Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра математики факультету інформатики

ПРОГНОЗУВАННЯ ЧАСОВИХ РЯДІВ МЕТОДОМ LSTM (TIME-SERIES FORECASTING USING LSTM)

Текстова частина до курсової роботи
за спеціальністю „Прикладна математика”

Керівник курсової роботи

к.т.н., доц. Щестюк Н. Ю.

(прізвище та ініціали)

_____

(підпис)

“11” травня 2021 р.

Виконав студент

Поляков М. Х.

(прізвище та ініціали)

“11” травня 2021 р.

Київ 2021

Table of contents

Introduction

Time-series forecasting is a complex problem. Forecasting includes using models to fit historical information and using it to predict future data points. There are multiple methods of forecasting time-series data, including autoregressive models, like ARIMA, using gradient boosting like XGBoost, and more recently, using neural networks. The different techniques might be better for different types of data, and most of the time, only comparison between models can show the best model. We would try creating an LSTM based neural network with a multi-step multiple output strategy in this work. The completed model is tested on two different datasets in nature, power usage, and stock market prices. Overall the work consists of five blocks:

- Time-series forecasting strategies
- RNN architecture description
- LSTM architecture description
- LSTM multi-step forecasting model for power usage
- LSTM multi-step forecasting model for stock prices prediction

1. Time-series forecasting strategies

Time-series predictions are an important area of machine learning with the goal of predicting the future. Prediction of future points plays a vital role in managing the decision-making of the same areas, like demand forecasting. Data that is needed for time-series predictions are classified into two types: one is time-series data, and another one is data with time points. Time-series data can be described mathematically as (Eq. 1):

$$X = (x_t; t = 1, \ldots, N) \tag{1}$$

Where *X* is the time-series, *t* is time over *N* points during that time. Estimation at a specific time point is x_t for time point *t* (Masum et al., 2018).

1.1 One-step

Choice of forecasting strategy depends on the basis of the application domain and the number of received data. One-step predictions are relevant where short-term predictions are needed. For example, durations of several seconds, or minutes are short-term. For such a situation, calculating a one-step-ahead prediction is useful. One-step forecast *(t+1)* is produced by passing the current and prior points *(t, t−1, ..., t−n)* to a chosen model (Eq. 2) (Figure 1):

$$F(t+1) = M(o(t), o(t-1), o(t-2), \ldots, o(t-n)) \tag{2}$$

where *F(t + 1)* is the prediction for time *(t + 1)*, *M* is the model, and *o(t)* is a value at time *t* (Masum et al., 2018).

1.2 Multi-step

Multi-step predictions are useful where the area of application needs long-term forecasting. There are multiple strategies for multi-step forecasting. Direct strategy develops *N* separate forecasting rules to predict *N* steps. For example, to determine the following two periods of any situation using a multi-step approach, the first forecast point *F(t + 1)* needs to be found through a model. The other model would be used to predict the second observation *F(t + 2)*. Nevertheless, the second observation is not dependent on the first observation estimate (Eq. 3-4) (Masum et al., 2018):

$$F(t+1) = M_1(o(t), o(t-1), o(t-2), \ldots, o(t-n)) \tag{3}$$

$$F(t+2) = M_2(o(t), o(t-1), o(t-2), \ldots, o(t-n)) \tag{4}$$

The multi-step approach can be represented as:

$$y_{t+h} = f_h\left(y_t, \ldots, y_{t-n+1}\right) \tag{5}$$

where *h* is the number of observations to project into the future, *n* is the order of the model, f_h is any some trainer. The direct approach does not consist of any collected errors because it does not use any predicted value as data. Nevertheless, it does not warrant any connection between prediction points as all models are trained independently. Another strategy is recursive when the prediction is a one-step forecast, and then it is appended to history to make the next prediction (Figure 1). In this work, we will be using a strategy in which one model outputs all forecasts at once. It is called a multiple output strategy (Brownlee, 2017).

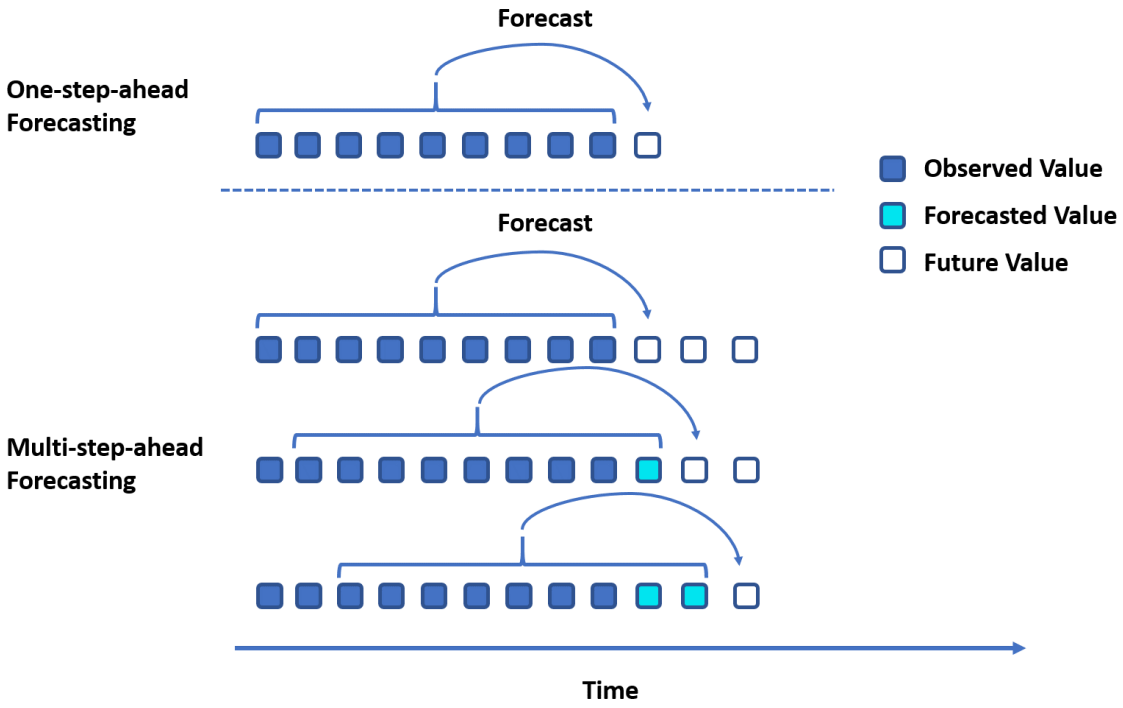

Figure 1. One-step-ahead and recursive multi-step-ahead forecasting (Li, 2019)

2. RNN architecture description

Traditional neural networks are not suited for time series forecasting because they consider all inputs and outputs independently. It would not be practical for time-series forecasting to use such a network since we predict based on historical data. Instead, we could use a Recurrent Neural Network (RNN), which considers the dependencies for past data points and is thus more effective for time-series predictions. Figure 2 shows a part of the neural network, A, which takes some input x_t and outputs a value h_t. A loop allows data to be moved from one step of the network to the following using a hidden state (Olah, 2015). However, RNN seems

to have problems learning long-term history, as explored by Bengio et al. in 1994. In short, the problem arises because of the vanishing gradient that is also happening in feed-forward traditional neural networks. The last layers of the neural network in backpropagation do not change the gradient much. In RNN, this is happening to the data in a time-series sequence since the first layers in the loop do not learn much (Figure 3). The problem can be solved using more advanced iterations of RNN such as LTSM (Long Short Term Memory) neural networks.

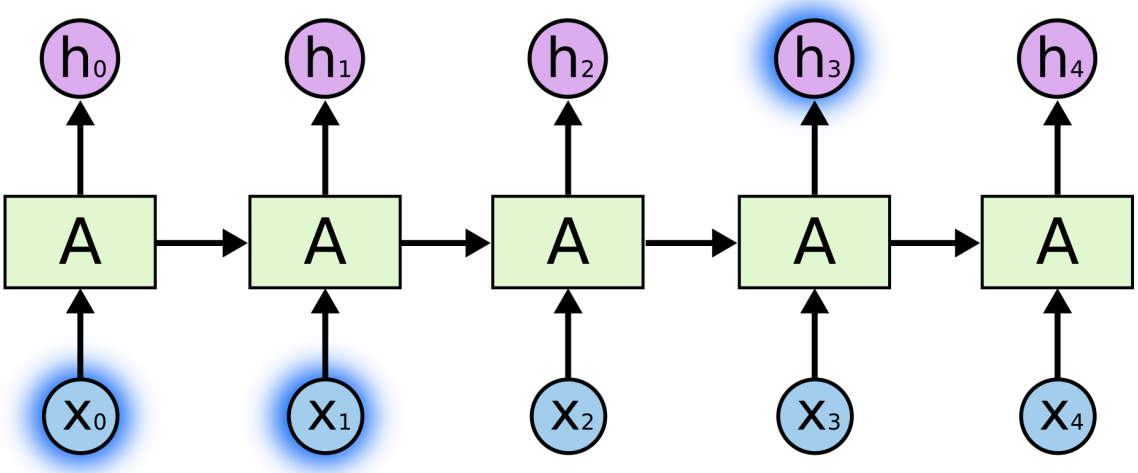

Figure 2. An unwrapped RNN (Olah, 2015)

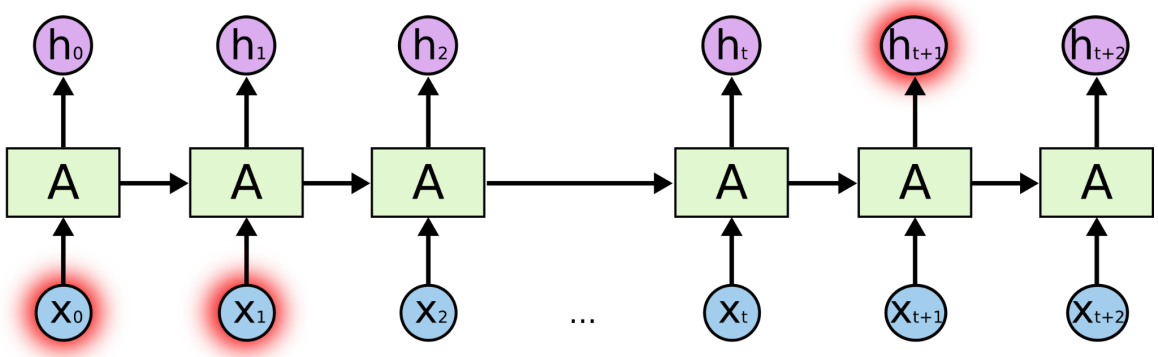

Figure 3. Vanishing gradient problem for RNN (Olah, 2015)

3. LSTM architecture description

The Long Short Term Memory unit (LSTM) was designed to overcome the weaknesses of a Recurrent neural network (RNN), such as the vanishing gradients issues. The RNN consists of a series of recurring modules of the neural network, where each module consists of a structure that includes a sole tanh layer. Tanh is used to squish values between *-1* and *1*. We save inputs in hidden state $h_{t,}$ which also has data from previous hidden state h_{t-1}. It is pretty simple. LSTM has the same series of recurring modules as an RNN, except the LSTM module structure is more complicated. Each module contains four layers rather than a sole layer as for an RNN, and it helps to solve RNN's vanishing gradient problem. Figure 4 shows the difference.

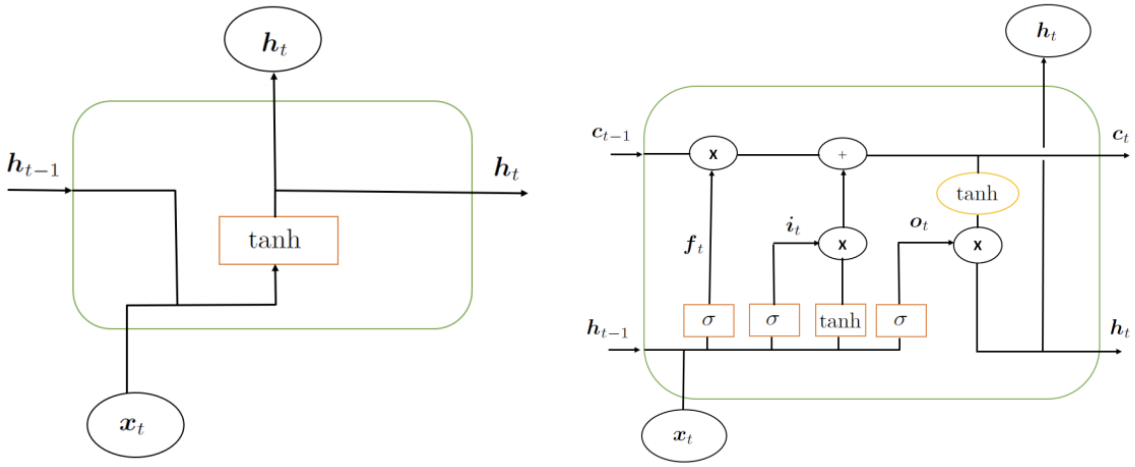

Figure 4. RNN (on the left) and LSTM (on the right) module (Masum et al., 2018)

Three gates manage data that will be added or removed to the cell state. An LSTM network computes a mapping from an input vector $x = (x_1, ..., x_t)$ to an output vector $y = (y_1, ..., y_t)$, where initially, the input gate activation vector i_t and the candidate values of the memory cell state $\dot{C}_t$ are calculated with (Eq. 6-7):

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \tag{6}$$

$$\dot{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{7}$$

where σ is the logistic sigmoid function, W is the respective weights, b are respective biases in Eq. 6-11. [ht-1 + xt] refers to concatenation of previous hidden state and input vectors. Ċt is a vector of new candidate values produced by each tanh layer to be added to the state. The forget gate activation vector ft is then determined using (Eq. 8):

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{8}$$

The forget gate serves to remove data from the cell state, and it also assists with resetting the memory cells. The estimated values of i_t, $\dot{C}_t$ and f_t are then used to determine the new state of the memory cell C_t (Eq. 9):

$$C_t = f_t * C_{t-1} + i_t * \dot{C}_t \tag{9}$$

The state is like a conveyor belt that goes through the whole chain. The cell state vector C_t is then used to calculate the output gate activation vector o_t, which can then be used to determine the output of the LSTM (Eq. 10-11):

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right) \tag{10}$$

$$h_t = o_t * \tanh\left(C_t\right) \tag{11}$$

4. LSTM multi-step forecasting model for power usage

4.1 Dataset

In this section, we will be working with PJM's Hourly Energy Consumption data (Mulla, 2018), a univariate time-series dataset of 10 years of hourly measurements collected from different US regions. We will be using the PJM East region data (Tennessee, Virginia, West Virginia), which has the hourly electricity consumption data from 2001 to 2018 (Figure 5).

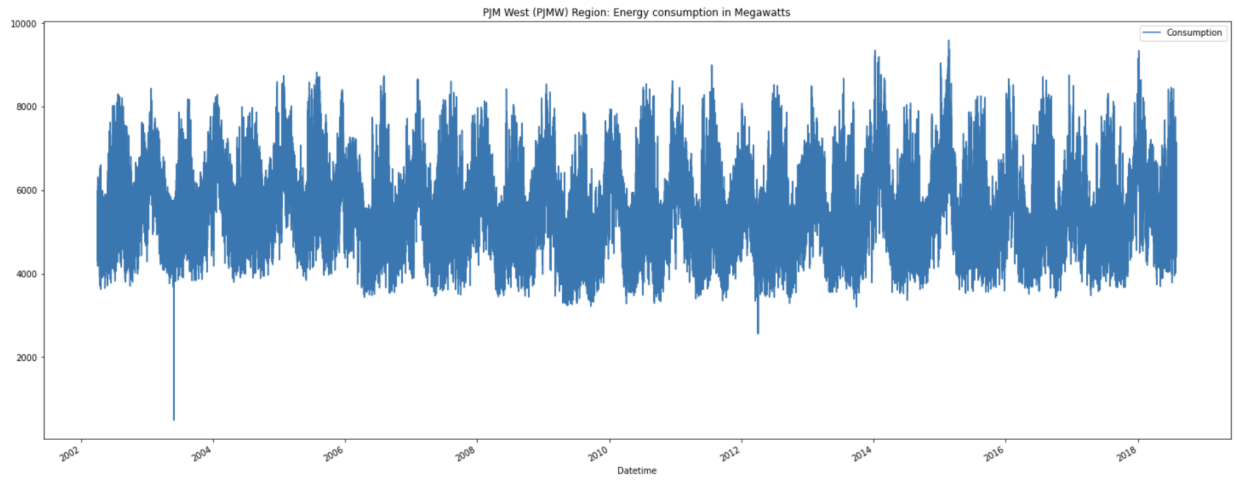

Figure 5. PJM East region electricity consumption dataset

4.2 Stacked LSTM Sequence To Sequence Autoencoder

An LSTM model is trained to map an input sequence to an output sequence. The seq2seq model contains two LSTMs, which are an encoder and a decoder. The encoder part changes the given input sequence to a context vector, which summarizes the input sequence. The context vector is given as input to the decoder, and the final encoder state is an original decoder state to forecast the output

sequence. We can use the sequence to sequence learning for time-series multi-step forecasting by adding two layers, a repeat vector layer and time distributed dense layer. A repeat vector layer is used to repeat the context vector we get from the encoder to pass it as an input to the decoder. We will repeat it for n-steps (*n* is the number of future steps we want to forecast). The output obtained from the decoder for each time step is incorporated. The time allocated densely will apply a fully connected dense layer on each time step and divides the output for each timestep. The time distributed densely is a wrapper that allows using a layer to every temporal part of an input. Stacking extra layers on the encoder part and the decoder part of the sequence model may improve our model's ability to learn more complex pictures of our time-series data in hidden layers by taking data at various levels (Figure 6) (Babu, 2020).

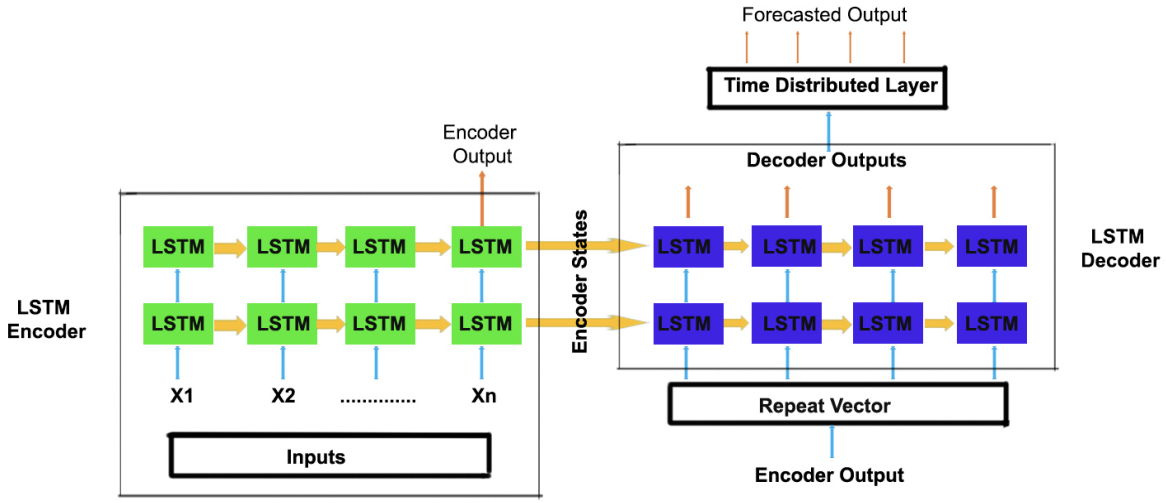

Figure 6. Stacked LSTM Sequence to Sequence Autoencoder (Babu, 2020).

4.3 Model training

First, the dataset is split into 3 parts (train 80%, test 10%, and validation 10%):

```
train_df, test_df = df[:int(len(df) * 0.8)], df[int(len(df) * 0.8):]
test_df, validate_df = test_df[:int(len(test_df) * 0.5)], test_df[int(len(test_df) * 0.5):]
```

There is a function that will use a sliding window method to transform series into samples of input past points and output future points:

```
def split_series(series, n_past, n_future):
 X, y = list(), list()
 for window_start in range(len(series)):
   past_end = window_start + n_past
   future_end = past_end + n_future
   if future_end > len(series):
     break
   # slicing the past and future parts of the window
   past, future = series[window_start:past_end, :], series[past_end:future_end, :]
   X.append(past)
   y.append(future)
 return np.array(X), np.array(y)
```

Afterward, we apply the *split_series* function along with *MixMaxScaler* to normalize neural network inputs (we would forecast 24 hours ahead with 48 hours history points):

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(-1,1))

n_features = 1
n_future = 24
n_past = n_future * 2

X_train, y_train = split_series(scaler.fit_transform(train_df.values), n_past, n_future)

X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], n_features))
y_train = y_train.reshape((y_train.shape[0], y_train.shape[1], n_features))

X_test, y_test = split_series(scaler.transform(test_df.values), n_past, n_future)
```

```
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], n_features))
y_test = y_test.reshape((y_test.shape[0], y_test.shape[1], n_features))
```

The neural network architecture is rather straightforward to build with TensorFlow:

```
import tensorflow as tf

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.LSTM(100, activation='relu', input_shape=(n_past, n_features)))
model.add(tf.keras.layers.RepeatVector(n_future))
model.add(tf.keras.layers.LSTM(100, activation='relu', return_sequences=True))
model.add(tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(1)))
model.compile(optimizer='adam', loss='mse')
model.summary()
```

The network is trained for five epochs with a loss of MSE 0.0042 and validation loss of MSE 0.0058 on the final epoch (Figure 7).

```
Epoch 1/5
3578/3578 [==============================] - 167s 46ms/step - loss: 0.0148 - val_loss: 0.0074
Epoch 2/5
3578/3578 [==============================] - 170s 48ms/step - loss: 0.0056 - val_loss: 0.0068
Epoch 3/5
3578/3578 [==============================] - 168s 47ms/step - loss: 0.0050 - val_loss: 0.0064
Epoch 4/5
3578/3578 [==============================] - 168s 47ms/step - loss: 0.0046 - val_loss: 0.0061
Epoch 5/5
3578/3578 [==============================] - 172s 48ms/step - loss: 0.0042 - val_loss: 0.0058
```

Figure 7. Training log for the neural network.

The prediction vs. actual plot for the 24-hour forecast looks good for arbitrary forecast points for the validation dataset (Figure 8).

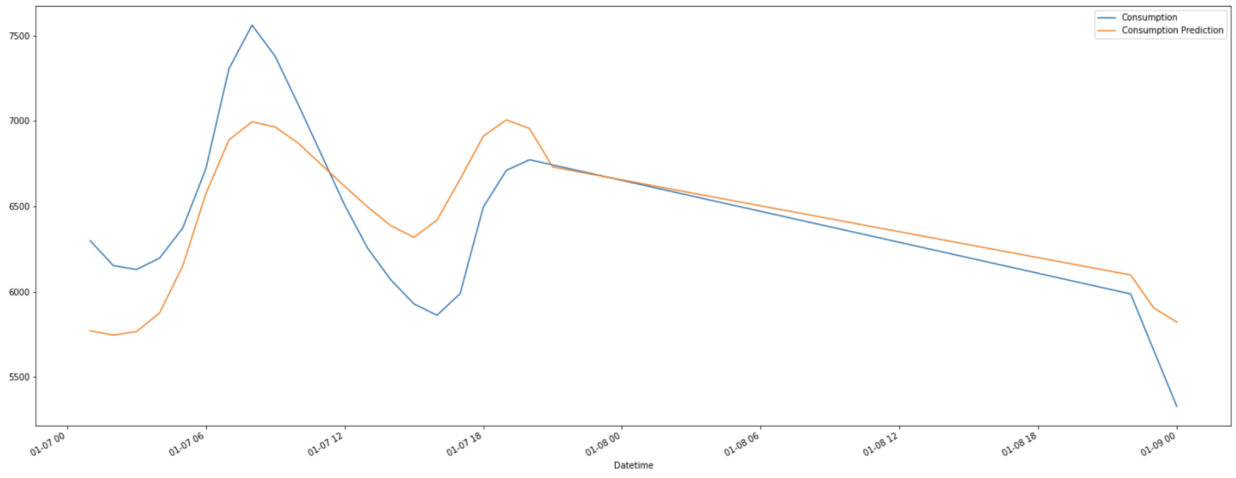

Figure 8. Prediction vs. actual plot for the 24-hour forecast.

4.4 Comparison with Prophet

To evaluate the results better, we can compare multi-step LSTM model predictions with Prophet. The prophet is an autoregressive model for predicting time-series data based on an additive model where non-linear trends are fit various seasonality, plus holiday effects. It works great with time-series that have solid seasonal data. In most cases, Prophet produces similar results as other autoregressive models like AutoARIMA (Facebook, 2021).

To compare results, predictions at points $t + 1, t + 6, t + 12, t + 18, t + 24$ at last 1000 forecast points for validation dataset that was not used in testing or training. Afterward, an RMSE error between prediction and actual is calculated for each future time point. For stacked LSTM sequence to sequence autoencoder, the best results are achieved at $t + 1$, which is expected. Accuracy slowly declines over the future values but still provides good results at point $t + 24$. By comparison, Prophet mode produced considerably worse results for each future time point, which is expected (Table 1). This proves that stacked LSTM sequence to sequence autoencoder can produce reliable short-term and long-term forecasts.

Time step / Model error	t + 1	t + 6	t + 12	t + 18	t + 24
seq2seq LSTM	209.5	410.0	498.4	528.4	554.4
Prophet	1512.9	1471.0	880.8	899.9	1267.9

Table 1. RMSE errors for seq2seq LSTM and Prophet for different time steps (power usage)

5. LSTM multi-step forecasting model for stock prices prediction

5.1 Dataset

The historical stock prices could be retrieved with Python library called *yfinance*, which is a wrapper to Yahoo! Finance API (Aroussi, 2017). For the sake of an example, let's try to forecast Microsoft's (MSTF) daily open stock prices (Figure 9).

```
import yfinance as yf
msft = yf.Ticker('MSFT')
df = msft.history(start='2014-03-01', end='2021-03-01', actions=False)
df = df[['Open']]
```

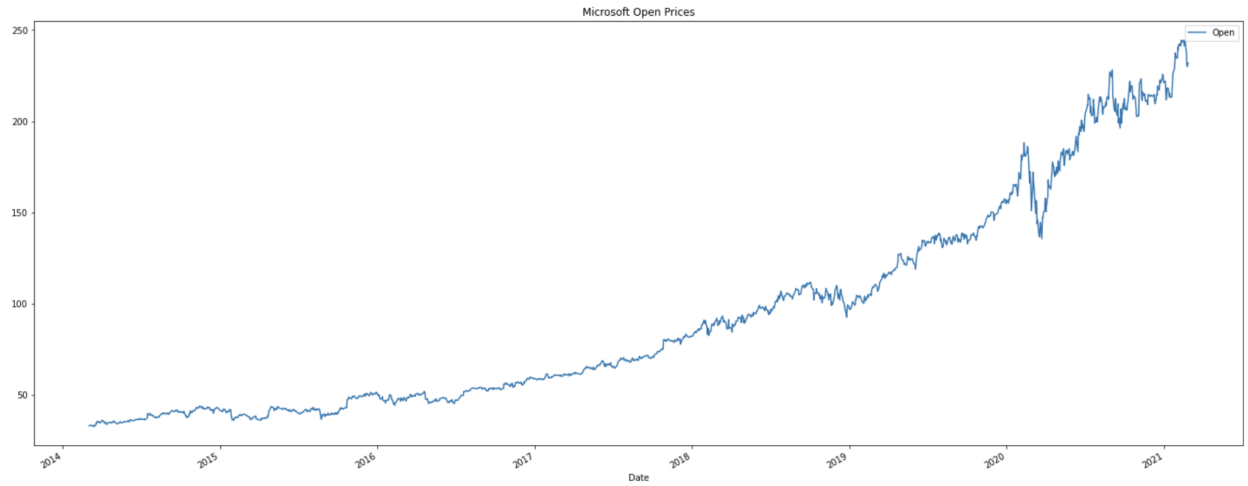

Figure 9. Microsoft stock prices dataset

5.2 Model training

The stacked LSTM sequence to sequence autoencoder is trained in the same way as in section 4.3 (we want to forecast for 24 days with 48 days history), except the neural network is trained with 15 epochs. The training loss is 0.0029, but the validation loss is on 0.1660 on epoch 15 (Figure 10).

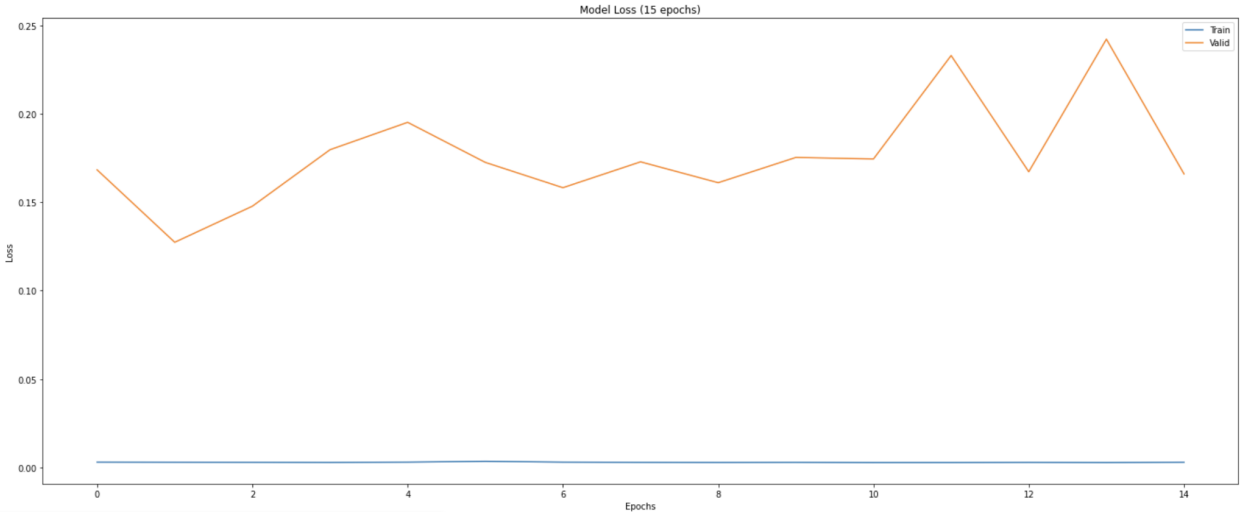

Figure 10. Model loss for 15 epochs.

This radical difference in training loss and validation loss is extremely uncommon, and it means that predictions are not great. This is confirmed by a prediction plot for arbitrarily chosen forecast point (Figure 11). The prediction for the $t + 1$ looks quite accurate (since it is always similar to t), but starting with the $t + 2$ model fails. Overall prediction quality is not satisfying.

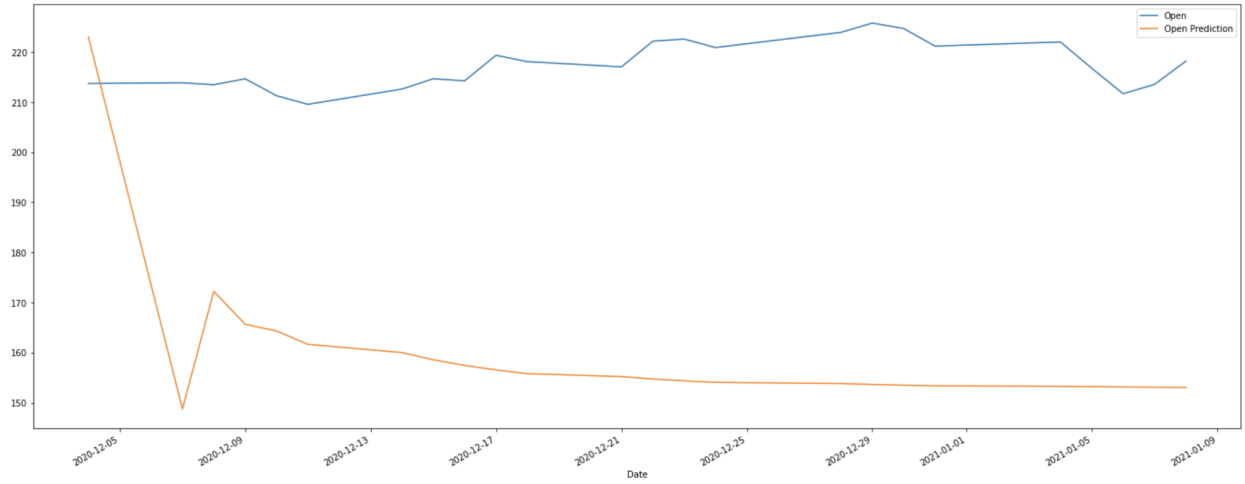

Figure 11. Seq2seq LSTM predictions for 24 days.

5.3 Results analysis

If the same architecture and preprocessing works great with one type of data (power usage) and poorly with another (stock prices), the problem might be with the data. Another way to prove it is to compare predictions to some baseline model prediction. The obvious baseline model uses the last historical value for forecasts for all time steps, as shown in Figure 12.

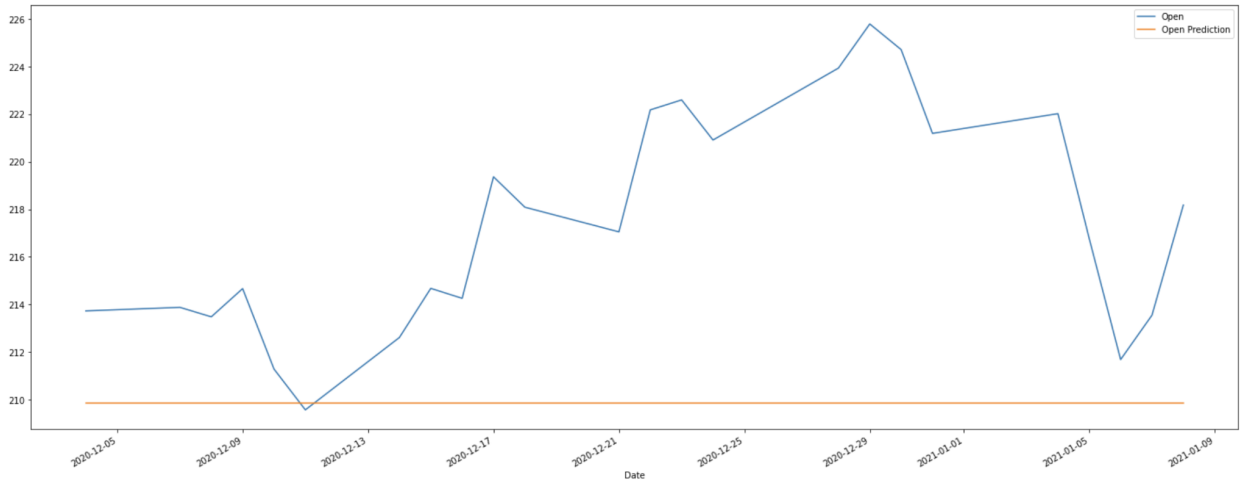

Figure 12. Baseline predictions for 24 days.

In table 2, we can see that the baseline model beats both seq2seq LSTM and Prophet for all time steps predictions, which means both seq2seq LSTM and Prophet do not work with stock market data and models are useless.

Time step / Model error	t + 1	t + 6	t + 12	t + 18	t + 24
seq2seq LSTM	14.39	55.14	63.35	67.32	69.49
Baseline	10.94	11.31	13.15	14.92	16.19
Prophet	44.43	45.59	47.21	48.55	49.35

Table 2. RMSE errors for seq2seq LSTM, Baseline, and Prophet for different time steps (stock market prediction).

5.4 Predicting the stock market is a particularly challenging task

The possibility of predicting the financial market has a connection to the notion of *market efficiency*. The term refers to the level to which stock prices represent all available data. There is no clear way to measure the market efficiency accurately. Despite this flaw, we can still make some judgments depending on how roughly efficient the market is. Collectively, it is known as Efficient Market Hypothesis (EMH) by Nobel laureate Eugene Fama (Investopedia Staff, 2020).

There are three forms of EMH: strong, semi-strong, and weak (Downey, 2020):

- If the market has strong efficiency, then all public and private information is already incorporated into the market prices. In this case, it is impossible to predict market prices since historical information is irrelevant to current or future data. Essentially, the historical trend line in the stock market is a random walk.

- Semi-strong efficiency says that only public information is included in the stock price. It suggests that to predict the stock market, one must have access to some private information.

- Weak form efficiency means that all past data is already in the price. As new information arises, the news is quickly incorporated into stock prices. This form makes market prediction possible, but the model must catch all arising information lighting fast, making the task very challenging.

Many oppose EMH since the real-world market can be inefficient. Market inefficiencies may exist due to information asymmetries, a lack of liquidity, market psychology, human bias, and human emotion. That is why investors such as Warren Buffett have consistently beaten the market over the years (Downey, 2020).

Nevertheless, even assuming inefficiencies in the market exists, the stock market prediction is an incredibly complex solution. The historical stock price is not an adequate feature for the analysis. The model should take into account thousands of different factors besides stock price simultaneously.

Conclusion

In this work, we trained the model called stacked LSTM sequence to sequence autoencoder on two datasets power usage and stock market prices. The advantage of this model is that it can output multiple time steps forecasts. For the power usage dataset, the LSTM-based model produced a great result on 24 hours forecast and beats Prophet autoregressive model by 37-75%, depending on the time step. On the other hand, the LSTM-based model with the same strategy produced bad results on stock market prices, getting worse results than a simple baseline model. Likely, this is related market efficiency concept. Turns out, forecasting the stock market is a highly complex problem and can't be solved with traditional time-series forecasting approaches. For future work, it would be good to add an ability to include more than one feature in stacked LSTM sequence to sequence autoencoder. This will allow feature generation before modeling and should improve results.

Bibliography

Aroussi, R. (2017). *Yahoo! Finance market data downloader*. GitHub. https://github.com/ranaroussi/yfinance

Babu, J. (2020). *Multivariate Multi-step Time Series Forecasting using Stacked LSTM sequence to sequence Autoencoder*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/10/multivariate-multi-step-time-series-forecasting-using-stacked-lstm-sequence-to-sequence-autoencoder-in-tensorflow-2-0-keras/

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *: IEEE Transactions on Neural Networks*, *5*(2). https://ieeexplore.ieee.org/document/279181

Brownlee, J. (2017). *4 Strategies for Multi-Step Time Series Forecasting*. Machine Learning Mastery. https://machinelearningmastery.com/multi-step-time-series-forecasting/

Downey, L. (2020). *Efficient Market Hypothesis (EMH)*. Investopedia. https://www.investopedia.com/terms/e/efficientmarkethypothesis.asp

Facebook. (2021). *Prophet*. Prophet. https://facebook.github.io/prophet/

Investopedia Staff. (2020). *Market Efficiency*. Investopedia. https://www.investopedia.com/terms/m/marketefficiency.asp

Li, R. (2019). *Multistep ahead forecast feature*. GitHub. https://github.com/sassoftware/python-dlpy/issues/101

Masum, S., Liu, Y., & Chiverton, J. (2018). *Multi-step Time Series Forecasting of Electric Load using Machine Learning Models*. School of Engineering, University of Portsmouth. https://core.ac.uk/download/pdf/159078987.pdf

Mulla, R. (2018). *Hourly Energy Consumption*. Kaggle. https://www.kaggle.com/robikscube/hourly-energy-consumption?select=PJMW_hourly.csv

Olah, C. (2015). *Understanding LSTM Networks*. Colah's blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs/