stock_pytorch

June 29, 2025

[41]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import datetime
import matplotlib.pyplot as plt
import math, time
from math import sqrt
from sklearn import preprocessing
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
import torch
import torch.nn as nn
from torch.autograd import Variable
import random
import itertools
import datetime
from operator import itemgetter
import urllib.request, json
import datetime as dt

# List all files under the input directory from kaggle
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

[42]:
```
data_source = 'alphavantage'
if data_source == 'alphavantage':
    # ====================== Loading Data from Alpha Vantage␣
 ↪==================================

    api_key = '4MFYSPOPV210V5WT'

    # American Airlines stock market prices
    ticker = "GOOG"

    # JSON file with all the stock market data for AAL from the last 20 years
```

```
    url_string = "https://www.alphavantage.co/query?
↪function=TIME_SERIES_DAILY&symbol=%s&outputsize=full&apikey=%s"%(ticker,api_key)

    # Save data to this file
    file_to_save = './data/stock_market_data-%s.csv'%ticker

    # If you haven't already saved data,
    # Go ahead and grab the data from the url
    # And store date, low, high, volume, close, open values to a Pandas␣
↪DataFrame
    if not os.path.exists(file_to_save):
        with urllib.request.urlopen(url_string) as url:
            data = json.loads(url.read().decode())
            # extract stock market data
            data = data['Time Series (Daily)']
            df = pd.DataFrame(columns=['Date','Low','High','Close','Open'])
            for k,v in data.items():
                date = dt.datetime.strptime(k, '%Y-%m-%d')
                data_row = [date.date(),float(v['3. low']),float(v['2. high']),
                            float(v['4. close']),float(v['1. open'])]
                df.loc[-1,:] = data_row
                df.index = df.index + 1
        print('Data saved to : %s'%file_to_save)
        df.to_csv(file_to_save)
```

[43]:
```
data_source = 'alphavantage'
if data_source == 'alphavantage':
    # ====================== Loading Data from Alpha Vantage␣
↪==================================

    api_key = '4MFYSPOPV210V5WT'

    # American Airlines stock market prices
    ticker = "AAL"

    # JSON file with all the stock market data for AAL from the last 20 years
    url_string = "https://www.alphavantage.co/query?
↪function=TIME_SERIES_DAILY&symbol=%s&outputsize=full&apikey=%s"%(ticker,api_key)

    # Save data to this file
    file_to_save = './data/stock_market_data-%s.csv'%ticker

    # If you haven't already saved data,
    # Go ahead and grab the data from the url
    # And store date, low, high, volume, close, open values to a Pandas␣
↪DataFrame
    if not os.path.exists(file_to_save):
```

```
        with urllib.request.urlopen(url_string) as url:
            data = json.loads(url.read().decode())
            # extract stock market data
            data = data['Time Series (Daily)']
            df = pd.DataFrame(columns=['Date','Low','High','Close','Open'])
            for k,v in data.items():
                date = dt.datetime.strptime(k, '%Y-%m-%d')
                data_row = [date.date(),float(v['3. low']),float(v['2. high']),
                            float(v['4. close']),float(v['1. open'])]
                df.loc[-1,:] = data_row
                df.index = df.index + 1
        print('Data saved to : %s'%file_to_save)
        df.to_csv(file_to_save)
```

[44]:
```
data_source = 'alphavantage'
if data_source == 'alphavantage':
    # ====================== Loading Data from Alpha Vantage␣
 ↪==================================

    api_key = '4MFYSPOPV210V5WT'

    # American Airlines stock market prices
    ticker = "AAL"

    # JSON file with all the stock market data for AAL from the last 20 years
    url_string = "https://www.alphavantage.co/query?
 ↪function=TIME_SERIES_DAILY&symbol=%s&outputsize=full&apikey=%s"%(ticker,api_key)

    # Save data to this file
    file_to_save = './data/stock_market_data-%s.csv'%ticker

    # If you haven't already saved data,
    # Go ahead and grab the data from the url
    # And store date, low, high, volume, close, open values to a Pandas␣
 ↪DataFrame
    if not os.path.exists(file_to_save):
        with urllib.request.urlopen(url_string) as url:
            data = json.loads(url.read().decode())
            # extract stock market data
            data = data['Time Series (Daily)']
            df = pd.DataFrame(columns=['Date','Low','High','Close','Open'])
            for k,v in data.items():
                date = dt.datetime.strptime(k, '%Y-%m-%d')
                data_row = [date.date(),float(v['3. low']),float(v['2. high']),
                            float(v['4. close']),float(v['1. open'])]
                df.loc[-1,:] = data_row
                df.index = df.index + 1
```

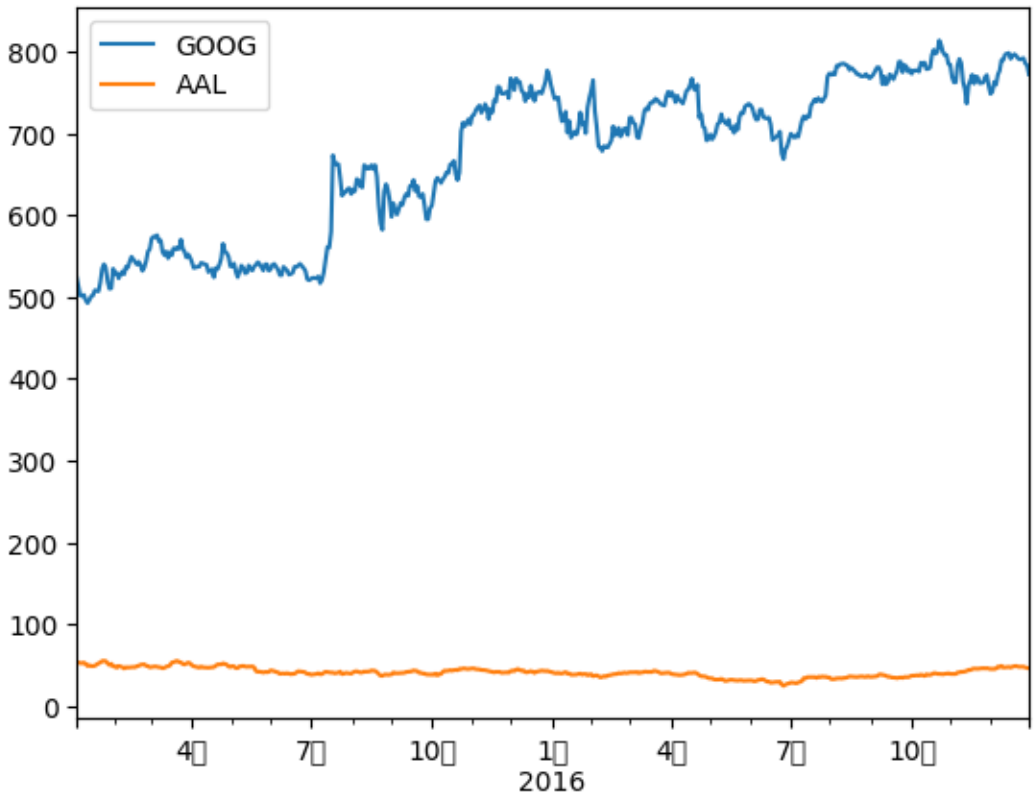
```
        print('Data saved to : %s'%file_to_save)
        df.to_csv(file_to_save)
```

[45]:
```
def stocksData(symbols, dates):
    df = pd.DataFrame(index=dates)
    for symbol in symbols:
        dfTemp = pd.read_csv("data/stock_market_data-{}.csv".format(symbol),␣
 ↪index_col='Date',
                parse_dates=True, usecols=['Date', 'Close'], na_values=['NaN'])
        dfTemp = dfTemp.rename(columns={'Close': symbol})
        # Add the column to the DataFrame:
        df = df.join(dfTemp)
    return df

# freq 'B' = business daily:
dates = pd.date_range('2015-01-02','2016-12-31',freq='B')
symbols = ['GOOG','AAL']
df = stocksData(symbols, dates)
# method 'pad': fill values forward(propagate last valid observation forward to␣
 ↪next valid backfill):
df.fillna(method='pad')
print(df)
df.interpolate().plot()
plt.show()
```

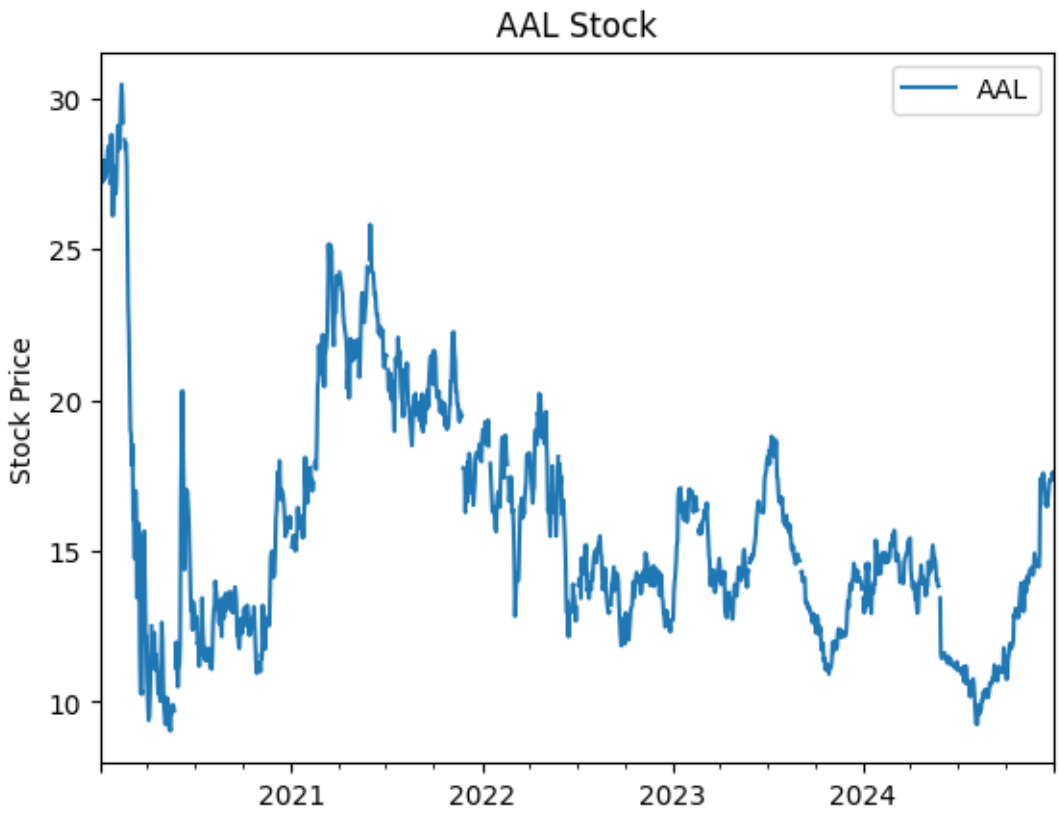
```
              GOOG     AAL
2015-01-02  524.81  53.910
2015-01-05  513.87  53.875
2015-01-06  501.96  53.040
2015-01-07  501.10  53.010
2015-01-08  502.68  53.660
…              …       …
2016-12-26     NaN     NaN
2016-12-27  791.55  48.610
2016-12-28  785.05  47.670
2016-12-29  782.79  47.250
2016-12-30  771.82  46.690

[521 rows x 2 columns]
```

```
/tmp/ipykernel_3244845/1971640994.py:16: FutureWarning: DataFrame.fillna with
'method' is deprecated and will raise in a future version. Use obj.ffill() or
obj.bfill() instead.
  df.fillna(method='pad')
/home/mzhyui/git/finai/.venv/lib/python3.10/site-
packages/IPython/core/pylabtools.py:170: UserWarning: Glyph 26376 (\N{CJK
UNIFIED IDEOGRAPH-6708}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```

```
/home/mzhyui/git/finai/.venv/lib/python3.10/site-
packages/IPython/core/pylabtools.py:170: UserWarning: Glyph 26376 (\N{CJK
UNIFIED IDEOGRAPH-6708}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```



[68]:
```
dates = pd.date_range('2020-01-02','2024-12-31',freq='B')
symbols = ['AAL']
df1 = pd.DataFrame(index=dates)
dfAAL = stocksData(symbols, dates)
dfAAL = df1.join(dfAAL)
dfAAL = dfAAL[['AAL']]
dfAAL.plot()
plt.ylabel("Stock Price")
plt.title("AAL Stock")
plt.show()
dfAAL.info()
```




```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1304 entries, 2020-01-02 to 2024-12-31
Freq: B
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   AAL     1258 non-null   float64
dtypes: float64(1)
memory usage: 20.4 KB
```

[69]:
```
# 'ffill'is like'pad': fill values forward(propagate last valid observation␣
 ↪forward to next valid backfill):
dfAal = dfAAL.fillna(method='ffill')

# MinMaxScaler: transform features by scaling each feature to a given range
scaler = MinMaxScaler(feature_range=(-1,1))
# reshape(-1,1): We have provided column as 1 but rows as unknown(numpy will␣
 ↪figure out)
dfAal['AAL'] = scaler.fit_transform(dfAal['AAL'].values.reshape(-1,1))
dfAal
```

```
/tmp/ipykernel_3244845/443460800.py:2: FutureWarning: DataFrame.fillna with
```

```
'method' is deprecated and will raise in a future version. Use obj.ffill() or
obj.bfill() instead.
  dfAal = dfAAL.fillna(method='ffill')
```

[69]:
```
                 AAL
2020-01-02  0.871209
2020-01-03  0.736818
2020-01-06  0.706020
2020-01-07  0.696687
2020-01-08  0.754550
…                …
2024-12-25 -0.224452
2024-12-26 -0.224452
2024-12-27 -0.224452
2024-12-30 -0.199253
2024-12-31 -0.216986

[1304 rows x 1 columns]
```

[83]:
```
# Function that creates train and test data through stock data and sequence␣
 ↪lenght
def loadData(stock, lookBack):
    data = []
    # Convert to numpy array
    dataRaw = stock.values

    # Create all possible sequences of length sequence length
    for i in range(len(dataRaw)-lookBack):
        data.append(dataRaw[i: i+lookBack])

    data = np.array(data)
    testSize = int(np.round(0.2*data.shape[0]))
    trainSize = data.shape[0]-testSize

    X_train = data[:trainSize,:-1,:]
    y_train = data[:trainSize,-1,:]
    X_test = data[trainSize:,:-1]
    y_test = data[trainSize:,-1,:]

    return X_train, X_test, y_train, y_test

# Sequence length:
lookBack = 20
X_train, X_test, y_train, y_test = loadData(dfAal, lookBack)
print("X_train and y_train shapes:", X_train.shape, y_train.shape)
print("X_test and y_test shapes:", X_test.shape, y_test.shape)
```

```
X_train and y_train shapes: (1027, 19, 1) (1027, 1)
```

```
X_test and y_test shapes: (257, 19, 1) (257, 1)
```

[84]:
```
# Training and test sets in Pytorch(turning into tensors)
X_train = torch.from_numpy(X_train).type(torch.Tensor)
X_test = torch.from_numpy(X_test).type(torch.Tensor)
y_train = torch.from_numpy(y_train).type(torch.Tensor)
y_test = torch.from_numpy(y_test).type(torch.Tensor)
```

[85]:
```
X_train.size(), y_train.size(), X_test.size(), y_test.size()
```

[85]:
```
(torch.Size([1027, 19, 1]),
 torch.Size([1027, 1]),
 torch.Size([257, 19, 1]),
 torch.Size([257, 1]))
```

[86]:
```
stepsNumb = lookBack-1
# Batch: number of training examples utilized in which(one) iteration of the␣
 ↪epochs
batchSize = 1606
# Epoch: the number of passes into the entire training dataset
numEpochs = 100

# Training and test dataset with torch:

train = torch.utils.data.TensorDataset(X_train,y_train)
test = torch.utils.data.TensorDataset(X_test,y_test)

trainLoader = torch.utils.data.DataLoader(dataset=train, batch_size=batchSize,␣
 ↪shuffle=False)
testLoader = torch.utils.data.DataLoader(dataset=test,␣
 ↪batch_size=batchSize,shuffle=False)
```

[87]:
```
# Build model

# Hyperparameters
inputDim = 1
hiddenDim = 32
numLayers = 2
outputDim = 1

# Defining the model as a class
# nn.Module: base class for all neural network modules, your models should also␣
 ↪subclass this class
class LSTM(nn.Module):
    def __init__(self, inputDim, hiddenDim, numLayers, outputDim):
        super(LSTM, self).__init__()
        # Hidden dimensions
```

```
        self.hiddenDim = hiddenDim
        # Number of hidden layers
        self.numLayers = numLayers

        # Building your LSTM
        # batch_first=True: if input and output tensors are provided as (batch,␣
↪seq, feature): the batch is the first access
        self.lstm = nn.LSTM(inputDim, hiddenDim, numLayers, batch_first=True)

        # Fully connected, Readout layer(parameters of the final non-recurrent␣
↪output layer)
        self.fc = nn.Linear(hiddenDim, outputDim)

    def forward(self, x):
        # Initialize hidden state with zeros
        # requires_grad_() = allows for fine grained exclusion of subgraphs␣
↪from gradient computation and can increase efficiency
        # Gradient: is another word for "slope"
        # x.size(0): number of examples sent into the batch size
        h0 = torch.zeros(self.numLayers, x.size(0), self.hiddenDim).
↪requires_grad_().to(device)

        # Initialize cell state
        # Cell state: horizontal line running through the top of the diagram.␣
↪It runs straight down the entire
        ## chain, with only some minor linear interactions. It's very easy for␣
↪information to just flow along it unchanged.
        c0 = torch.zeros(self.numLayers, x.size(0), self.hiddenDim).
↪requires_grad_().to(device)

        # One time step
        # We need to detach as we are doing truncated backpropagation through␣
↪time (BPTT)
        # If we don't, we'll backprop all the way to the start even after going␣
↪through another batch
        # datach(): returns a new Tensor, detached from the current graph. The␣
↪result will never require gradient
        out, (hiddenState, cellState) = self.lstm(x, (h0.detach(), c0.detach()))

        # Hidden state index of last time step
        # out.size() --> 100, 28, 100
        out = self.fc(out[:, -1, :]) # --> 100, 100

        # out.size() --> 100, 10
        return out
```

```
model = LSTM(inputDim=inputDim, hiddenDim=hiddenDim, numLayers=numLayers,␣
 ↪outputDim=outputDim).to(device)

# MSE: Creates a criterion that measures the mean squared error between each␣
 ↪element in the input x and target y
lossFn = torch.nn.MSELoss()

# Optimiser: will hold the current state and will update the parameters based␣
 ↪on the computed gradients
# lr: learning rate
optimiser = torch.optim.Adam(model.parameters(), lr=0.01)
print(model)
print(len(list(model.parameters())))
for i in range(len(list(model.parameters()))):
    print(list(model.parameters())[i].size())
```

```
LSTM(
  (lstm): LSTM(1, 32, num_layers=2, batch_first=True)
  (fc): Linear(in_features=32, out_features=1, bias=True)
)
10
torch.Size([128, 1])
torch.Size([128, 32])
torch.Size([128])
torch.Size([128])
torch.Size([128, 32])
torch.Size([128, 32])
torch.Size([128])
torch.Size([128])
torch.Size([1, 32])
torch.Size([1])
```

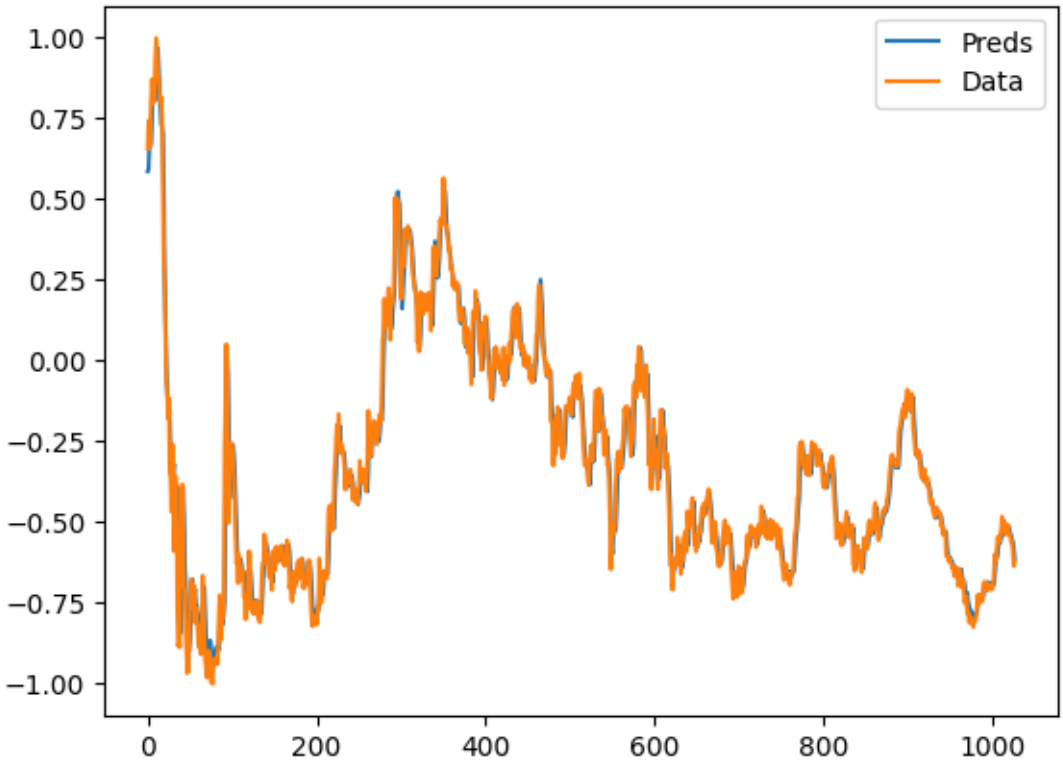
[88]:
```
# Train model

trainLoss = np.zeros(numEpochs)

# Number of steps to unroll
seqDim = lookBack-1

device = torch.device("cuda:0")

for e in range(numEpochs):
    # Forward pass
    yTrainPred = model(X_train.to(device))

    loss = lossFn(yTrainPred, y_train.to(device))
    if e % 10 == 0 and e != 0:
```

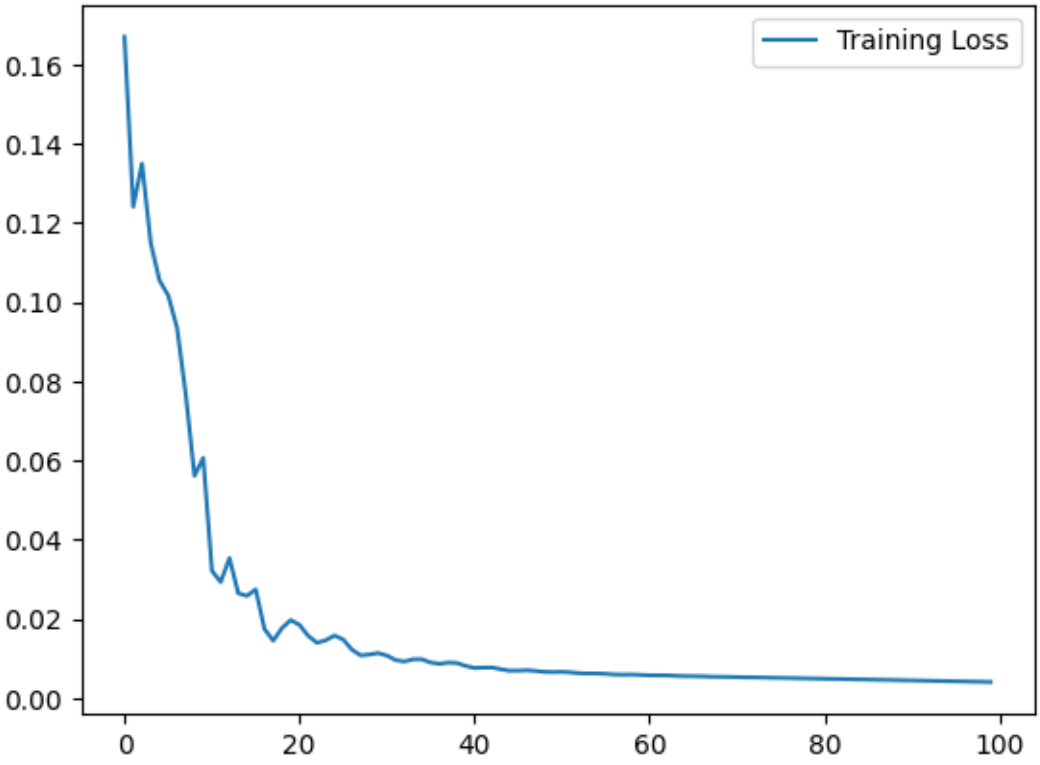
```
        print("Epoch: ", e, "MSE: ", loss.item())
    trainLoss[e] = loss.item()

    # Zero out gradient, else they will accumulate between epochs
    optimiser.zero_grad()

    # Backwark pass
    loss.backward()

    # Update parameters
    optimiser.step()
```

```
Epoch:  10 MSE:  0.03208833560347557
Epoch:  20 MSE:  0.018452443182468414
Epoch:  30 MSE:  0.010680662468075752
Epoch:  40 MSE:  0.0075613753870129585
Epoch:  50 MSE:  0.006579800974577665
Epoch:  60 MSE:  0.005701241549104452
Epoch:  70 MSE:  0.005250984337180853
Epoch:  80 MSE:  0.004841009620577097
Epoch:  90 MSE:  0.00441976310685277
```

[89]:
```
print("yTrainPred shape: ", np.shape(yTrainPred))

plt.plot(yTrainPred.detach().cpu().numpy(), label="Preds")
plt.plot(y_train.detach().cpu().numpy(), label="Data")
plt.legend()
plt.show()

plt.plot(trainLoss, label="Training Loss")
plt.legend()
plt.show()
```

```
yTrainPred shape:  torch.Size([1027, 1])
```

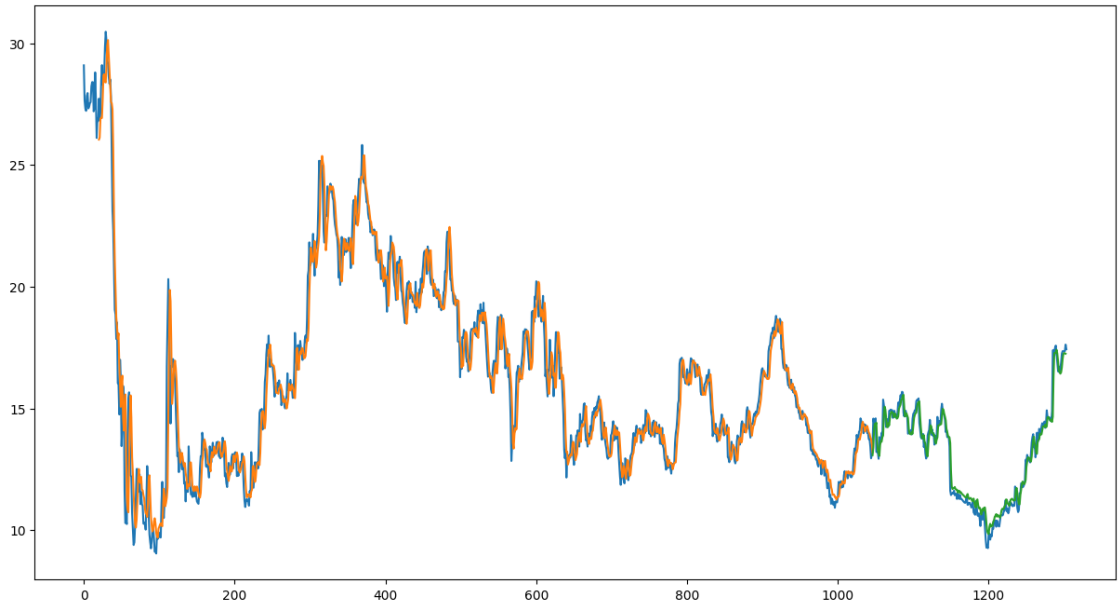

Preds
Data
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
200
400
600
800
1000


Training Loss
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
20
40
60
80
100

[90]:
```
# Make predictions
yTestPred = model(X_test.to(device))

# Invert predictions(scale back the data to the original representation)
yTrainPred = scaler.inverse_transform(yTrainPred.detach().cpu().numpy())
y_train = scaler.inverse_transform(y_train.detach().cpu().numpy())
yTestPred = scaler.inverse_transform(yTestPred.detach().cpu().numpy())
y_test = scaler.inverse_transform(y_test.detach().cpu().numpy())

# Calculate Root Mean Squared Error
trainScore = math.sqrt(mean_squared_error(yTrainPred[:,0], y_train[:,0]))
testScore = math.sqrt(mean_squared_error(yTestPred[:,0], y_test[:,0]))
print('Train Score: ', trainScore,'RMSE')
print('Test Score: ', testScore,'RMSE')
```

```
Train Score:  0.6795963796286784 RMSE
Test Score:  0.4266559121271588 RMSE
```

[91]:
```
# Shift train predictions for plotting
# np.empty_like: returns a new array with the same shape and type as a given␣
 ↪array
trainPredictPlot = np.empty_like(dfAal)
trainPredictPlot[:,:] = np.nan
trainPredictPlot[lookBack:len(yTrainPred)+lookBack,:] = yTrainPred

# Shift test predictions for plotting
testPredictPlot = np.empty_like(dfAal)
testPredictPlot[:,:] = np.nan
testPredictPlot[len(yTrainPred)+lookBack-1:len(dfAal)-1,:] = yTestPred

# Plot baseline and predictions
plt.figure(figsize=(15,8))
plt.plot(scaler.inverse_transform(dfAal))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```



[109]:
```python
def predict_multiple_steps(model, last_sequence, num_steps, device):
    """


    Args:
        model:   LSTM
        last_sequence:      (1, seq_len, 1)
        num_steps:
        device:

    Returns:
        predictions:
    """
    model.eval()
    predictions = []
    current_sequence = last_sequence.clone()

    with torch.no_grad():
        for _ in range(num_steps):
            #
            pred = model(current_sequence.to(device))
            predictions.append(pred.cpu().numpy())

            #
            current_sequence = torch.cat([
                current_sequence[:, 1:, :],  #
                pred.unsqueeze(1)            #
```

```
            ], dim=1)

    return np.array(predictions).reshape(-1, 1)
def enhanced_multi_step_prediction(model, X_test, num_future_steps, device,␣
 ↪scaler):
    """

    """
    model.eval()
    all_predictions = []

    #
    with torch.no_grad():
        for i in range(len(X_test)):
            sequence = torch.FloatTensor(X_test[i:i+1]).to(device)  #
            predictions = predict_multiple_steps(model, sequence,␣
 ↪num_future_steps, device)
            all_predictions.append(predictions)

    return np.array(all_predictions)

#
future_steps = 20
start_idx = 0
end_idx = len(X_test)-10  #
plt.figure(figsize=(15, 10))
plt.plot(scaler.inverse_transform(dfAal), label='Historical Data', alpha=0.7)
for j in range(start_idx,end_idx, 50):
    print(f"Predicting for sample {j+1}/{len(X_test)}...")
    future_preds = enhanced_multi_step_prediction(model, X_test[j:j+10],␣
 ↪future_steps, device, scaler)

    #

    for i in range(min(5, len(future_preds))):  #   5
        # start_idx = len(dfAal) - len(X_test) + i
        future_index = range(len(dfAal) - len(X_test)+j, len(dfAal) -␣
 ↪len(X_test)+j + future_steps)
        plt.plot(future_index, scaler.inverse_transform(future_preds[i]),
                 alpha=0.6, label=f'Prediction {i+1}')

plt.legend()
plt.title(f'Multiple {future_steps}-Step Predictions')
plt.show()
```

```
Predicting for sample 1/257…
Predicting for sample 51/257…
```

```
Predicting for sample 101/257…
Predicting for sample 151/257…
Predicting for sample 201/257…
```