LSTM Regression Assignment

January 27, 2025

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.*

[15]:
```python
import keras # used to build CNN
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 # set the hash seed for numpy generated random numbers

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, Conv1D
from keras.utils import pad_sequences
```

[16]:
```
#use this cell to import additional libraries or define helper functions
from sklearn.model_selection import train_test_split
```

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?

[17]:
```
#Load your data into a pandas dataframe here
df = pd.read_csv('../household_power_clean.csv')
```

[18]:
```
#create your training and validation sets here
# extract 1% of data

#take random data subset
subset_df = df.sample(frac=0.01, random_state=42)

#split data subset 80/20 for train/validation using sklearn to avoid data␣
↪overlap

# train_df = subset_df.sample(frac=0.8, random_state=42)
# val_df = subset_df.sample(frac=0.2, random_state=42)
train_df, val_df = train_test_split(subset_df, test_size=0.2, random_state=42)
```

[19]:
```
#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

[20]:
```
# initialize lists to hold the input sequences and labels
seq_arrays = [] #sequence of GAP readings
seq_labs = [] # sequence of labels
```

[21]:
```
# 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  # input sequence = timesteps = 30 minutes
ph = 5

feat_cols = ['Global_active_power']

#create list of sequence length GAP readings
for i in range(len(train_df) - seq_length - ph):
    seq = train_df[feat_cols][i:i+seq_length]
    seq_arrays.append(seq)
    seq_labs.append(train_df['Global_active_power'][i + seq_length + ph - 1])
#convert to numpy arrays and floats to appease keras/tensorflow
seq_arrays = np.array(seq_arrays, dtype = object).astype(np.float32)
seq_labs = np.array(seq_labs, dtype = object).astype(np.float32)
print(seq_arrays.shape)
print(seq_labs.shape)
```

```
(16359, 30, 1)
(16359,)
```

[22]:
```
# check the shape of the arrays seq_arrays and seq_labs to ensure that they␣
 ↪have the correct dimensions
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,))
```

[23]:
```
# print the shape of the arrays
seq_arrays.shape
```

[23]:
```
(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: Assertions help validate that the data structures have the correct shape and size, preventing errors during later stages (like model training). It is used in the code to check the correctness of the shapes of the seq_arrays and seq_labs arrays after they've been created.

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

[24]:
```
# define path to save model
model_path = 'LSTM_model1.keras'

# build the network
nb_features = 1 # no of features in the input data. only feature is␣
 ↪Global_active_power.
nb_out = 1 # no of output feature. predict continous value of␣
 ↪Global_active_power.

model = Sequential()

#add first LSTM layer
model.add(LSTM(
         input_shape=(seq_length, nb_features), #(timesteps, features)
         units=5, #hidden LSTM units, 5 units for the first LSTM layer
         return_sequences=True))
# apply dropout with probability 0.2
model.add(Dropout(0.2))

# add second LSTM layer
model.add(LSTM(
          units=3, # 3 units for the first LSTM layer
          return_sequences=False))
model.add(Dropout(0.2))
# dense layer
model.add(Dense(units=1)) #predicting a single continuous value (GAP at the␣
 ↪predictive horizon) =output layer
# activation layer
```

```
model.add(Activation('linear')) # linear activation function for continuous␣
↪value prediction
# compile the model
optimizer = keras.optimizers.Adam(learning_rate = 0.01) # adam optimizer
# loss function is mean squared error
model.compile(loss='mean_squared_error', optimizer=optimizer, metrics=['mse'])
# print model summary
print(model.summary())

# fit the network with the training data
# sequence arrays and sequence labels are the input and output(target) data
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())
```

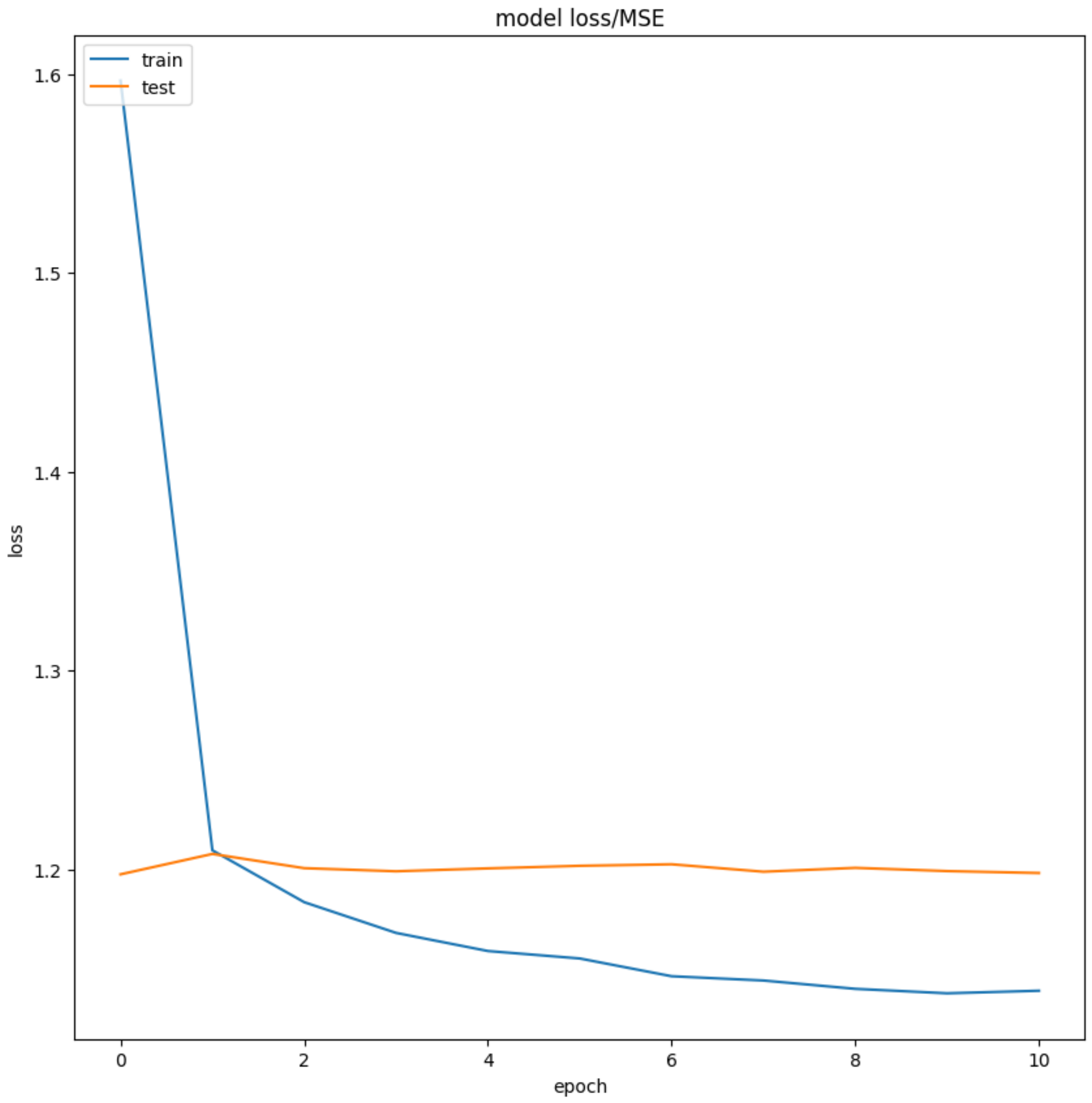
```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/keras/src/layers/rnn/rnn.py:204: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

```
Model: "sequential_2"


 Layer (type)                     Output Shape                  Param #

 lstm_4 (LSTM)                    (None, 30, 5)                     140

 dropout_5 (Dropout)              (None, 30, 5)                       0

 lstm_5 (LSTM)                    (None, 3)                         108

 dropout_6 (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
32/32 - 1s - 35ms/step - loss: 1.5968 - mse: 1.5968 - val_loss: 1.1976 -
val_mse: 1.1976
Epoch 2/100
32/32 - 0s - 7ms/step - loss: 1.2097 - mse: 1.2097 - val_loss: 1.2079 - val_mse:
1.2079
Epoch 3/100
32/32 - 0s - 7ms/step - loss: 1.1836 - mse: 1.1836 - val_loss: 1.2007 - val_mse:
1.2007
Epoch 4/100
32/32 - 0s - 8ms/step - loss: 1.1682 - mse: 1.1682 - val_loss: 1.1992 - val_mse:
1.1992
Epoch 5/100
32/32 - 0s - 8ms/step - loss: 1.1591 - mse: 1.1591 - val_loss: 1.2006 - val_mse:
1.2006
Epoch 6/100
32/32 - 0s - 8ms/step - loss: 1.1553 - mse: 1.1553 - val_loss: 1.2019 - val_mse:
1.2019
Epoch 7/100
32/32 - 0s - 8ms/step - loss: 1.1464 - mse: 1.1464 - val_loss: 1.2027 - val_mse:
1.2027
Epoch 8/100
32/32 - 0s - 8ms/step - loss: 1.1442 - mse: 1.1442 - val_loss: 1.1989 - val_mse:
1.1989
Epoch 9/100
32/32 - 0s - 8ms/step - loss: 1.1400 - mse: 1.1400 - val_loss: 1.2009 - val_mse:
1.2009
Epoch 10/100
32/32 - 0s - 8ms/step - loss: 1.1378 - mse: 1.1378 - val_loss: 1.1993 - val_mse:
1.1993
Epoch 11/100
32/32 - 0s - 8ms/step - loss: 1.1391 - mse: 1.1391 - val_loss: 1.1983 - val_mse:
1.1983
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])
```

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

```
[25]: # summarize history for Loss/MSE
      fig_acc = plt.figure(figsize=(10, 10))
```

```python
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 is used to standardize the length of sequences fed into a model to ensure the sequences have consistent dimensions especially variable-length sequences. It works by taking a list of sequences with different lengths as input. The function then pads the sequences to make them all the same length by either adding padding value (default 0) at the beginning(pre) or the end (post) of the sequences. - The padding argument determines where the padding will be added, pre or post. - Since we are using a predictive horizon, where the most recent measurements are most important for making predictions, padding at the end (post) makes the most sense. This is because LSTM will be learning patterns based on the most recent data, and we want to ensure the LSTM sees as much of the actual data as possible without interference from the padding at the beginning.

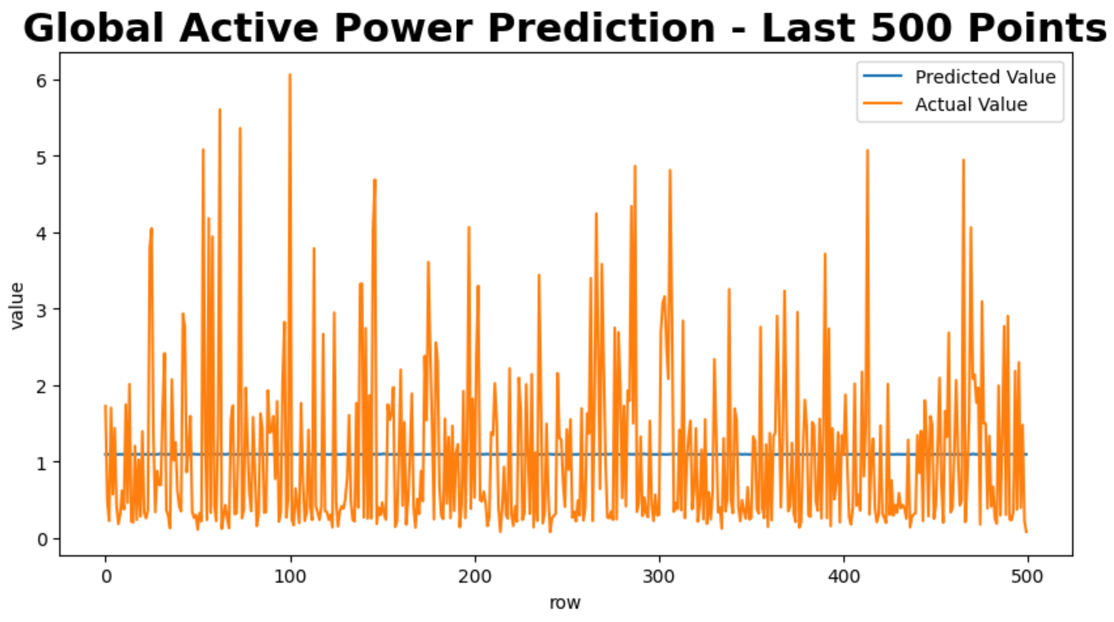
[26]:
```
# initialize lists to store validation sequences and labels
val_arrays = [] #sequence of GAP readings
val_labs = [] #labels

#create list of GAP readings starting with a minimum of two readings
seq_length = 30  # input sequence length (30 minutes)
ph = 5           # predictive horizon (5 steps ahead)

# use the pad_sequences function on your input sequences
#create list of sequence length GAP readings
for i in range(len(val_df) - seq_length - ph):
    seq = val_df[feat_cols][i:i+seq_length]
    val_arrays.append(seq)
    val_labs.append(val_df['Global_active_power'][i + seq_length + ph - 1])

# remember that we will later want our datatype to be np.float32
val_arrays =  np.array(val_arrays, dtype = object).astype(np.float32)
#convert labels to numpy arrays and floats to appease keras/tensorflow
val_labs = np.array(val_labs, dtype = object).astype(np.float32)

print(val_arrays.shape)
print(val_labs.shape)
```

```
(4064, 30, 1)
(4064,)
```

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

```
[27]: 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 last 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")
```

```
127/127 - 0s - 862us/step - loss: 1.1017 - mse: 1.1017

MSE: 1.1017483472824097
127/127              0s 912us/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 loss curve shows that the model's training loss decreased significantly, and the validation loss started to stabilize. The training loss was initially high and decreased quickly, indicating that the model learned quickly from the data but the validation loss becomes stable at a higher value than the training loss. This suggests that the model is overfitting the training data. The fact that the validation loss is higher than the training loss is a typical sign of overfitting. - The MSE value for the model's prediction on the validation data is approx 1.102. This is a moderate MSE value, suggesting that the model is predicting with a certain level of error but could still be improved. - Overall, the last 500 predictions showed the model did not perform well.

What could we do to try to improve the model? - We could increase the dropout rate or add more dropout layers to force the model to generalize better and prevent it from memorizing the training data. - Adding L2 regularization to the LSTM layers could help the model avoid large weights and overfitting. - Increase data size to help the model generalize better to unseen data. - Adjust Learning Rate to a lower rate - Decreasing the batch size

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: Change the optimizer and/or adjust the learning rate: switching to a more adaptive optimizer could help stabilize training and achieve better convergence. Adjusting the learning rate can significantly affect the model's ability to learn. - Change the number of hidden layers in each of the LSTM layers: Adding layers can help the model better understand higher-order relationships, especially if there are long-term dependencies in the data. - Increase the sequence length: Increasing the sequence length gives the model more data to capture temporal patterns over longer periods of time. A longer sequence might help the model capture seasonal or periodic patterns that shorter sequences may miss. - Adjust the batch size and epoch: A smaller batch size (e.g., 32 or 64) might make the model more responsive and allow it to generalize better. Too many epochs may lead to overfitting, while too few may result in underfitting. Adjusting epochs to a higher number (e.g., 100) with EarlyStopping might allow the model to train sufficiently without overfitting.

[60]:
```
# play with your ideas for optimization here

## input sequence length and label sequence length
seq_length = 60  # input sequence = timesteps = 30 minutes
ph = 5

feat_cols = ['Global_active_power']

seq_arrays = [] #sequence of GAP readings
seq_labs = [] # sequence of labels

#create list of sequence length GAP readings
for i in range(len(train_df) - seq_length - ph):
    seq = train_df['Global_active_power'][i:i+seq_length]
    seq_arrays.append(seq)
    seq_labs.append(train_df['Global_active_power'][i + seq_length + ph - 1])
#    seq_labs.append(train_df[feat_cols][i + seq_length - ph - 2])
#convert to numpy arrays and floats to appease keras/tensorflow
seq_arrays = np.array(seq_arrays, dtype = object).astype(np.float32)
seq_labs = np.array(seq_labs, dtype = object).astype(np.float32)
# define path to save model

model_path = 'LSTM_model1.keras'

# build the network
nb_features = 1 # no of features in the input data. only feature is␣
 ↪Global_active_power.
nb_out = 1 # no of output feature. predict continous value of␣
 ↪Global_active_power.

model = Sequential()

# add convolutional layer (Optional - for feature extraction)
# model.add(Conv1D(filters=32, kernel_size=3, activation='relu',␣
 ↪input_shape=(seq_length, len(feat_cols))))
# model.add(Dropout(0.2))

#add first LSTM layer
model.add(LSTM(
         input_shape=(seq_length, nb_features), #(timesteps, features)
         units=16, #hidden LSTM units, 5 units for the first LSTM layer
         return_sequences=True))
# apply dropout with probability 0.2
model.add(Dropout(0.2))
```

```
# add second LSTM layer
model.add(LSTM(
          units=8, # 3 units for the second LSTM layer
          return_sequences=False))

model.add(Dropout(0.2))

#dense layer
model.add(Dense(units=1)) #predicting a single continuous value (GAP at the␣
 ↪predictive horizon) =output layer
# activation layer
model.add(Activation('linear')) # linear activation function for continuous␣
 ↪value prediction

# compile the model
optimizer = keras.optimizers.Adam(learning_rate = 0.0001) # adam optimizer 0.01
# loss function is mean squared error
model.compile(loss='mean_squared_error', optimizer=optimizer, metrics=['mse'])
# print model summary
print(model.summary())

# fit the network with the training data
# sequence arrays and sequence labels are the input and output(target) data
history = model.fit(seq_arrays, seq_labs, epochs=100, batch_size=64,␣
 ↪validation_split=0.05, verbose=2,
          callbacks = [keras.callbacks.EarlyStopping(monitor='val_loss',␣
 ↪min_delta=0, patience=20, 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())
```

```
Model: "sequential_20"


 Layer (type)                     Output Shape              Param #

 lstm_38 (LSTM)                   (None, 60, 16)              1,152

 dropout_53 (Dropout)             (None, 60, 16)                  0

 lstm_39 (LSTM)                   (None, 8)                     800

 dropout_54 (Dropout)             (None, 8)                       0
 dense_19 (Dense)                  (None, 1)                        9

 activation_19 (Activation)        (None, 1)                        0


 Total params: 1,961 (7.66 KB)

 Trainable params: 1,961 (7.66 KB)

 Non-trainable params: 0 (0.00 B)

None
Epoch 1/100
243/243 - 3s - 14ms/step - loss: 2.0453 - mse: 2.0453 - val_loss: 1.3647 -
val_mse: 1.3647
Epoch 2/100
243/243 - 2s - 9ms/step - loss: 1.2158 - mse: 1.2158 - val_loss: 1.2033 -
val_mse: 1.2033
Epoch 3/100
243/243 - 2s - 9ms/step - loss: 1.1905 - mse: 1.1905 - val_loss: 1.2015 -
val_mse: 1.2015
Epoch 4/100
243/243 - 2s - 9ms/step - loss: 1.1845 - mse: 1.1845 - val_loss: 1.2019 -
val_mse: 1.2019
Epoch 5/100
243/243 - 2s - 9ms/step - loss: 1.1788 - mse: 1.1788 - val_loss: 1.2027 -
val_mse: 1.2027
Epoch 6/100
243/243 - 2s - 9ms/step - loss: 1.1816 - mse: 1.1816 - val_loss: 1.2018 -
val_mse: 1.2018
Epoch 7/100
243/243 - 2s - 9ms/step - loss: 1.1804 - mse: 1.1804 - val_loss: 1.2047 -
val_mse: 1.2047
Epoch 8/100
243/243 - 2s - 9ms/step - loss: 1.1760 - mse: 1.1760 - val_loss: 1.2035 -
val_mse: 1.2035
Epoch 9/100
243/243 - 2s - 9ms/step - loss: 1.1694 - mse: 1.1694 - val_loss: 1.1999 -
val_mse: 1.1999
Epoch 10/100
243/243 - 2s - 9ms/step - loss: 1.1767 - mse: 1.1767 - val_loss: 1.2029 -
val_mse: 1.2029
Epoch 11/100
243/243 - 2s - 9ms/step - loss: 1.1748 - mse: 1.1748 - val_loss: 1.2004 -
val_mse: 1.2004
```

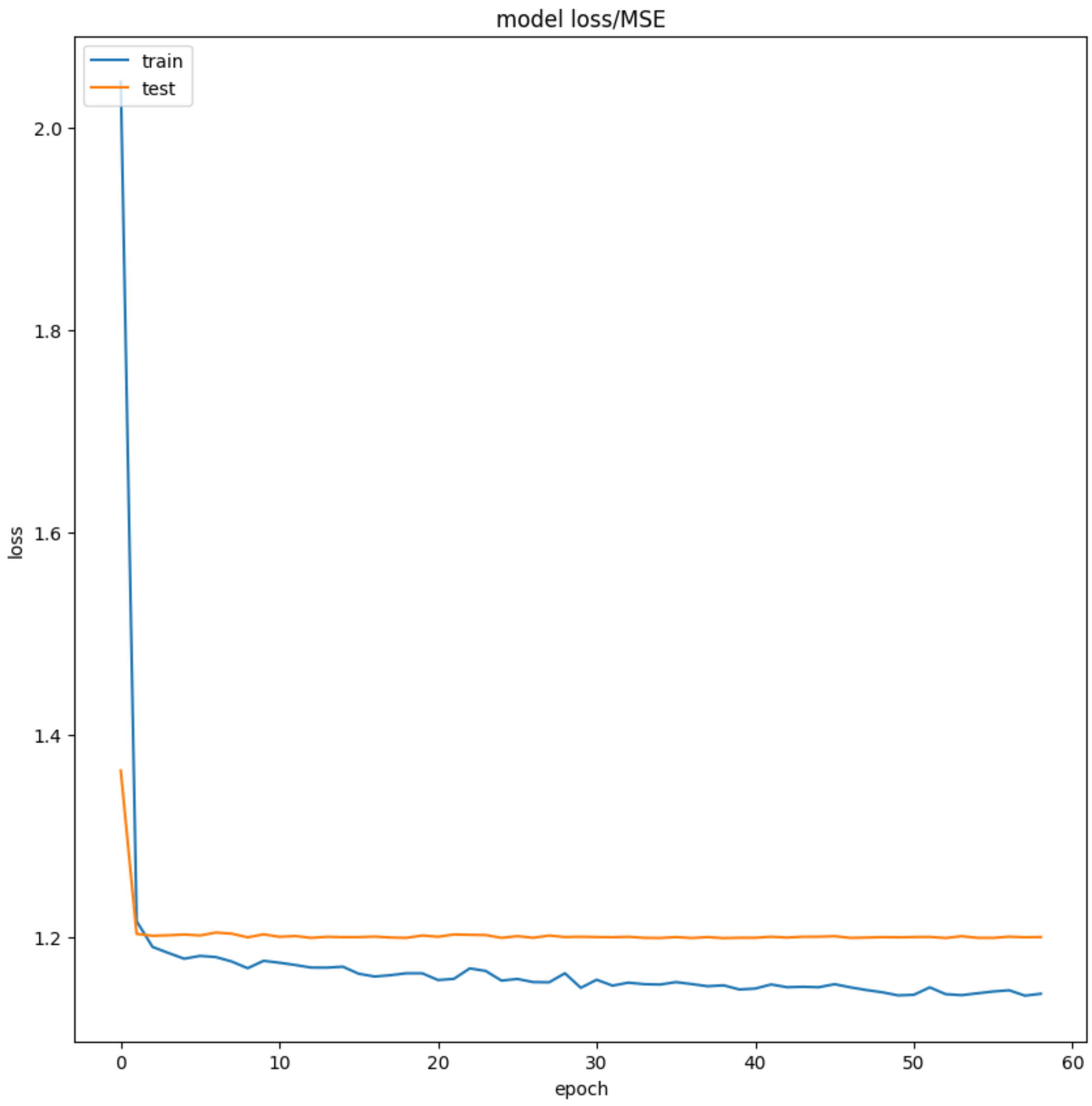
```
Epoch 12/100
243/243 - 2s - 9ms/step - loss: 1.1726 - mse: 1.1726 - val_loss: 1.2012 -
val_mse: 1.2012
Epoch 13/100
243/243 - 2s - 9ms/step - loss: 1.1700 - mse: 1.1700 - val_loss: 1.1994 -
val_mse: 1.1994
Epoch 14/100
243/243 - 2s - 9ms/step - loss: 1.1700 - mse: 1.1700 - val_loss: 1.2004 -
val_mse: 1.2004
Epoch 15/100
243/243 - 2s - 9ms/step - loss: 1.1709 - mse: 1.1709 - val_loss: 1.2001 -
val_mse: 1.2001
Epoch 16/100
243/243 - 2s - 9ms/step - loss: 1.1639 - mse: 1.1639 - val_loss: 1.2001 -
val_mse: 1.2001
Epoch 17/100
243/243 - 2s - 9ms/step - loss: 1.1612 - mse: 1.1612 - val_loss: 1.2006 -
val_mse: 1.2006
Epoch 18/100
243/243 - 2s - 9ms/step - loss: 1.1625 - mse: 1.1625 - val_loss: 1.1997 -
val_mse: 1.1997
Epoch 19/100
243/243 - 2s - 9ms/step - loss: 1.1644 - mse: 1.1644 - val_loss: 1.1994 -
val_mse: 1.1994
Epoch 20/100
243/243 - 2s - 9ms/step - loss: 1.1644 - mse: 1.1644 - val_loss: 1.2017 -
val_mse: 1.2017
Epoch 21/100
243/243 - 2s - 9ms/step - loss: 1.1577 - mse: 1.1577 - val_loss: 1.2005 -
val_mse: 1.2005
Epoch 22/100
243/243 - 2s - 9ms/step - loss: 1.1589 - mse: 1.1589 - val_loss: 1.2027 -
val_mse: 1.2027
Epoch 23/100
243/243 - 2s - 9ms/step - loss: 1.1692 - mse: 1.1692 - val_loss: 1.2024 -
val_mse: 1.2024
Epoch 24/100
243/243 - 2s - 9ms/step - loss: 1.1667 - mse: 1.1667 - val_loss: 1.2021 -
val_mse: 1.2021
Epoch 25/100
243/243 - 2s - 9ms/step - loss: 1.1572 - mse: 1.1572 - val_loss: 1.1994 -
val_mse: 1.1994
Epoch 26/100
243/243 - 2s - 9ms/step - loss: 1.1588 - mse: 1.1588 - val_loss: 1.2011 -
val_mse: 1.2011
Epoch 27/100
243/243 - 2s - 9ms/step - loss: 1.1557 - mse: 1.1557 - val_loss: 1.1995 -
val_mse: 1.1995
```

```
Epoch 28/100
243/243 - 2s - 9ms/step - loss: 1.1555 - mse: 1.1555 - val_loss: 1.2017 -
val_mse: 1.2017
Epoch 29/100
243/243 - 2s - 9ms/step - loss: 1.1644 - mse: 1.1644 - val_loss: 1.2002 -
val_mse: 1.2002
Epoch 30/100
243/243 - 2s - 9ms/step - loss: 1.1500 - mse: 1.1500 - val_loss: 1.2005 -
val_mse: 1.2005
Epoch 31/100
243/243 - 2s - 9ms/step - loss: 1.1580 - mse: 1.1580 - val_loss: 1.2002 -
val_mse: 1.2002
Epoch 32/100
243/243 - 2s - 9ms/step - loss: 1.1522 - mse: 1.1522 - val_loss: 1.2000 -
val_mse: 1.2000
Epoch 33/100
243/243 - 2s - 9ms/step - loss: 1.1551 - mse: 1.1551 - val_loss: 1.2004 -
val_mse: 1.2004
Epoch 34/100
243/243 - 2s - 9ms/step - loss: 1.1537 - mse: 1.1537 - val_loss: 1.1994 -
val_mse: 1.1994
Epoch 35/100
243/243 - 2s - 9ms/step - loss: 1.1533 - mse: 1.1533 - val_loss: 1.1993 -
val_mse: 1.1993
Epoch 36/100
243/243 - 2s - 9ms/step - loss: 1.1556 - mse: 1.1556 - val_loss: 1.2002 -
val_mse: 1.2002
Epoch 37/100
243/243 - 2s - 9ms/step - loss: 1.1537 - mse: 1.1537 - val_loss: 1.1992 -
val_mse: 1.1992
Epoch 38/100
243/243 - 2s - 9ms/step - loss: 1.1516 - mse: 1.1516 - val_loss: 1.2001 -
val_mse: 1.2001
Epoch 39/100
243/243 - 2s - 9ms/step - loss: 1.1525 - mse: 1.1525 - val_loss: 1.1991 -
val_mse: 1.1991
Epoch 40/100
243/243 - 2s - 9ms/step - loss: 1.1484 - mse: 1.1484 - val_loss: 1.1995 -
val_mse: 1.1995
Epoch 41/100
243/243 - 2s - 9ms/step - loss: 1.1493 - mse: 1.1493 - val_loss: 1.1995 -
val_mse: 1.1995
Epoch 42/100
243/243 - 2s - 9ms/step - loss: 1.1534 - mse: 1.1534 - val_loss: 1.2005 -
val_mse: 1.2005
Epoch 43/100
243/243 - 2s - 9ms/step - loss: 1.1506 - mse: 1.1506 - val_loss: 1.1997 -
val_mse: 1.1997
```

```
Epoch 44/100
243/243 - 2s - 9ms/step - loss: 1.1510 - mse: 1.1510 - val_loss: 1.2005 -
val_mse: 1.2005
Epoch 45/100
243/243 - 2s - 9ms/step - loss: 1.1506 - mse: 1.1506 - val_loss: 1.2005 -
val_mse: 1.2005
Epoch 46/100
243/243 - 2s - 9ms/step - loss: 1.1536 - mse: 1.1536 - val_loss: 1.2010 -
val_mse: 1.2010
Epoch 47/100
243/243 - 2s - 9ms/step - loss: 1.1505 - mse: 1.1505 - val_loss: 1.1994 -
val_mse: 1.1994
Epoch 48/100
243/243 - 2s - 9ms/step - loss: 1.1478 - mse: 1.1478 - val_loss: 1.1997 -
val_mse: 1.1997
Epoch 49/100
243/243 - 2s - 9ms/step - loss: 1.1456 - mse: 1.1456 - val_loss: 1.2001 -
val_mse: 1.2001
Epoch 50/100
243/243 - 2s - 9ms/step - loss: 1.1426 - mse: 1.1426 - val_loss: 1.1999 -
val_mse: 1.1999
Epoch 51/100
243/243 - 2s - 9ms/step - loss: 1.1431 - mse: 1.1431 - val_loss: 1.2002 -
val_mse: 1.2002
Epoch 52/100
243/243 - 2s - 9ms/step - loss: 1.1505 - mse: 1.1505 - val_loss: 1.2003 -
val_mse: 1.2003
Epoch 53/100
243/243 - 2s - 9ms/step - loss: 1.1438 - mse: 1.1438 - val_loss: 1.1993 -
val_mse: 1.1993
Epoch 54/100
243/243 - 2s - 9ms/step - loss: 1.1427 - mse: 1.1427 - val_loss: 1.2010 -
val_mse: 1.2010
Epoch 55/100
243/243 - 2s - 9ms/step - loss: 1.1446 - mse: 1.1446 - val_loss: 1.1994 -
val_mse: 1.1994
Epoch 56/100
243/243 - 2s - 9ms/step - loss: 1.1464 - mse: 1.1464 - val_loss: 1.1994 -
val_mse: 1.1994
Epoch 57/100
243/243 - 2s - 9ms/step - loss: 1.1476 - mse: 1.1476 - val_loss: 1.2006 -
val_mse: 1.2006
Epoch 58/100
243/243 - 2s - 9ms/step - loss: 1.1423 - mse: 1.1423 - val_loss: 1.2000 -
val_mse: 1.2000
Epoch 59/100
243/243 - 2s - 9ms/step - loss: 1.1442 - mse: 1.1442 - val_loss: 1.2002 -
val_mse: 1.2002
```

```
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])
```

[61]:
```
# 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_opt.png")
```



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: The model changes, such as reducing the number of LSTM units, adjusting the learning rate, and reducing the batch size slightly improved stability, as seen in the validation loss curve. However, the performance improvement was minimal, and the model still shows some signs of overfitting, as evidenced by the gap between the training and validation loss. The reduced learning rate helped to smooth the convergence, but it also slowed the model's learning, potentially limiting its ability to capture complex patterns. - For optimization in production, I would try the following: 1. Increase the sequence length to capture more context and long-term dependencies. 2. Experiment with more regularization techniques although using L2 regularization in this model did not improve the performance, but I would try other regularization methods. 3. Use more features (e.g., voltage, time-of-day) to improve model accuracy. 4. Fine-tune the optimizer (try SGD with momentum) and learning rate schedules (like learning rate decay).

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: The LSTM models in this assignment outperformed the linear regression model in terms of flexibility to apture complex temporal dependencies in the data. However, the linear regression model was simpler, faster to train, and computationally more efficient, making it more suitable for resource-constrained IoT devices.

For an IoT system predicting GAP for a single household with a 5-minute predictive horizon, Linear Regression would be the better choice due to its lower computational requirements, time-saving, simplicity, and sufficient accuracy for short-term predictions. LSTM models, while more accurate for capturing complex patterns, it overkilled this specific use case and introduced unnecessary complexity and resource demands.