Time Series Practice

Data AI Lab

School of Electrical Engineering

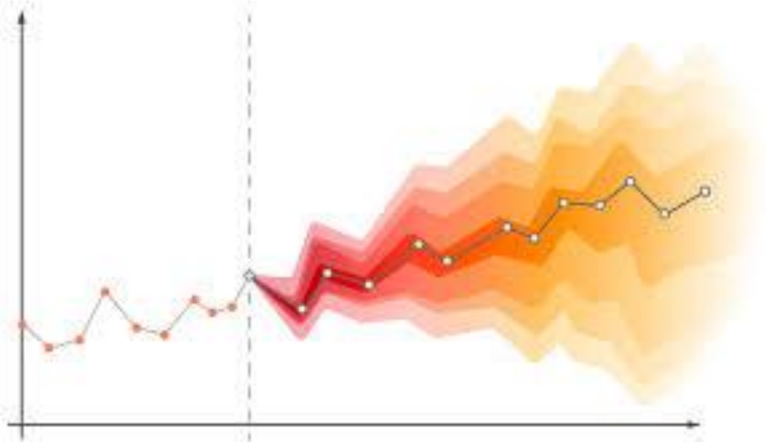
KAIST

Outline

1. **Introduction**
2. Data Processing
3. Background of Practice Model
4. Practice
5. (Optional) Encoder-decoder structure

Time Series Forecasting

- Predict future values based on historical data.
  - e.g., weather forecasting, traffic prediction, sales forecasting …



Time Series and Deep Learning

- Learns complex, non-linear patterns.
- Popular Models
  - **RNN/LSTM**: Sequential dependencies over time.
  - **CNN**: Efficiently captures local patterns and stationarity.
  - **Transformer**: Handles global dependencies with attention mechanisms.

Time Series Practice

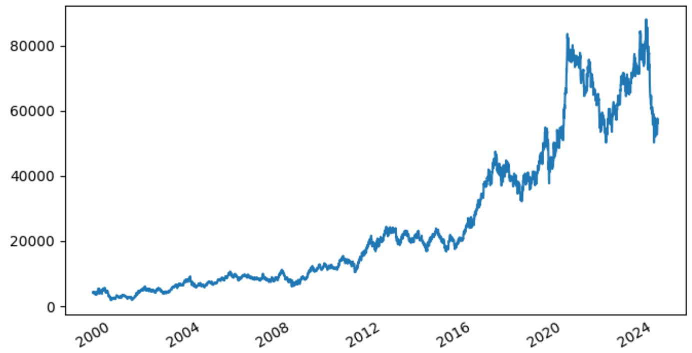
- Build a model for time series forecasting.
- Steps:
  - Data Collection
  - Data Preprocessing
  - Model Training
  - Evaluation

Outline

1. Introduction
2. **<u>Data Processing</u>**
3. Background of Practice Model
4. Practice
5. (Optional) Encoder-decoder structure

Introduction

- **Objective**:
  - Forecast financial time series data using deep learning models.
- **Data Source**: Yahoo Finance (yFinance API)
  - It offers a Pythonic way to fetch financial data from Yahoo!® finance.
  - Documentation website: https://ranaroussi.github.io/yfinance/index.html
  - **[Caveat] It is intended for research and educational purposes.**



Experimental Setup and Environment

- **Install** required packages and configure GPU settings

```python
# Math and data preprocessing libraries
import math
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler

# For handling dataset
import yfinance as yf
from datetime import date

# For visualization
import seaborn as sns
import matplotlib.pyplot as plt
```

```python
# For deep learning
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# For evaluation
from sklearn.metrics \
import root_mean_squared_error, mean_absolute_percentage_error
```

```python
device = torch.device('cuda' if \
                      torch.cuda.is_available() else 'cpu')

print(device)
```

Dataset Preparation

- **Download** the market data
  - Try other stock tickers: ‘AAPL’, ‘NVDA’, ‘005930.KS’

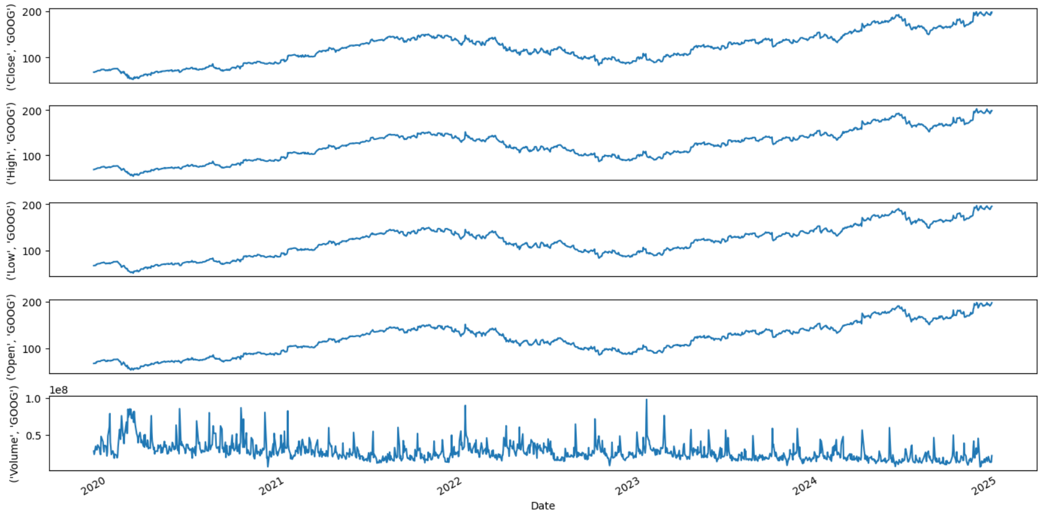
```
start_date = '2020-01-01'
end_date = '2024-12-31'

df = yf.download('GOOG', start=start_date, end=end_date)

# Inspect the data
print()
print(df.head())
print(df.info())
```

```
Price            Close       High        Low       Open    Volume
Ticker            GOOG       GOOG       GOOG       GOOG      GOOG
Date
2020-01-02   68.123726  68.162086  66.837348  66.837348  28132000
2020-01-03   67.789429  68.379312  67.036336  67.151721  23728000
2020-01-06   69.460922  69.575007  67.258334  67.258334  34646000
2020-01-07   69.417572  69.898343  69.270099  69.646752  30054000
2020-01-08   69.964615  70.326314  69.293024  69.354799  30560000
```

Data Visualization

- **Draw line plots** for each feature:
    - There are five features used (Open, High, Low, Close, Volume)

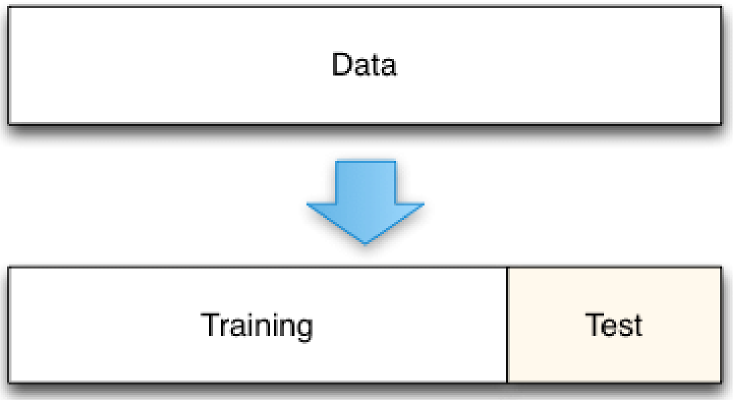
```
ncols = 1
nrows = int(round(df.shape[1] / ncols, 0))
fig, ax = plt.subplots(nrows=nrows, ncols=ncols, \
                       sharex=True, figsize=(14, 7))
for i, ax in enumerate(fig.axes):
    sns.lineplot(data=df.iloc[:, i], ax=ax)
    ax.tick_params(axis="x", rotation=30, \
                   labelsize=10, length=0)
fig.tight_layout()
plt.show()
```



Data Preprocessing: Train-Test Split

- Why do we split the data?
  - Ensures that the model is evaluated on unseen data.
  - Prevents overfitting by testing on separate data which is not used during training.
- **Split** into training and test data:

```
# Train test split
train_ratio = 0.8
training_data_len = math.ceil(len(df) * train_ratio)

# Splitting the dataset
train_data = df[:training_data_len][['Open']]
test_data = df[training_data_len:][['Open']]
print(train_data.shape)
print(test_data.shape)
```



Data Preprocessing: Scaling

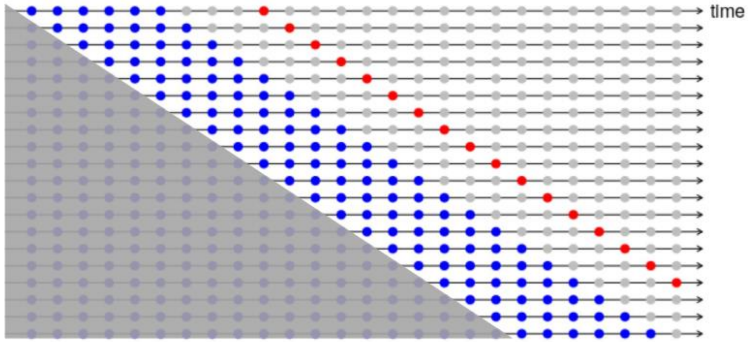
- Why do we scale the data?
  - Prevents features with larger magnitudes from dominating the training process.
- **Scale** the data to normalize values between 0 and 1:

```
scaler = MinMaxScaler(feature_range=(0, 1))
train_scaled = scaler.fit_transform(train_data.values)
test_scaled = scaler.transform(test_data.values)
```

Data Preprocessing: Sliding Window

- Why do we convert (time-series) data into sequences?
  - Time series models require sequences to capture temporal dependencies.
    - The model can learn specific patterns for predicting future values.
  - Preprocess the data to enable time-dependent sequential input.
    - Sequence length defines the look-back window for predicting future values.
- **Create labeled training pairs** by sliding window:

```python
def convert_data_into_tensors(data_seq):
  features, labels = [], []
  for i in range(len(data_seq) - sequence_length):
      features.append(data_seq[i:i + sequence_length])
      labels.append(data_seq[i + sequence_length, 0])
  features, labels = np.array(features), np.array(labels)

  features = torch.tensor(features, dtype=torch.float32)
  labels = torch.tensor(labels, dtype=torch.float32)
  return features, labels
```



Data Preprocessing: Batching

- Why do we make batch data?
  - If using the entire training dataset to update model parameters, the training process can become **slow**, and the entire dataset may **not fit into memory**.
- **Create data loaders** for efficient batch processing:

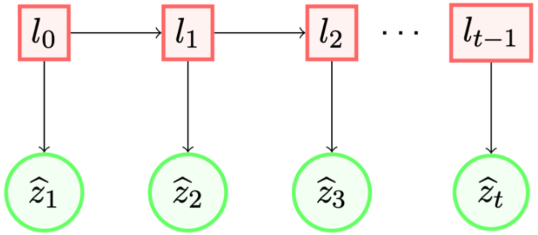
```
batch_size = 32

def to_loader(x, y, batch_size, shuffle):
  dataset = torch.utils.data.TensorDataset(x, y)
  return torch.utils.data.DataLoader(dataset, batch_size, shuffle)

train_loader = to_loader(X_train, y_train, batch_size, shuffle=True)
test_loader = to_loader(X_test, y_test, batch_size, shuffle=False)
```

Outline

1. Introduction
2. Data Processing
3. **<u>Background of Practice Model</u>**
4. Practice
5. (Optional) Encoder-decoder structure

Recap: State Space Models

- **State Space Model (SSM)**
  - **Input**: last state $\boldsymbol{l}_{t-1}$
  - **Update**: $\boldsymbol{l}_t = \mathbf{F}_t \boldsymbol{l}_{t-1} + \boldsymbol{g}_t \epsilon_t$
    - White noise ϵ_t, parameters $\{\boldsymbol{F}_t, \boldsymbol{g}_t\}$
  - **Output**: $z_t = \boldsymbol{a}_t^\top \boldsymbol{l}_{t-1} + \epsilon_t$
    - Parameter $\boldsymbol{a}_t$

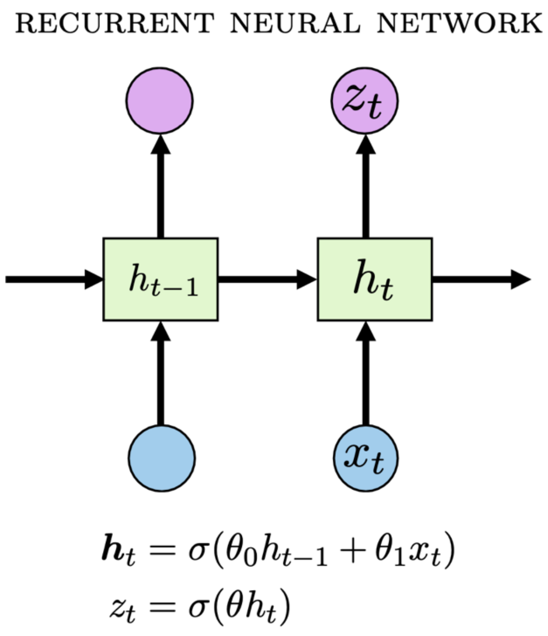


Measurements $\quad z_t = \boldsymbol{a}_t^T \boldsymbol{l}_{t-1} + \epsilon_t, \quad \epsilon_t \sim N(0, \sigma^2)$

State transition $\quad \boldsymbol{l}_t = \boldsymbol{F}_t \boldsymbol{l}_{t-1} + \boldsymbol{g}_t \epsilon_t, \quad \boldsymbol{l}_0 \sim N(\boldsymbol{\mu}_0, \mathrm{diag}(\sigma_0^2)).$

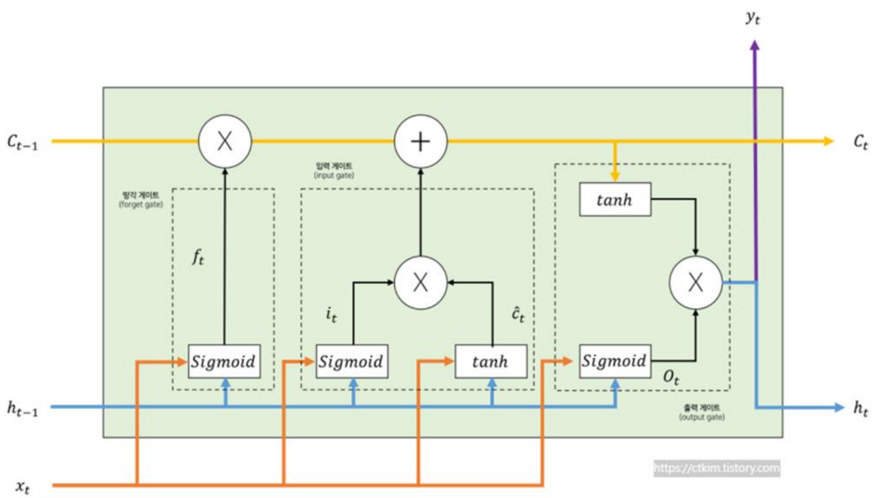
Recap: Recurrent Neural Networks (RNN)

- **RNN(Recurrent Neural Network)**
  - **Input**: last state h_{t-1}, current feature x_t
  - **Update**: $h_t = \sigma(\boldsymbol{\theta}_0 h_{t-1} + \boldsymbol{\theta}_1 x_t)$
    - Activation function σ, learnable parameters $\{\theta_0, \theta_1\}$
  - **Output**: $z_t = \sigma(\boldsymbol{\theta} h_t)$
    - Learnable parameter θ



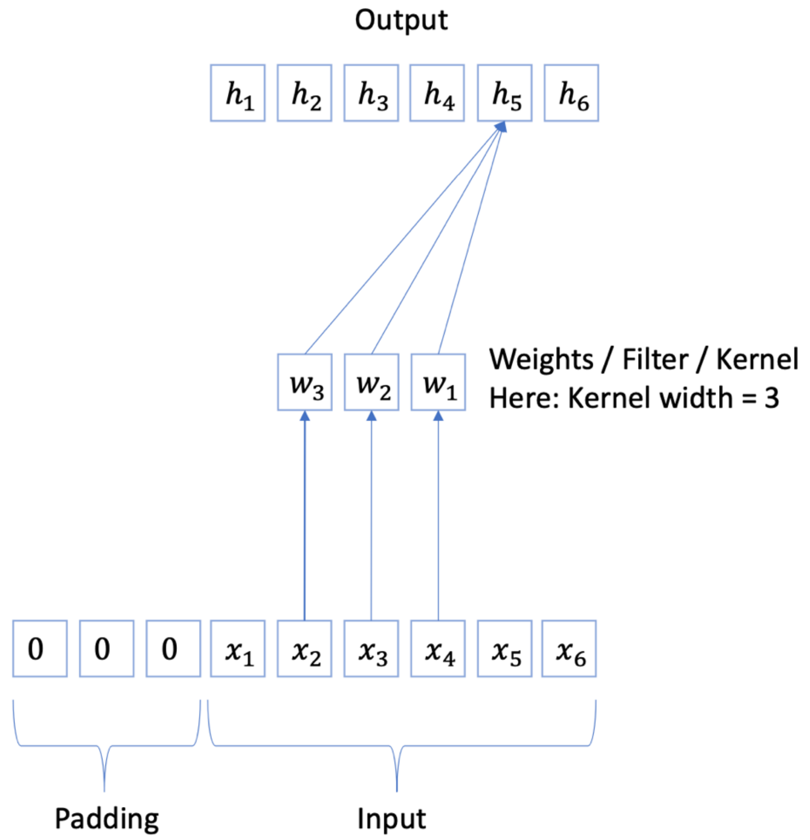
Recap: Long Short-Term Memory (LSTM)

- **LSTM(Long Short Term Memory)**
  - **Input**: last short-term state h_{t-1}, last long-term state C_{t-1}, current feature x_t
  - **Update**: $C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}_t$, $h_t = o_t \odot \tanh(C_t)$
    - Forget Gate: $f_t = \sigma\left(\mathbf{W}_{hf} h_{t-1} + \mathbf{W}_{xf} x_t\right)$
    - Input Gate: $i_t = \sigma(\mathbf{W}_{hi} h_{t-1} + \mathbf{W}_{xi} x_t), \hat{C}_t = \tanh(\mathbf{W}_{hc} h_{t-1} + \mathbf{W}_{xc} x_t)$
    - Output Gate: $o_t = \sigma(\mathbf{W}_{ho} h_{t-1} + \mathbf{W}_{xo} x_t)$
  - **Output**: $z_t = h_t$



Recap: Convolutional Neural Networks (CNN)

- **CNN(Convolutional Neural Networks)**
  - **Input**: previous features $\{x_i\}_{i=t-D}^{t-1}$
  - **Output** : $h_t = \sum_{d=1}^{D} w_d x_{t-d}$
    - Kernel $W = [w_1, \cdots, w_D]$
  - After convolution layer, passing more layers like pooling, flatten, fully-connected layer



Outline

LSTM Model Architecture

- **Objective**: Build an LSTM model for time series forecasting
- **TODO**:
  - Sequential data passes through the LSTM layer.
  - The fully connected layer maps the last LSTM output to a single value.

```python
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        # (lstm): LSTM(1, 64, num_layers=2, batch_first=True)

        self.linear = nn.Linear(hidden_size, 1)
        # (linear): Linear(in_features=64, out_features=1, bias=True)

    def forward(self, x):
        out, _ = self.lstm(x)
        return self.linear(out)

model = LSTMModel(input_size, hidden_size, num_layers).to(device)
print(model)
```

Train

- **Objective**: Implement a training loop
- **TODO**:
  - Compute predictions
  - Calculate the train loss
  - Perform optimization

```python
# train
model.train()
for batch_x, batch_y in train_loader:
  # (1) Move input and target tensors to the device (e.g., GPU)
  batch_x, batch_y =

  # (2) Pass the input (batch_x) through the model
  #     The model outputs shape [batch_size, sequence_length, 1]
  #     Take the prediction at the last timestep (index -1) and feature index 0
  #     → model(batch_x)[:, -1, 0]
  pred =

  # (3) Compute the loss between pred and batch_y by using loss_fn
  loss =

  # (4) Clear previous gradients to avoid accumulation
  optimizer.

  # (5) Perform backpropagation to compute gradients
  loss.

  # (6) Update the model parameters using the optimizer
  optimizer.

  total_train_loss += loss.item()

avg_loss = total_train_loss / len(train_loader)
train_hist.append(avg_loss)
```

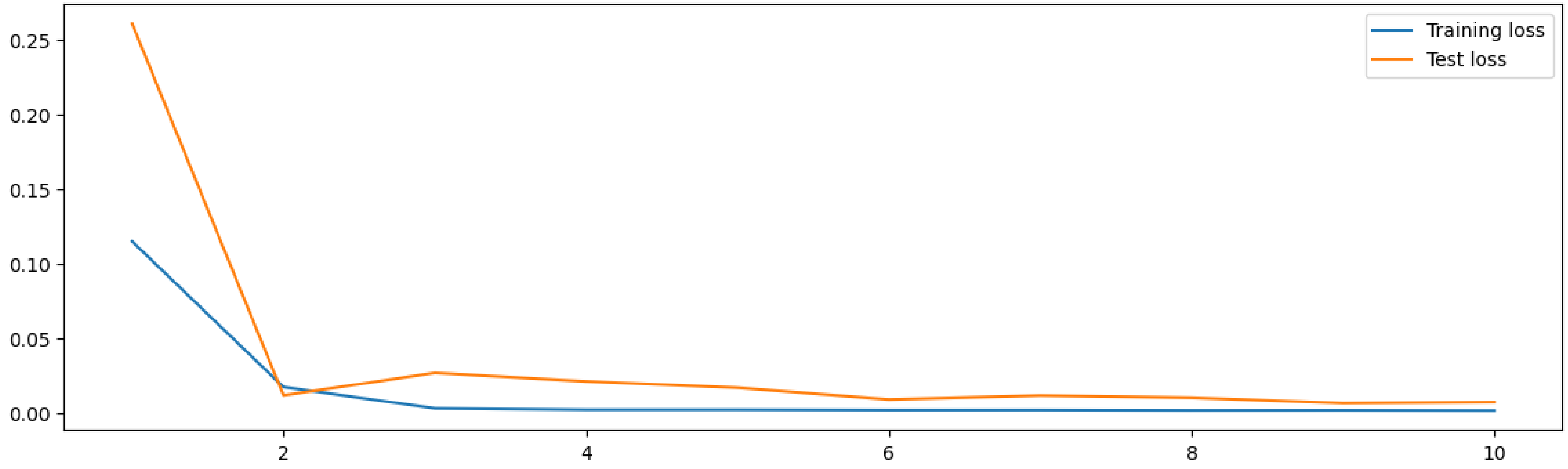
Test

- **Objective**: Implement evaluation
- **TODO**:
  - Compute predictions
  - Calculate the test loss

```python
# evaluate
model.eval()
with torch.no_grad():
  for test_x, test_y in test_loader:
    # TODO
    # (1) Move input and target tensors to the device (e.g., GPU)
    test_x, test_y =

    # (2) Pass the input (test_x) through the model
    #     The model outputs shape [batch_size, sequence_length, 1]
    #     Take the prediction at the last timestep (index -1) and feature index 0
    #     → model(test_x)[:, -1, 0]
    test_pred =

    # (3) Compute the loss between test_pred and test_y by using loss_fn
    test_loss =

    total_test_loss += test_loss.item()

avg_test_loss = total_test_loss / len(test_loader)
test_hist.append(avg_test_loss)
```

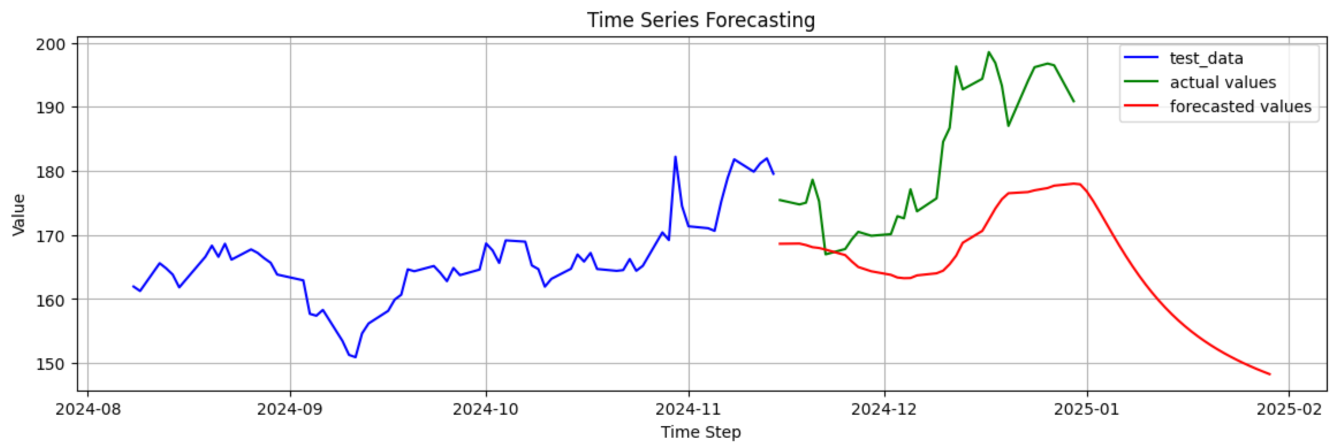
Loss Visualization

- **Objective**: Compare training loss and test loss over the epochs
- **Interpretation**:
  - Training loss: Stabilizes after a few epochs, suggesting convergence.
  - Test loss: Follows a similar trend as training loss but remains slightly higher



Forecasting Visualization

- **Objective**: Compare actual values with the model's forecasted values
- **Interpretation**:
  - Test data: unseen test data that the model is being evaluated on
  - Actual values: ground truth values used to validate the model's forecasts
  - Forecasted values: the model's predicted values for the future



Model Evaluation

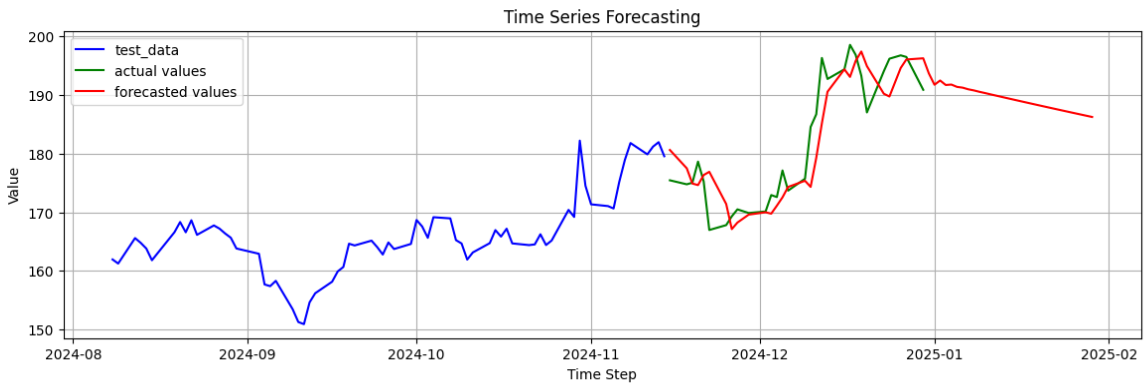
- **Objective**: Evaluate the performance of the LSTM model
  - Root Mean Squared Error (RMSE)
  - Mean Absolute Percentage Error (MAPE)
- **TODO**:
  - Measure RMSE and MAPE

```python
def test(model, X_test, y_test):
  model.eval()
  with torch.no_grad():
      test_predictions = []
      for batch_X_test in X_test:
          batch_X_test = batch_X_test.to(device).unsqueeze(0)
          test_predictions.append(model(batch_X_test) \
                                 .cpu().numpy().flatten()[0])

  test_predictions = np.array(test_predictions)
  y_test = y_test.cpu().numpy()

  rmse = root_mean_squared_error(, ) # TODO
  mape = mean_absolute_percentage_error(, ) # TODO

  return rmse, mape
```

Other Models

- **Objective**: Compare the performance of Conv1D and RNN models
  - Conv1D focuses on capturing local temporal patterns
  - RNNs are designed for sequential dependencies over time

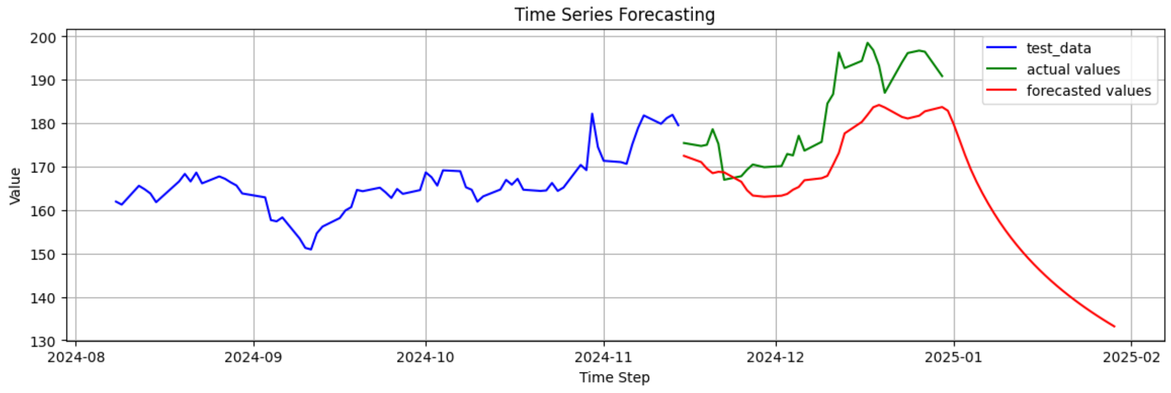
```
class Conv1DModel(nn.Module):
  def __init__(self, input_size, hidden_size):
    super(Conv1DModel, self).__init__()
    # TODO
    self.conv1d = nn.Conv1d(in_channels=, out_channels=, \
                            kernel_size=2, stride=1)
    self.fc = nn.Linear(, 1)

  def forward(self, x):
    x = x.transpose(1, 2)
    x = self.conv1d(x)
    x = x.transpose(1, 2)
    return self.fc(x)
```



Other Models

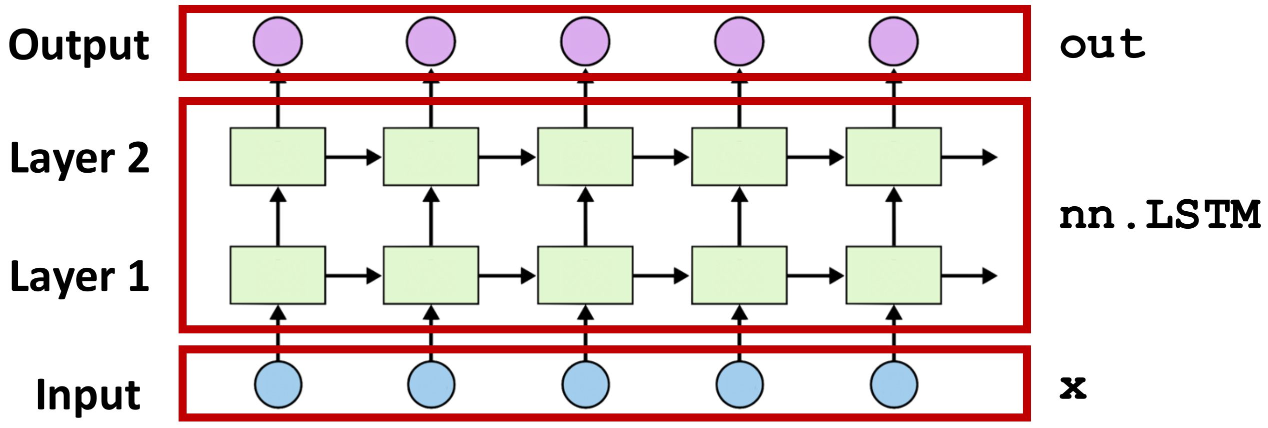
- **Objective**: Compare the performance of Conv1D and RNN models
  - Conv1D focuses on capturing local temporal patterns
  - RNNs are designed for sequential dependencies over time

```python
class RNNModel(nn.Module):
  def __init__(self, input_size, hidden_size, num_layers):
      super(RNNModel, self).__init__()
      # TODO
      self.rnn = nn.RNN(, , , batch_first=True)
      self.fc = nn.Linear(, 1)

  def forward(self, x):
      # TODO
```



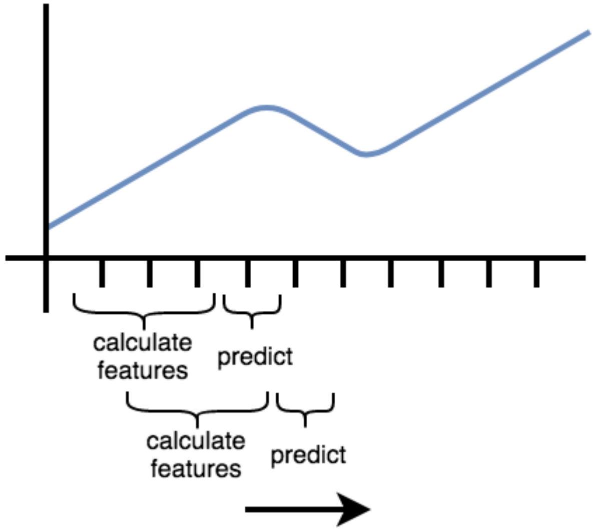
Code Explanation: LSTM Architecture

- `out,(hn,cn) = nn.LSTM(x)`
  - `x`: [batch_size, sequence_length, num_features] # [32, 50, 1]
  - `out`: [batch_size, sequence_length, hidden_size] # [32, 50, 64]
  - `hn`: [num_layers, batch_size, hidden_size] # [2, 32, 64]
  - `cn`: [num_layers, batch_size, hidden_size] # [2, 32, 64]
  - Only the output features from the final LSTM layer are typically required.



Code Explanation: Rolling Forecast

- `input_data = np.roll(input_data, shift=-1)`
  - For example, [10, 20, 30, 40, 50] → [20, 30, 40, 50, 10]
    - Then replace the last value (10).
  - Used to shift the input window for autoregressive forecasting.
    - If ground-truth future value is available, insert it at the end of the input window.
    - Otherwise, insert the model's predicted value.

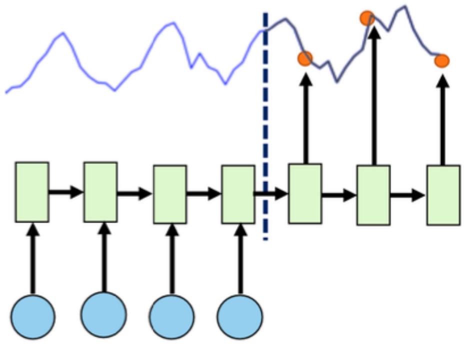


Outline

1. Introduction
2. Data processing
3. Background of practice model
4. Practice
5. **(Optional) Encoder-decoder structure**

Recap: Encoder-Decoder Structure

- **Encoder**: Captures the sequence of temporal dependencies from the past observations.
- **Decoder**: Uses the encoded information from the encoder to produce the future predictions.



$$f_{encoder} : \{z_1, \cdots, z_{T_e}\} \mapsto \boldsymbol{h}_{T_e}$$
$$f_{decoder} : \boldsymbol{h}_{T_e} \mapsto \{z_{T_e+1}, \cdots, z_{T_e+T_d}\}$$

Multi-Step Forecasting

- **Forecasting multi-step**: The model is trained on sequences of past observations and sequences of future values.

```
sequence_length = 50
target_len = 10

def create_enc_dec_sequences(data):
    features, labels = [], []
    for i in range(len(data) - sequence_length - target_len):
        features.append(data[i:i + sequence_length])
        labels.append(data[i + sequence_length : \
                           i + sequence_length + target_len])

    features = np.array(features, dtype=np.float32)
    labels = np.array(labels, dtype=np.float32)

    features = torch.tensor(features, dtype=torch.float32)
    labels = torch.tensor(labels, dtype=torch.float32)
    return features, labels
```

Functions for Training

- In multi-step forecasting, the format for creating training pairs changes.
  - Unlike one-step forecasting, **multi-step forecasting requires output sequences** that predict several future time steps at once.

```
def train_(model, train_loader, test_loader):
```

```
def plot_forecasting_(model, X_test, y_test):
```

```
def test_(model, X_test, y_test):
```

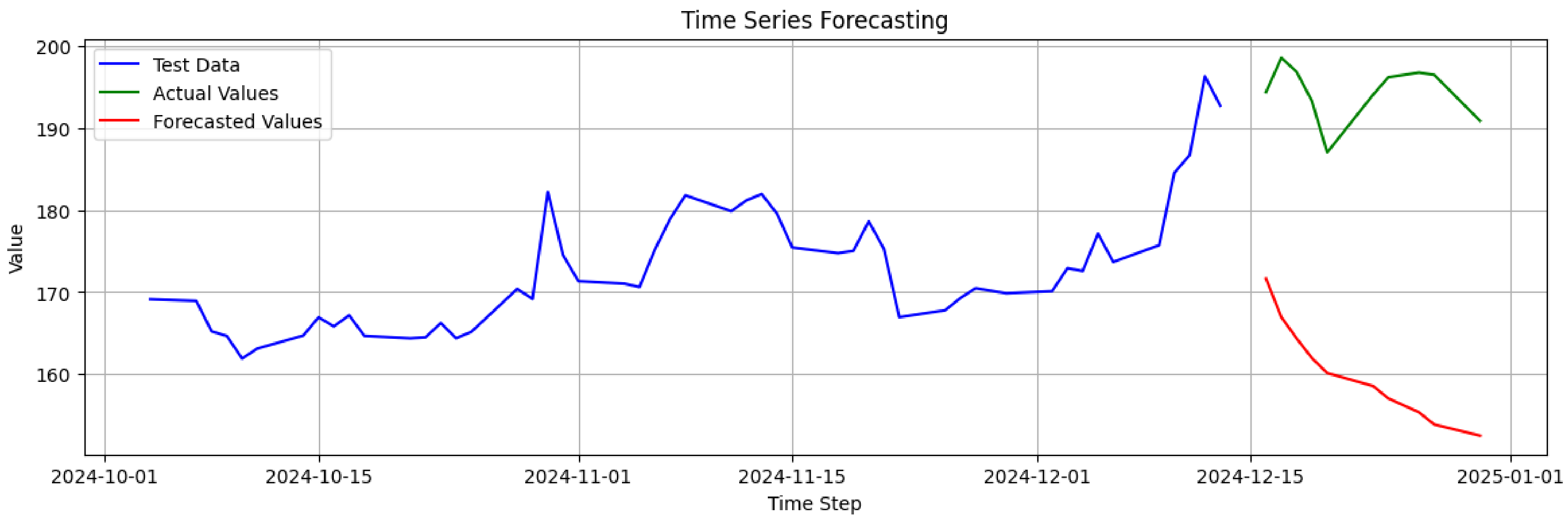
RNN-RNN Model Architecture

- **Objective**: Build an RNN-RNN architecture
- **TODO**:
  - Implement the __init__ method
  - Complete the forward function

```
class EncoderRNN(nn.Module):
  def __init__(self, input_size, hidden_size, num_layers):
    # TODO

  def forward(self, x):
    # TODO

class DecoderRNN(nn.Module):
  def __init__(self, input_size, hidden_size, num_layers):
    # TODO

  def forward(self, x, h):
    # TODO
```

Forecasting Visualization

- **Objective**: Display the test data, actual values, and the forecasted values to evaluate the model's performance.



References

- [NIPS '14] Sequence to Sequence Learning with Neural Networks
  - https://arxiv.org/abs/1409.3215