```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
import yfinance as yf
import seaborn as sns
from keras.layers import Dense, LSTM, Dropout
from keras.models import Sequential
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
pd.options.mode.chained_assignment = None
import tensorflow as tf
from zipfile import ZipFile
import keras
tf.random.set_seed(0)
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False

if IN_COLAB:
    !pip install keras-tuner
    !mkdir "logs/lstm_kt"
```

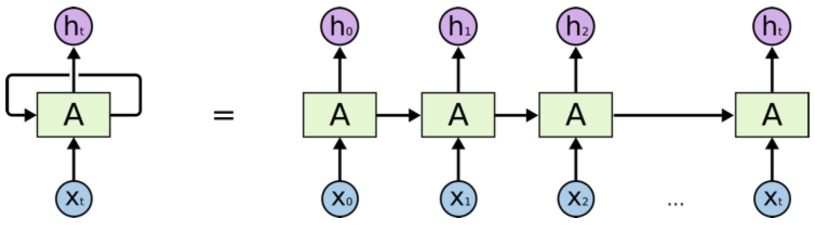
```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: keras-tuner in /usr/local/lib/python3.9/dist-packages (1.3.4)
Requirement already satisfied: kt-legacy in /usr/local/lib/python3.9/dist-packages (from keras-tuner) (1.0.4)
Requirement already satisfied: packaging in /usr/local/lib/python3.9/dist-packages (from keras-tuner) (23.0)
Requirement already satisfied: protobuf<=3.20.3 in /usr/local/lib/python3.9/dist-packages (from keras-tuner) (3.20.3)
Requirement already satisfied: tensorflow>=2.0 in /usr/local/lib/python3.9/dist-packages (from keras-tuner) (2.12.0)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from keras-tuner) (2.27.1)
Requirement already satisfied: wrapt<1.15,>=1.11.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (1.14.1)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (1.16.0)
Requirement already satisfied: keras<2.13,>=2.12.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (2.12.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (67.6.1)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (1.53.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (1.6.3)
Requirement already satisfied: tensorboard<2.13,>=2.12 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (2.12.1)
Requirement already satisfied: jax>=0.3.15 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (0.4.7)
Requirement already satisfied: gast<=0.4.0,>=0.2.1 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (0.4.0)
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (3.8.0)
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (16.0.0)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (4.5.0)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (3.3.0)
Requirement already satisfied: tensorflow-estimator<2.13,>=2.12.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (2.12.
Requirement already satisfied: flatbuffers>=2.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (23.3.3)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (0.3
Requirement already satisfied: numpy<1.24,>=1.22 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (1.22.4)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (2.2.0)
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (0.2.0)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.9/dist-packages (from tensorflow>=2.0->keras-tuner) (1.4.0)
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner) (2.0.12)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner) (1.26.15)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner) (2022.12.7)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests->keras-tuner) (3.4)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.9/dist-packages (from astunparse>=1.6.0->tensorflow>=2.0->keras-tuner) (0.
Requirement already satisfied: scipy>=1.7 in /usr/local/lib/python3.9/dist-packages (from jax>=0.3.15->tensorflow>=2.0->keras-tuner) (1.10.1)
Requirement already satisfied: ml-dtypes>=0.0.3 in /usr/local/lib/python3.9/dist-packages (from jax>=0.3.15->tensorflow>=2.0->keras-tuner) (0.0.4)
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorflow>=2.0->keras-t
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorflow>=2.0-
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorfl
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorflow>=2.0->keras-tuner)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorflow>=2.0->keras-tuner)
Requirement already satisfied: google-auth-oauthlib<1.1,>=0.5 in /usr/local/lib/python3.9/dist-packages (from tensorboard<2.13,>=2.12->tensorflow>=2.0
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python3.9/dist-packages (from google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.9/dist-packages (from google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->t
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.9/dist-packages (from google-auth<3,>=1.6.3->tensorboard<2.13,>=2.12->tensorflo
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.9/dist-packages (from google-auth-oauthlib<1.1,>=0.5->tensorboard<2.
Requirement already satisfied: importlib-metadata>=4.4 in /usr/local/lib/python3.9/dist-packages (from markdown>=2.6.8->tensorboard<2.13,>=2.12->tenso
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.9/dist-packages (from werkzeug>=1.0.1->tensorboard<2.13,>=2.12->tensorflow>
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.9/dist-packages (from importlib-metadata>=4.4->markdown>=2.6.8->tensorboard<2.13,>=
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.9/dist-packages (from pyasn1-modules>=0.2.1->google-auth<3,>=1.6.3->tens
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.9/dist-packages (from requests-oauthlib>=0.7.0->google-auth-oauthlib<1.1,>=0.
mkdir: cannot create directory ‘logs/lstm_kt’: File exists
```

```
# Helper to plot loss
def plot_loss(history):
  plt.plot(history.history['loss'], label='loss')
  #plt.plot(history.history['val_loss'], label='val_loss')
  plt.legend()
  plt.grid(True)
```

```
epochs = 50
ticker = "MSFT"
n_lookback = 60  # length of input sequences (lookback period)
n_forecast = 10  # length of output sequences (forecast period)
per = "1y"
layer_size = 4
batch = 4
metrics = ["mse"]
```

Recurrent Neural Networks

The RNN type of network and its uses mirror that of a CNN - while the convolutional neural networks excel at capturing spatial relationships in data, RNNs excel at capturing temporal relationships, or things that change over time. This leads to this type of network to be well suited to time series types of problems, and also things like text processing, where the words that occur earlier in a sentence are connected to those that occur later.



The unique part of RNNs is that they can loop back, thus allowing the model to discover temporal relationships, or relationships that span over sequential data.

RNN Structure

We will not focus much on a vanilla RNN model, we'll look at a variant - the Long Short-Term Memory model below. We should briefly look at the basics that make up a RNN. The structure of an RNN is similar to that of other neural networks, but with a few key differences:

- The first is that the input is a sequence of data, rather than a single image.
- The second is that the output is a sequence of data, rather than a single value.
- The third is that the network has looping, which allows it to remember things that happened in the past.

Clearly, the attributes of RNNs make them well suited to time series problems, where the input is a sequence of data, and the output is also a sequence of data; they are also heavily used in text processing and NLP problems, where the input is a sequence of words, and the output is also a sequence of words. Until relatively recently, RNNs were the state of the art for NLP problems, particularly generative models, but the advent of the Transformer architecture has made them less popular.

RNN Memory

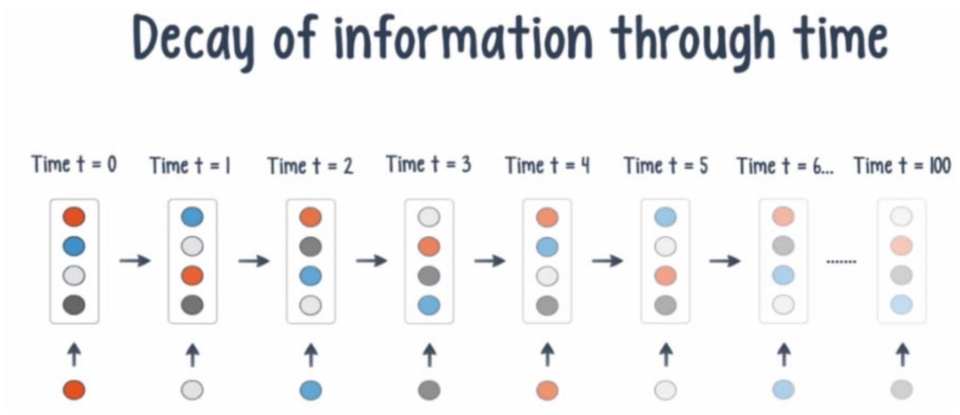
The defining characteristic of a RNN model is that it holds a "state", or a memory of the data that came from previous time steps. In NLP applications this state remembers information from what came previously in a sentence. In a time series application, this state remembers information from previous time steps. Just as the convolution operation in a CNN captures spatial relationships, the RNN operation captures temporal relationships with this state. As the model progresses through the sequential data, the state that it holds both informs the output and is updated by the data in the current time step.

Vanishing (or Exploding) Gradients

A main reason that plain implementations of RNN models aren't used all that much is the problem of vanishing, or sometimes exploding, gradients. Recall that the gradient when we are doing gradient descent is the slope of the loss curve with respect to the weights, and we use that gradient along with the learning rate to adjust the weights up or down to make a great model. One problem with a RNN is that we need to also perform that gradient descent update of weights back through the time steps that make up our model. This means that the gradient is multiplied by the weights at each time step, and if the weights are small we can end up with gradients that are basically flat (no slope, or gradient to the curve) which can mean that the weights aren't really "learning", as they aren't being adjusted substantially as the model trains. This is called the vanishing gradient problem, the opposite, exploding gradients, is when the gradient is large and compounds over time. the problem of vanishing gradients is most easy to think of by thinking about NLP problems. If we have a sentence such as:

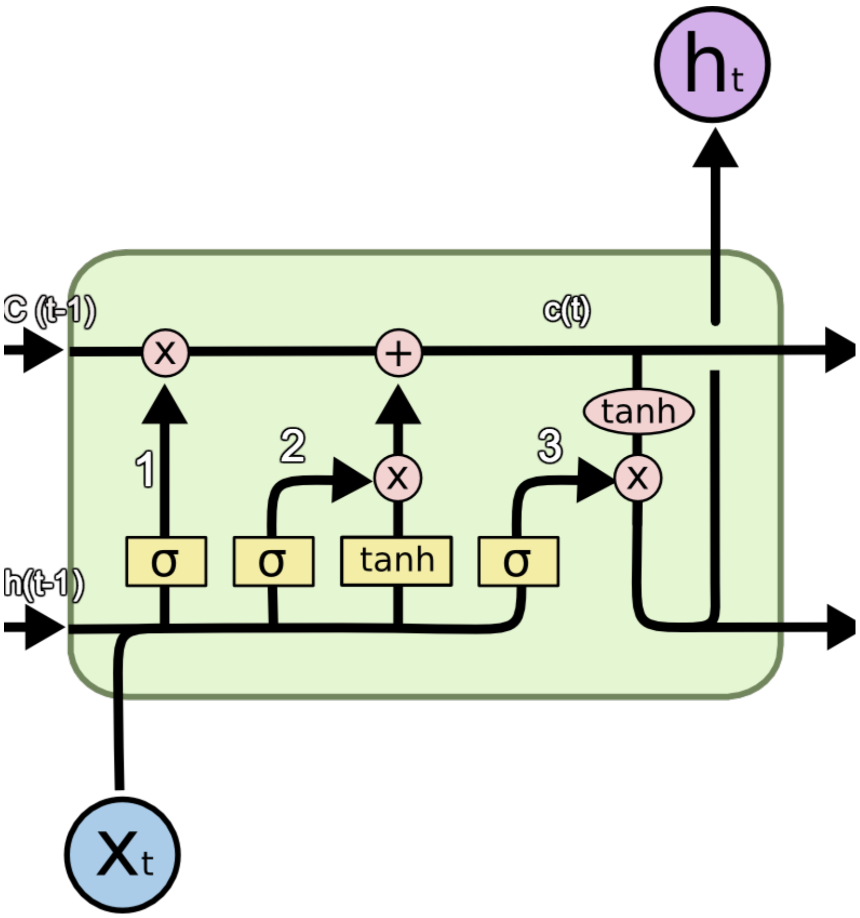
- "France is where I grew up, I speak fluent French."

The early information that someone grew up in France will likely inform the language that they speak fluently. If our model is experiencing a vanishing gradient problem, it may not be able to reach "fluent" and then remember that France is in the memory, as the gradients going back through all the layers and all the time steps is so small that the problem of "I speak SOME LANGUAGE fluently" isn't able to be learned with the knowledge that "they are from France", as the small gradients aren't changing, or learning, all the way back to the early weights. We can visualize it like this, the early steps and the late steps have a connection that fades over time.



Long Short Term Memory

Long Short Term Memory (lstm) models are a type of RNN that we can commonly use to make time series predictions. LSTM models function to "remember" certain data and carry that forward, and forget other data. This gives LSTM models the ability to combat the vanishing gradient to "remember" certain data and carry that forward, and forget other data. This gives LSTM models the ability to combat the vanishing gradient problem and maintain a "better memory" than a plain implementation of a RNN. LSTM models work by holding a couple of "states", or memory, and control the flow of data through the network and into those states with a set of gates. These LSTM models are near state of the art for sequental problems, surpassed only recently by the Transformer architecture that is in large language models like the GPTs.



LSTM models have some internal magic that allows them to remember data:

- Cell state (C) - the "long term memory" of the model.
- Hidden state (h) - the "short term memory" of the model.
- Forget gate (1) - determines which old data can be dropped.
- Input gate (2) - processes new data.
- Output gate (3) - combines the "held" old data with the new data to generate the output.

LSTM States

The LSTM model holds two types of memory, the long and the short term. Each step of the model's training involves updating both of these states with the new incoming data at that time step. Each state is passed from time step to time step through the model, and the new data is used to update the states by each gate.

LSTM Gates

The flow of data into the states is controlled by a series of gates, each using either the tanh or the signmoid activation functions, strategically. Each activation function serves a different purpose here:

- Sigmoid - acts to scale the importance of weights.
- Tanh - acts to embed the current state in a normalized space.

The gates are:

- Forget gate - determines which data to forget. The forget gate uses the sigmoid activation function on the current input and the previous hidden state to determine which data to forget. This is passed up to the cell state and multiplied by the current cell state. This acts to raise values that are "important" and lower values that are "unimportant".
- Input gate - determines which data to update. This works in conjunction with the "candidate gate", which is the tanh part - techincally it is its own thing, but it is effectively half of the input gate. The input gate generates both a representation and an importance scale of the hidden state and input data with the tanh and sigmoid activation functions. The representation is then multiplied by the importance scale and added to the current cell state. This "adds" the new data to the current cell state, but only the data that is "important" is added.
- Output gate - determines which data to output. This takes a representation of the *new* cell state multiplied by the sigmoid activation function of the hidden state and input data. This takes the importance of the current time step combined with the understanding of the current state. This value is also what is ultimately output from the model, to go into the dense layers for the final prediction.

LSTM Model Data

We'll use the closing price as our target here, we can download the data from the yahoo finance API. One of the good things about using LSTM models is that they are much more accepting than the more basic ARIMA-ish models in terms of data preparation. We don't need to deconstruct the data to try to make it stationary as the model is much better able to capture the underlying patterns in the data. The ARIMA model is basically a linear regression model, which we know is not super great at capturing non-linear relationships. The LSTM model has no such restriction, and if we have enough data and a reasonably repeatable pattern in the data, we can likely capture that pattern in the model itself. Personally, I find all the elaborate time series data preparation to be a pain, using the LSTM appoach makes it much more similar to regular regression/classification problems.

```
# download the data
df = yf.download(tickers=[ticker], period=per)
df.head()
```

```
[*********************100%***********************]  1 of 1 completed
```

	Open	High	Low	Close	Adj Close	Volume
Date						
2022-04-12	289.239990	290.739990	280.489990	282.059998	279.319763	30966700
2022-04-13	282.730011	288.579987	281.299988	287.619995	284.825745	21907200
2022-04-14	288.089996	288.309998	279.320007	279.829987	277.111420	28221600
2022-04-18	278.910004	282.459991	278.339996	280.519989	277.794739	20778000
2022-04-19	279.380005	286.170013	278.410004	285.299988	282.528259	22297700

```
y = df['Close'].fillna(method='ffill')
y = y.values.reshape(-1, 1)
y.shape
```

```
(250, 1)
```

```
# scale the data
scaler = MinMaxScaler(feature_range=(0, 1))
scaler = scaler.fit(y)
y = scaler.transform(y)
print(y[:10])
```

```
[[0.85597067]
 [0.92615494]
 [0.82782108]
 [0.83653103]
 [0.89686931]
 [0.91024974]
 [0.84019183]
 [0.7546074 ]
 [0.8390558 ]
 [0.70651351]]
```

Generate Data for Predictions

For each point in our data we generate additional data at that point. LSTM models expect data to always be a 3D tensor of the dimensions:

- [batch_size, timesteps, feature]

The outcome of this will be data where each time point is now no longer just 1 value, like a normal time series, but it is a bunch of dates - this is the "long term" memory part. The X's shape is:

- (# of rows of data, # of lookback rows, number of features)
- I.e. each "row" now has the past 60 values as part of it.

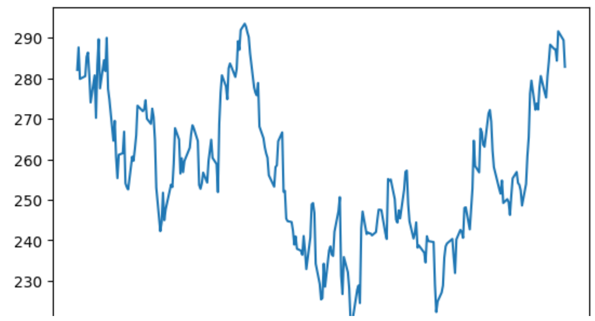
The Y's shape is:

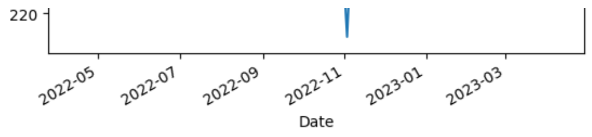
- (# of rows of data, # of forecasting rows, number of features)
- I.e. each row is a prediction into the future.

Each of the inputs for our data "remembers" the past 60 days into the past! This is what allows these models to do such a good job on sequential data.

```
df['Close'].plot()
```

```
<Axes: xlabel='Date'>
```





```
# generate the input and output sequences
X = []
Y = []

for i in range(n_lookback, len(y)):
    X.append(y[i - n_lookback: i])
    y_tmp = y[i: i + n_forecast]
    Y.append(y_tmp[0][0])

X = np.array(X)
Y = np.array(Y)
print(X.shape, Y.shape)
```

```
    (190, 60, 1) (190,)
```

```
X[1]
```

```
    array([[0.92615494],
           [0.82782108],
           [0.83653103],
           [0.89686931],
           [0.91024974],
           [0.84019183],
           [0.7546074 ],
           [0.8390558 ],
           [0.70651351],
           [0.87061348],
           [0.95152744],
           [0.7986618 ],
           [0.88639233],
           [0.85243622],
           [0.95594559],
           [0.7965161 ],
           [0.7634437 ],
           [0.63531918],
           [0.69742488],
           [0.58444821],
           [0.51880845],
           [0.59164345],
           [0.59644028],
           [0.66359513],
           [0.50277709],
           [0.49091137],
           [0.48358996],
           [0.58571059],
           [0.57270884],
           [0.60931568],
           [0.65198174],
           [0.74463506],
           [0.72734151],
           [0.73428443],
           [0.76154993],
           [0.70398874],
           [0.68795758],
           [0.73529411],
           [0.70891193],
           [0.63797031],
           [0.48901799],
           [0.35357226],
           [0.38172185],
           [0.47349147],
           [0.38778087],
           [0.42161062],
           [0.49848529],
           [0.49078521],
           [0.56311518],
           [0.6747035 ],
           [0.63923269],
           [0.53307259],
           [0.58078779],
           [0.53749035],
           [0.57220381],
           [0.61348151],
           [0.65589486],
           [0.68353942],
```

Fit Model

Our data is now ready to be used to fit a model, importantly, each X value is now a sequence containing the data of the past 60 days, rather than "just" a single value. So our input shape is now (60, 1). Each prediction takes in one value, and 60 time steps of it in the past; each prediction is a single value, and one time step of it.

We can now make a model and train it. The long-short term memory layers are mostly simple to use, we need to be aware of a few things:

- Input shape - this input shape is the size of the previous records and the number of features. Here we have 60 for the n_lookback and 1 for the number of features.
- Return Sequences - this controls if the LSTM layers return everything or just the end result. We want all layers that are feeding another LSTM layer to return sequences, but the last LSTM layer should not.

- Output - each forecast gets its own output neuron, we are predicting exactly one time step value.

The units is our size parameter for each layer. Like in other models, the larger the size, the more complex the model can be, but the more likely it is to overfit. There are a few more notes on sizing below.

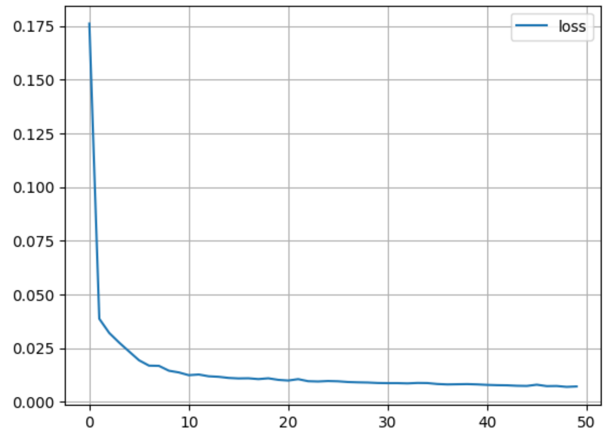
```
# fit the model
model = Sequential()
model.add(LSTM(units=layer_size, return_sequences=True, input_shape=(n_lookback, 1)))
model.add(LSTM(units=layer_size))
model.add(Dense(1))

model.compile(loss='mean_squared_error', optimizer='adam', metrics=metrics)
history = model.fit(X, Y, epochs=epochs, batch_size=batch, verbose=1)
tmp = model.predict(X)
```

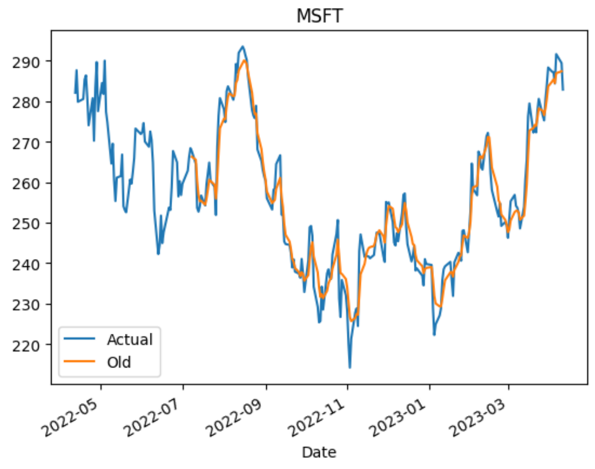
```
    Epoch 1/50
    48/48 [==============================] - 4s 8ms/step - loss: 0.1761 - mse: 0.1761
    Epoch 2/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0386 - mse: 0.0386
    Epoch 3/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0320 - mse: 0.0320
    Epoch 4/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0275 - mse: 0.0275
    Epoch 5/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0234 - mse: 0.0234
    Epoch 6/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0193 - mse: 0.0193
    Epoch 7/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0168 - mse: 0.0168
    Epoch 8/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0167 - mse: 0.0167
    Epoch 9/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0144 - mse: 0.0144
    Epoch 10/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0136 - mse: 0.0136
    Epoch 11/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0124 - mse: 0.0124
    Epoch 12/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0127 - mse: 0.0127
    Epoch 13/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0118 - mse: 0.0118
    Epoch 14/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0116 - mse: 0.0116
    Epoch 15/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0111 - mse: 0.0111
    Epoch 16/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0108 - mse: 0.0108
    Epoch 17/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0109 - mse: 0.0109
    Epoch 18/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0105 - mse: 0.0105
    Epoch 19/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0109 - mse: 0.0109
    Epoch 20/50
    48/48 [==============================] - 0s 7ms/step - loss: 0.0102 - mse: 0.0102
    Epoch 21/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0099 - mse: 0.0099
    Epoch 22/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0105 - mse: 0.0105
    Epoch 23/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0096 - mse: 0.0096
    Epoch 24/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0094 - mse: 0.0094
    Epoch 25/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0096 - mse: 0.0096
    Epoch 26/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0095 - mse: 0.0095
    Epoch 27/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0092 - mse: 0.0092
    Epoch 28/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0090 - mse: 0.0090
    Epoch 29/50
    48/48 [==============================] - 0s 8ms/step - loss: 0.0089 - mse: 0.0089
```

Print Results

```
# Print
old_preds = []
for i in range(len(tmp)):
    old_preds.append(tmp[i][0])

# organize the results in a data frame
df_past = df[['Close']].reset_index()
#print(len(df_past))
df_past.rename(columns={'index': 'Date', 'Close': 'Actual'}, inplace=True)
df_past['Date'] = pd.to_datetime(df_past['Date'])

df_past['Old'] = np.nan
for i in range(len(old_preds)):
    df_past["Old"].iloc[i+n_lookback-1] = scaler.inverse_transform(np.array(old_preds[i]).reshape(1,-1))

results = df_past.set_index('Date')
# plot the results
plot_loss(history)
results.plot(title=ticker)

[illegible]
```

```
<Axes: title={'center': 'MSFT'}, xlabel='Date'>
```





```
df_tmp = df_past[~df_past["Old"].isna()]
trainScore = math.sqrt(mean_squared_error(df_tmp["Actual"], df_tmp["Old"]))
print('Train Score: %.2f RMSE' % (trainScore))
```

```
Train Score: 4.06 RMSE
```

Forward Forecasts - Model with Larger Y

We can also try to predict more than one day into the future as a core feature of our model. With stats-based models, if we want to project multiple time steps into the future we have to use our initial predictions to feed the later predictions. With a LSTM we can generate a standard output prediction that is multiple timesteps to begin with. To do this we need to change the shape of the Y data that is being predicted, rather than predicting one value, we can create a Y set that is the next 10 values, giving us 2 weeks of price forecasts. So, for each date in the data, we have:

- X data this is the past 60 days of data.
- Y data this is the next 10 days of data.

These two values are the middle dimension of our data, the number of time steps. So, we need to make our dataset to model this - each prediction is now a sequence of 10 values, rather than a single value. Our 10 day future prediction isn't a run of the "next 10" predictions that come from the end of our model, they are one unitary output of our model that is 10 values long! Note that we can use the other method of feeding in predictions that we generate as a source for the next time-step's prediction, but this appoach is more elegant, normally easier to implement, and makes our model more "complete" in the sense that it will always generate its full output in one go. This may or may not be more accurate, that is not assured.

```
def stockPredPrep(data, column="Close", n_lookback=60, n_forecast=10):
    scaler = MinMaxScaler(feature_range=(0, 1))
    y = data[[column]].fillna(method='ffill')
    y = scaler.fit_transform(y)
    start = n_lookback
    end = len(y) - n_forecast
    #end = len(y)
    X = []
    Y = []
    print(start,end)
    for i in range(start, end):
        X.append(y[i - n_lookback: i])
        y_tmp = y[i: i + n_forecast]
        Y.append(np.array(y_tmp))
    return np.array(X), np.array(Y), scaler
```

```
X_n, Y_n, scaler_n = stockPredPrep(df, n_lookback=n_lookback, n_forecast=n_forecast)
```

```
X_p, Y_p, scaler_p = stockPredPrep(df, n_lookback=n_lookback, n_forecast=n_forecast)
print(X_p.shape, Y_p.shape)
print(Y_p[0])
```

```
    60 240
    (180, 60, 1) (180, 10, 1)
    [[0.63443586]
     [0.49760158]
     [0.4856097 ]
     [0.50277709]
     [0.536102  ]
     [0.50492299]
     [0.57157281]
     [0.60615991]
     [0.63860131]
     [0.58204979]]
```

Multi-Step Predictions

Each y value is now a sequence of 10 values - the next 10 days of price. The output layer of our network also needs to be rearranged to match this, with one neuron for each prediction that is part of one Y value. This also changes the way the model is being trained, as now the 10 timestep prediction is the thing that is being evaluated for loss. We are technically not creating the model that minimizes the loss of one prediction, we are creating the model that minimizes the loss of the 10 predictions. Whether this is good or bad depends somewhat on perspective. It does make the structure of the model easier to use, as we don't need to feed predictions back into a model to get longer range predictions. This note is important, in a normal time-series model we were always generating predictions for the next time step in some ever-rolling manner if we wanted to make projections farther into the future (potentially outside of the Facebook Prohet models, which are more sophisticated internally). This means that there was no direct relationship in those models between "now" and "2 weeks from now" in the model, we were basically trusting that our "tomorrow" predictions would stack up to be accurate into the future. This is also a reason that deconstructing the trends and cyclicality of the data in those time series models was very important, we needed that so the model knew how to structure the patterns of its predictions.

In a multi-step prediction with our LSTM model we can construct our target for the model itself to by "N timesteps" rather than the next time step, like we are used to. This means that we are always generating a "N step" prediction and always evaluting the accuracy of those N steps when calculating the loss and optimizing our model. The result here is a "real" prediction that "based on the current information at this time step, we expect the value N steps ahead to be M", and that prediction will be based on the accuracy of the previous "N days out" predictions that we've made during training. Combined with the large capacity of the models and the ability to construct deep models, this opens the door to the accurate "far out" forecasting that models are capable of, given substantial data. This isn't a guarantee that these multi-step predictions will be the most accurate, we are still predicting something that hasn't happened yet, and reality is tricky. This is also one of the most relatable reasons I can think of to define a custom loss function, the way we treat error for predictions of tomorrow vs predictions of 2 weeks from now can reasonably be very different in different scenarios. For example, a stock price forecasting model needs to be very accuracte (potentially less than a cent) for predictions used to make automated high frequency trading decisions, but it doesn't need to be as accurate for predictions of stocks that are likely to climb over the next year. Defining a custom loss function that evaluates the accuracy of a model's predictions based on the real world impact of those errors is pretty likely if a time series model is fit into a very specific situation.

Modelling Options

We can also try adding some more complexity to the model, just like any other network. One note is that if we add more lstm layers, we need to turn on the return sequences flag, so we are passing all the data to the next layer. This is basically a "keep LSTMing" or "stop LSTMing" flag that we always leave off on only the final LSTM layer and never think about again. There are a few other things that we *could* do with a model using this and another similar option, return_state, that returns the hidden state information. These can be used to develop customized architectures when using LSTM in functional models - the kind that we can customize almost limitlessly. Doing this is generally both complex and context dependent, for example, if we were developing audio recognition models for a company like Shazam (phone app that hears a song and looks it up for you) we may want to take advantage of things like this to create a model that is specifically good at exactly what we want to do. The Shazam model needs to extract a song from other audio like talking in an often noisy environment, process the audio that has been "distorted" by different audio systems (e.g. nightclub, car radio, tiny built in TV speakers...), match it to one of all other songs (bazillion way softmax classification?) based on any small clip, and do all of this within a couple of seconds. I am definately not in charge of audio engineering at Shazam, but I can go out on a pretty stable limb and say that they have models that are highly tailored to specific aspects of audio processing that are far more important for the odd challenge they have at hand. Long story short, this is one of the things that one could use to create such types of models, but it goes beyond our scope.

```
# fit the model
model = Sequential()
model.add(LSTM(units=layer_size, return_sequences=True, input_shape=(n_lookback, 1)))
model.add(LSTM(units=layer_size))
model.add(Dropout(0.2))
model.add(Dense(10))

model.compile(loss='mean_squared_error', optimizer='adam', metrics=metrics)
history = model.fit(X_p, Y_p, epochs=epochs, batch_size=batch, verbose=1)
tmp = model.predict(X_p)
```

```
    Epoch 1/50
    45/45 [==============================] - 3s 7ms/step - loss: 0.2569 - mse: 0.2569
    Epoch 2/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.1750 - mse: 0.1750
    Epoch 3/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.1284 - mse: 0.1284
    Epoch 4/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0956 - mse: 0.0956
    Epoch 5/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0892 - mse: 0.0892
    Epoch 6/50
    45/45 [==============================] - 0s 6ms/step - loss: 0.0714 - mse: 0.0714
    Epoch 7/50
    45/45 [==============================] - 0s 6ms/step - loss: 0.0646 - mse: 0.0646
    Epoch 8/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0626 - mse: 0.0626
    Epoch 9/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0604 - mse: 0.0604
    Epoch 10/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0515 - mse: 0.0515
    Epoch 11/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0491 - mse: 0.0491
    Epoch 12/50
    45/45 [==============================] - 0s 6ms/step - loss: 0.0543 - mse: 0.0543
    Epoch 13/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0521 - mse: 0.0521
    Epoch 14/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0452 - mse: 0.0452
    Epoch 15/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0499 - mse: 0.0499
    Epoch 16/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0446 - mse: 0.0446
    Epoch 17/50
    45/45 [==============================] - 0s 8ms/step - loss: 0.0464 - mse: 0.0464
    Epoch 18/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0404 - mse: 0.0404
    Epoch 19/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0379 - mse: 0.0379
    Epoch 20/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0436 - mse: 0.0436
    Epoch 21/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0395 - mse: 0.0395
    Epoch 22/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0414 - mse: 0.0414
    Epoch 23/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0411 - mse: 0.0411
    Epoch 24/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0404 - mse: 0.0404
    Epoch 25/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0375 - mse: 0.0375
    Epoch 26/50
    45/45 [==============================] - 0s 8ms/step - loss: 0.0409 - mse: 0.0409
    Epoch 27/50
    45/45 [==============================] - 0s 8ms/step - loss: 0.0395 - mse: 0.0395
    Epoch 28/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0332 - mse: 0.0332
    Epoch 29/50
    45/45 [==============================] - 0s 7ms/step - loss: 0.0359 - mse: 0.0359
```

Check a Prediction

We can check a prediction to see what it looks like. The final prediction is now 10 numbers long, we have 10 predictions for the next 10 days!

```
scaler_p.inverse_transform(tmp[-1].reshape(-1, 1))
```

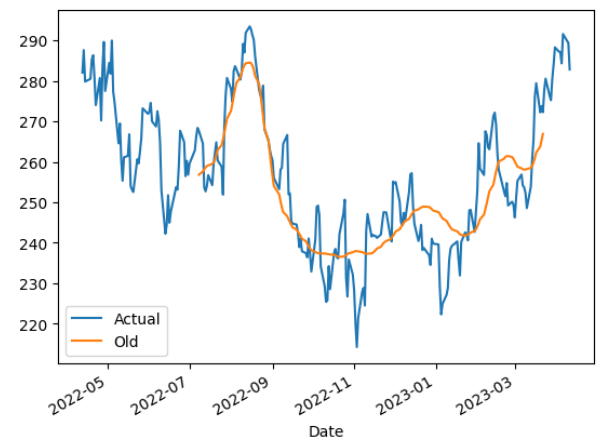
```
    array([[278.34912],
           [277.14767],
           [278.61234],
           [276.31705],
           [277.2228 ],
           [276.26025],
           [277.66287],
           [277.3053 ],
           [275.50757],
           [277.78903]], dtype=float32)
```

Plot Predictions

Because we have a bunch of predictions for each individual record, we need to collapse them together somehow to be able to make a simple plot of our predictions. There are several ways to do this, depending on what you want, but here we will just take the average of the predictions for each day. This gives us one value for each day, which we can then plot. You might alternatively want to take the most recent, median, or a weighted average of the predictions - the full stacks of predictions is also returned, so alternate calculations are possible. We haven't investigated it here, but we are likely seeing some fairly inaccurate predictions for the predictions made 9 or 10 days in advance being averaged with some pretty accurate predictions for the predictions made 1 or 2 days in advance.

```
def mergePreds(preds, forecasts):
    tmp_preds = []
    mean_preds = []
    for i in range(len(preds)):
        end = i
        start = end - forecasts
        item_preds = []
        if start < 0:
            start = 0
        record = forecasts-1
        for n in range(start, end):
            item = preds[n][record]
            #print(item, item.shape, type(item))
            item_preds.append(item)
            record -= 1
        if len(item_preds) > 1:
            tmp_preds.append(item_preds)
            mean_preds.append(np.mean(item_preds))
    return tmp_preds, mean_preds

def plotStockPred(preds, real, scaler, y_size=10):
    # organize the results in a data frame
    # organize the results in a data frame
    df_past = real[['Close']].reset_index()
    #print(len(df_past))
    df_past.rename(columns={'index': 'Date', 'Close': 'Actual'}, inplace=True)
    df_past['Date'] = pd.to_datetime(df_past['Date'])

    df_past['Old'] = np.nan
    for i in range(len(preds)):
        #df_past["Old"].iloc[i+n_lookback-1] = scaler.inverse_transform(np.array(preds[i]).reshape(1,-1))
        val = scaler.inverse_transform(preds[i].reshape(1,-1))[0][0]
        #print(i, val)
        df_past["Old"].iloc[i+n_lookback-1] = val

    results = df_past.set_index('Date')
    return results


full_preds, mean_preds = mergePreds(tmp, n_forecast)

res = plotStockPred(mean_preds, df, scaler_p, n_forecast)
res.plot()
```

```
    <Axes: xlabel='Date'>
```



Future Predictions

We can make predictions into the future, we just need to feed the model the data up to the most recent record, and then have it predict the next 10 days. Our input X value for each prediciton is one record of data, which we've seen contains the past 60 days of data. To make a prediction, we need to assemble our "last day" row of data that contains the previous 60 days worth of data that we need.

```
future_data = stockPredPrep(df, n_forecast=0)
future_data[0][-1] # The last record
```

```
    60 250
    array([[0.30623572],
           [0.31532435],
           [0.32946233],
           [0.27215346],
           [0.22317587],
           [0.32782127],
           [0.35761173],
           [0.35079516],
           [0.33274426],
           [0.42602877],
           [0.42804851],
           [0.35925279],
           [0.42363036],
           [0.48598838],
           [0.63557189],
           [0.55667768],
           [0.536733  ],
           [0.67293609],
           [0.6624591 ],
           [0.62320114],
           [0.61663728],
           [0.72039897],
           [0.73112866],
           [0.69515282],
           [0.60464521],
           [0.55301688],
           [0.48497851],
           [0.4703357 ],
           [0.51148704],
           [0.44142894],
           [0.45329466],
           [0.44395352],
           [0.40419091],
           [0.46528654],
           [0.5180509 ],
           [0.53799539],
           [0.50366061],
           [0.49798026],
           [0.48056055],
           [0.43347634]
```

```
        [0.43347654],
        [0.50075735],
        [0.58747801],
        [0.64617523],
        [0.78199964],
        [0.82277192],
        [0.73188601],
        [0.75145163],
        [0.73264337],
        [0.80042922],
        [0.83716241],
        [0.7842717 ],
        [0.76975524],
        [0.83640506],
        [0.88109047],
        [0.93473853],
        [0.92123214],
        [0.92060075],
```

Shape and Predict

Mucho reshaping - we are getting the most recent set of data to predict for, since that'll give us a prediciton 10 days into the future. We are also reshaping that data - one batch, "however many" time steps, and one feature. So our shapes are:

- X shape: (1, 60, 1) - one batch, 60 time steps, 1 feature
- Y shape: (1, 10, 1) - one batch, 10 time steps, 1 feature

Both the X and Y are multidimensional here, but this is one record as far as the model is concerned. Similar to how with CNN models each record is a 2D (plus color depth) image, here each record is a 3D tensor. To make a prediction, we feed the X and get the Y, just as we are used to.

```
future_preds = model.predict(future_data[0][-1].reshape(1,-1,1))
future_preds = scaler_p.inverse_transform(future_preds.reshape(-1, 1))
future_preds
```

```
1/1 [==============================] - 0s 23ms/step
array([[279.9707 ],
       [279.59503],
       [279.26288],
       [277.5374 ],
       [279.18353],
       [278.12296],
       [279.45596],
       [278.3903 ],
       [277.50595],
       [278.888  ]], dtype=float32)
```

Combine with the old data. I will stack the original data together with the new predictions, and then plot the results.

```
df_pred = df["Close"]
df_pred = df_pred.to_frame()
df_pred["Date"] = df_pred.index

df_tmp = pd.DataFrame(columns=["Date", "Close"])
df_tmp["Date"] = pd.date_range(start=df_pred["Date"].iloc[-1] + pd.Timedelta(days=1), periods=len(future_preds))
df_tmp["Close"] = future_preds

df_pred.set_index("Date", inplace=True)
df_tmp.set_index("Date", inplace=True)

results = df_pred.append(df_tmp)
results.tail(10)
```

```
<ipython-input-66-bf9b5cb2fa1d>:12: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pand
  results = df_pred.append(df_tmp)
```

	Close
Date	
2023-04-12	279.970703
2023-04-13	279.595032
2023-04-14	279.262878
2023-04-15	277.537415
2023-04-16	279.183533
2023-04-17	278.122955
2023-04-18	279.455963
2023-04-19	278.390289
2023-04-20	277.505951
2023-04-21	278.888000

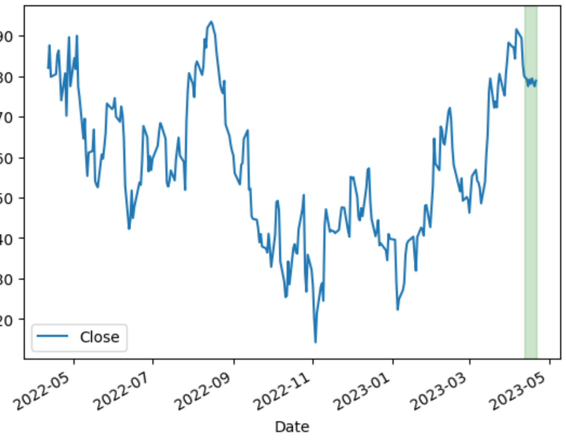
Plot It

I am going to just shade the prediction part on the plot here.

```
s = df_tmp.index[0]
```

```
e = df_tmp.index[-1]


results.plot()
plt.axvspan(s,e, color='g', alpha=0.2)
```

```
<matplotlib.patches.Polygon at 0x7f4abfca5460>
```



Datasets with More Features

We can add more features to the model as well. One of the benefits to using a neural network is that adding more features does not require the same work and effort as with traditional time series models. Since we are generating a prediciton of N days into the future at any given time, we aren't restricted to the values that "we can know for the future" or offset calculations like we had to do with the ordinary time series models. The increased capacity of neural networks to learn relationships means that our model will accept more data with few changes.

Reconstructing the data to accomodate more features is a bit more cumbersome, we need to create data that is:

- Y - the next N days of data.
  - Shape - batch size, number of timesteps to predict, 1 value (usually).
- X - the past N days of data, plus the features that we want to use.
  - Shape - batch size, number of timesteps to use (M days), number of features.

So for any one prediction, the X input is the M day lookback of both the price and all the other features. The Y output is the next N days of price.

Nicer Data Prep Function

We can pair this with a more generalized function to prep our data. This one should, assuming I didn't mess it up, allow us to use any number of features and set the time windows to look both forward and backward to whatever we want. The columns parameter limits the data to those columns, and the y_col specifies which of these is the target.

```
def prepare_stock_data(ticker, columns, y_col="Close", lookback=60, lookahead=10, timeframe="5y"):
    # Download stock prices for the past 5 years
    data = yf.download(ticker, period=timeframe)

    # Select the desired columns
    data = data[columns]

    # Prepare the data for an LSTM model
    x = []
    y = []
    for i in range(lookback, len(data)-lookahead):
        x.append(data.iloc[i-lookback:i].values)
        y.append(data[y_col][i:i+lookahead].values)
    x = np.array(x)
    y = np.array(y)

    # Reshape the data for the LSTM model
    x = np.reshape(x, (x.shape[0], x.shape[1], x.shape[2]))
    y = np.reshape(y, (y.shape[0], y.shape[1], 1))

    return x, y
```

For "real" predictions using the open, low, and close values doesn't make a lot of sense, but for a demonstration it's fine.

```
columns = ["Close", "Volume", "Open", "High", "Low"]

x_tmp, y_tmp = prepare_stock_data("AAPL", columns=columns, y_col="Close", lookback=n_lookback, lookahead=n_forecast)
print(x_tmp.shape, y_tmp.shape)
```

```
[*********************100%***********************]  1 of 1 completed
(1188, 60, 5) (1188, 10, 1)
```

Fit Model

The input shape for X is different now:

- Batch size, by...
- Number of lookback timesteps (M days), by...
- Number of features (price + volume + whatever else)

The output shape remains the same as before, we are still predicting 10 days into the future, so we output to 10 neurons and each output is a list of 10 individual predictions of one single value.

```
# fit the model
model = Sequential()
model.add(LSTM(units=layer_size, return_sequences=True, input_shape=(n_lookback, len(columns))))
model.add(LSTM(units=layer_size))
model.add(Dense(n_forecast))

model.compile(loss='mean_squared_error', optimizer='adam', metrics=metrics)
history = model.fit(x_tmp, y_tmp, epochs=epochs, batch_size=batch, verbose=1)
```

```
    Epoch 1/50
    297/297 [==============================] - 5s 6ms/step - loss: 13229.9600 - mse: 13229.9600
    Epoch 2/50
    297/297 [==============================] - 2s 7ms/step - loss: 12821.5078 - mse: 12821.5078
    Epoch 3/50
    297/297 [==============================] - 2s 6ms/step - loss: 12475.8975 - mse: 12475.8975
    Epoch 4/50
    297/297 [==============================] - 2s 6ms/step - loss: 12160.3682 - mse: 12160.3682
    Epoch 5/50
    297/297 [==============================] - 2s 7ms/step - loss: 11860.2686 - mse: 11860.2686
    Epoch 6/50
    297/297 [==============================] - 2s 7ms/step - loss: 11571.6494 - mse: 11571.6494
    Epoch 7/50
    297/297 [==============================] - 2s 6ms/step - loss: 11291.9727 - mse: 11291.9727
    Epoch 8/50
    297/297 [==============================] - 2s 7ms/step - loss: 11019.4814 - mse: 11019.4814
    Epoch 9/50
    297/297 [==============================] - 2s 6ms/step - loss: 10752.9795 - mse: 10752.9795
    Epoch 10/50
    297/297 [==============================] - 2s 6ms/step - loss: 10492.0977 - mse: 10492.0977
    Epoch 11/50
    297/297 [==============================] - 2s 6ms/step - loss: 10236.9854 - mse: 10236.9854
    Epoch 12/50
    297/297 [==============================] - 2s 6ms/step - loss: 9986.8662 - mse: 9986.8662
    Epoch 13/50
    297/297 [==============================] - 2s 6ms/step - loss: 9740.7920 - mse: 9740.7920
    Epoch 14/50
    297/297 [==============================] - 2s 7ms/step - loss: 9499.0684 - mse: 9499.0684
    Epoch 15/50
    297/297 [==============================] - 2s 6ms/step - loss: 9262.5771 - mse: 9262.5771
    Epoch 16/50
    297/297 [==============================] - 2s 6ms/step - loss: 9031.0137 - mse: 9031.0137
    Epoch 17/50
    297/297 [==============================] - 2s 6ms/step - loss: 8803.7900 - mse: 8803.7900
    Epoch 18/50
    297/297 [==============================] - 2s 6ms/step - loss: 8580.7344 - mse: 8580.7344
    Epoch 19/50
    297/297 [==============================] - 2s 6ms/step - loss: 8361.7949 - mse: 8361.7949
    Epoch 20/50
    297/297 [==============================] - 2s 7ms/step - loss: 8147.0532 - mse: 8147.0532
    Epoch 21/50
    297/297 [==============================] - 2s 6ms/step - loss: 7936.3950 - mse: 7936.3950
    Epoch 22/50
    297/297 [==============================] - 2s 6ms/step - loss: 7729.9780 - mse: 7729.9780
    Epoch 23/50
    297/297 [==============================] - 2s 6ms/step - loss: 7527.9385 - mse: 7527.9385
    Epoch 24/50
    297/297 [==============================] - 2s 6ms/step - loss: 7330.1455 - mse: 7330.1455
    Epoch 25/50
    297/297 [==============================] - 2s 6ms/step - loss: 7136.3779 - mse: 7136.3779
    Epoch 26/50
    297/297 [==============================] - 2s 7ms/step - loss: 6946.8306 - mse: 6946.8306
    Epoch 27/50
    297/297 [==============================] - 2s 6ms/step - loss: 6761.2393 - mse: 6761.2393
    Epoch 28/50
    297/297 [==============================] - 2s 6ms/step - loss: 6579.6416 - mse: 6579.6416
    Epoch 29/50
    297/297 [==============================] - 2s 7ms/step - loss: 6401.8906 - mse: 6401.8906
```

LSTM - Tuning Performance

Like any other neural network model, we can change some things to try to improve the performance of our model. Like any other network, the true test of our model will come down to experimentation and testing.

Unit Size and Layers

The "number of neurons" argument in creating an LSTM layer is the number of units in the layer. This parameter defines the size of the hidden state vector that is passed between the layers and combines with the size of the input to define the rest of the sizes of the weight matricies. The units is not related to the time steps, it is better though of as roughly the size of the data that we are processing at each time step. This works like the number of neurons in a normal dense layer, the larger the value, the higher the capacity of that layer to learn.

Adding layers to the LSTM model is the other general strategy to creating a more powerful model - the more layers we add, the more complex the model can be, and the more it can learn. The basic structure of an LSTM model is similar to what we saw in a CNN, we have the "specialized stuff" in the first layers, either LSTM or CNN, then we add on some amount to dense layers to generate a final prediction. Models that have multiple LSTM layers are often called "stacked LSTM" models.

As with dense models, we need to decide how large to make our model in capacity, and how to balance that betweeen the two ways to add capacity - "width", or the number of units in each layer, and "depth", or the number of layers. This is something that ultimately needs to be tested to be firmly determined, but we can use some rules of thumb to give us good starting points:

- If we have a "simple" pattern, like the examples that were easy to predict with moving average based strategies, one LSTM layer should be enough, likely with fewer than 50 units, maybe far fewer.
- For more complex patterns:
    - A good starting point is to use about 50 units the first LSTM layer. If the problem appears complex, and maybe one other layer with a similar number of units.
    - If the patterns in the data appear to be largely defined by **interactions between multiple input features** or by **pattern blocks**, such as those identified by technical stock trading above, then we should try more units.
    - If the patterns in the data appear to be **highly abstract**, such as parts of speech/language being components of full sentences, then we should try more layers. This lines up with very deep models being good at things like language translation, the structure and capacity of the model is well suited to the problem of language, with very abstract relationships. This is conceptually similar to how deeper CNN models are able to transform images through multiple layers and extract useful features on which to base a prediction - the deep LSTM models are able to transform temporal data through multiple layers and extract useful features.

The relative balance between the two ways to add capacity can't be too far out of balance, which is a guideline without specifics; we are less likely to see good results with an LSTM layer of 3,000 units and one layer, or 2 units and 75 layers. In general, things that are based on unstructured data such as NLP based applications will tend to be better suited with a deep model, things like a sales forecast can likely be well served by a more shallow model. The training times for very deep models with large timesteps can get very large, as can the memory requirements. There is a rule of thumb that we can use for an estimate for a total number of units, but it is, at best, a very rough guideline:

- Units = samples in training data / ((# input neurons + # output neurons) * alpha)
- Where alpha is a constant normally between 2 and 10, with 5 being a good starting point.

So an algorithmic approach to deciding on a rough starting size for a model would be to calculate the total neurons above, using an alpha that mirrors the complexity, then divide that by one layer for a simple problem, two or more for a complex problem. The farther we get from a simple time-series problem, the less likely this rule of thumb will be precise.

Dropouts in LSTM

Dropout layers are commonly used in LSTM models as a way to prevent overfitting. There are two basic ways that dropout layers can be used in an LSTM model:

- Dropout between LSTM layers.
- Dropout after LSTM layers, before the output layer.

Of the two, placing the dropout between LSTM layers will generally give more variance in the impact of the layer - potentially in both directions. There are also many other varieties of dropout strategies being researched, all seeking to be situationally smarter variations of the standard dropout layer we've applied here. While dropout layers are extremely common to see in LSTM models, they are not certain to improve model performance and their usefullness needs to be tested to be truly evaluated. It may or may not be useful in any given scenario, I am somewhat skeptical of the general usefullness of placing dropouts between the LSTM layers and would tend to lean towards trying the second approach first, where we sit the dropout after the LSTM layers.

Keras Tuner

We can use the Keras Tuner to do some hyperparameter tuning for the best configuration. The same principles apply as with any other model, we need to define a search space, and then run the search. One thing that we can do to make our search space smaller and allow more time for productive search, we can likely create smaller models that visually seem to "follow the trend" if we plot the training data - take that model that looks decent, and search +/- some units, +/- some layers, and try some variations such as dropouts or an additional dense layer.

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir="logs/lstm_kt")
```

```
import keras_tuner as kt

def build_model(hp):
    model = Sequential()
    model.add(LSTM(hp.Int('input_unit',min_value=4,max_value=128,step=4),return_sequences=True, input_shape=(n_lookback, len(columns))))
    for i in range(hp.Int('n_layers', 1, 5)):
        model.add(LSTM(hp.Int(f'lstm_{i}_units',min_value=4,max_value=128,step=4),return_sequences=True))
    model.add(LSTM(6))
    model.add(Dropout(hp.Float('Dropout_rate',min_value=0,max_value=0.5,step=0.1)))
    model.add(Dense(6))
    model.add(Dropout(hp.Float('Dropout_rate',min_value=0,max_value=0.5,step=0.1)))
    model.add(Dense(10))
    model.compile(loss='mean_squared_error', optimizer='adam',metrics=metrics)
    return model
```

```
import time
timstamp = str(time.strftime("%Y%m%d-%H%M%S"))
proj = "lstm_kt" + timstamp
bayesian_opt_tuner = kt.tuners.BayesianOptimization(
    build_model,
    objective='mse',
    max_trials=5,
    executions_per_trial=3,
    directory="logs/lstm_kt",
    project_name=proj,
    overwrite=True)
bayesian_opt_tuner.search_space_summary()
```

```
Search space summary
Default search space size: 4
input_unit (Int)
{'default': None, 'conditions': [], 'min_value': 4, 'max_value': 128, 'step': 4, 'sampling': 'linear'}
n_layers (Int)
{'default': None, 'conditions': [], 'min_value': 1, 'max_value': 5, 'step': 1, 'sampling': 'linear'}
lstm_0_units (Int)
{'default': None, 'conditions': [], 'min_value': 4, 'max_value': 128, 'step': 4, 'sampling': 'linear'}
Dropout_rate (Float)
{'default': 0.0, 'conditions': [], 'min_value': 0.0, 'max_value': 0.5, 'step': 0.1, 'sampling': 'linear'}
```

```
# Change this for a more realistic search, but much longer.
#search_epochs = int(epochs/3)
search_epochs = 3
bayesian_opt_tuner.search(x_tmp, y_tmp,epochs=search_epochs,validation_split=0.2,verbose=1, callbacks=[tensorboard_callback])
```

```
Trial 5 Complete [00h 00m 54s]
mse: 11006.791666666666

Best mse So Far: 10891.7041015625
Total elapsed time: 00h 03m 14s
```

Hyperparameter Tuning Complete - Results

We can take a look at the results and take the best model to go do our predicting...

```
best_model = bayesian_opt_tuner.get_best_models(num_models=1)[0]
best_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 60, 52)            12064

 lstm_1 (LSTM)               (None, 60, 128)           92672

 lstm_2 (LSTM)               (None, 60, 52)            37648

 lstm_3 (LSTM)               (None, 6)                 1416

 dropout (Dropout)           (None, 6)                 0

 dense (Dense)               (None, 6)                 42

 dropout_1 (Dropout)         (None, 6)                 0

 dense_1 (Dense)             (None, 10)                70

=================================================================
Total params: 143,912
Trainable params: 143,912
Non-trainable params: 0
_________________________________________________________________
```

Exercise

Predict the temperature for the next 24 hours, the target column is "T (degC)". Use the past 30 days of data to do so as a starting point, if you run into resource issues, switch that to 15 or 7 and give it a try. If you have time, attempt to make a function that constructs both the X and Y data for you, and can take in a number of days to look back, a number of days to predict forward, and ideally, a list of columns to use as features. As well, test one feature vs a bunch of features, and see if you can get a better result with more features.

```
uri = "https://storage.googleapis.com/tensorflow/tf-keras-datasets/jena_climate_2009_2016.csv.zip"
zip_path = keras.utils.get_file(origin=uri, fname="jena_climate_2009_2016.csv.zip")
zip_file = ZipFile(zip_path)
zip_file.extractall()
csv_path = "jena_climate_2009_2016.csv"

weather_dataframe = pd.read_csv(csv_path)
weather_dataframe.head(10)
```

```
WARNING:tensorflow:Detecting that an object or model or tf.train.Checkpoint is being deleted with unrestored values. See the following logs for the sp
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.1
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.2
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.3
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.4
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.5
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.6
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.7
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.8
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.9
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.10
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.11
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.12
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.13
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.14
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.15
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.16
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.17
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.18
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.19
```

```
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.20
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.21
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.22
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.23
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.24
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.25
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer._variables.26
```

	Date Time	p (mbar)	T (degC)	Tpot (K)	Tdew (degC)	rh (%)	VPmax (mbar)	VPact (mbar)	VPdef (mbar)	sh (g/kg)	H2OC (mmol/mol)	rho (g/m**3)	wv (m/s)	max. wv (m/s)	wd (deg)
0	01.01.2009 00:10:00	996.52	-8.02	265.40	-8.90	93.3	3.33	3.11	0.22	1.94	3.12	1307.75	1.03	1.75	152.3
1	01.01.2009 00:20:00	996.57	-8.41	265.01	-9.28	93.4	3.23	3.02	0.21	1.89	3.03	1309.80	0.72	1.50	136.1
2	01.01.2009 00:30:00	996.53	-8.51	264.91	-9.31	93.9	3.21	3.01	0.20	1.88	3.02	1310.24	0.19	0.63	171.6
3	01.01.2009 00:40:00	996.51	-8.31	265.12	-9.07	94.2	3.26	3.07	0.19	1.92	3.08	1309.19	0.34	0.50	198.0
4	01.01.2009 00:50:00	996.51	-8.27	265.15	-9.04	94.1	3.27	3.08	0.19	1.92	3.09	1309.00	0.32	0.63	214.3
5	01.01.2009 01:00:00	996.50	-8.05	265.38	-8.78	94.4	3.33	3.14	0.19	1.96	3.15	1307.86	0.21	0.63	192.7
6	01.01.2009 01:10:00	996.50	-7.62	265.81	-8.30	94.8	3.44	3.26	0.18	2.04	3.27	1305.68	0.18	0.63	166.5

```
weather_dataframe.tail(10)
```

	Date Time	p (mbar)	T (degC)	Tpot (K)	Tdew (degC)	rh (%)	VPmax (mbar)	VPact (mbar)	VPdef (mbar)	sh (g/kg)	H2OC (mmol/mol)	rho (g/m**3)	wv (m/s)	max. wv (m/s)	wd (deg)
420541	31.12.2016 22:30:00	1000.44	-4.08	269.05	-7.89	74.60	4.51	3.37	1.15	2.10	3.37	1293.55	1.27	2.48	192.1
420542	31.12.2016 22:40:00	1000.45	-4.45	268.68	-7.15	81.30	4.39	3.57	0.82	2.22	3.57	1295.24	0.80	1.44	183.8
420543	31.12.2016 22:50:00	1000.32	-4.09	269.05	-7.23	78.60	4.51	3.54	0.96	2.21	3.54	1293.37	1.25	1.60	199.2
420544	31.12.2016 23:00:00	1000.21	-3.76	269.39	-7.95	72.50	4.62	3.35	1.27	2.09	3.35	1291.71	0.89	1.30	223.7
420545	31.12.2016 23:10:00	1000.11	-3.93	269.23	-8.09	72.60	4.56	3.31	1.25	2.06	3.31	1292.41	0.56	1.00	202.6
420546	31.12.2016 23:20:00	1000.07	-4.05	269.10	-8.13	73.10	4.52	3.30	1.22	2.06	3.30	1292.98	0.67	1.52	240.0
	31.12.2016														

Prepare Data

The data has records every 10 minutes, meaning each day is 6 * 24 = 144 records. We have many features that we could use to generate a model. I think that making a prediction for every 10 minutes is a bit much, so I am going to collapse the 6 rows per hour down to 1. This is an arbitrary choice that does surrender some information, but it will make our data much more manageable. With a massive amount of data that might cause us issues later, this also may help with resource usage...

I'm going to construct my datasets to be in this structure:

- X - batch size, 720 time steps (30 days), 13 features.
    - Each time step is the average of 6 individual time steps.
- Y - batch size, 24 time steps (to predict - 1 day), 1 feature (temperature).

You could structure the dataset differently here, but make sure that what is going in and out makes sense here, and that you can roughly picture what would change if we were to make some changes to our choices:

- What if I wanted to predict the next 48 hours instead of 24?
- What if I wanted to predict the temperature every 30 minutes instead of every 10 minutes?
- What if I wanted to base my predictions on the past 60 days instead of 30?
- What if I wanted to use only wind velocity and barometric pressure as my features, instead of everything?

It isn't really necissary that you do each of these, but you should be able to adapt your solution to accomodate these changes if you needed to, without it being totally confusing. Each change is just a reshaping of the data going into and out of our model. The multi-dimensional shape of the data isn't the most intuitive thing in the world, so it it totally fine if your mind doesn't immediately manipulate the data in your head. If you diagram out the dimensions, it should be doable though.

```
PREDICTION_LENGTH = 24
LOOKBACK = 24*7
LEN = 6
EXERCISE_EPOCHS = 10
EXERCISE_BATCH = 8
```

```
def collapseHours(dataF, date_col="Date Time", spacing=LEN):
    i = 0
    processed = []
    while i < len(dataF):
        processed.append(dataF[i:i+spacing].mean())
        i += spacing
    tmp_df = pd.DataFrame(processed)
    tmp_df = pd.DataFrame(processed)
    tmp_df.columns = dataF.columns[1:]
    return tmp_df


import warnings
with warnings.catch_warnings():
    warnings.simplefilter(action='ignore', category=FutureWarning)

    collapsed = collapseHours(weather_dataframe, date_col="Date Time", spacing=LEN)
    collapsed.head()
```

Create a function to construct datsets for us.

```
def prepare_weather_data(dataF, cols, y_col="T (degC)", lookback=720, lookahead=144):
    # Select the desired columns
    scaler = MinMaxScaler()
    data = dataF[cols]

    # Prepare the data for an LSTM model
    x = []
    y = []
    for i in range(lookback, len(data)-lookahead):
        x.append(data.iloc[i-lookback:i].values)
        y.append(data[y_col][i:i+lookahead].values)
    x = np.array(x)
    y = np.array(y)

    # Reshape the data for the LSTM model
    x = np.reshape(x, (x.shape[0], x.shape[1], x.shape[2]))
    y = np.reshape(y, (y.shape[0], y.shape[1], 1))

    return x, y
```

Only One Feature

Try with a strightforward model that only uses one feature, and see how well it does.

```
X_one, y_one = prepare_weather_data(collapsed, cols=["T (degC)", "wd (deg)"], y_col="T (degC)", lookback=LOOKBACK, lookahead=PREDICTION_LENGTH)
print(X_one.shape, y_one.shape)
```

```
    (69900, 168, 2) (69900, 24, 1)
```

```
# fit the model
model = Sequential()
model.add(LSTM(units=layer_size, return_sequences=True, input_shape=(LOOKBACK, X_one.shape[2])))
model.add(LSTM(units=layer_size))
model.add(Dropout(0.2))
model.add(Dense(PREDICTION_LENGTH))

model.compile(loss='mean_squared_error', optimizer='adam', metrics=metrics)
history = model.fit(X_one, y_one, epochs=EXERCISE_EPOCHS, batch_size=EXERCISE_BATCH, verbose=1)
```

```
    Epoch 1/10
    8738/8738 [==============================] - 107s 12ms/step - loss: 43.4530 - mse: 43.4530
    Epoch 2/10
    8738/8738 [==============================] - 103s 12ms/step - loss: 20.2881 - mse: 20.2881
    Epoch 3/10
    8738/8738 [==============================] - 103s 12ms/step - loss: 18.9020 - mse: 18.9020
    Epoch 4/10
    8738/8738 [==============================] - 102s 12ms/step - loss: 18.0966 - mse: 18.0966
    Epoch 5/10
    8738/8738 [==============================] - 103s 12ms/step - loss: 17.0168 - mse: 17.0168
    Epoch 6/10
    8738/8738 [==============================] - 103s 12ms/step - loss: 15.5268 - mse: 15.5268
    Epoch 7/10
    8738/8738 [==============================] - 102s 12ms/step - loss: 14.8341 - mse: 14.8341
    Epoch 8/10
    8738/8738 [==============================] - 103s 12ms/step - loss: 14.5408 - mse: 14.5408
    Epoch 9/10
    8738/8738 [==============================] - 103s 12ms/step - loss: 14.4028 - mse: 14.4028
    Epoch 10/10
    8738/8738 [==============================] - 103s 12ms/step - loss: 14.2463 - mse: 14.2463
```

Train Multi-Feature Model

Now that our data is prepared, we can model and predict. Note the input and output shapes in our model:

- Input shape - the size of one batch: (# of time steps to look back) * (# of features)
- Output shape - the size of the output layer: (# of time steps to predict) * (# of targets)

No matter how the data was processed above, it should fit this format for the LSTM model.

Note: if the lookback and the prediction are large numbers, we are possibly going to run into performance issues in terms of either processing time or memory usage. Try turning down the lookback to 7 days instead of 30 to reduce memory usage.

```
weather_columns = collapsed.columns
print("Columns Used:", weather_columns)
X_weather, y_weather = prepare_weather_data(collapsed, cols=weather_columns, y_col="T (degC)", lookback=LOOKBACK, lookahead=PREDICTION_LENGTH)
print(X_weather.shape, y_weather.shape)
```

```
Columns Used: Index(['p (mbar)', 'T (degC)', 'Tpot (K)', 'Tdew (degC)', 'rh (%)',
       'VPmax (mbar)', 'VPact (mbar)', 'VPdef (mbar)', 'sh (g/kg)',
       'H2OC (mmol/mol)', 'rho (g/m**3)', 'wv (m/s)', 'max. wv (m/s)',
       'wd (deg)'],
      dtype='object')
(69900, 168, 14) (69900, 24, 1)
```

```
# fit the model
model = Sequential()
model.add(LSTM(units=layer_size, return_sequences=True, input_shape=(LOOKBACK, len(weather_columns))))
model.add(LSTM(units=layer_size))
model.add(Dropout(0.2))
model.add(Dense(PREDICTION_LENGTH))

model.compile(loss='mean_squared_error', optimizer='adam', metrics=metrics)
history = model.fit(X_weather, y_weather, epochs=EXERCISE_EPOCHS, batch_size=EXERCISE_BATCH, verbose=1)
```

```
Epoch 1/10
8738/8738 [==============================] - 110s 12ms/step - loss: 79.6410 - mse: 79.6410
Epoch 2/10
8738/8738 [==============================] - 105s 12ms/step - loss: 72.9247 - mse: 72.9247
Epoch 3/10
8738/8738 [==============================] - 105s 12ms/step - loss: 72.1234 - mse: 72.1234
Epoch 4/10
8738/8738 [==============================] - 104s 12ms/step - loss: 71.5346 - mse: 71.5346
Epoch 5/10
8738/8738 [==============================] - 103s 12ms/step - loss: 71.2663 - mse: 71.2663
Epoch 6/10
8738/8738 [==============================] - 103s 12ms/step - loss: 70.8618 - mse: 70.8618
Epoch 7/10
8738/8738 [==============================] - 103s 12ms/step - loss: 70.5501 - mse: 70.5501
Epoch 8/10
8738/8738 [==============================] - 103s 12ms/step - loss: 70.4479 - mse: 70.4479
Epoch 9/10
8738/8738 [==============================] - 104s 12ms/step - loss: 70.2477 - mse: 70.2477
Epoch 10/10
8738/8738 [==============================] - 104s 12ms/step - loss: 70.1866 - mse: 70.1866
```