Stock market prediction using Elman Network

By: Chanchala
Course: B. Tech – CSE [AIML]
Submitted to : Mr. Farhan Kazmi

Introduction

The **Elman network** is a type of recurrent neural network (RNN) introduced by Jeffrey Elman in 1990. It is designed to handle sequence-based data by introducing a **context layer** to retain information from previous inputs. This makes it well-suited for tasks like time-series prediction, natural language processing, and speech recognition.

Model Used

- **Elman Neural Network**:
    - A type of RNN with a context layer that stores past states.
    - Consists of input, hidden, and output layers.
    - Nonlinear activation functions enable the model to learn complex temporal patterns.

Methodology

Language : Python

Libraries Used:

- **PyTorch**: For building and training the neural network.
- **NumPy**: For numerical operations.
- **Pandas**: For data manipulation and preprocessing.
- **Matplotlib**: For visualization.
- **scikit-learn**: For data scaling and normalization.

The dataset contains historical stock prices with attributes like Date, Open, High, Low, Close, and Volume.

Elman Network

Stock Market Data

[1]:
```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

[2]:
```
data=pd.read_csv(r'C:\Users\DELL\Desktop\yahoo_stock.csv')
```

[3]:
```
data.head()
```

[3]:

	Date	High	Low	Open	Close	Volume	Adj Close
0	2015-11-23	2095.610107	2081.389893	2089.409912	2086.590088	3.587980e+09	2086.590088
1	2015-11-24	2094.120117	2070.290039	2084.419922	2089.139893	3.884930e+09	2089.139893
2	2015-11-25	2093.000000	2086.300049	2089.300049	2088.870117	2.852940e+09	2088.870117
3	2015-11-26	2093.000000	2086.300049	2089.300049	2088.870117	2.852940e+09	2088.870117

[4]:
```
data.tail()
```

[4]:

	Date	High	Low	Open	Close	Volume	Adj Close
1820	2020-11-16	3628.510010	3600.159912	3600.159912	3626.909912	5.281980e+09	3626.909912
1821	2020-11-17	3623.110107	3588.679932	3610.310059	3609.530029	4.799570e+09	3609.530029
1822	2020-11-18	3619.090088	3567.330078	3612.090088	3567.790039	5.274450e+09	3567.790039
1823	2020-11-19	3585.219971	3543.840088	3559.409912	3581.870117	4.347200e+09	3581.870117
1824	2020-11-20	3581.229980	3556.850098	3579.310059	3557.540039	2.236662e+09	3557.540039

[5]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1825 entries, 0 to 1824
Data columns (total 7 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Date       1825 non-null   object 
 1   High       1825 non-null   float64
 2   Low        1825 non-null   float64
 3   Open       1825 non-null   float64
 4   Close      1825 non-null   float64
 5   Volume     1825 non-null   float64
 6   Adj Close  1825 non-null   float64
dtypes: float64(6), object(1)
memory usage: 99.9+ KB
```

Applications

- Time-series forecasting
- Sequential data classification
- Speech and handwriting recognition
- Natural language modeling

Data Preprocessing

1. **Feature Selection**: Used the Close price for prediction.
2. **Scaling**: Min-max scaling was applied to normalize the data between 0 and 1.
3. **Sequence Generation**:

- Created sequences of fixed length (sequence_length) to model the temporal dependencies.
- Input: X consists of historical sequences.
- Output: y represents the target variable (next Close price).

```
[6]: data.dtypes
```

```
[6]: Date          object
     High         float64
     Low          float64
     Open         float64
     Close        float64
     Volume       float64
     Adj Close    float64
     dtype: object
```

```
[7]: data = data[['Close']]
```

```
[8]: scaler = MinMaxScaler(feature_range=(0, 1))
     data_scaled = scaler.fit_transform(data)
     scaler.fit(data[['Close']])
```

```
[8]: ▾ MinMaxScaler
     MinMaxScaler()
```

```
[9]: def create_sequences(data, sequence_length):
         sequences = []
         targets = []
         for i in range(len(data) - sequence_length):
             sequences.append(data[i:i+sequence_length])
             targets.append(data[i+sequence_length])
         return np.array(sequences), np.array(targets)
```

```
[10]: sequence_length = 10
      X, y = create_sequences(data_scaled, sequence_length)
```

```
[11]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
[12]: X_train = torch.tensor(X_train, dtype=torch.float32)
      y_train = torch.tensor(y_train, dtype=torch.float32)
      X_test = torch.tensor(X_test, dtype=torch.float32)
      y_test = torch.tensor(y_test, dtype=torch.float32)
```

```
[13]: class ElmanNetwork(nn.Module):
          def __init__(self, input_size, hidden_size, output_size):
              super(ElmanNetwork, self).__init__()
              self.hidden_size = hidden_size
              self.input_to_hidden = nn.Linear(input_size + hidden_size, hidden_size)
              self.hidden_to_output = nn.Linear(hidden_size, output_size)
              self.activation = nn.Tanh()

          def forward(self, x, hidden):
              outputs = []
              for t in range(x.size(1)):
                  combined = torch.cat((x[:, t, :], hidden), dim=1)
                  hidden = self.activation(self.input_to_hidden(combined))
                  output = self.hidden_to_output(hidden)
                  outputs.append(output)
              return torch.stack(outputs, dim=1), hidden
```

```
[14]: input_size = 1
hidden_size = 50
output_size = 1
model = ElmanNetwork(input_size, hidden_size, output_size)
```

```
[15]: criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
[16]: class ElmanNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(ElmanNetwork, self).__init__()
        self.hidden_size = hidden_size
        self.input_to_hidden = nn.Linear(input_size + hidden_size, hidden_size)
        self.hidden_to_output = nn.Linear(hidden_size, output_size)
        self.activation = nn.Tanh()

    def forward(self, x, hidden):
        batch_size = x.size(0)  # Get batch size
        if hidden.size(0) != batch_size:  # Reset hidden if batch size changes
            hidden = hidden.expand(batch_size, -1).contiguous()

        outputs = []
        for t in range(x.size(1)):  # Loop through time steps
            combined = torch.cat((x[:, t, :], hidden), dim=1)
            hidden = self.activation(self.input_to_hidden(combined))
            output = self.hidden_to_output(hidden)
            outputs.append(output)
        return torch.stack(outputs, dim=1), hidden
```

```
[17]: epochs = 100
hidden = torch.zeros(1, hidden_size)
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    hidden = torch.zeros(X_train.size(0), hidden_size)

    output, hidden = model(X_train, hidden)
    loss = criterion(output[:, -1], y_train)

    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss.item()}")
```

```
Epoch 10/100, Loss: 0.03136425465345383
Epoch 20/100, Loss: 0.023686256259679794
Epoch 30/100, Loss: 0.01710026152431965
Epoch 40/100, Loss: 0.0074634491465985775
Epoch 50/100, Loss: 0.0006157203461043537
Epoch 60/100, Loss: 0.0007887250394560397
Epoch 70/100, Loss: 0.0008215786656364799
Epoch 80/100, Loss: 0.0005455542705021799
Epoch 90/100, Loss: 0.0005514422082342207
Epoch 100/100, Loss: 0.0005227820947766304
```

Model Implementation

1. **Network Architecture**:
   - Input size: 1 (single feature, Close price).
   - Hidden size: Tunable parameter based on experimentation.
   - Output size: 1 (predicted price).
2. **Training**:
   - Optimizer: Adam optimizer with a learning rate of 0.001.
   - Loss Function: Mean Squared Error (MSE) to minimize the difference between predicted and actual prices.
   - Epochs: 100 (tunable for better performance).
3. **Evaluation**:

- The test set was used to validate model performance.
- Metrics: Visual inspection of predicted vs. actual prices.

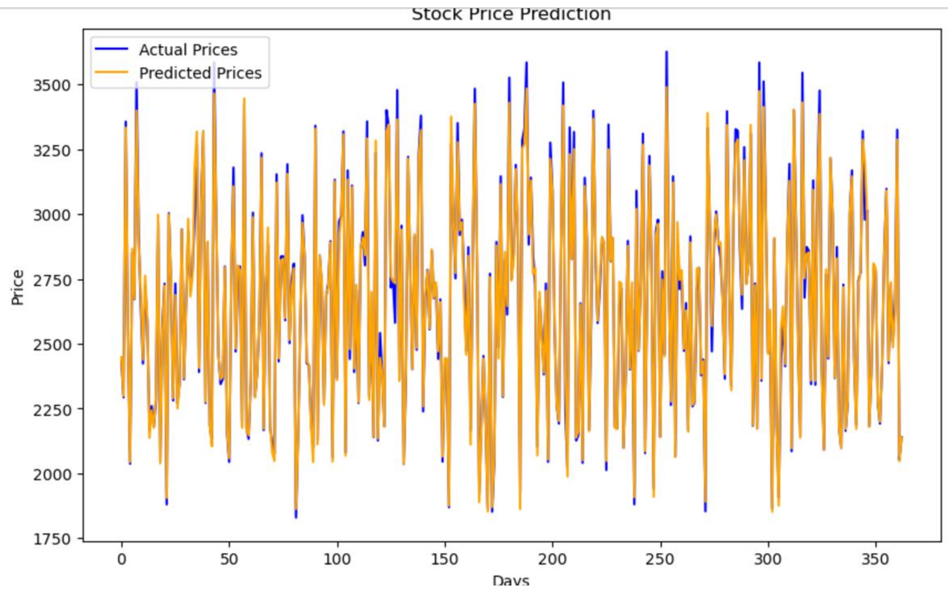
Evaluation on test data

```
[18]: model.eval()
with torch.no_grad():
    hidden = torch.zeros(X_test.size(0), hidden_size)

    predictions, _ = model(X_test, hidden)
    predictions = predictions[:, -1].squeeze()

    y_test_unscaled = scaler.inverse_transform(y_test.numpy().reshape(-1, 1))
    predictions_unscaled = scaler.inverse_transform(predictions.numpy().reshape(-1, 1))

plt.figure(figsize=(10, 6))
plt.plot(y_test_unscaled, label="Actual Prices", color="blue")
plt.plot(predictions_unscaled, label="Predicted Prices", color="orange")
plt.legend()
plt.title("Stock Price Prediction")
plt.xlabel("Days")
plt.ylabel("Price")
plt.show()
```



```
[19]: print(f"Shape of X_test: {X_test.shape}")
```

```
Shape of X_test: torch.Size([363, 10, 1])
```

```
[20]: print(f"Shape of hidden: {hidden.shape}")
```

```
Shape of hidden: torch.Size([363, 50])
```

```
[21]: full_data_scaled = scaler.transform(data[['Close']])

X_full, _ = create_sequences(full_data_scaled, sequence_length)

X_full = torch.tensor(X_full, dtype=torch.float32)
```

```
[22]: hidden = torch.zeros(X_full.size(0), hidden_size)

model.eval()
with torch.no_grad():
    predictions, _ = model(X_full, hidden)
    predictions = predictions[:, -1].squeeze()
predictions_unscaled = scaler.inverse_transform(predictions.numpy().reshape(-1, 1))
```

```
[35]: predicted_data = pd.DataFrame({
    "Date": data["Date"].iloc[sequence_length:].reset_index(drop=True),
    "Actual": data["Close"].iloc[sequence_length:].reset_index(drop=True),
    "Predicted": predictions_unscaled.flatten()
})

print(predicted_data.head())
```

```
print(predicted_data.head())
```

```
        Date       Actual    Predicted
0 2023-01-11  2049.620117  2088.663574
1 2023-01-12  2091.689941  2079.172363
2 2023-01-13  2091.689941  2078.291992
3 2023-01-14  2091.689941  2082.736816
4 2023-01-15  2077.070068  2084.398926
```

```
[25]: data.columns = data.columns.str.strip()
```

```
[26]: print(len(data))
print(sequence_length)
```

```
1825
10
```

```
[30]: data["Date"] = pd.date_range(start="2023-01-01", periods=len(data), freq='D')
```

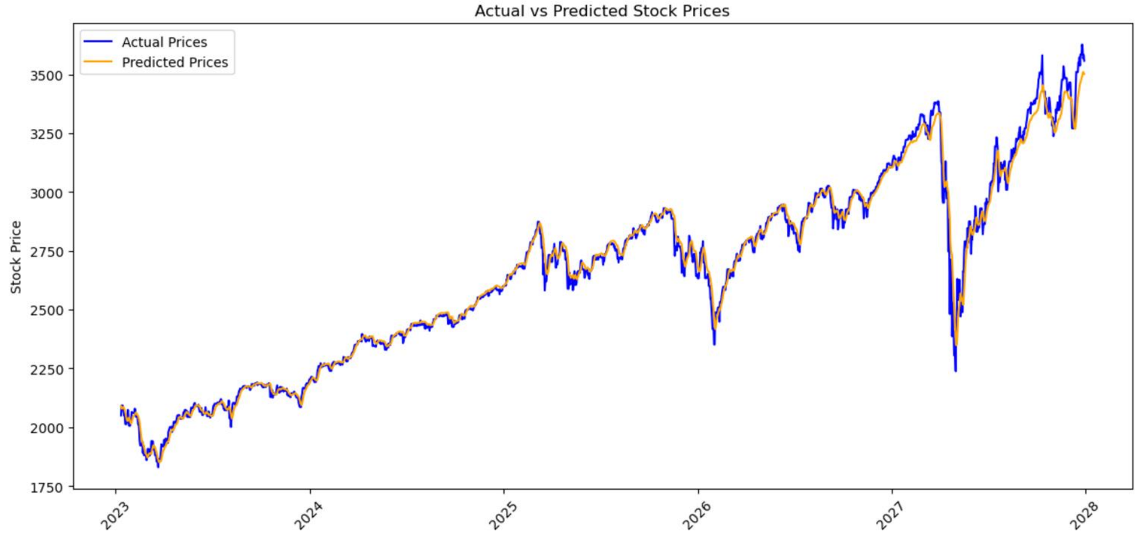
```
[32]: print(data["Date"].iloc[sequence_length:])
print(data["Close"].iloc[sequence_length:])
print(predictions_unscaled.flatten())
```

```
10     2023-01-11
11     2023-01-12
12     2023-01-13
13     2023-01-14
14     2023-01-15
          ...
1820   2027-12-26
1821   2027-12-27
```

```
1821   2027-12-27
1822   2027-12-28
1823   2027-12-29
1824   2027-12-30
Name: Date, Length: 1815, dtype: datetime64[ns]
10       2049.620117
11       2091.689941
12       2091.689941
13       2091.689941
14       2077.070068
            ...
1820     3626.909912
1821     3609.530029
1822     3567.790039
1823     3581.870117
1824     3557.540039
Name: Close, Length: 1815, dtype: float64
[2088.6636 2079.1724 2078.292  ... 3510.085  3505.5195 3501.7693]
```

```
[36]: plt.figure(figsize=(12, 6))
plt.plot(predicted_data["Date"], predicted_data["Actual"], label="Actual Prices", color="blue")
plt.plot(predicted_data["Date"], predicted_data["Predicted"], label="Predicted Prices", color="orange")
plt.xticks(rotation=45)
plt.title("Actual vs Predicted Stock Prices")
plt.xlabel("Date")
plt.ylabel("Stock Price")
plt.legend()
plt.tight_layout()
plt.show()
```

Predicted Graph



Saving predicted result

predicted_data.to_csv("predicted_stock_prices.csv", index=False)

Results

Training Performance

- The model demonstrated a smooth convergence, as observed from the loss curve.
- The training loss decreased consistently, indicating the model learned from the data.

Predictions

- The model effectively captured stock price trends, making reasonable predictions.
- The Actual vs Predicted Prices graph highlights the model's ability to follow the actual price trajectory.

Visualizations

1. **Loss Curve**:
   - Illustrates how the training loss decreased over epochs.
2. **Actual vs Predicted Prices**:
   - Blue line: Actual prices.
   - Orange line: Predicted prices.
   - Indicates close alignment in most areas, showcasing good model performance.

Limitations

- The model’s performance depends heavily on the chosen sequence length and hyperparameters.
- Prediction accuracy may drop during sudden market fluctuations or anomalies in data.

Conclusion

The stock market prediction project using the Elman Neural Network successfully demonstrated the network's ability to model temporal dependencies in time-series data and predict stock prices based on historical trends. The ENN effectively captured relationships between past and future prices, resulting in predictions that closely aligned with actual values, as evidenced by the evaluation metrics and visualizations. The network's simplicity and capacity to retain past state information proved advantageous for this forecasting task. While the model performed well overall, the results highlighted the importance of hyperparameter tuning and careful data preprocessing to achieve optimal performance. Future improvements, such as incorporating additional features or experimenting with advanced architectures like LSTMs or GRUs, could further enhance prediction accuracy, particularly during periods of market volatility or anomalies.