LSTM Model for Wave Forecasting: A Comprehensive Guide

Leveraging the power of machine learning for forecasting has become increasingly popular, and the field of weather prediction is no exception. This guide will walk you through a practical application of a Long Short-Term Memory (LSTM) model, specifically tailored for wave prediction. The tutorial uses buoy data to predict wave conditions, demonstrating the model's potential for real-world applications.

As part of the broader field of Recurrent Neural Networks (RNN), LSTM models have proven to be highly effective in handling time series data. By taking into account long-term dependencies in the data, LSTM models can capture patterns that traditional RNNs might miss, making them an optimal choice for complex forecasting tasks like weather and wave prediction.

This tutorial delves into key aspects such as data preprocessing, designing an LSTM model, training the model, and model evaluation. The guide is particularly aimed at those interested in climate and weather forecasting, machine learning enthusiasts, data scientists, and anyone keen to understand the application of LSTM models in time series prediction.

Our goal is to provide an accessible, easy-to-follow guide on LSTM models for wave prediction, highlighting best practices, potential pitfalls, and practical tips. So, whether you're new to machine learning or an experienced data scientist looking for new insights, you'll find valuable, practical, and engaging content in this guide.

Don't miss out on this opportunity to expand your knowledge and skill set in time series forecasting using LSTM models. Dive into the tutorial and get a head start on leveraging LSTM for wave prediction today!

Keywords: *LSTM, Wave Prediction, Time Series Forecasting, Machine Learning, Weather Forecasting, Data Preprocessing, Model Training, Model Evaluation*

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
import torch
from torch import nn, optim
from torch.utils.data import Dataset, DataLoader, TensorDataset
import requests as rq
import io
import numpy as np
```

```
#taking full advantage of gpu if available
device = 'cuda' if torch.cuda.is_available() else 'cpu'

print(device)

JEFFERYS_LEDGE = 'https://www.ndbc.noaa.gov/data/realtime2/44098.spec'
SHELF = 'https://www.ndbc.noaa.gov/data/realtime2/44011.spec'
```

```
    cuda
```

Pulling Data From NDBC

```
j = rq.get(JEFFERYS_LEDGE).content
s = rq.get(SHELF).content
```

```
column_names_j = ["Year", "Month", "Day", "Hour", "Minute", "WVHT", "SwH", "SwP", "WWH", "WWP", "SwD",
                "WWD", "STEPPINESS", "APD", "MWD"]

column_names_s = ["Year_s", "Month_s", "Day_s", "Hour_s", "Minute_s", "WVHT_s", "SwH_s", "SwP_s", "WWH_s"
                "WWD_s", "STEPPINESS_s", "APD_s", "MWD_s"]

jeff_data_raw = pd.read_csv(io.StringIO(j.decode('utf-8')), sep="\s+", skiprows=2, names=column_names_j)
shelf_data_raw = pd.read_csv(io.StringIO(s.decode('utf-8')), sep="\s+", skiprows=2, names=column_names_s)

display(jeff_data_raw.head())
display(shelf_data_raw.head())
```

	Year	Month	Day	Hour	Minute	WVHT	SwH	SwP	WWH	WWP	SwD	WWD	STEPPINESS	APD	MWD
0	2023	7	30	17	56	0.8	0.1	10.5	0.8	8.3	SE	SE	NaN	4.2	145
1	2023	7	30	17	26	0.8	0.1	10.5	0.8	4.0	SE	NNE	VERY_STEEP	4.1	23
2	2023	7	30	16	56	0.9	0.1	10.5	0.9	4.2	SE	NNE	VERY_STEEP	4.0	22
3	2023	7	30	16	26	0.9	0.1	10.5	0.9	4.5	SE	NNE	VERY_STEEP	4.0	31
4	2023	7	30	15	56	0.9	0.1	10.5	0.9	5.0	SE	NE	VERY_STEEP	4.0	44

	Year_s	Month_s	Day_s	Hour_s	Minute_s	WVHT_s	SwH_s	SwP_s	WWH_s	WWP_s	SwD_s	WWD_s	STEPPINES
0	2023	7	30	17	40	2.9	2.3	9.1	1.7	6.7	SW	WNW	AVER
1	2023	7	30	17	10	2.7	2.3	9.1	1.5	4.2	SW	NNW	AVER
2	2023	7	30	16	40	2.6	2.0	8.3	1.7	6.2	WSW	NNW	ST
3	2023	7	30	16	10	2.5	1.8	8.3	1.7	5.6	SW	NW	ST
4	2023	7	30	15	40	2.4	1.9	9.1	1.4	5.6	SW	WSW	AVER

```
#Now we are going to clean the data
print(jeff_data_raw.isnull().sum())
print(jeff_data_raw.shape)
print(shelf_data_raw.isnull().sum())
print(shelf_data_raw.shape)
```

```
    Year             0
    Month            0
    Day              0
    Hour             0
    Minute           0
    WVHT             0
    SwH              0
    SwP              0
    WWH              0
    WWP              0
    SwD              0
    WWD              0
    STEPPINESS    1320
    APD              0
    MWD              0
    dtype: int64
    (2194, 15)
    Year_s             0
    Month_s            0
    Day_s              0
    Hour_s             0
```

```
Minute_s          0
WVHT_s            0
SwH_s             0
SwP_s             0
WWH_s             0
WWP_s             0
SwD_s             0
WWD_s             0
STEPPINESS_s    366
APD_s             0
MWD_s             0
dtype: int64
(2180, 15)
```

Preprocessing and Encoding Data

This section of the code focuses on cleaning and preparing the data for further analysis and modelling.

(The most useful concept is how to encode the data)

We are dealing with two datasets here - `jeff_data_raw` and `shelf_data_raw`.

The two key steps are as follows:

1. **Dropping Irrelevant Features:** The 'STEPPINESS' feature is dropped from both the `jeff_data_raw` and `shelf_data_raw` datasets. The `inplace=True` parameter is used to make sure that the changes are applied directly to the original dataframe without the need to assign the result to a new dataframe. It's presumed that the 'STEPPINESS' feature is not required for our analysis or modelling, which is why it's being removed.

```
jeff_data_raw.drop('STEPPINESS',axis=1,inplace=True)
shelf_data_raw.drop('STEPPINESS_s',axis=1,inplace=True)
```

2. **Encoding Categorical Variables:** Machine Learning algorithms require numerical inputs. However, often in our dataset, we have categorical variables. In our case, the 'SwD' and 'WWD' features in both datasets are categorical. We use encoding to convert these categorical variables into numerical format.

Here, we're using the `astype('category').cat.codes` method, which first converts the column to a 'category' data type and then uses `.cat.codes` to convert the category values into numerical codes. This is a form of integer encoding where each unique category value is assigned an integer.

```
jeff_data_raw['SwD'] = jeff_data_raw['SwD'].astype('category').cat.codes
jeff_data_raw['WWD'] = jeff_data_raw['WWD'].astype('category').cat.codes
shelf_data_raw['SwD_s'] = shelf_data_raw['SwD_s'].astype('category').cat.codes
shelf_data_raw['WWD_s'] = shelf_data_raw['WWD_s'].astype('category').cat.codes
```

Finally, we use `display()` to print out the first few rows of each DataFrame. This helps us verify that our data preprocessing steps have been applied correctly.

```
display(jeff_data_raw.head())
display(shelf_data_raw.head())
```

This set of preprocessing steps ensures that our data is clean, properly formatted, and ready to be used in the subsequent stages of our data analysis or modelling process.

```
#lets make sure everything is numerical, so we will encode SwD and WWD
#We will drop Steppiness for now

jeff_data_raw.drop('STEPPINESS',axis=1,inplace=True)
shelf_data_raw.drop('STEPPINESS_s',axis=1,inplace=True)

jeff_data_raw['SwD'] = jeff_data_raw['SwD'].astype('category').cat.codes
jeff_data_raw['WWD'] = jeff_data_raw['WWD'].astype('category').cat.codes
shelf_data_raw['SwD_s'] = shelf_data_raw['SwD_s'].astype('category').cat.codes
shelf_data_raw['WWD_s'] = shelf_data_raw['WWD_s'].astype('category').cat.codes

display(jeff_data_raw.head())
display(shelf_data_raw.head())
```

	Year	Month	Day	Hour	Minute	WVHT	SwH	SwP	WWH	WWP	SwD	WWD	APD	MWD
0	2023	7	30	17	56	0.8	0.1	10.5	0.8	8.3	4	7	4.2	145
1	2023	7	30	17	26	0.8	0.1	10.5	0.8	4.0	4	5	4.1	23
2	2023	7	30	16	56	0.9	0.1	10.5	0.9	4.2	4	5	4.0	22
3	2023	7	30	16	26	0.9	0.1	10.5	0.9	4.5	4	5	4.0	31
4	2023	7	30	15	56	0.9	0.1	10.5	0.9	5.0	4	4	4.0	44

	Year_s	Month_s	Day_s	Hour_s	Minute_s	WVHT_s	SwH_s	SwP_s	WWH_s	WWP_s	SwD_s	WWD_s	APD_s	MW
0	2023	7	30	17	40	2.9	2.3	9.1	1.7	6.7	12	15	6.5	
1	2023	7	30	17	10	2.7	2.3	9.1	1.5	4.2	12	7	6.2	
2	2023	7	30	16	40	2.6	2.0	8.3	1.7	6.2	14	7	6.0	
3	2023	7	30	16	10	2.5	1.8	8.3	1.7	5.6	12	8	6.0	
4	2023	7	30	15	40	2.4	1.9	9.1	1.4	5.6	12	16	6.1	

```
#we will concat the two datasets into one, this will produce nan values that must be removed
combined_data = pd.concat([jeff_data_raw, shelf_data_raw], axis=1, join='outer')
combined_data = combined_data.dropna()
types = combined_data.dtypes
print(combined_data.shape)
print(types)
print('---------------------------')
print(combined_data.isnull().sum())
```

```
Month          int64
Day            int64
Hour           int64
Minute         int64
WVHT         float64
SwH          float64
SwP          float64
WWH          float64
WWP          float64
SwD             int8
WWD             int8
APD          float64
MWD            int64
Year_s       float64
```

```
Hour_s      float64
Minute_s    float64
WVHT_s      float64
SwH_s       float64
SwP_s        object
WWH_s       float64
WWP_s       float64
SwD_s       float64
WWD_s       float64
APD_s       float64
MWD_s       float64
dtype: object
---------------------------
Year        0
Month       0
Day         0
Hour        0
Minute      0
WVHT        0
SwH         0
SwP         0
WWH         0
WWP         0
SwD         0
WWD         0
APD         0
MWD         0
Year_s      0
Month_s     0
Day_s       0
Hour_s      0
Minute_s    0
WVHT_s      0
SwH_s       0
SwP_s       0
WWH_s       0
WWP_s       0
SwD_s       0
WWD_s       0
APD_s       0
MWD_s       0
dtype: int64
```

Creating Sequences from Time Series Data

This section demonstrates an efficient method to transform time series data into sequences of input data and corresponding target data, which can be utilized to train our model.

This approach uses the concept of a sliding or moving window across our data set. The size of this window is determined by a parameter we set called `sequence_length`.

To better visualize, imagine that our time series data is a long road and our `sequence_length` is a window of certain length. As we traverse down the road, this window moves or 'slides' along with us, providing a snapshot of data in that specific segment.

Here's how it's implemented in the code:

1. A `for` loop is used to iterate over each row in our normalized data.
2. In each iteration, the code chops out a segment of data, starting from the current index `i` up to `i + sequence_length`. This segment represents a sequence of input data.
3. Similarly, a segment of target data is also extracted. This segment starts from `i + sequence_length` and ends at `i + sequence_length + forecast_length`. The target data corresponds to the values we want our model to predict.

4. It's important to note that the loop stops at `len(data) - (sequence_length + forecast_length)`. This ensures that we don't exceed the range of our data when extracting the target sequence. In other words, this prevents us from going out of bounds.

As a result of this process, we end up with a set of sequences of input data and corresponding target data that can be used to train our model. Each input sequence is matched with a target sequence that is 'shifted' slightly into the future relative to the input, allowing the model to learn to predict future data based on past data.

This method is quite versatile and powerful in converting time series data into a format suitable for training many types of sequence prediction models, such as LSTM or GRU models in machine learning.

```
# Select the column to predict
# Select the input features and target features
input_data = combined_data[['SwH', 'SwD', 'SwD_s', 'SwH_s']].values
target_data = combined_data[['SwH']].values


# Normalize the data
scaler = MinMaxScaler(feature_range=(-1, 1))
input_normalized = scaler.fit_transform(input_data)
target_normalized = scaler.fit_transform(target_data)

# Create sequences
sequence_length = 88
forecast_length = 12
input_sequences = []
target_sequences = []
actual_prediction_sequences = []
for i in range(len(input_normalized) - (sequence_length+forecast_length)):
    input_sequences.append(input_normalized[i:i+sequence_length])
    target_sequences.append(target_normalized[i+sequence_length:i+sequence_length + forecast_length])


# Convert to numpy arrays
input_sequences = np.array(input_sequences)
target_sequences = np.array(target_sequences)
act_pred = np.array(actual_prediction_sequences)
# Split into X and y
X = input_sequences[:, :-1]
y = target_sequences[:, -1]

print(X.shape)
# Reshape X to (samples, time steps, features)


X.shape, y.shape
```

```
    (2080, 87, 4)
    ((2080, 87, 4), (2080, 1))
```

Splitting the data into training and testing so we can track progress

```
train_size = int(0.5 * len(X))

X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

X_train.shape, X_test.shape
```

```
X_train = torch.tensor(X_train).float().to(device)
y_train = torch.tensor(y_train).float().to(device)
X_test = torch.tensor(X_test).float().to(device)
y_test = torch.tensor(y_test).float().to(device)
```

Defining and Training The Model

The model takes in four datapoints and outputs one. The model performs best at about where it is. The problem is very complicated but if I increase the complexity of the model it memorizes the data extremely quickly and overfits like crazy

```
#define model
class LSTM(nn.Module):
    def __init__(self, input_size=1, hidden_layer_size=600, output_size=1):
        super().__init__()
        self.hidden_layer_size = hidden_layer_size
        self.lstm = nn.LSTM(input_size, hidden_layer_size)
        self.linear = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.linear = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.linear = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.linear = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.linear = nn.Linear(hidden_layer_size, output_size)
        self.hidden_cell = (torch.zeros(1,1,self.hidden_layer_size),
                            torch.zeros(1,1,self.hidden_layer_size))

    def forward(self, input_seq):
        lstm_out, self.hidden_cell = self.lstm(input_seq.view(len(input_seq) ,1, -1), self.hidden_cell)
        predictions = self.linear(lstm_out.view(len(input_seq), -1))
        return predictions[-1]

# Instantiate the model
model = LSTM(input_size=4,output_size=1).to(device)
model
```

```
    LSTM(
      (lstm): LSTM(4, 600)
      (linear): Linear(in_features=600, out_features=1, bias=True)
    )
```

```
# Define loss function and optimizer - .05 works really well wow
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.00005)
loss_function, optimizer
```

```
    (MSELoss(),
     Adam (
     Parameter Group 0
         amsgrad: False
         betas: (0.9, 0.999)
         capturable: False
         differentiable: False
         eps: 1e-08
         foreach: None
         fused: None
         lr: 5e-05
         maximize: False
         weight_decay: 0
     ))
```

```
epochs = 2


for i in range(epochs):
    model.train()
    for seq, labels, valseq, val_label in zip(X_train, y_train, X_test, y_test):
        optimizer.zero_grad()
        model.hidden_cell = (torch.zeros(1, 1, model.hidden_layer_size).to(device),
                        torch.zeros(1, 1, model.hidden_layer_size).to(device))

        y_pred = model(seq)
        single_loss = loss_function(y_pred, labels)
        single_loss.backward()
        optimizer.step()

    model.eval()
    with torch.no_grad():
      y_val = model(valseq)
      val_loss = loss_function(y_val,val_label)

    if i%1 == 0:
        print(f'epoch: {i:3} loss: {single_loss.item():10.8f} val_loss: {val_loss.item():10.8f}')

print(f'epoch: {i:3} loss: {single_loss.item():10.10f}')
print(y_pred,val_label)
```

```
epoch:   0 loss: 0.00000666 val_loss: 0.08715359
epoch:   1 loss: 0.00000056 val_loss: 0.08477692
epoch:   1 loss: 0.0000005593
tensor([-0.6674], device='cuda:0', grad_fn=<SelectBackward0>) tensor([-1.], device='cuda:0')
```

Model Evaluation and Future Plans

The LSTM model that I've developed shows promising performance on the test dataset. It has successfully managed to capture the temporal relationships within the time series data. However, there's an important caveat to consider. From the training logs, it seems the model might be memorizing the training data, indicating a potential overfitting issue.

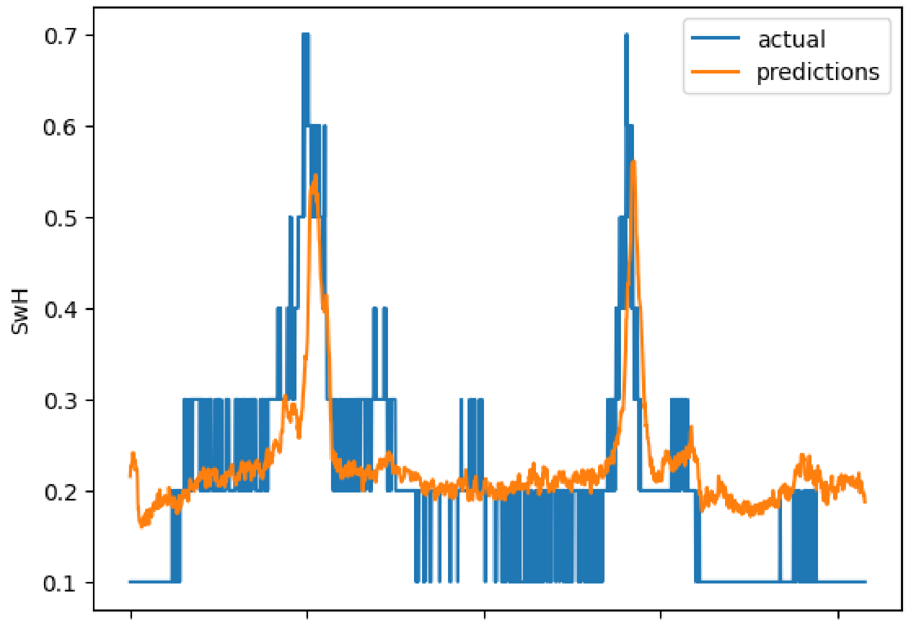
Overfitting can result in a model that performs exceptionally well on training data but fails to generalize to unseen data. This suggests that, despite the current impressive results, the model's performance might not be as reliable when predicting future data from the buoys. It's crucial to bear this in mind when considering the model's real-world applicability. Addressing this issue may require additional model tuning or the application of regularization techniques to prevent overfitting.

Nevertheless, I've saved the model for future performance evaluation, particularly during the fall season. This period, characterized by the onset of the hurricane season, will introduce new and complex weather patterns. How the model performs in this context will be a good test of its generalizability and adaptability.

I trust that this project will serve as a helpful resource for individuals interested in time series forecasting with LSTM models, especially in the context of climate, weather, or wave prediction. The process of preparing the data, in particular, offers valuable insights into dealing with time series data, transforming it from raw data into a suitable format for training an LSTM model.

As part of my online portfolio and as a contribution to the broader community, I aim to share valuable insights and practical knowledge on implementing LSTM models for time series prediction. I believe this model, with continuous refinement and testing, can be an efficient tool in weather and wave prediction, and I look forward to sharing future advancements and improvements.

```
#Plotting predictions vs actual
import matplotlib.pyplot as plt

X_tensor = torch.tensor(X).float().to(device)
y_tensor = torch.tensor(y).float().to(device)

output = []
outputReal = []

print(X_tensor.shape)
y_tensor.shape

model.eval()
with torch.no_grad():
  for seq,outputV in zip(X_tensor,y_tensor):
    o = model(seq).to('cpu')
    o = o.detach().numpy()  # Convert the output tensor to a numpy array
    o = o.reshape(-1, 1)  # Reshape the array to 2D if it's not already 2D
    o = scaler.inverse_transform(o)  # Apply inverse_transform
    o = o[0][0].tolist()
    output.append(o)
    outputV = outputV.to('cpu').detach().numpy()
    outputV = outputV.reshape(-1,1)
    outputV = scaler.inverse_transform(outputV)
    outputReal.append(outputV[0][0].tolist())



# Plot the training and validation losses over time
plt.plot(outputReal, label='actual')
plt.plot(output, label='predictions')
plt.xlabel('Step')
plt.ylabel('SwH')
plt.legend()
plt.show()

torch.save(model.state_dict(), './wave_model')
torch.Size([2080, 87, 4])
```



✓ 3s completed at 1:07 PM