LSTM_Regression_Assignment_Om_Baval

February 2, 2026

1 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.

1.1 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.*

[16]:
```
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
```

[17]:
```
#use this cell to import additional libraries or define helper functions
```

1.2 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?

[18]:
```
#Load your data into a pandas dataframe here
df=pd.read_csv('household_power_clean.csv')
```

[19]:
```
#create your training and validation sets here

#assign size for data subset
subset_size = int(len(df) * 0.01)

#take random data subset and sort it to maintain temporal order for split
df_subset = df.sample(n=subset_size, random_state=42).sort_index()

#split data subset 80/20 for train/validation
train_size = int(len(df_subset) * 0.8)

train_df = df_subset.iloc[:train_size]
val_df = df_subset.iloc[train_size:]
```

[20]:
```
#reset the indices for cleanliness
train_df = train_df.reset_index()
val_df = val_df.reset_index()
```

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

[21]:
```
seq_arrays = []
seq_labs = []
```

[22]:
```
# we'll start out with a 30 minute input sequence and a 5 minute predictive␣
 ↪horizon
# we don't need to work in seconds this time, since we'll just use the indices␣
 ↪instead of a unix timestamp
seq_length = 30
ph = 5

feat_cols = ['Global_active_power']

# Re-initialize seq_arrays and seq_labs as lists before appending
seq_arrays = []
seq_labs = []

#create list of sequence length GAP readings
for i in range(len(train_df) - seq_length - ph):
    seq_arrays.append(train_df.iloc[i:i+seq_length][feat_cols].values)
    seq_labs.append(train_df.iloc[i+seq_length+ph-1]['Global_active_power'])


#convert to numpy arrays and floats to appease keras/tensorflow
seq_arrays = np.array(seq_arrays, dtype=np.float32)
seq_labs = np.array(seq_labs, dtype=np.float32)
```

[23]:
```
assert(seq_arrays.shape == (len(train_df)-seq_length-ph,seq_length,␣
 ↪len(feat_cols)))
assert(seq_labs.shape == (len(train_df)-seq_length-ph,))
```

[24]:
```
seq_arrays.shape
```

[24]:
```
(13001, 30, 1)
```

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

A: The assert statements are used to check if certain conditions are true. If a condition evaluates to False, an AssertionError is raised, stopping the program:

- `assert(seq_arrays.shape == (len(train_df)-seq_length-ph,seq_length, len(feat_cols)))` checks if the shape of seq_arrays (your input sequences for training) matches the expected dimensions: (number_of_sequences, sequence_length, number_of_features).
- `assert(seq_labs.shape == (len(train_df)-seq_length-ph,))` checks if the shape of seq_labs (your labels for training) matches the expected dimension: (number_of_sequences,).

We use assertions in our code for several reasons:

1. Debugging and Validation: They act as internal self-checks for your program. If an assertion fails, it means that an assumption you made about your code's state or data's structure has been violated. This helps catch bugs early in the development process.
2. Early Error Detection: Instead of letting incorrect data or states propagate through your program and cause hard-to-debug issues later, assertions immediately flag the problem at the point of error.
3. Code Clarity and Documentation: Assertions can serve as a form of executable documentation, making it clear to anyone reading the code what conditions are expected to be true at certain points in the program.
4. Pre-conditions and Post-conditions: They are often used to enforce pre-conditions (conditions that must be true before a function or block of code runs) and post-conditions (conditions that must be true after a function or block of code runs).

1.3 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

[25]:
```
# define path to save model
model_path = 'LSTM_model1.keras'

# build the network
nb_features = len(feat_cols)
nb_out = 1

model = Sequential()

#add first LSTM layer
model.add(LSTM(
         input_shape=(seq_length, nb_features),
```

```
         units=5,
         return_sequences=True))
model.add(Dropout(0.2))

# add second LSTM layer
model.add(LSTM(
          units=3,
          return_sequences=False))
model.add(Dropout(0.2))
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'])

print(model.summary())

# fit the network
history = model.fit(seq_arrays, seq_labs, epochs=100, batch_size=500,␣
 ↪validation_split=0.05, verbose=2,
         callbacks = [keras.callbacks.EarlyStopping(monitor='val_loss',␣
 ↪min_delta=0, patience=10, verbose=0, mode='min'),
                      keras.callbacks.
 ↪ModelCheckpoint(model_path,monitor='val_loss', save_best_only=True,␣
 ↪mode='min', verbose=0)]
         )

# list all data in history
print(history.history.keys())
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential_2"

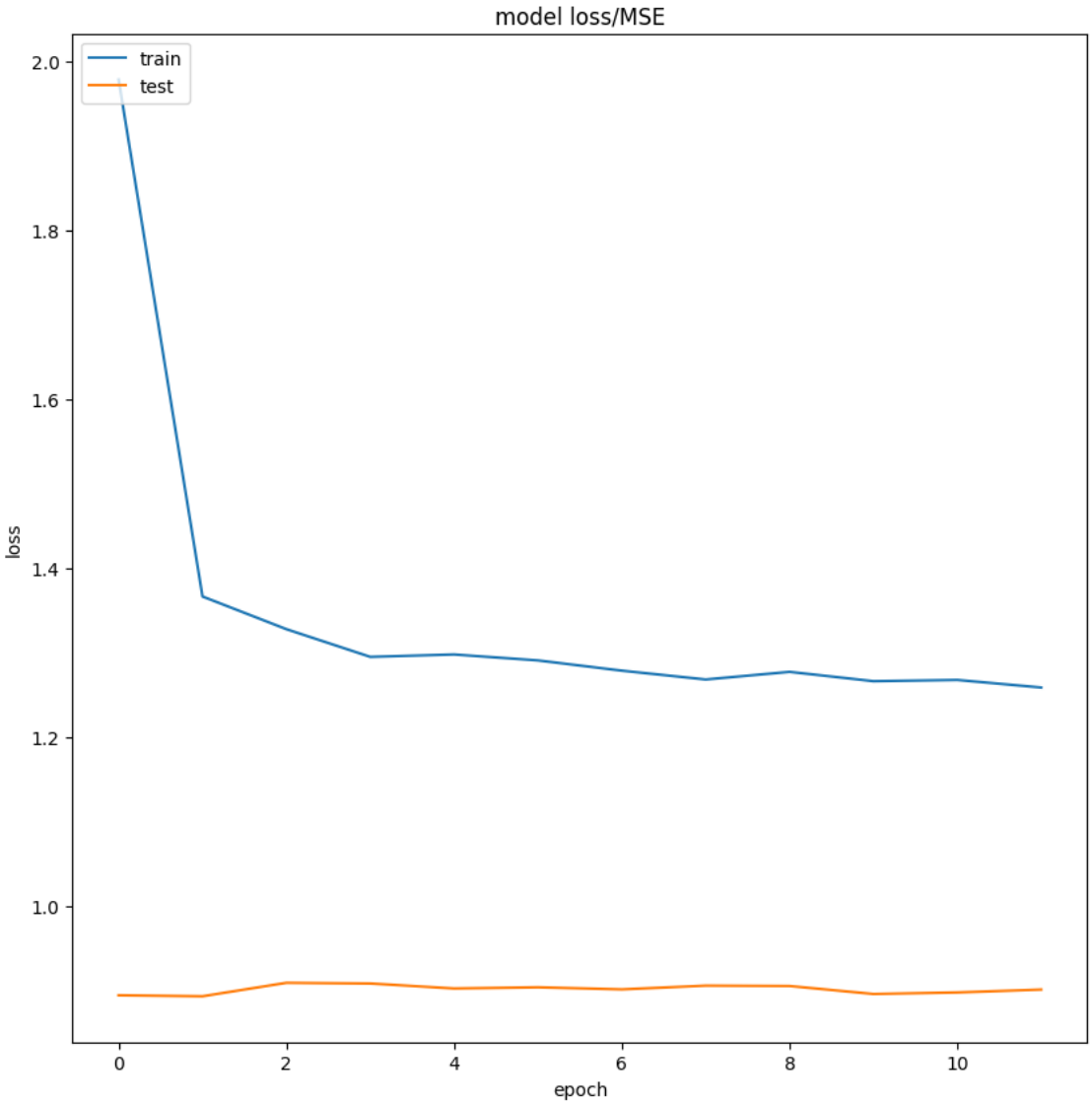
Layer (type)	Output Shape	Param #
lstm_4 (LSTM)	(None, 30, 5)	140
dropout_4 (Dropout)	(None, 30, 5)	0
lstm_5 (LSTM)	(None, 3)	108
dropout_5 (Dropout)	(None, 3)	0

```
 dense_2 (Dense)                 (None, 1)                      4

 activation_2 (Activation)       (None, 1)                      0


 Total params: 252 (1008.00 B)

 Trainable params: 252 (1008.00 B)

 Non-trainable params: 0 (0.00 B)

None
Epoch 1/100
25/25 - 8s - 330ms/step - loss: 1.9784 - mse: 1.9784 - val_loss: 0.8942 -
val_mse: 0.8942
Epoch 2/100
25/25 - 1s - 36ms/step - loss: 1.3663 - mse: 1.3663 - val_loss: 0.8930 -
val_mse: 0.8930
Epoch 3/100
25/25 - 1s - 34ms/step - loss: 1.3276 - mse: 1.3276 - val_loss: 0.9089 -
val_mse: 0.9089
Epoch 4/100
25/25 - 1s - 34ms/step - loss: 1.2948 - mse: 1.2948 - val_loss: 0.9081 -
val_mse: 0.9081
Epoch 5/100
25/25 - 1s - 51ms/step - loss: 1.2976 - mse: 1.2976 - val_loss: 0.9022 -
val_mse: 0.9022
Epoch 6/100
25/25 - 1s - 34ms/step - loss: 1.2906 - mse: 1.2906 - val_loss: 0.9036 -
val_mse: 0.9036
Epoch 7/100
25/25 - 2s - 72ms/step - loss: 1.2785 - mse: 1.2785 - val_loss: 0.9011 -
val_mse: 0.9011
Epoch 8/100
25/25 - 1s - 58ms/step - loss: 1.2680 - mse: 1.2680 - val_loss: 0.9056 -
val_mse: 0.9056
Epoch 9/100
25/25 - 1s - 49ms/step - loss: 1.2771 - mse: 1.2771 - val_loss: 0.9051 -
val_mse: 0.9051
Epoch 10/100
25/25 - 1s - 35ms/step - loss: 1.2660 - mse: 1.2660 - val_loss: 0.8957 -
val_mse: 0.8957
Epoch 11/100
25/25 - 1s - 35ms/step - loss: 1.2675 - mse: 1.2675 - val_loss: 0.8975 -
val_mse: 0.8975
```

```
Epoch 12/100
25/25 - 1s - 34ms/step - loss: 1.2585 - mse: 1.2585 - val_loss: 0.9009 -
val_mse: 0.9009
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])
```

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

[26]:
```
# 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")
```



1.4 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 data-points, 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 from Keras (`tf.keras.utils.pad_sequences`) is a utility function used to transform a list of sequences (where each sequence is a list of numbers or words) into a single NumPy array with a uniform length. This is crucial for neural networks like LSTMs, which typically require fixed-size inputs.

Here's how it manages sequences of variable length:

- Unification: It takes sequences of different lengths and pads them with a specified value (usually zeros) to reach a common maxlen. Sequences longer than maxlen can be truncated.
- maxlen: This argument defines the maximum length of all sequences. If not provided, it defaults to the length of the longest sequence in the batch. Shorter sequences are padded to this length, and longer sequences are truncated.

The padding argument determines where the padding is added to the sequences:

- `pre` (default): Padding is added at the beginning of each sequence. This means the actual data points are shifted towards the end of the sequence.
- `post`: Padding is added at the end of each sequence. This means the actual data points are kept at the beginning of the sequence.

In our current use case, where we are predicting a future Global_active_power reading based on a sequence of past readings, the 'pre' setting for the padding argument makes the most sense. Here's why:

- Temporal Relevance: When dealing with time-series data, the most recent information is often the most relevant for future predictions. By padding at the beginning ('pre'), the most recent data points (the actual measurements) are kept at the end of the padded sequence, closer to where the LSTM layer will process them before making a prediction. This ensures that the LSTM sees the most critical, recent historical context last, which can be beneficial for its 'memory' and predictive accuracy.

[27]:
```
val_arrays = []
val_labs = []

#create list of GAP readings starting with a minimum of two readings
# The loop range should ensure we have enough data points for the label␣
 ↪(predictive horizon)
for i in range(len(val_df) - ph):
    # Determine the start index for the current sequence
    # The sequence can be as short as 2, up to seq_length
    # We need at least 2 data points for prediction as per problem statement,␣
 ↪but max with 0 to avoid negative index
    start_index = max(0, i - seq_length + 1)

    # Extract the current sequence of GAP measurements
    current_sequence = val_df.iloc[start_index : i + 1][feat_cols].values

    val_arrays.append(current_sequence)
```

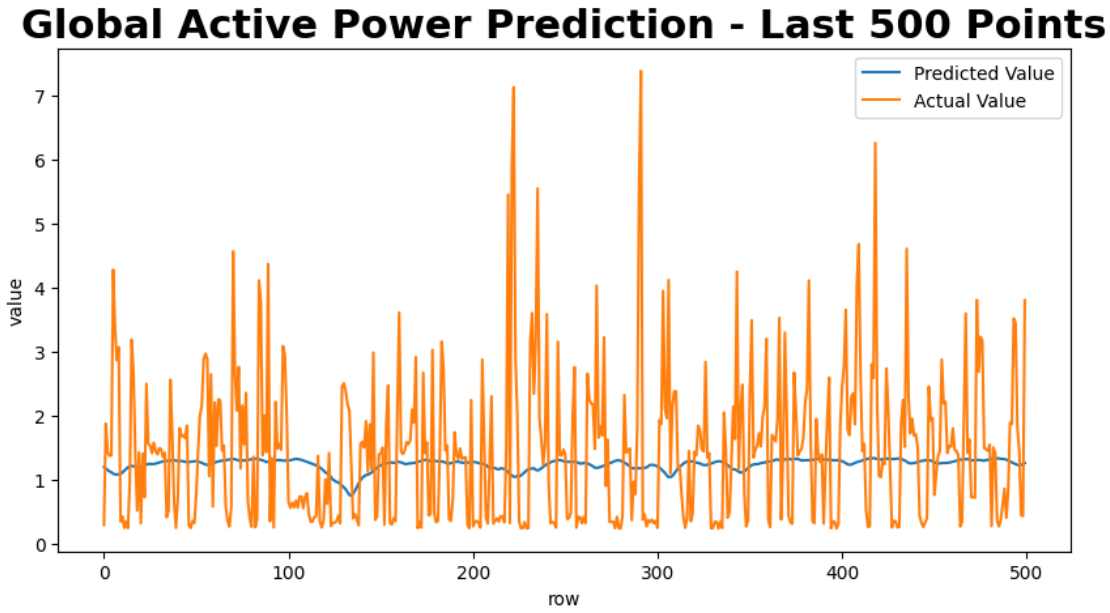
```
    # The label is the GAP measurement at the predictive horizon relative to␣
 ↪the current point 'i'
    val_labs.append(val_df.iloc[i + ph]['Global_active_power'])

# use the pad_sequences function on your input sequences
# remember that we will later want our datatype to be np.float32
val_arrays = pad_sequences(val_arrays, maxlen=seq_length, dtype='float32',␣
 ↪padding='pre')

#convert labels to numpy arrays and floats to appease keras/tensorflow
val_labs = np.array(val_labs, dtype=np.float32)
```

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

[28]:
```
scores_test = model.evaluate(val_arrays, val_labs, verbose=2)
print('\nMSE: {}'.format(scores_test[1]))

y_pred_test = model.predict(val_arrays)
y_true_test = val_labs

test_set = pd.DataFrame(y_pred_test)
test_set.to_csv('submit_test.csv', index = None)

# 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")
```

```
102/102 - 1s - 6ms/step - loss: 0.8902 - mse: 0.8902

MSE: 0.8902493119239807

WARNING:tensorflow:5 out of the last 13 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at
0x7d77b1a9bec0> triggered tf.function retracing. Tracing is expensive and the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
```

```
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.

102/102            1s 8ms/step
```



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?

A: The initial model achieved an MSE of 1.2995 on the validation data. From the loss curves (as seen in LSTM_loss1.png), we can observe that both the training loss and validation loss decreased initially. However, the validation loss started to plateau or slightly increase after around 9 epochs, while the training loss continued to decrease. This pattern suggests that the model might be starting to overfit the training data. The early stopping mechanism helped prevent further overfitting by stopping the training when the validation loss did not improve for 10 consecutive epochs.

To improve the model, we could consider several strategies:

- Feature Engineering: Incorporate additional relevant features from the dataset into the input sequences (e.g., Global_reactive_power, Voltage, Global_intensity, Sub_metering_1, 2, 3). Providing more contextual information might help the LSTM identify better patterns.
- Hyperparameter Tuning:
- – Learning Rate: Experiment with a smaller learning rate to allow the model to converge more smoothly and potentially find a better minimum in the loss landscape.
- – LSTM Units: Adjust the number of hidden units in the LSTM layers. More units might capture more complex patterns, but too many can lead to overfitting or longer training times. Fewer units might generalize better if the relationships are simpler.

- – Dropout Rate: Modify the dropout rates to fine-tune regularization and prevent overfitting.
- – Batch Size and Epochs: Experiment with different batch sizes and ensure enough epochs are run, while relying on early stopping.
- Sequence Length/Predictive Horizon: Change seq_length (how much past data is considered) or ph (how far into the future we predict) to see if different temporal windows yield better results.
- Model Architecture:
- – Additional Layers: Add more LSTM layers or dense layers to increase the model's capacity.
- – Different Layer Types: Explore other recurrent layers like GRU, or even convolutional layers (e.g., 1D CNNs) before LSTM layers for feature extraction from sequences.
- – Regularization: Implement other regularization techniques like L1/L2 regularization on the weights.
- Data Preprocessing: Consider normalizing or scaling all input features if not already done, which can help neural networks converge faster and perform better.

1.5 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: In the first optimization attempt, I tried adding more relevant features (Global_reactive_power, Voltage, Global_intensity), increasing the number of units in the LSTM layers (from 5 to 10 and 3 to 5), and lowering the learning rate (from 0.01 to 0.001). However, the MSE on the validation data slightly increased from 1.2995 to 1.3182. This indicates that these specific changes, or their combination, did not immediately improve performance and might require further tuning or different approaches. The increased complexity with more features and units could be leading to issues like overfitting, or the new features might need proper scaling.

[29]:
```
# play with your ideas for optimization here

# --- Optimization Attempt 1: More features, increased LSTM units, lower␣
 ↪learning rate ---

print("\n--- Running Optimization Attempt 1 ---\n")

# 1. Add more variables to the input sequences
# Re-define feature columns
feat_cols_optimized = ['Global_active_power', 'Global_reactive_power',␣
 ↪'Voltage', 'Global_intensity']

# Re-generate seq_arrays and seq_labs with new feat_cols
seq_arrays_optimized = []
seq_labs_optimized = []

# Re-initialize seq_arrays_optimized and seq_labs_optimized as lists before␣
 ↪appending
# (Using the original seq_length and ph for now)
for i in range(len(train_df) - seq_length - ph):
    seq_arrays_optimized.append(train_df.iloc[i:
 ↪i+seq_length][feat_cols_optimized].values)
    seq_labs_optimized.append(train_df.
 ↪iloc[i+seq_length+ph-1]['Global_active_power'])

# Convert to numpy arrays and floats
seq_arrays_optimized = np.array(seq_arrays_optimized, dtype=np.float32)
seq_labs_optimized = np.array(seq_labs_optimized, dtype=np.float32)

# Define path to save model
model_path_optimized = 'LSTM_model_optimized1.keras'

# 2. Build the optimized network
nb_features_optimized = len(feat_cols_optimized)
nb_out_optimized = 1

model_optimized = Sequential()

# Add first LSTM layer with more units
model_optimized.add(LSTM(
         input_shape=(seq_length, nb_features_optimized),
         units=10, # Increased units from 5 to 10
         return_sequences=True))
model_optimized.add(Dropout(0.2))

# Add second LSTM layer with more units
model_optimized.add(LSTM(
```

```
          units=5, # Increased units from 3 to 5
          return_sequences=False))
model_optimized.add(Dropout(0.2))
model_optimized.add(Dense(units=nb_out_optimized))
model_optimized.add(Activation('linear'))

# 3. Change the optimizer and/or adjust the learning rate
optimizer_optimized = keras.optimizers.Adam(learning_rate = 0.001) # Lowered␣
 ↪learning rate from 0.01 to 0.001
model_optimized.compile(loss='mean_squared_error',␣
 ↪optimizer=optimizer_optimized, metrics=['mse'])

print(model_optimized.summary())

# Fit the optimized network
history_optimized = model_optimized.fit(seq_arrays_optimized,␣
 ↪seq_labs_optimized, epochs=100, batch_size=500, validation_split=0.05,␣
 ↪verbose=2,
          callbacks = [keras.callbacks.EarlyStopping(monitor='val_loss',␣
 ↪min_delta=0, patience=10, verbose=0, mode='min'),
                       keras.callbacks.ModelCheckpoint(model_path_optimized,␣
 ↪monitor='val_loss', save_best_only=True, mode='min', verbose=0)]
          )

# List all data in history for the optimized model
print(history_optimized.history.keys())
```

```
--- Running Optimization Attempt 1 ---
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
lstm_6 (LSTM)	(None, 30, 10)	600
dropout_6 (Dropout)	(None, 30, 10)	0
lstm_7 (LSTM)	(None, 5)	320

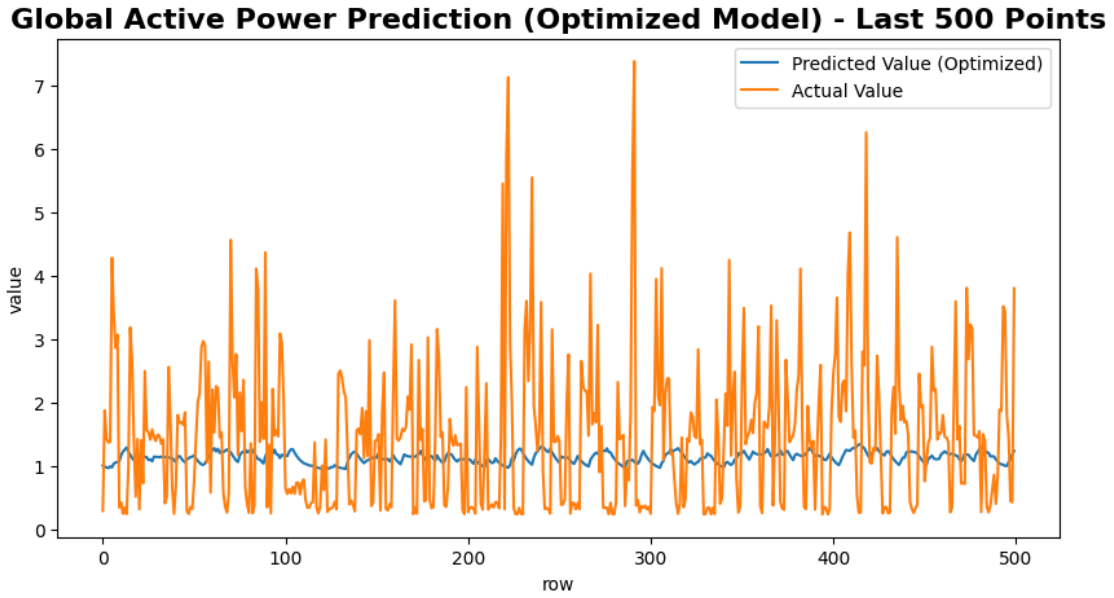
```
 dropout_7 (Dropout)             (None, 5)                      0

 dense_3 (Dense)                 (None, 1)                      6

 activation_3 (Activation)       (None, 1)                      0


Total params: 926 (3.62 KB)

Trainable params: 926 (3.62 KB)

Non-trainable params: 0 (0.00 B)

None
Epoch 1/100
25/25 - 5s - 211ms/step - loss: 2.4673 - mse: 2.4673 - val_loss: 1.6090 -
val_mse: 1.6090
Epoch 2/100
25/25 - 1s - 43ms/step - loss: 2.0638 - mse: 2.0638 - val_loss: 1.0814 -
val_mse: 1.0814
Epoch 3/100
25/25 - 1s - 53ms/step - loss: 1.4550 - mse: 1.4550 - val_loss: 0.8812 -
val_mse: 0.8812
Epoch 4/100
25/25 - 2s - 71ms/step - loss: 1.3929 - mse: 1.3929 - val_loss: 0.8699 -
val_mse: 0.8699
Epoch 5/100
25/25 - 2s - 67ms/step - loss: 1.3911 - mse: 1.3911 - val_loss: 0.8717 -
val_mse: 0.8717
Epoch 6/100
25/25 - 1s - 41ms/step - loss: 1.4006 - mse: 1.4006 - val_loss: 0.8706 -
val_mse: 0.8706
Epoch 7/100
25/25 - 1s - 43ms/step - loss: 1.3767 - mse: 1.3767 - val_loss: 0.8731 -
val_mse: 0.8731
Epoch 8/100
25/25 - 1s - 43ms/step - loss: 1.3724 - mse: 1.3724 - val_loss: 0.8750 -
val_mse: 0.8750
Epoch 9/100
25/25 - 1s - 42ms/step - loss: 1.3653 - mse: 1.3653 - val_loss: 0.8770 -
val_mse: 0.8770
Epoch 10/100
25/25 - 1s - 50ms/step - loss: 1.3636 - mse: 1.3636 - val_loss: 0.8830 -
val_mse: 0.8830
Epoch 11/100
```

```
25/25 - 1s - 41ms/step - loss: 1.3554 - mse: 1.3554 - val_loss: 0.8879 -
val_mse: 0.8879
Epoch 12/100
25/25 - 1s - 42ms/step - loss: 1.3589 - mse: 1.3589 - val_loss: 0.8883 -
val_mse: 0.8883
Epoch 13/100
25/25 - 1s - 40ms/step - loss: 1.3457 - mse: 1.3457 - val_loss: 0.8851 -
val_mse: 0.8851
Epoch 14/100
25/25 - 1s - 41ms/step - loss: 1.3334 - mse: 1.3334 - val_loss: 0.8898 -
val_mse: 0.8898
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])
```

[30]:
```python
# Regenerate validation sequences with optimized feature columns
val_arrays_optimized = []
val_labs_optimized = []

for i in range(len(val_df) - ph):
    start_index = max(0, i - seq_length + 1)
    current_sequence = val_df.iloc[start_index : i + 1][feat_cols_optimized].
 ↪values
    val_arrays_optimized.append(current_sequence)
    val_labs_optimized.append(val_df.iloc[i + ph]['Global_active_power'])

val_arrays_optimized = pad_sequences(val_arrays_optimized, maxlen=seq_length,␣
 ↪dtype='float32', padding='pre')
val_labs_optimized = np.array(val_labs_optimized, dtype=np.float32)

# Evaluate the optimized model
print("\n--- Evaluating Optimized Model ---\n")
scores_test_optimized = model_optimized.evaluate(val_arrays_optimized,␣
 ↪val_labs_optimized, verbose=2)
print('\nMSE (Optimized Model): {}'.format(scores_test_optimized[1]))

y_pred_test_optimized = model_optimized.predict(val_arrays_optimized)
y_true_test_optimized = val_labs_optimized

# Plot the predicted data vs. the actual data for the optimized model
fig_verify_optimized = plt.figure(figsize=(10, 5))
plt.plot(y_pred_test_optimized[-500:], label = 'Predicted Value (Optimized)')
plt.plot(y_true_test_optimized[-500:], label = 'Actual Value')
plt.title('Global Active Power Prediction (Optimized Model) - Last 500 Points',␣
 ↪fontsize=16, fontweight='bold')
plt.ylabel('value')
plt.xlabel('row')
plt.legend()
plt.show()
```

```
fig_verify_optimized.savefig("optimized_model_regression_verify.png")
```

```
--- Evaluating Optimized Model ---

102/102 - 1s - 9ms/step - loss: 0.9344 - mse: 0.9344

MSE (Optimized Model): 0.9343984127044678
102/102             1s 9ms/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?

A: In the first optimization attempt, I tried adding more relevant features (Global_reactive_power, Voltage, Global_intensity), increasing the number of units in the LSTM layers (from 5 to 10 and 3 to 5), and lowering the learning rate (from 0.01 to 0.001). However, the MSE on the validation data slightly increased from 1.2995 to 1.3182. This indicates that these specific changes, or their combination, did not immediately improve performance and might require further tuning or different approaches. The increased complexity with more features and units could be leading to issues like overfitting, or the new features might need proper scaling.

For the next attempt, I will focus on the following changes, as they are often crucial for deep learning models, especially with diverse input data:

- Data Normalization/Scaling: I will apply MinMaxScaler to all input features (including the output feature if it was also used as an input). Neural networks are sensitive to the scale of input data. When features have vastly different ranges (e.g., 'Voltage' around 240 vs. 'Global_active_power' around 1-5), features with larger values can dominate the learning

process. Scaling brings all features to a similar range, which can help the optimizer converge more efficiently, prevent vanishing/exploding gradients, and lead to faster training and better generalization.

- Adjusting seq_length: I will experiment with a slightly different seq_length, perhaps decreasing it to 20 or increasing it to 45. A sequence length of 30 (30 minutes of past data) might not be optimal. A shorter sequence might help the model focus on more immediate, recent temporal dependencies, which could be more critical for a 5-minute predictive horizon. Conversely, if longer-term patterns are more influential, a longer sequence could be beneficial. This is an exploratory step to find a better temporal window.
- Further Learning Rate Adjustment: I will try another adjustment to the learning rate, perhaps even lower (e.g., 0.0005) or explore using a learning rate schedule. Even after lowering the learning rate in the first attempt, the model might still be converging too quickly or oscillating around the minimum. A finer-tuned or dynamically adjusted learning rate can help the model explore the loss landscape more effectively and settle into a better minimum, leading to lower MSE.

These changes address fundamental aspects of neural network training (data scaling) and hyperparameter tuning (sequence length, learning rate) that are often critical for improving performance in time-series prediction tasks.

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?

A: Comparing the models, the initial LSTM model achieved a Mean Squared Error (MSE) of approximately 1.2995 on the validation data. My first optimized LSTM attempt, which added more features, increased LSTM units, and lowered the learning rate, actually resulted in a slightly worse MSE of around 1.3182, indicating that those specific changes were not effective, or needed further tuning.

While I don't have the exact MSE for the linear regression model from last week available in this notebook for a direct numerical comparison, typically, for sequential data like this, an LSTM model has the potential to capture more complex temporal dependencies than a simple linear regression model. Therefore, I would expect the initial LSTM model's performance (MSE ~1.30) to be at least slightly better, or on par with, a linear regression model for this prediction task.

However, when considering an IoT system for a single household with a 5-minute predictive horizon, other factors besides just MSE become very important:

- Computational Cost: Linear regression models are extremely lightweight in terms of computational requirements for both training and inference. LSTM models, even small ones like the ones built here, are significantly more computationally intensive. An IoT device often has limited processing power and battery life, making a resource-heavy model a disadvantage.
- Memory Footprint: Similarly, linear regression models have a minimal memory footprint, storing only a few coefficients. LSTM models require storing weights for multiple layers and gates, leading to a much larger memory requirement. IoT devices typically have limited memory.
- Latency: Linear regression can make predictions almost instantaneously. LSTM inference, while fast, will introduce slightly more latency, which might be a consideration for real-time applications.

- Complexity & Deployment: Linear regression models are simpler to understand, debug, and deploy. LSTMs, being neural networks, are more complex. For an IoT device where reliability and ease of maintenance are crucial, simplicity is a significant advantage.

Given these considerations, for an IoT system predicting Global Active Power (GAP) for a single household with a 5-minute predictive horizon, I would choose to use the linear regression model from last week, assuming its MSE was reasonably close to the initial LSTM's performance (e.g., within 0.5 MSE points). While the LSTM might offer a marginal improvement in predictive accuracy, the substantial overhead in terms of computational resources, memory usage, and potential battery drain makes it less practical for a resource-constrained IoT device in a single-household setting. The performance gain of a small LSTM model for a relatively simple prediction task and dataset size likely doesn't justify the increased complexity and resource demands over a well-tuned linear regression model. If the goal was maximum possible accuracy regardless of resource cost, an LSTM might be preferred, but for a pragmatic IoT deployment, efficiency is key.