)  
 All assertions passed!  
**Model: "sequential\_25"**

Layer (type)	Output Shape	
lstm_39 (LSTM)	(None, 30, 16)	
dropout_52 (Dropout)	(None, 30, 16)	
lstm_40 (LSTM)	(None, 30, 32)	
dropout_53 (Dropout)	(None, 30, 32)	
lstm_41 (LSTM)	(None, 30, 64)	
dropout_54 (Dropout)	(None, 30, 64)	
lstm_42 (LSTM)	(None, 30, 32)	
dropout_55 (Dropout)	(None, 30, 32)	
lstm_43 (LSTM)	(None, 16)	
dropout_56 (Dropout)	(None, 16)	
dense_24 (Dense)	(None, 1)	
activation_10 (Activation)	(None, 1)	



Total params: 47,825 (186.82 KB)

Trainable params: 47,825 (186.82 KB)

Non-trainable params: 0 (0.00 B)

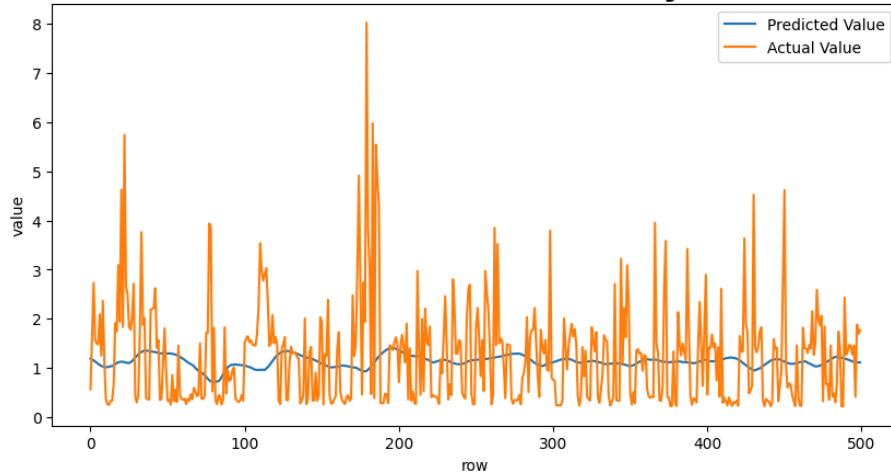
None  
Epoch 1/100  
32/32 - 19s - 584ms/step - loss: 1.3930 - mse: 1.3930 - val\_loss: 1.6131 - val\_mse: 1.6131  
Epoch 2/100  
32/32 - 11s - 329ms/step - loss: 1.1732 - mse: 1.1732 - val\_loss: 1.4805 - val\_mse: 1.4805  
Epoch 3/100  
32/32 - 11s - 331ms/step - loss: 1.1562 - mse: 1.1562 - val\_loss: 1.5127 - val\_mse: 1.5127  
Epoch 4/100  
32/32 - 9s - 268ms/step - loss: 1.1659 - mse: 1.1659 - val\_loss: 1.4882 - val\_mse: 1.4882  
Epoch 5/100  
32/32 - 10s - 307ms/step - loss: 1.1500 - mse: 1.1500 - val\_loss: 1.5051 - val\_mse: 1.5051  
Epoch 6/100  
32/32 - 12s - 379ms/step - loss: 1.1518 - mse: 1.1518 - val\_loss: 1.4719 - val\_mse: 1.4719  
Epoch 7/100  
32/32 - 10s - 327ms/step - loss: 1.1522 - mse: 1.1522 - val\_loss: 1.4609 - val\_mse: 1.4609  
Epoch 8/100  
32/32 - 8s - 248ms/step - loss: 1.1486 - mse: 1.1486 - val\_loss: 1.4690 - val\_mse: 1.4690  
Epoch 9/100  
32/32 - 12s - 373ms/step - loss: 1.1509 - mse: 1.1509 - val\_loss: 1.4423 - val\_mse: 1.4423  
Epoch 10/100  
32/32 - 10s - 306ms/step - loss: 1.1504 - mse: 1.1504 - val\_loss: 1.4784 - val\_mse: 1.4784  
Epoch 11/100  
32/32 - 8s - 246ms/step - loss: 1.1504 - mse: 1.1504 - val\_loss: 1.4793 - val\_mse: 1.4793  
Epoch 12/100  
32/32 - 11s - 358ms/step - loss: 1.1451 - mse: 1.1451 - val\_loss: 1.4859 - val\_mse: 1.4859  
Epoch 13/100  
32/32 - 10s - 307ms/step - loss: 1.1412 - mse: 1.1412 - val\_loss: 1.4873 - val\_mse: 1.4873  
Epoch 14/100  
32/32 - 8s - 257ms/step - loss: 1.1452 - mse: 1.1452 - val\_loss: 1.4766 - val\_mse: 1.4766  
Epoch 15/100  
32/32 - 11s - 354ms/step - loss: 1.1456 - mse: 1.1456 - val\_loss: 1.4658 - val\_mse: 1.4658  
Epoch 16/100  
32/32 - 11s - 354ms/step - loss: 1.1433 - mse: 1.1433 - val\_loss: 1.5017 - val\_mse: 1.5017  
Epoch 17/100  
32/32 - 8s - 249ms/step - loss: 1.1336 - mse: 1.1336 - val\_loss: 1.4587 - val\_mse: 1.4587  
Epoch 18/100  
32/32 - 10s - 319ms/step - loss: 1.1368 - mse: 1.1368 - val\_loss: 1.4770 - val\_mse: 1.4770  
Epoch 19/100  
32/32 - 10s - 319ms/step - loss: 1.1368 - mse: 1.1368 - val\_loss: 1.4770 - val\_mse: 1.4770

32/32 - 12s - 377ms/step - loss: 1.1441 - mse: 1.1441 - val\_loss: 1.5097 - val\_mse: 1.5097  
dict\_keys(['loss', 'mse', 'val\_loss', 'val\_mse'])  
Padded validation sequences shape: (4093, 30)  
Validation labels shape: (4093,)  
Shape of val\_arrays: (4093, 30, 1)  
Shape of val\_labs: (4093,)  
128/128 - 2s - 15ms/step - loss: 0.8229 - mse: 0.8229

MSE: 0.8229178786277771

128/128 ————— 4s 26ms/step

## Global Active Power Prediction (LSTM 5 layers)- Last 500 Points



```
In [ ]: # II. Architectural change - 2 convolutional layer, max pooling layer, Dense(Fully
# Play with your ideas for optimization here

# Read the .txt file into a DataFrame
df = pd.read_csv(file_path, parse_dates=['Datetime'])

# Display the first few rows of the DataFrame to verify it loaded correctly
print(df.head())

#create training and validation sets here
# Sort the data by Datetime to ensure it's in chronological order
df = df.sort_values('Datetime')

# Assign size for data subset (1% of the entire dataset)
subset_size = int(0.01 * len(df))

# Take random data subset
#n=subset_size: This argument specifies the number of rows needed in random subset.
#random_state=42: This is crucial for reproducibility. It sets the seed for the ra
# Incase this code is run multiple times with the same random_state, this w
# debugging, and sharing results. IF this argument is None, then the same
df_subset = df.sample(n=subset_size, random_state=42)

# Sort the subset by Datetime again to maintain chronological order
df_subset = df_subset.sort_values('Datetime')

# Split data subset 80/20 for train/validation
# Split the df subset which is a random sample of the original DataFrame, its index
```

```

# We use the index to split to maintain the time order
#len(df_subset): Gets the number of rows in the df_subset.
#0.8 * len(df_subset): Calculates 80% of the number of rows. This determines the sp
#int(...): Converts the result to an integer. This is necessary because slice indic
#split_index: Stores the calculated index value. This index marks the boundary betw
split_index = int(0.8 * len(df_subset))

# This creates the training set. It takes all rows from df_subset up to (but not in
# Because the original index from the dataframe is used, and because the subset was
# this subset will maintain the time order.
# train_df: Stores the resulting training DataFrame
train_df = df_subset[:split_index]

# Create the validation set. It takes all rows from df_subset starting from the row
val_df = df_subset[split_index:]

#reset the indices for cleanliness
#After splitting the data, the original index from the full dataset is retained. Re
#This makes it easier to work with the data, especially when iterating over rows or
train_df = train_df.reset_index()
val_df = val_df.reset_index()

print(f"Training set shape: {train_df.shape}")
print(f"Validation set shape: {val_df.shape}")

seq_arrays = []
seq_labs = []

# Get GAP readings from your DataFrame column
gap_readings = train_df['Global_active_power'].values # Direct numpy array access

#use 30 consecutive measurements to predict value 5 steps in future
seq_length = 30 # 30-minute window
ph = 5 # Predict 5 minutes ahead
feat_cols = ['Global_active_power'] # Define feature columns

# Create sequences with alignment for predictive horizon
seq_arrays = []
seq_labs = []

# Calculate valid range for sequence creation

# Total Data Points: len(gap_readings) = 16,359 historical GAP measurements.
#Sequence Length: Each input sequence uses 30 consecutive measurements.
#Predictive Horizon: To predict the GAP value 5 steps ahead of the input sequence.
#Input Sequence: Each sequence starts at index i and ends at i + seq_length - 1.
#Output Value: Predicted value is at index i + seq_length + ph - 1.
#Last Valid Sequence:
#Start index i must satisfy:
#i + seq_length + ph - 1 < len(gap_readings)
#For the dataset with len = 16359:
# i + 30 + 5 - 1 < 16,359 → i < 16,359 - 34 → i ≤ 16,324
# Total sequences: 16,325 (i ranges from 0 to 16,324).

```

```

#Sequence creation loop
#Input Sequence: Sliding window of 30 consecutive values
#Output Value: Value 5 steps after the end of the input window

for i in range(num_samples):
    # Input: 30 consecutive GAP measurements
    input_seq = gap_readings[i:i+seq_length]

    # Output: GAP value 5 cycles after sequence end
    output_val = gap_readings[i + seq_length + ph - 1]

    # append(input_seq): adds the current input_seq (which is itself a sequence of
    # So, seq_arrays will become a list of lists (or a list of arrays).
    #input_seq is a sequence of 30 consecutive GAP measurements (because seq_length
    #seq_arrays is a list that will contain all these sequences.
    #.append() adds input_seq as a new element to the end of seq_arrays.
    #After this operation, seq_arrays becomes a list of lists, where each inner list
    # Example:
    # seq_arrays = [
    # [0.5, 0.6, ..., 0.8], # First sequence (30 values)
    # [0.6, 0.7, ..., 0.9], # Second sequence (30 values)
    # ...
    seq_arrays.append(input_seq) #store the input sequence of time series data in n

    # add the output_val to the seq_labs list
    # output_val is a single GAP measurement, 5 steps ahead of the end of the input
    # seq_labs is a list that will contain all these future values.
    # append(): This adds output_val as a new element to the end of seq_labs.
    # After this operation, seq_labs becomes a list of individual GAP values, each
    # Example:
    # python
    # seq_labs = [0.7, 0.8, 0.9, ...] # Each value corresponds to a sequence in seq_arrays
    seq_labs.append(output_val)

# Convert to numpy arrays with proper typing
seq_arrays = np.array(seq_arrays, dtype=np.float32).reshape(-1, seq_length, len(features))
seq_labs = np.array(seq_labs, dtype=np.float32)

print("Input shape:", seq_arrays.shape)
print("Output shape:", seq_labs.shape)

# Verify dimensions with corrected calculation
expected_samples = len(train_df) - seq_length - ph + 1
expected_shape = (expected_samples, seq_length, len(features))

try:
    assert seq_arrays.shape == expected_shape, (
        f"Sequence array shape mismatch. Got {seq_arrays.shape}, "
        f"expected {expected_shape}. Check data length: {len(train_df)} "
        f"should be ≥ {seq_length + ph}"
    )

    assert seq_labs.shape == (expected_samples,), (
        f"Label shape mismatch. Got {seq_labs.shape}, "
        f"expected {(expected_samples,)}"
    )

```



```

)
    print("All assertions passed!")
except AssertionError as e:
    print(f"Assertion Error: {e}")
    print(f"Required data length: {seq_length + ph - 1}")
    print(f"Actual data length: {len(train_df)}")

assert(seq_arrays.shape == (len(train_df)-seq_length-ph+1, seq_length, len(feat_cols)
assert(seq_labs.shape == (len(train_df)-seq_length-ph+1,))
# define path to save model
model_path = 'LSTM_model1.keras'

# build the network
# Onfe Feature which is GAP and one Output value (GAP) is predicted
nb_features = 1
nb_out = 1

# Initialize a Sequential model
model = Sequential()

# Input shape: (seq_length, nb_features)
input_shape = (seq_length, nb_features)

# 1D Convolutional Layer
inputs = Input(shape=input_shape)
x = Conv1D(filters=64, kernel_size=3, activation='relu')(inputs)

model.add(MaxPooling1D(pool_size=2)) # Downsample the output
model.add(BatchNormalization()) # Normalize the activations
model.add(Dropout(0.3)) # Regularization

# Second 1D Convolutional Layer
model.add(Conv1D(filters=128, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(BatchNormalization())
model.add(Dropout(0.3))

# Global Max Pooling to reduce sequence dimension
model.add(GlobalMaxPooling1D()) # Output shape: (batch_size, 128)

# Dense (Fully Connected) Layers
model.add(Dense(units=64, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.3))

model.add(Dense(units=32, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.3))

# Output Layer
model.add(Dense(units=nb_out, activation='linear')) # Linear activation for regres

# Compile the model with Huber Loss
optimizer = Adam(learning_rate=0.001)
optimizer=optimizer, metrics=['mse'])

```

```

# Print model summary
print(model.summary())

# fit the network (train it)
# max number of training epochs = 100
# number of samples per gradient update (batch size) = 500
# use 5% of data for validation
# verbosity mode = 2, one line per epoch
# callback - early stopping to prevent overfitting and if no more learning is happening
# monitor='val_loss', # Monitor validation loss
# min_delta=0: Minimum change to qualify as improvement
# patience=10: Number of epochs with no improvement after which training will stop
# verbose=0: Verbosity mode
# mode='min' # The direction is automatically inferred if not set

# Model checkpoint to save best model
# model_path, # Path to save the model
# monitor='val_loss', # Monitor validation loss
# save_best_only=True, # Only save when the model is better than the previous best
# mode='min', # Lower validation loss is better
# verbose=0 # Verbosity mode

history = model.fit(
    seq_arrays, # Input sequences
    seq_labels, # Output labels
    epochs=100,
    batch_size=500,
    validation_split=0.05,
    verbose=2,
    callbacks=[
        keras.callbacks.EarlyStopping(monitor='val_loss', patience=10, mode='min',
        keras.callbacks.ModelCheckpoint(model_path, monitor='val_loss', save_best_o
    ]
)

# List all data in history
# After training, it prints the keys of the history object, which typically include
print(history.history.keys())

# show me how one or two of your different models perform
# using the code from the "Validating our model" section above
val_arrays = []
val_labels = []

# Create sequences starting with minimum of 2 readings, building up to seq_length
for i in range(2, len(val_df['Global_active_power'])):
    # Get all available readings up to current index
    raw_seq = val_df['Global_active_power'].values[:i] # Get readings from start to
    # Get label (ph steps ahead)
    if i + ph - 1 < len(val_df): # Check if we have enough data for the label
        label = val_df['Global_active_power'].values[i + ph - 1]

```

```

else: # Handle edge case at end of dataset
    break

val_arrays.append(raw_seq)

# Pad sequences to uniform length (seq_length) with pre-padding (zeros at beginning)
val_arrays = pad_sequences(
    val_arrays,
    maxlen=seq_length, # Force all sequences to be seq_length long
    dtype='float32',    # Match TensorFlow's default dtype
    padding='pre',      # Add zeros at BEGINNING of sequences (preserve recent data)
    truncating='pre',   # Truncate from beginning if sequences are too long (should be 'post')
    value=0             # Use 0 for padding (consider using mean value if better for task)
)

# Convert labels to numpy array with proper dtype
val_labs = np.array(val_labs, dtype=np.float32)

print("Padded validation sequences shape:", val_arrays.shape)
print("Validation labels shape:", val_labs.shape)

# Reshape val_arrays to add the feature dimension
val_arrays = np.reshape(val_arrays, (val_arrays.shape[0], val_arrays.shape[1], nb_filters))

# Ensure val_labs is a 1D array
val_labs = np.reshape(val_labs, (-1,))

# Debugging: Print shapes
print("Shape of val_arrays:", val_arrays.shape)
print("Shape of val_labs:", val_labs.shape)

# Evaluate the model
scores_test = model.evaluate(val_arrays, val_labs, verbose=2)
print(f'\nValidation Loss (Huber): {scores_test[0]}')
print(f'\nMSE: {}'.format(scores_test[1]))

# Make predictions
y_pred_test = model.predict(val_arrays)
y_true_test = val_labs

# Save predictions to CSV
test_set = pd.DataFrame(y_pred_test)
test_set.to_csv('submit_test.csv', index=None)

# Plot the predicted data vs. the actual data
fig_verify = plt.figure(figsize=(10, 5))
plt.plot(y_pred_test[-500:], label='Predicted Value')
plt.plot(y_true_test[-500:], label='Actual Value')
plt.title('Global Active Power Prediction (2CNN, Max Pool, Dense and Output Layer)-')
plt.ylabel('value')
plt.xlabel('row')
plt.legend()
plt.show()
fig_verify.savefig("model_regression_verify.png")

```

	Unnamed: 0	Date	Time	Global_active_power	\
0	0	2006-12-16	17:24:00	4.216	
1	1	2006-12-16	17:25:00	5.360	
2	2	2006-12-16	17:26:00	5.374	
3	3	2006-12-16	17:27:00	5.388	
4	4	2006-12-16	17:28:00	3.666	

	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	\
0	0.418	234.84	18.4	0.0	
1	0.436	233.63	23.0	0.0	
2	0.498	233.29	23.0	0.0	
3	0.502	233.74	23.0	0.0	
4	0.528	235.68	15.8	0.0	

	Sub_metering_2	Sub_metering_3	Datetime	gap_monthly	\
0	1.0	17.0	2006-12-16 17:24:00	NaN	
1	1.0	16.0	2006-12-16 17:25:00	NaN	
2	2.0	17.0	2006-12-16 17:26:00	NaN	
3	1.0	17.0	2006-12-16 17:27:00	NaN	
4	1.0	17.0	2006-12-16 17:28:00	NaN	

	grp_monthly	v_monthly	gi_monthly
0	NaN	NaN	NaN
1	NaN	NaN	NaN
2	NaN	NaN	NaN
3	NaN	NaN	NaN
4	NaN	NaN	NaN

Training set shape: (16393, 16)  
Validation set shape: (4099, 16)  
Input shape: (16359, 30, 1)  
Output shape: (16359,)  
All assertions passed!  
**Model: "sequential\_26"**

Layer (type)	Output Shape	
max_pooling1d_8 ( <a href="#">MaxPooling1D</a> )	?	
batch_normalization_16 ( <a href="#">BatchNormalization</a> )	?	0
dropout_57 ( <a href="#">Dropout</a> )	?	
conv1d_9 ( <a href="#">Conv1D</a> )	?	0
max_pooling1d_9 ( <a href="#">MaxPooling1D</a> )	?	
batch_normalization_17 ( <a href="#">BatchNormalization</a> )	?	0
dropout_58 ( <a href="#">Dropout</a> )	?	
global_max_pooling1d_4 ( <a href="#">GlobalMaxPooling1D</a> )	?	
dense_25 ( <a href="#">Dense</a> )	?	0
batch_normalization_18 ( <a href="#">BatchNormalization</a> )	?	0
dropout_59 ( <a href="#">Dropout</a> )	?	
dense_26 ( <a href="#">Dense</a> )	?	0
batch_normalization_19 ( <a href="#">BatchNormalization</a> )	?	0
dropout_60 ( <a href="#">Dropout</a> )	?	
dense_27 ( <a href="#">Dense</a> )	?	0



Total params: 0 (0.00 B)

Trainable params: 0 (0.00 B)

Non-trainable params: 0 (0.00 B)

None

Epoch 1/100  
 32/32 - 7s - 230ms/step - loss: 1.0315 - mse: 3.8379 - val\_loss: 0.8666 - val\_mse: 2.8935

Epoch 2/100  
 32/32 - 1s - 28ms/step - loss: 0.7682 - mse: 2.5771 - val\_loss: 0.7452 - val\_mse: 2.4550

Epoch 3/100  
 32/32 - 1s - 39ms/step - loss: 0.6503 - mse: 2.1043 - val\_loss: 0.6827 - val\_mse: 2.2282

Epoch 4/100  
 32/32 - 2s - 50ms/step - loss: 0.5915 - mse: 1.8721 - val\_loss: 0.6328 - val\_mse: 2.0455

Epoch 5/100  
 32/32 - 1s - 44ms/step - loss: 0.5519 - mse: 1.7321 - val\_loss: 0.5826 - val\_mse: 1.8585

Epoch 6/100  
 32/32 - 2s - 67ms/step - loss: 0.5201 - mse: 1.6188 - val\_loss: 0.5584 - val\_mse: 1.7613

Epoch 7/100  
 32/32 - 1s - 25ms/step - loss: 0.5002 - mse: 1.5368 - val\_loss: 0.5756 - val\_mse: 1.8331

Epoch 8/100  
 32/32 - 1s - 27ms/step - loss: 0.4887 - mse: 1.4897 - val\_loss: 0.5576 - val\_mse: 1.7608

Epoch 9/100  
 32/32 - 1s - 39ms/step - loss: 0.4768 - mse: 1.4454 - val\_loss: 0.5566 - val\_mse: 1.7559

Epoch 10/100  
 32/32 - 1s - 27ms/step - loss: 0.4614 - mse: 1.3846 - val\_loss: 0.5532 - val\_mse: 1.7415

Epoch 11/100  
 32/32 - 1s - 39ms/step - loss: 0.4546 - mse: 1.3683 - val\_loss: 0.5516 - val\_mse: 1.7329

Epoch 12/100  
 32/32 - 1s - 27ms/step - loss: 0.4463 - mse: 1.3285 - val\_loss: 0.5411 - val\_mse: 1.6883

Epoch 13/100  
 32/32 - 1s - 37ms/step - loss: 0.4438 - mse: 1.3183 - val\_loss: 0.5432 - val\_mse: 1.6985

Epoch 14/100  
 32/32 - 1s - 41ms/step - loss: 0.4391 - mse: 1.3089 - val\_loss: 0.5433 - val\_mse: 1.6988

Epoch 15/100  
 32/32 - 1s - 44ms/step - loss: 0.4330 - mse: 1.2946 - val\_loss: 0.5440 - val\_mse: 1.7024

Epoch 16/100  
 32/32 - 2s - 52ms/step - loss: 0.4299 - mse: 1.2857 - val\_loss: 0.5395 - val\_mse: 1.6838

Epoch 17/100  
 32/32 - 2s - 68ms/step - loss: 0.4304 - mse: 1.2763 - val\_loss: 0.5386 - val\_mse: 1.6789

Epoch 18/100  
 32/32 - 1s - 34ms/step - loss: 0.4267 - mse: 1.2672 - val\_loss: 0.5354 - val\_mse: 1.6652

32/32 - 1s - 38ms/step - loss: 0.4230 - mse: 1.2585 - val\_loss: 0.5355 - val\_mse: 1.6671  
Epoch 20/100  
32/32 - 2s - 55ms/step - loss: 0.4208 - mse: 1.2509 - val\_loss: 0.5356 - val\_mse: 1.6673  
Epoch 21/100  
32/32 - 3s - 80ms/step - loss: 0.4179 - mse: 1.2378 - val\_loss: 0.5361 - val\_mse: 1.6695  
Epoch 22/100  
32/32 - 2s - 64ms/step - loss: 0.4152 - mse: 1.2322 - val\_loss: 0.5355 - val\_mse: 1.6658  
Epoch 23/100  
32/32 - 1s - 30ms/step - loss: 0.4156 - mse: 1.2309 - val\_loss: 0.5340 - val\_mse: 1.6593  
Epoch 24/100  
32/32 - 2s - 51ms/step - loss: 0.4145 - mse: 1.2289 - val\_loss: 0.5319 - val\_mse: 1.6486  
Epoch 25/100  
32/32 - 2s - 67ms/step - loss: 0.4115 - mse: 1.2200 - val\_loss: 0.5338 - val\_mse: 1.6571  
Epoch 26/100  
32/32 - 1s - 34ms/step - loss: 0.4089 - mse: 1.2094 - val\_loss: 0.5326 - val\_mse: 1.6530  
Epoch 27/100  
32/32 - 2s - 51ms/step - loss: 0.4101 - mse: 1.2122 - val\_loss: 0.5301 - val\_mse: 1.6406  
Epoch 28/100  
32/32 - 1s - 36ms/step - loss: 0.4088 - mse: 1.2071 - val\_loss: 0.5303 - val\_mse: 1.6427  
Epoch 29/100  
32/32 - 1s - 31ms/step - loss: 0.4064 - mse: 1.2058 - val\_loss: 0.5317 - val\_mse: 1.6490  
Epoch 30/100  
32/32 - 1s - 38ms/step - loss: 0.4061 - mse: 1.1976 - val\_loss: 0.5310 - val\_mse: 1.6460  
Epoch 31/100  
32/32 - 1s - 26ms/step - loss: 0.4053 - mse: 1.1987 - val\_loss: 0.5288 - val\_mse: 1.6353  
Epoch 32/100  
32/32 - 1s - 39ms/step - loss: 0.4070 - mse: 1.2020 - val\_loss: 0.5302 - val\_mse: 1.6420  
Epoch 33/100  
32/32 - 1s - 28ms/step - loss: 0.4047 - mse: 1.1936 - val\_loss: 0.5275 - val\_mse: 1.6291  
Epoch 34/100  
32/32 - 2s - 51ms/step - loss: 0.4046 - mse: 1.1872 - val\_loss: 0.5276 - val\_mse: 1.6287  
Epoch 35/100  
32/32 - 3s - 83ms/step - loss: 0.4040 - mse: 1.1941 - val\_loss: 0.5301 - val\_mse: 1.6413  
Epoch 36/100  
32/32 - 1s - 30ms/step - loss: 0.4031 - mse: 1.1910 - val\_loss: 0.5282 - val\_mse: 1.6322  
Epoch 37/100  
32/32 - 1s - 35ms/step - loss: 0.4023 - mse: 1.1890 - val\_loss: 0.5281 - val\_mse: 1.6320

Epoch 38/100  
 32/32 - 1s - 40ms/step - loss: 0.4024 - mse: 1.1889 - val\_loss: 0.5291 - val\_mse: 1.6351  
 Epoch 39/100  
 32/32 - 1s - 42ms/step - loss: 0.4009 - mse: 1.1821 - val\_loss: 0.5267 - val\_mse: 1.6235  
 Epoch 40/100  
 32/32 - 1s - 36ms/step - loss: 0.4010 - mse: 1.1773 - val\_loss: 0.5273 - val\_mse: 1.6281  
 Epoch 41/100  
 32/32 - 1s - 40ms/step - loss: 0.4014 - mse: 1.1823 - val\_loss: 0.5277 - val\_mse: 1.6296  
 Epoch 42/100  
 32/32 - 1s - 25ms/step - loss: 0.4010 - mse: 1.1808 - val\_loss: 0.5276 - val\_mse: 1.6296  
 Epoch 43/100  
 32/32 - 1s - 27ms/step - loss: 0.4003 - mse: 1.1770 - val\_loss: 0.5265 - val\_mse: 1.6241  
 Epoch 44/100  
 32/32 - 1s - 39ms/step - loss: 0.4002 - mse: 1.1788 - val\_loss: 0.5263 - val\_mse: 1.6240  
 Epoch 45/100  
 32/32 - 2s - 54ms/step - loss: 0.3999 - mse: 1.1820 - val\_loss: 0.5279 - val\_mse: 1.6318  
 Epoch 46/100  
 32/32 - 1s - 40ms/step - loss: 0.3999 - mse: 1.1766 - val\_loss: 0.5265 - val\_mse: 1.6250  
 Epoch 47/100  
 32/32 - 1s - 41ms/step - loss: 0.3989 - mse: 1.1732 - val\_loss: 0.5271 - val\_mse: 1.6281  
 Epoch 48/100  
 32/32 - 2s - 65ms/step - loss: 0.3995 - mse: 1.1761 - val\_loss: 0.5255 - val\_mse: 1.6187  
 Epoch 49/100  
 32/32 - 1s - 41ms/step - loss: 0.3987 - mse: 1.1734 - val\_loss: 0.5246 - val\_mse: 1.6150  
 Epoch 50/100  
 32/32 - 1s - 28ms/step - loss: 0.3988 - mse: 1.1703 - val\_loss: 0.5244 - val\_mse: 1.6142  
 Epoch 51/100  
 32/32 - 1s - 40ms/step - loss: 0.3987 - mse: 1.1741 - val\_loss: 0.5233 - val\_mse: 1.6089  
 Epoch 52/100  
 32/32 - 1s - 38ms/step - loss: 0.3973 - mse: 1.1678 - val\_loss: 0.5225 - val\_mse: 1.6047  
 Epoch 53/100  
 32/32 - 1s - 39ms/step - loss: 0.3981 - mse: 1.1714 - val\_loss: 0.5234 - val\_mse: 1.6089  
 Epoch 54/100  
 32/32 - 1s - 39ms/step - loss: 0.3985 - mse: 1.1725 - val\_loss: 0.5249 - val\_mse: 1.6174  
 Epoch 55/100  
 32/32 - 1s - 40ms/step - loss: 0.3967 - mse: 1.1694 - val\_loss: 0.5227 - val\_mse: 1.6049  
 Epoch 56/100  
 32/32 - 1s - 41ms/step - loss: 0.3964 - mse: 1.1624 - val\_loss: 0.5231 - val\_mse: 1.6049



```

6078
Epoch 57/100
32/32 - 3s - 80ms/step - loss: 0.3972 - mse: 1.1653 - val_loss: 0.5262 - val_mse: 1.6236
Epoch 58/100
32/32 - 2s - 62ms/step - loss: 0.3971 - mse: 1.1675 - val_loss: 0.5248 - val_mse: 1.6177
Epoch 59/100
32/32 - 1s - 39ms/step - loss: 0.3965 - mse: 1.1616 - val_loss: 0.5259 - val_mse: 1.6226
Epoch 60/100
32/32 - 1s - 24ms/step - loss: 0.3956 - mse: 1.1681 - val_loss: 0.5230 - val_mse: 1.6080
Epoch 61/100
32/32 - 1s - 40ms/step - loss: 0.3973 - mse: 1.1637 - val_loss: 0.5271 - val_mse: 1.6285
Epoch 62/100
32/32 - 1s - 24ms/step - loss: 0.3974 - mse: 1.1687 - val_loss: 0.5281 - val_mse: 1.6332
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])
Padded validation sequences shape: (4093, 30)
Validation labels shape: (4093,)
Shape of val_arrays: (4093, 30, 1)
Shape of val_labs: (4093,)
128/128 - 0s - 2ms/step - loss: 0.3209 - mse: 0.8381

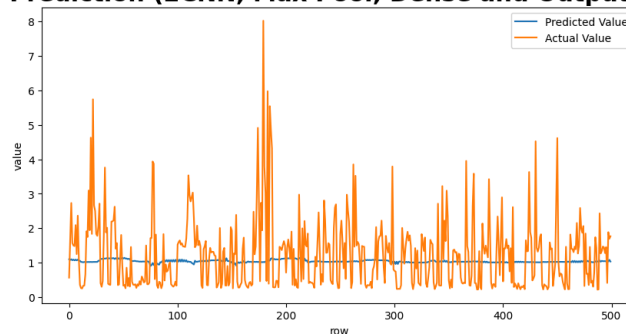
```

Validation Loss (Huber): 0.32089662551879883

MSE: 0.8380647301673889

128/128 ————— 0s 2ms/step

**Global Active Power Prediction (2CNN, Max Pool, Dense and Output Layer)- Last 500 Points**



```

In [ ]: # play with your ideas for optimization here
# III. Change sequence_length to 50 (from 30)

df = pd.read_csv(file_path, parse_dates=['Datetime'])

# Display the first few rows of the DataFrame to verify it loaded correctly
print(df.head())

#create training and validation sets here
# Sort the data by Datetime to ensure it's in chronological order
df = df.sort_values('Datetime')

# Assign size for data subset (1% of the entire dataset)

```

```

# Take random data subset
#n=subset_size: This argument specifies the number of rows needed in random subset.
#random_state=42: This is crucial for reproducibility. It sets the seed for the random
#               Incase this code is run multiple times with the same random_state, this will
#               debugging, and sharing results. IF this argument is None, then the same
df_subset = df.sample(n=subset_size, random_state=42)

# Sort the subset by Datetime again to maintain chronological order
df_subset = df_subset.sort_values('Datetime')

# Split data subset 80/20 for train/validation
# Split the df_subset which is a random sample of the original DataFrame, its index
# We use the index to split to maintain the time order
#len(df_subset): Gets the number of rows in the df_subset.
#0.8 * len(df_subset): Calculates 80% of the number of rows. This determines the split
#int(...): Converts the result to an integer. This is necessary because slice indices
#split_index: Stores the calculated index value. This index marks the boundary between

split_index = int(0.8 * len(df_subset))

# This creates the training set. It takes all rows from df_subset up to (but not including)
# Because the original index from the dataframe is used, and because the subset was sorted
# this subset will maintain the time order.
# train_df: Stores the resulting training DataFrame

train_df = df_subset[:split_index]

# Create the validation set. It takes all rows from df_subset starting from the row
val_df = df_subset[split_index:]

#reset the indices for cleanliness
#After splitting the data, the original index from the full dataset is retained. Re-indexing
#This makes it easier to work with the data, especially when iterating over rows or
train_df = train_df.reset_index()
val_df = val_df.reset_index()

print(f"Training set shape: {train_df.shape}")
print(f"Validation set shape: {val_df.shape}")

# show me how one or two of your different models perform
# using the code from the "Validating our model" section above

seq_arrays = []
seq_labs = []
import numpy as np

# Get GAP readings from your DataFrame column
gap_readings = train_df['Global_active_power'].values # Direct numpy array access

#use 30 consecutive measurements to predict value 5 steps in future
seq_length = 50 # 50-minute window
ph = 5 # Predict 5 minutes ahead

seq_arrays = train_df['Global_active_power'].values # Define feature columns

```

```

# Create sequences with alignment for predictive horizon
seq_arrays = []
seq_labs = []

# Calculate valid range for sequence creation

# Total Data Points: Len(gap_readings) = 16,359 historical GAP measurements.
#Sequence Length: Each input sequence uses 30 consecutive measurements.
#Predictive Horizon: To predict the GAP value 5 steps ahead of the input sequence.
#Input Sequence: Each sequence starts at index i and ends at i + seq_length - 1.
#Output Value: Predicted value is at index i + seq_length + ph - 1.
#Last Valid Sequence:
#Start index i must satisfy:
#i + seq_length + ph - 1 < Len(gap_readings)
#For the dataset with Len = 16359:
# i + 30 + 5 - 1 < 16,359 → i < 16,359 - 34 → i ≤ 16,324
# Total sequences: 16,325 (i ranges from 0 to 16,324).

num_samples = len(gap_readings) - seq_length - ph + 1

#Sequence creation loop
#Input Sequence: Sliding window of 30 consecutive values
#Output Value: Value 5 steps after the end of the input window

for i in range(num_samples):
    # Input: 30 consecutive GAP measurements
    input_seq = gap_readings[i:i+seq_length]

    # Output: GAP value 5 cycles after sequence end
    output_val = gap_readings[i + seq_length + ph - 1]

    # append(input_seq): adds the current input_seq (which is itself a sequence of
    # So, seq_arrays will become a list of lists (or a list of arrays).
    #input_seq is a sequence of 30 consecutive GAP measurements (because seq_length
    #seq_arrays is a list that will contain all these sequences.
    #.append() adds input_seq as a new element to the end of seq_arrays.
    #After this operation, seq_arrays becomes a list of lists, where each inner list
    # Example:
    # seq_arrays = [
    # [0.5, 0.6, ..., 0.8], # First sequence (30 values)
    # [0.6, 0.7, ..., 0.9], # Second sequence (30 values)
    # ...
    seq_arrays.append(input_seq) #store the input sequence of time series data in n

    # add the output_val to the seq_labs list
    # output_val is a single GAP measurement, 5 steps ahead of the end of the input
    # seq_labs is a list that will contain all these future values.
    # append(): This adds output_val as a new element to the end of seq_labs.
    # After this operation, seq_labs becomes a list of individual GAP values, each
    # Example:
    # python
    # seq_labs = [0.7, 0.8, 0.9, ...] # Each value corresponds to a sequence in seq_arrays
    seq_labs.append(output_val)

```

```

seq_arrays = np.array(seq_arrays, dtype=np.float32).reshape(-1, seq_length, len(feats))
seq_labs = np.array(seq_labs, dtype=np.float32)

print("Input shape:", seq_arrays.shape)
print("Output shape:", seq_labs.shape)

# Verify dimensions with corrected calculation
expected_samples = len(train_df) - seq_length - ph + 1
expected_shape = (expected_samples, seq_length, len(feats_cols))

try:
    assert seq_arrays.shape == expected_shape, (
        f"Sequence array shape mismatch. Got {seq_arrays.shape}, "
        f"expected {expected_shape}. Check data length: {len(train_df)} "
        f"should be ≥ {seq_length + ph}"
    )

    assert seq_labs.shape == (expected_samples,), (
        f"Label shape mismatch. Got {seq_labs.shape}, "
        f"expected ({expected_samples},)"
    )

    print("All assertions passed!")
except AssertionError as e:
    print(f"Assertion Error: {e}")
    print(f"Required data length: {seq_length + ph - 1}")
    print(f"Actual data length: {len(train_df)}")

assert(seq_arrays.shape == (len(train_df)-seq_length-ph+1, seq_length, len(feats_cols)))
assert(seq_labs.shape == (len(train_df)-seq_length-ph+1,))
# define path to save model
model_path = 'LSTM_model1.keras'

# build the network
# Onfe Feature which is GAP and one Output value (GAP) is predicted
nb_features = 1
nb_out = 1

# Initialize a Sequential model
model = Sequential()
model.add(Input(shape=(seq_length, nb_features))) # (30, 1) based on our sequence
model.add(LSTM(units=5, return_sequences=True)) # Number of LSTM units (neurons) in

#add first LSTM layer
model.add(Dropout(0.2)) # Return full sequence for next LSTM layer

# add second LSTM layer
model.add(LSTM(
    units=3, #number of LSTM units in this layer 2 of LSTM
    return_sequences=False)) # do not return

# Add dropout for regularization
model.add(Dropout(0.2))

# Add Dense (fully connected) output layer
model.add(Dense(units=nb_out)) # Output layer with 1 unit for regression
model.add(Activation("linear")) # Linear activation for regression

```

```

# Define optimizer with Learning rate
optimizer = keras.optimizers.Adam(learning_rate = 0.01)

# Loss function is MSE, Adam optimizer, track MSE during training
# uses Mean Squared Error as both the loss function and a metric, appropriate for regression
model.compile(loss='mean_squared_error', optimizer=optimizer, metrics=['mse'])

print(model.summary())

# fit the network (train it)
# max number of training epochs = 100
# number of samples per gradient update (batch size) = 500
# use 5% of data for validation
# verbosity mode = 2, one line per epoch
# callback - early stopping to prevent overfitting and if no more learning is happening
# monitor='val_loss', # Monitor validation loss
# min_delta=0: Minimum change to qualify as improvement
# patience=10: Number of epochs with no improvement after which training will stop
# verbose=0: Verbosity mode
# mode='min' # The direction is automatically inferred if not set

# Model checkpoint to save best model
# model_path, # Path to save the model
# monitor='val_loss', # Monitor validation loss
# save_best_only=True, # Only save when the model is better than the previous best
# mode='min', # Lower validation loss is better
# verbose=0 # Verbosity mode

history = model.fit(seq_arrays, seq_labels, epochs=100, batch_size=500, validation_split=0.05,
                    callbacks = [keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0.001, patience=10),
                                keras.callbacks.ModelCheckpoint(model_path, monitor='val_loss', save_best_only=True)]
                    )

# List all data in history
# After training, it prints the keys of the history object, which typically include
print(history.history.keys())

# show me how one or two of your different models perform
# using the code from the "Validating our model" section above

val_arrays = []
val_labels = []

# Create sequences starting with minimum of 2 readings, building up to seq_length
for i in range(2, len(val_df['Global_active_power'])):
    # Get all available readings up to current index
    raw_seq = val_df['Global_active_power'].values[:i] # Get readings from start to current index

    # Get label (ph steps ahead)
    if i + ph - 1 < len(val_df): # Check if we have enough data for the label
        val_labels.append(val_df['Global_active_power'].values[i + ph - 1])
    else: # Handle edge case at end of dataset
        break

```

```

# Pad sequences to uniform length (seq_length) with pre-padding (zeros at beginning)
val_arrays = pad_sequences(
    val_arrays,
    maxlen=seq_length, # Force all sequences to be seq_length long
    dtype='float32',    # Match TensorFlow's default dtype
    padding='pre',      # Add zeros at BEGINNING of sequences (preserve recent data)
    truncating='pre',   # Truncate from beginning if sequences are too long (should be 'post')
    value=0             # Use 0 for padding (consider using mean value if better for model)
)

# Convert labels to numpy array with proper dtype
val_labs = np.array(val_labs, dtype=np.float32)

print("Padded validation sequences shape:", val_arrays.shape)
print("Validation labels shape:", val_labs.shape)

# Reshape val_arrays to add the feature dimension
val_arrays = np.reshape(val_arrays, (val_arrays.shape[0], val_arrays.shape[1], nb_features))

# Ensure val_labs is a 1D array
val_labs = np.reshape(val_labs, (-1,))

# Debugging: Print shapes
print("Shape of val_arrays:", val_arrays.shape)
print("Shape of val_labs:", val_labs.shape)

# Evaluate the model
scores_test = model.evaluate(val_arrays, val_labs, verbose=2)
print('\nMSE: {}'.format(scores_test[1]))

# Make predictions
y_pred_test = model.predict(val_arrays)
y_true_test = val_labs

# Save predictions to CSV
test_set = pd.DataFrame(y_pred_test)
test_set.to_csv('submit_test.csv', index=None)

# Plot the predicted data vs. the actual data
fig_verify = plt.figure(figsize=(10, 5))
plt.plot(y_pred_test[-500:], label='Predicted Value')
plt.plot(y_true_test[-500:], label='Actual Value')
plt.title('Global Active Power Prediction (seq_len = 50)- Last 500 Points', fontsize=14)
plt.ylabel('value')
plt.xlabel('row')
plt.legend()
plt.show()
fig_verify.savefig("model_regression_verify.png")

```

```

      Unnamed: 0      Date      Time      Global_active_power  \
0      0      2006-12-16  17:24:00      4.216
1      1      2006-12-16  17:25:00      5.360
2      2      2006-12-16  17:26:00      5.374
3      3      2006-12-16  17:27:00      5.388
4      4      2006-12-16  17:28:00      3.666

      Global_reactive_power  Voltage  Global_intensity  Sub_metering_1  \
0      0.418      234.84      18.4      0.0
1      0.436      233.63      23.0      0.0
2      0.498      233.29      23.0      0.0
3      0.502      233.74      23.0      0.0
4      0.528      235.68      15.8      0.0

      Sub_metering_2  Sub_metering_3      Datetime  gap_monthly  \
0      1.0      17.0  2006-12-16  17:24:00      NaN
1      1.0      16.0  2006-12-16  17:25:00      NaN
2      2.0      17.0  2006-12-16  17:26:00      NaN
3      1.0      17.0  2006-12-16  17:27:00      NaN
4      1.0      17.0  2006-12-16  17:28:00      NaN

      grp_monthly  v_monthly  gi_monthly
0      NaN      NaN      NaN
1      NaN      NaN      NaN
2      NaN      NaN      NaN
3      NaN      NaN      NaN
4      NaN      NaN      NaN

```

Training set shape: (16393, 16)  
 Validation set shape: (4099, 16)  
 Input shape: (16339, 50, 1)  
 Output shape: (16339,)  
 All assertions passed!  
**Model: "sequential\_27"**

Layer (type)	Output Shape	
lstm_44 (LSTM)	(None, 50, 5)	
dropout_61 (Dropout)	(None, 50, 5)	
lstm_45 (LSTM)	(None, 3)	
dropout_62 (Dropout)	(None, 3)	
dense_28 (Dense)	(None, 1)	
activation_11 (Activation)	(None, 1)	



Total params: 252 (1008.00 B)  
 Trainable params: 252 (1008.00 B)  
 Non-trainable params: 0 (0.00 B)

None

Epoch 1/100  
 32/32 - 5s - 160ms/step - loss: 1.5139 - mse: 1.5139 - val\_loss: 1.6885 - val\_mse: 1.6885

Epoch 2/100  
 32/32 - 1s - 45ms/step - loss: 1.2132 - mse: 1.2132 - val\_loss: 1.5449 - val\_mse: 1.5449

Epoch 3/100  
 32/32 - 1s - 44ms/step - loss: 1.1881 - mse: 1.1881 - val\_loss: 1.5307 - val\_mse: 1.5307

Epoch 4/100  
 32/32 - 2s - 66ms/step - loss: 1.1654 - mse: 1.1654 - val\_loss: 1.5127 - val\_mse: 1.5127

Epoch 5/100  
 32/32 - 2s - 72ms/step - loss: 1.1634 - mse: 1.1634 - val\_loss: 1.4895 - val\_mse: 1.4895

Epoch 6/100  
 32/32 - 2s - 54ms/step - loss: 1.1559 - mse: 1.1559 - val\_loss: 1.4858 - val\_mse: 1.4858

Epoch 7/100  
 32/32 - 3s - 78ms/step - loss: 1.1440 - mse: 1.1440 - val\_loss: 1.4896 - val\_mse: 1.4896

Epoch 8/100  
 32/32 - 3s - 79ms/step - loss: 1.1346 - mse: 1.1346 - val\_loss: 1.4820 - val\_mse: 1.4820

Epoch 9/100  
 32/32 - 3s - 80ms/step - loss: 1.1254 - mse: 1.1254 - val\_loss: 1.4723 - val\_mse: 1.4723

Epoch 10/100  
 32/32 - 2s - 49ms/step - loss: 1.1295 - mse: 1.1295 - val\_loss: 1.4912 - val\_mse: 1.4912

Epoch 11/100  
 32/32 - 3s - 105ms/step - loss: 1.1223 - mse: 1.1223 - val\_loss: 1.4614 - val\_mse: 1.4614

Epoch 12/100  
 32/32 - 1s - 46ms/step - loss: 1.1172 - mse: 1.1172 - val\_loss: 1.4522 - val\_mse: 1.4522

Epoch 13/100  
 32/32 - 1s - 43ms/step - loss: 1.1193 - mse: 1.1193 - val\_loss: 1.4584 - val\_mse: 1.4584

Epoch 14/100  
 32/32 - 1s - 44ms/step - loss: 1.1154 - mse: 1.1154 - val\_loss: 1.4835 - val\_mse: 1.4835

Epoch 15/100  
 32/32 - 1s - 44ms/step - loss: 1.1140 - mse: 1.1140 - val\_loss: 1.4653 - val\_mse: 1.4653

Epoch 16/100  
 32/32 - 3s - 79ms/step - loss: 1.1159 - mse: 1.1159 - val\_loss: 1.4403 - val\_mse: 1.4403

Epoch 17/100  
 32/32 - 3s - 89ms/step - loss: 1.1445 - mse: 1.1445 - val\_loss: 1.5281 - val\_mse: 1.5281

Epoch 18/100  
 32/32 - 3s - 98ms/step - loss: 1.1137 - mse: 1.1137 - val\_loss: 1.4592 - val\_mse: 1.4592

Epoch 19/100  
 32/32 - 3s - 98ms/step - loss: 1.1137 - mse: 1.1137 - val\_loss: 1.4592 - val\_mse: 1.4592



```

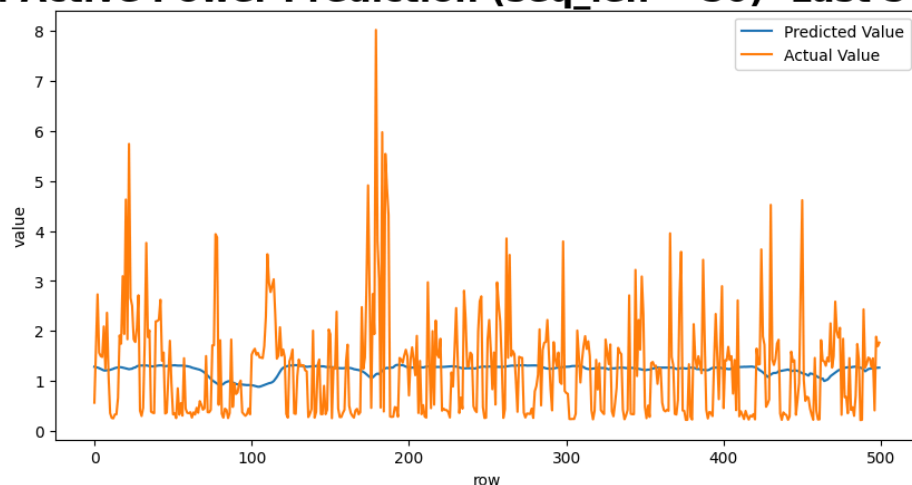
32/32 - 2s - 52ms/step - loss: 1.1103 - mse: 1.1103 - val_loss: 1.4641 - val_mse: 1.4641
Epoch 20/100
32/32 - 2s - 78ms/step - loss: 1.1109 - mse: 1.1109 - val_loss: 1.4588 - val_mse: 1.4588
Epoch 21/100
32/32 - 3s - 80ms/step - loss: 1.1120 - mse: 1.1120 - val_loss: 1.4535 - val_mse: 1.4535
Epoch 22/100
32/32 - 1s - 44ms/step - loss: 1.1115 - mse: 1.1115 - val_loss: 1.4675 - val_mse: 1.4675
Epoch 23/100
32/32 - 3s - 87ms/step - loss: 1.1141 - mse: 1.1141 - val_loss: 1.4597 - val_mse: 1.4597
Epoch 24/100
32/32 - 3s - 100ms/step - loss: 1.1083 - mse: 1.1083 - val_loss: 1.4548 - val_mse: 1.4548
Epoch 25/100
32/32 - 2s - 52ms/step - loss: 1.1096 - mse: 1.1096 - val_loss: 1.4570 - val_mse: 1.4570
Epoch 26/100
32/32 - 2s - 72ms/step - loss: 1.1117 - mse: 1.1117 - val_loss: 1.4663 - val_mse: 1.4663
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])
Padded validation sequences shape: (4093, 50)
Validation labels shape: (4093,)
Shape of val_arrays: (4093, 50, 1)
Shape of val_labs: (4093,)
128/128 - 1s - 7ms/step - loss: 0.8302 - mse: 0.8302

```

MSE: 0.830170214176178

128/128 ————— 1s 9ms/step

## Global Active Power Prediction (seq\_len = 50)- Last 500 Points



```

In [ ]: # play with your ideas for optimization here
        # IV Model change - Adding Multivariate (Voltage, Global Intensity and Global React

        # Load data
        df = pd.read_csv('/content/drive/MyDrive/AAI-530-DataAnalyticsAndIOT/household_powe

        # Data preprocessing

```

```

df = df.sort_values('Datetime')
subset_size = int(0.01 * len(df))
df_subset = df.sample(n=subset_size, random_state=42).sort_values('Datetime')
split_index = int(0.8 * len(df_subset))
train_df = df_subset[:split_index].reset_index()
val_df = df_subset[split_index:].reset_index()

# Define feature columns
feat_cols = ['Global_active_power', 'Voltage', 'Global_intensity', 'Global_reactive

# Sequence creation with multiple features
seq_length = 30
ph = 5
num_samples = len(train_df) - seq_length - ph + 1

# show me how one or two of your different models perform
# using the code from the "Validating our model" section above
seq_arrays = []
seq_labs = []

for i in range(num_samples):
    input_seq = train_df[feat_cols].values[i:i+seq_length]
    output_val = train_df['Global_active_power'].values[i + seq_length + ph - 1]
    seq_arrays.append(input_seq)
    seq_labs.append(output_val)

seq_arrays = np.array(seq_arrays, dtype=np.float32)
seq_labs = np.array(seq_labs, dtype=np.float32)

# Model building
model_path = 'LSTM_model_multivariate.keras'
nb_features = len(feat_cols)
nb_out = 1

model = Sequential()

model.add(Input(shape=(seq_length, nb_features))) # Explicit input layer
model.add(LSTM(units=32, return_sequences=True))
model.add(Dropout(0.2))

model.add(LSTM(units=16, return_sequences=False))
model.add(Dropout(0.2))
model.add(Dense(units=8, activation='relu'))
model.add(Dense(units=nb_out))
model.add(Activation("linear"))

optimizer = keras.optimizers.Adam(learning_rate=0.01)
model.compile(loss='mean_squared_error', optimizer=optimizer, metrics=['mse'])

# Model training
history = model.fit(
    seq_arrays, seq_labs,
    epochs=100,
    batch_size=500,
    validation_split=0.05,

```

```

callbacks=[
    keras.callbacks.EarlyStopping(monitor='val_loss', patience=10, mode='min'),
    keras.callbacks.ModelCheckpoint(model_path, monitor='val_loss', save_best_o
]
)

# Validation data preparation
val_arrays = []
val_labs = []

for i in range(2, len(val_df)):
    if i + ph - 1 < len(val_df):
        raw_seq = val_df[feat_cols].values[:i]
        val_arrays.append(raw_seq)
        val_labs.append(val_df['Global_active_power'].values[i + ph - 1])


val_arrays = pad_sequences(val_arrays, maxlen=seq_length, dtype='float32', padding=
val_labs = np.array(val_labs, dtype=np.float32)
val_arrays = np.reshape(val_arrays, (val_arrays.shape[0], val_arrays.shape[1], nb_f

# Model evaluation and prediction
scores_test = model.evaluate(val_arrays, val_labs, verbose=2)
print(f'MSE: {scores_test[1]}')
y_pred_test = model.predict(val_arrays)
y_true_test = val_labs

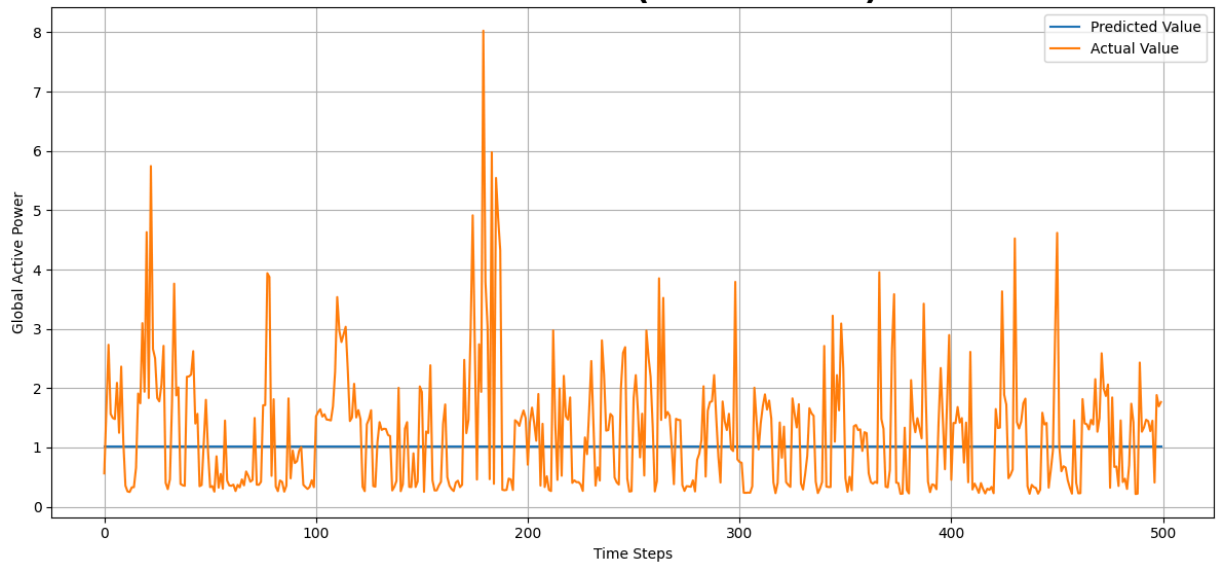
# Visualization
plt.figure(figsize=(12, 6))
plt.plot(y_pred_test[-500:], label='Predicted Value')
plt.plot(y_true_test[-500:], label='Actual Value')
plt.title('Global Active Power Prediction (Multivariate) - Last 500 Points',
          fontsize=22, fontweight='bold')
plt.ylabel('Global Active Power')
plt.xlabel('Time Steps')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("multivariate_model_prediction.png")
plt.show()

# Plot training history
plt.figure(figsize=(10, 5))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss Over Time')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.savefig("training_history.png")
plt.show()

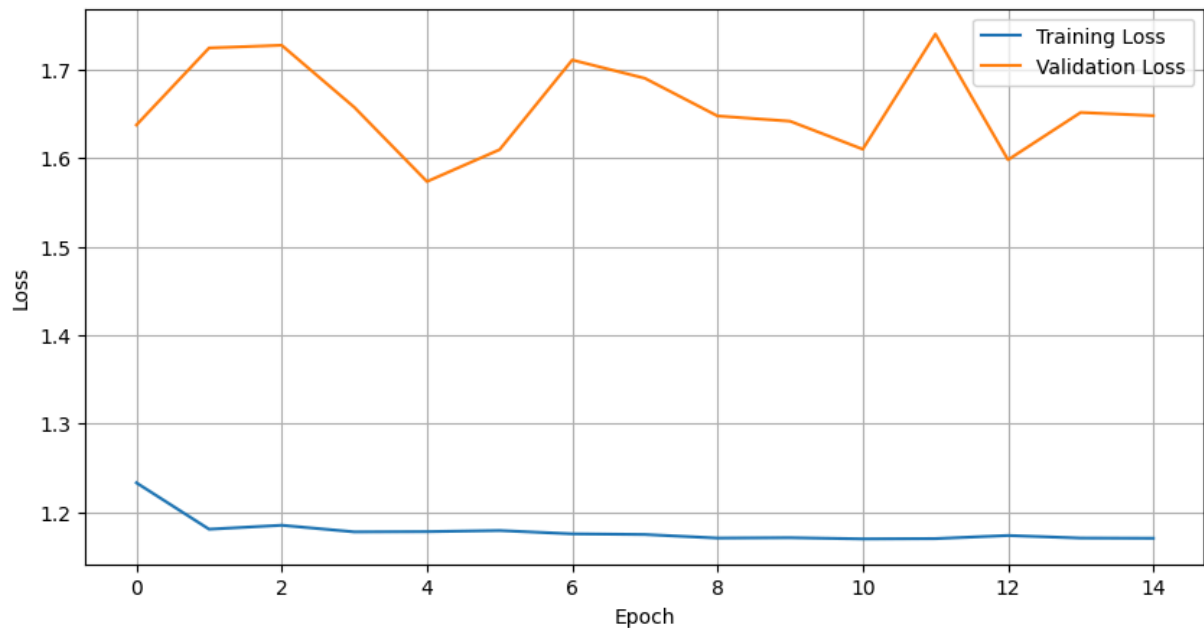
```

Epoch 1/100  
32/32 - 9s - 268ms/step - loss: 1.2329 - mse: 1.2329 - val\_loss: 1.6376 - val\_mse: 1.6376  
Epoch 2/100  
32/32 - 3s - 81ms/step - loss: 1.1805 - mse: 1.1805 - val\_loss: 1.7246 - val\_mse: 1.7246  
Epoch 3/100  
32/32 - 5s - 167ms/step - loss: 1.1849 - mse: 1.1849 - val\_loss: 1.7277 - val\_mse: 1.7277  
Epoch 4/100  
32/32 - 3s - 82ms/step - loss: 1.1775 - mse: 1.1775 - val\_loss: 1.6575 - val\_mse: 1.6575  
Epoch 5/100  
32/32 - 5s - 151ms/step - loss: 1.1778 - mse: 1.1778 - val\_loss: 1.5736 - val\_mse: 1.5736  
Epoch 6/100  
32/32 - 3s - 80ms/step - loss: 1.1790 - mse: 1.1790 - val\_loss: 1.6097 - val\_mse: 1.6097  
Epoch 7/100  
32/32 - 4s - 124ms/step - loss: 1.1752 - mse: 1.1752 - val\_loss: 1.7110 - val\_mse: 1.7110  
Epoch 8/100  
32/32 - 3s - 108ms/step - loss: 1.1745 - mse: 1.1745 - val\_loss: 1.6903 - val\_mse: 1.6903  
Epoch 9/100  
32/32 - 2s - 71ms/step - loss: 1.1705 - mse: 1.1705 - val\_loss: 1.6477 - val\_mse: 1.6477  
Epoch 10/100  
32/32 - 3s - 79ms/step - loss: 1.1709 - mse: 1.1709 - val\_loss: 1.6418 - val\_mse: 1.6418  
Epoch 11/100  
32/32 - 2s - 75ms/step - loss: 1.1695 - mse: 1.1695 - val\_loss: 1.6102 - val\_mse: 1.6102  
Epoch 12/100  
32/32 - 4s - 125ms/step - loss: 1.1699 - mse: 1.1699 - val\_loss: 1.7404 - val\_mse: 1.7404  
Epoch 13/100  
32/32 - 3s - 106ms/step - loss: 1.1734 - mse: 1.1734 - val\_loss: 1.5982 - val\_mse: 1.5982  
Epoch 14/100  
32/32 - 3s - 80ms/step - loss: 1.1705 - mse: 1.1705 - val\_loss: 1.6517 - val\_mse: 1.6517  
Epoch 15/100  
32/32 - 2s - 72ms/step - loss: 1.1703 - mse: 1.1703 - val\_loss: 1.6481 - val\_mse: 1.6481  
128/128 - 1s - 8ms/step - loss: 0.8545 - mse: 0.8545  
MSE: 0.8545377254486084  
**128/128**  **2s** 11ms/step

## Global Active Power Prediction (Multivariate) - Last 500 Points



## Model Loss Over Time



In [ ]: *# V. Architerctural change - CNN feature extraction, Bidirectional LSTM with attent*

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import (Input, Conv1D, LSTM, Dense, Dropout,
                                     Bidirectional, LayerNormalization, Attention,
                                     concatenate)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

```

# Load the dataset
file_path = '/content/drive/MyDrive/AAI-530-DataAnalyticsAndIoT/household_power_cle

df = pd.read_csv(file_path, parse_dates=['Datetime'])

# 1. Data Preparation Functions
def load_data(file_path):
    df = pd.read_csv(file_path, parse_dates=['Datetime'], index_col='Datetime')
    return df.sort_index()

def create_sequences(data, seq_length, pred_horizon):
    xs, ys = [], []
    for i in range(len(data)-seq_length-pred_horizon+1):
        x = data[i:(i+seq_length)]
        y = data[i+seq_length+pred_horizon-1]
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)

# 2. Enhanced Model Architecture
def build_cnn_lstm_attention(seq_length=30, n_features=1):
    inputs = Input(shape=(seq_length, n_features))

    # CNN Feature Extraction
    x = Conv1D(64, kernel_size=3, activation='relu', padding='same')(inputs)
    x = LayerNormalization()(x)
    x = Dropout(0.3)(x)

    # Bidirectional LSTM
    x = Bidirectional(LSTM(128, return_sequences=True))(x)

    # Attention Mechanism
    attention = Attention()([x, x])
    x = concatenate([x, attention])
    x = LayerNormalization()(x)
    x = Dropout(0.4)(x)

    # Temporal Pooling
    x = LSTM(64, return_sequences=False)(x)
    x = Dense(32, activation='relu', kernel_regularizer='l2')(x)

    # Output
    outputs = Dense(1)(x)

    return Model(inputs=inputs, outputs=outputs)

# 3. Training Configuration
def train_model(model, X_train, y_train):
    optimizer = Adam(learning_rate=0.001)
    model.compile(loss='mse', optimizer=optimizer, metrics=['mae'])

    callbacks = [
        EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True),
        ModelCheckpoint('best_model.keras', save_best_only=True)
    ]

```

```

history = model.fit(
    X_train, y_train,
    epochs=2,
    batch_size=128,
    validation_split=0.2,
    callbacks=callbacks,
    verbose=1
)
return history

# 4. Main Workflow
if __name__ == "__main__":
    # Load and preprocess data
    #df = load_data('household_power_clean.csv')
    data = df[['Global_active_power']].values

    # Parameters
    SEQ_LENGTH = 30
    PRED_HORIZON = 5

    # Create sequences
    X, y = create_sequences(data, SEQ_LENGTH, PRED_HORIZON)
    X = X.reshape(-1, SEQ_LENGTH, 1) # Add feature dimension

    # Train-test split
    split = int(0.8 * len(X))
    X_train, y_train = X[:split], y[:split]
    X_test, y_test = X[split:], y[split:]

    # Build and train model
    model = build_cnn_lstm_attention(SEQ_LENGTH)
    history = train_model(model, X_train, y_train)

    # Evaluate
    test_loss = model.evaluate(X_test, y_test, verbose=0)
    print(f"\nTest MSE: {test_loss[0]:.4f}, MAE: {test_loss[1]:.4f}")

    # Generate predictions
    y_pred = model.predict(X_test)

    # Visualization
    plt.figure(figsize=(12,6))
    plt.plot(y_test[-500:], label='True')
    plt.plot(y_pred[-500:], label='Predicted')
    plt.title("CNN-LSTM with Attention: Prediction vs Actual")
    plt.xlabel("Time Steps")
    plt.ylabel("Global Active Power")
    plt.legend()
    plt.show()

```

Mounted at /content/drive

Epoch 1/2

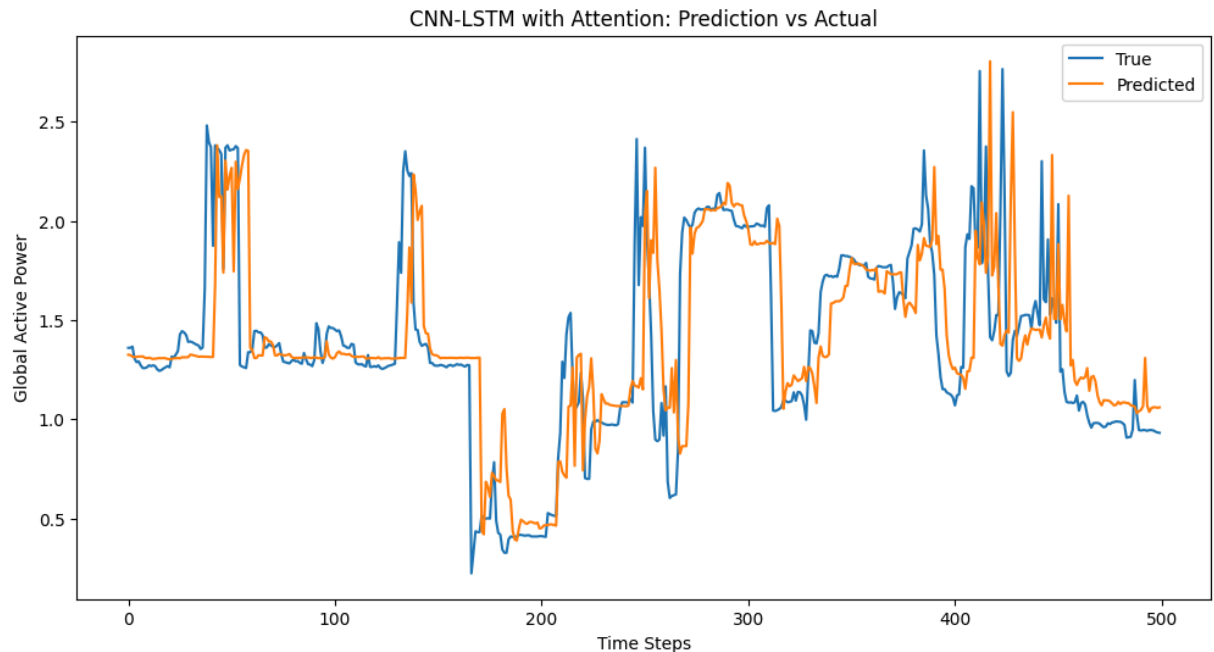
10247/10247 ————— 4494s 438ms/step - loss: 0.3704 - mae: 0.2869 - val\_loss: 0.2304 - val\_mae: 0.2445

Epoch 2/2

10247/10247 ————— 4456s 435ms/step - loss: 0.2668 - mae: 0.2540 - val\_loss: 0.2272 - val\_mae: 0.2454

Test MSE: 0.1924, MAE: 0.2314

12808/12808 ————— 697s 54ms/step



**Q: How did your model changes affect performance on the validation data? Why do you think they were/were not effective? If you were trying to optimize for production, what would you try next?**

Answer:

## I. HOW MODEL CHANGE AFFECTED PERFORMANCE:

The following are the Models Built in this assignment and their MSE and their performance on validation data.

1. LSTM((Increasing number of LSTM layers to 5 - Sequence Length = 30, PH = 5): MSE (0.8221). This model has high MSE and was not predicting the power well learning the dynamic aspects of Power consumption. When the Actual power consumption is at the peak or lowest, the model doesn't predict the power consumed.
2. Architectural change 2 convolutional layers with max pooling layer, Dense(Fully connected) layers and output layer): MSE: (0.8413). This model has high MSE and does not predict the power well by learning the dynamic aspects of Power consumption.



When the Actual power consumption is at the peak or lowest, the model doesn't predict the power consumed.

3. LSTM (2 Layers - Sequence Length = 50, PH = 5): MSE (0.829): This model has high MSE and does not predict the power well learning the dynamic aspects of Power consumption. When the Actual power consumption is at the peak or lowest, the model doesn't predict the power consumed.
4. Adding Multivariate (Voltage, Global Intensity and Global Reactive power) as Dependent variables), MSE (0.854): This model has high MSE and does not predict the power well learning the dynamic aspects of Power consumption. When the Actual power consumption is at the peak or lowest, the model doesn't predict the power consumed.
5. Architectural change - CNN feature Extraction and Bidirectional LSTM with attention mechanism, Temporal Pooling and output layer, MSE with 2 Epochs( 0.2314): This model has comparatively low MSE and aligns with the high and low power consumption points with a lag which indicates that the model is not effective in predicting the power consumption before 5 minutes for a household.

## II. OPTIMIZATION STEPS TO IMPROVE THE MODEL:

With the tests inferring that Linear Regression model with Moving Average outperforms more complex models like LSTM and CNN-Bidirectional LSTM in terms of MSE (0.038), but lags in capturing peaks and low points of power consumption, the model to choose is the model with improved ability to capture non-linear patterns and temporal dependencies in the data.

## Selection of Hybrid Models:

1. Combining the Linear Regression model with a non-linear model (e.g., Random Forest, Gradient Boosting, or Neural Networks) to capture both linear and non-linear patterns could be tried.
2. Ensemble approach: Using an ensemble approach (e.g., stacking or weighted averaging) to combine predictions from multiple models.

### 3. Advanced Time Series Models:

Below Time series models could be tried.

- a. Prophet: A time series forecasting model developed by Facebook that handles seasonality, trends, and holidays well.
- b. ARIMA/SARIMA: Traditional time series models that can capture seasonality and trends.

c. XGBoost/LightGBM: Gradient boosting models that can handle non-linear relationships and feature interactions. It can handle non-linear relationships and feature interactions, making it well-suited for capturing complex patterns.

It can incorporate temporal and seasonal features (e.g., time of day, day of week, month, festivals) effectively.

#### 4. Model Architecture Improvements

a. LSTM/GRU with Attention: Addition of an attention mechanism to LSTM/GRU models to focus on important time steps (e.g., peaks and low points).

b. Transformer for Time Series: Using a Transformer-based model with self-attention to capture long-term dependencies and seasonal patterns.

c. TCN (Temporal Convolutional Network): Using a TCN with dilated convolutions to capture long-term dependencies efficiently.

5 ## Hyperparameter Tuning Usage of tools like Grid Search, Random Search, or Bayesian Optimization to find the best hyperparameters for your models.

6. Focus on tuning:

a. Learning rate, number of layers, number of units (for neural networks).

b. Window size, number of lags, and rolling statistics (for feature engineering).

7. Error Analysis

Analysis of the residuals (errors) of the Linear Regression model to identify patterns (e.g., systematic underprediction during peaks).

**Q: How did the models that you built in this assignment compare to the linear regression model from last week? Think about model performance and other IoT device considerations; Which model would you choose to use in an IoT system that predicts GAP for a single household with a 5-minute predictive horizon, and why?**

Answer:

## 1. I MODELS BUILT IN THIS ASSIGNMENT and its performance:

A: The following are the Models Built in this assignment and their MSE

1. LSTM((Increasing number of LSTM layers to 5 - Sequence Length = 30, PH = 5): MSE (0.8221). This model has high MSE and was not predicting the power well learning the

dynamic aspects of Power consumption. When the Actual power consumption is at the peak or lowest, the model doesn't predict the power consumed.

2. Architectural change 2 convolutional layers with max pooling layer, Dense(Fully connected) layers and output layer): MSE: (0.8413). This model has high MSE and does not predict the power well by learning the dynamic aspects of Power consumption. When the Actual power consumption is at the peak or lowest, the model doesn't predict the power consumed.
3. LSTM (2 Layers - Sequence Length = 50, PH = 5): MSE (0.829): This model has high MSE and does not predict the power well learning the dynamic aspects of Power consumption. When the Actual power consumption is at the peak or lowest, the model doesn't predict the power consumed.
4. Adding Multivariate (Voltage, Global Intensity and Global Reactive power) as Dependent variables), MSE (0.854): This model has high MSE and does not predict the power well learning the dynamic aspects of Power consumption. When the Actual power consumption is at the peak or lowest, the model doesn't predict the power consumed.
5. Architectural change - CNN feature Extraction and Bidirectional LSTM with attention mechanism, Temporal Pooling and output layer, MSE with 2 Epochs( 0.2314): This model has comparatively low MSE and aligns with the high and low power consumption points with a lag which indicates that the model is not effective in predicting the power consumption before 5 minutes for a household.

## II. INFERENCE:

### COMPARISON OF MODELS WITH LINEAR REGRESSION MODEL:

The "Linear Regression model with moving Average" which I had implemented and build shows the lowest MSE (0.038) in predicting the Global Active Power, significantly outperforming the more complex LSTM models(0.822) and CNN with Bidirectional LSTM with 2 Epochs (0.2314)

The Linear Regression moving average based model has improved well when compared to the Prediction Horizon and Forgetting factor based model which I had built. The moving average (MA) model which is a common approach for analyzing and forecasting time series data. The model assumes that the current value is a linear combination of past errors. It smooths out short-term fluctuations and highlights longer-term trends in the data. The order  $q$  determines how many past error terms are considered. MA model uses past forecast errors (the difference between the actual value and the forecasted value) to predict the current value. It assumes that these past errors provide information about the current and future

Though the Linear Regression with moving average model performed better ( $MSE = 0.038$ ), there was lag: at certain points (particularly in high peaks and low power), in the predicted power compared to the Actual power. This could be due to the model's inability to perfectly predict the timing of power changes.

### III. DEEP LEARNING MODEL:

The CNN Feature Extraction, Bidirectional LSTM with attention mechanism, Temporal Pooling and output layer, MSE with 2 Epochs showed better performance with MSE of 0.2314. Below is the description of this model

#### A. CNN Feature Extraction:

Convolutional Neural Networks (CNNs) are excellent at extracting spatial features from input data, such as images or sequences. This step helps in capturing important patterns and reducing the dimensionality of the input.

#### B. Bidirectional LSTM with Attention Mechanism:

Long Short-Term Memory (LSTM) networks are a type of Recurrent Neural Network (RNN) that can capture long-term dependencies in sequential data. The bidirectional aspect means the model processes the sequence in both forward and backward directions, which can improve context understanding. The attention mechanism allows the model to focus on specific parts of the input sequence, enhancing its ability to capture relevant information.

#### C. Temporal Pooling:

This step aggregates information over time, which can help in summarizing the sequence data and reducing the model's complexity.

#### D. Output Layer:

This layer produces the final predictions based on the processed features from the previous layers.

Mean Squared Error (MSE) with 2 Epochs: MSE is a common loss function for regression tasks, measuring the average squared difference between predicted and actual values. Training for 2 epochs means the model has gone through the entire training dataset twice and the model could perform if iterated through more Epochs.

This model has comparatively low MSE and aligns with the high and low power consumption points with a lag which indicates that the model is not effective in predicting the power

consumption before 5 minutes for a household.

## WHICH MODEL WOULD BE BEST TO BE CHOSEN IN AN IOT SYSTEM THAT PREDICTS GAP FOR A SINGLE HOUSEHOLD WITH 5 MINUTE PREDICTIVE HORIZON:

With the tests inferring that Linear Regression model with Moving Average outperforms more complex models like LSTM and CNN-Bidirectional LSTM in terms of MSE (0.038), but lags in capturing peaks and low points of power consumption, the model to choose is the model with improved ability to capture non-linear patterns and temporal dependencies in the data.

### Selection of Hybrid Models:

1. Combining the Linear Regression model with a non-linear model (e.g., Random Forest, Gradient Boosting, or Neural Networks) to capture both linear and non-linear patterns could be tried.
2. Ensemble approach: Using an ensemble approach (e.g., stacking or weighted averaging) to combine predictions from multiple models.

### 3. Advanced Time Series Models:

Below Time series models could be tried.

- a. Prophet: A time series forecasting model developed by Facebook that handles seasonality, trends, and holidays well.
- b. ARIMA/SARIMA: Traditional time series models that can capture seasonality and trends.
- c. XGBoost/LightGBM: Gradient boosting models that can handle non-linear relationships and feature interactions. It can handle non-linear relationships and feature interactions, making it well-suited for capturing complex patterns.

It can incorporate temporal and seasonal features (e.g., time of day, day of week, month, festivals) effectively.

## 4. Model Architecture Improvements

### a. LSTM/GRU with Attention:

Addition of an attention mechanism to LSTM/GRU models to focus on important time steps (e.g., peaks and low points).

## **b. Transformer for Time Series:**

Using a Transformer-based model with self-attention to capture long-term dependencies and seasonal patterns.

## **c.TCN (Temporal Convolutional Network):**

Using a TCN with dilated convolutions to capture long-term dependencies efficiently.

## **5. Hyperparameter Tuning**

Usage of tools like Grid Search, Random Search, or Bayesian Optimization to find the best hyperparameters for your models.

## **6. Focus on tuning:**

- a. Learning rate, number of layers, number of units (for neural networks).
- b. Window size, number of lags, and rolling statistics (for feature engineering).

## **7. Error Analysis**

Analysis of the residuals (errors) of the Linear Regression model to identify patterns (e.g., systematic underprediction during peaks).