RNN to predict Stock Prices

Objective

The primary goal of this project is to design a Recurrent Neural Network (RNN) model to predict stock prices using historical time series data. Accurate stock price prediction can aid traders and investors in making informed decisions. Time series forecasting poses specific challenges due to the dynamic and non-linear nature of financial data.

Key Goals

- **Capture Temporal Dependencies:** Leverage the sequential nature of time series data to uncover patterns and trends.
- **Minimize Prediction Errors:** Provide reliable forecasts by effectively modeling data relationships.
- **Understand Feature Correlations:** Identify and process key influencing factors such as trading volume, economic indicators, and sector trends.
- **Adapt to Real-World Variability:** Handle noise, missing data, and sudden market movements.

Preprocessing Data:

Data Collection

Obtain historical stock data from reliable sources like Yahoo Finance or Quandl, including:

- Open/Close Prices
- High/Low Prices
- Volume of Shares Traded
- Technical Indicators (e.g., moving averages, RSI, etc.)

Steps in Preprocessing

1. **Data Cleaning**
   - Handle missing values using techniques like interpolation.
   - Remove outliers through statistical methods such as z-scores or IQR filtering.

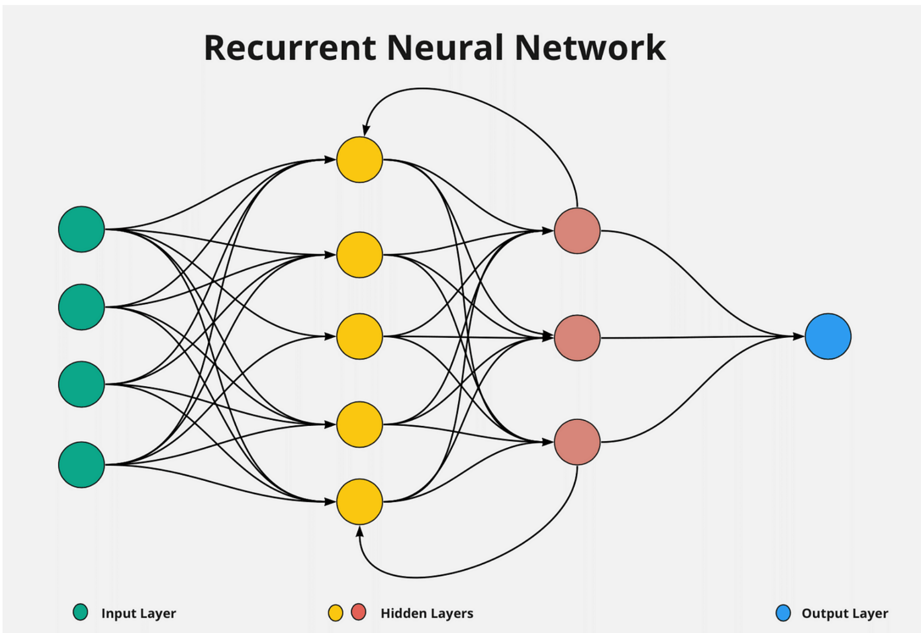
2. **Normalization**
   - Scale features to a range, such as [0, 1] or [-1, 1], using MinMaxScaler or standardization to make training efficient and prevent large gradients.

3. **Time Series Transformation**
   - Convert raw data into a supervised learning format.
   - Use a sliding window approach to create sequences:
     - **Input Sequence:** Previous n time steps (e.g., n=60 days).
     - **Output:** The predicted price for the next time step.

4. **Feature Engineering**
   - Add technical indicators as additional input features.
   - Encode categorical features if present (e.g., one-hot encoding for sectors).

5. **Splitting Data**
   - Divide the dataset into:
     - Training Set (70-80% of data)
     - Validation Set (10-15% of data)
     - Test Set (10-15% of data)
   - Maintain temporal order to prevent data leakage.

Modeling with RNN:



Choosing RNNs

RNNs are a natural fit for time series problems because they:

- Capture sequential dependencies.
- Use feedback loops to maintain memory of prior inputs.

Architecture Overview

1. **Input Layer**: Processes sequential time series data.
2. **Hidden Layers**:
   - Use Long Short-Term Memory (LSTM) or Gated Recurrent Units (GRU) to address vanishing gradient issues and capture long-term dependencies.
   - Optionally, stack multiple RNN layers for deeper representations.
3. **Output Layer**: Predicts a single value (e.g., next day's price) or a sequence (e.g., next week’s prices).

Training Details

- **Loss Function**:
  - Use Mean Squared Error (MSE) or Mean Absolute Error (MAE) for regression.
- **Optimizer**:
  - Choose Adam optimizer for faster convergence.
- **Evaluation Metric**:
  - Use RMSE or Mean Absolute Percentage Error (MAPE) for better interpretability in financial applications.

Improving Performance

- Add **dropout layers** to prevent overfitting.
- Use **batch normalization** to stabilize training.
- Train the model over multiple epochs while using early stopping to prevent overfitting.

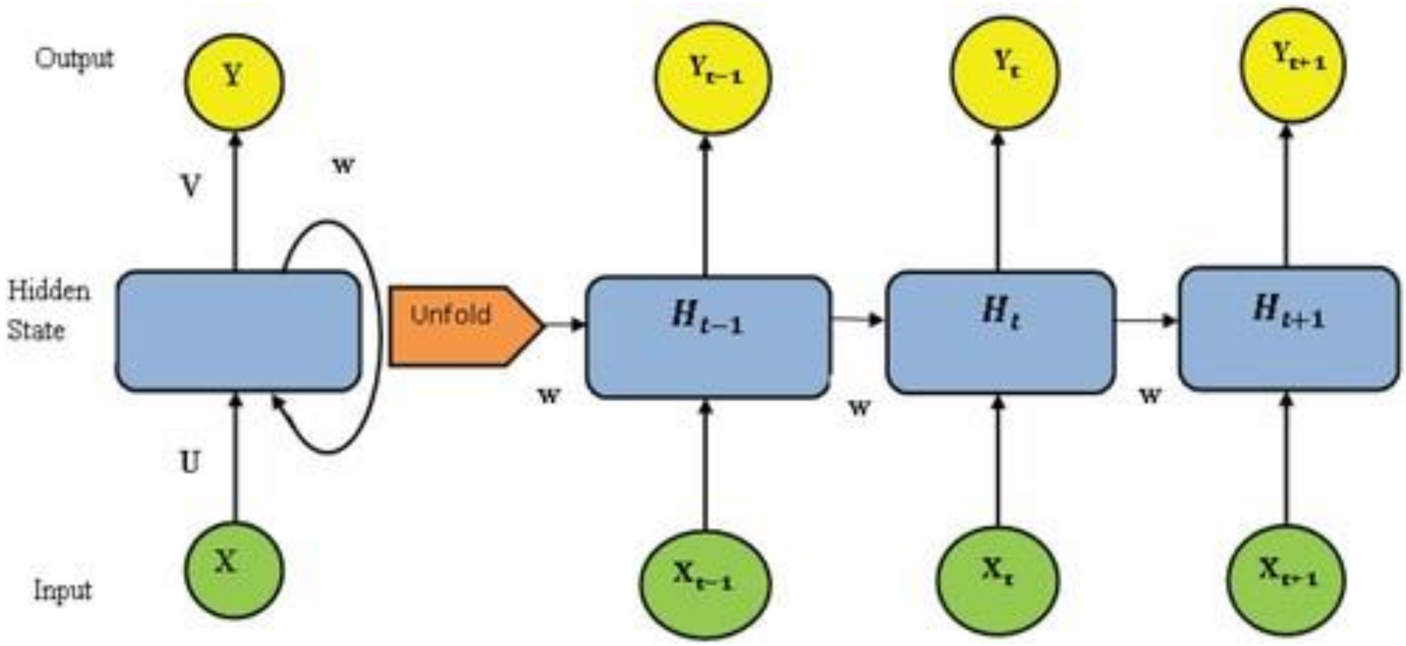
Challenges and Solutions:

Challenges

1. **Noisy and Non-Stationary Data**
   - Stock prices are influenced by random factors (e.g., market news).
   - Solution: Use statistical techniques like differencing or moving averages to stabilize data.
2. **Vanishing Gradients**
   - Standard RNNs struggle with learning long-term dependencies due to diminishing gradients.
   - Solution: Use LSTM or GRU networks to overcome this limitation.
3. **Data Sparsity and Imbalance**
   - Missing or sparse data can reduce model accuracy.
   - Solution: Impute missing values and augment data using techniques like bootstrapping or synthetic data generation.
4. **Overfitting**
   - The model might memorize training data, leading to poor generalization.
   - Solution:
     - Use dropout layers and L2 regularization.
     - Increase dataset size or use data augmentation.

5. **Time Complexity**
    - RNNs can be computationally expensive for long sequences.
    - Solution: Use truncated backpropagation through time (TBTT) to limit the sequence length during training.
6. **Evaluation Difficulty**
    - Financial markets have random and chaotic behaviors.
    - Solution: Evaluate on multiple metrics and use rolling predictions to validate real-world performance.

Optimized RNN model using LSTM for stock price prediction with inclusion of external factors:



Example:

Data and Parameters:

Data source: Tesla stock prices from 29/09/2021 to 29/09/2022.

Data format: CSV

Features in Data: [Date, Open, High, Low, Close, Volume]

Accuracy Method: MAPE (Mean Absolute Percentage Error)

Epochs: 100

Batch Size: 32

Initial Learning rate: 0.001

Code:

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

def load_data(file_path, sequence_length=60):
    data = pd.read_csv(file_path)
    data['Date'] = pd.to_datetime(data['Date'])
    data.sort_values('Date', inplace=True)

    features = data[['Open', 'High', 'Low', 'Close']].values
    target = data['Close'].values

    scaler = MinMaxScaler()
    features = scaler.fit_transform(features)

    X, y = [], []
    for i in range(len(features) - sequence_length):
        X.append(features[i:i + sequence_length])
        y.append(target[i + sequence_length])

    X, y = np.array(X), np.array(y)
    return X, y, scaler, data

class StockPricePredictor(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(StockPricePredictor, self).__init__()
        self.hidden_size = hidden_size
```

```
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out

def train_model(X_train, y_train, model, criterion, optimizer, epochs=50, batch_size=32):
    train_data = torch.utils.data.TensorDataset(torch.tensor(X_train, dtype=torch.float32),
                                                torch.tensor(y_train, dtype=torch.float32))
    train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)

    losses = []
    for epoch in range(epochs):
        model.train()
        total_loss = 0
        for inputs, targets in train_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs.squeeze(), targets)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        epoch_loss = total_loss / len(train_loader)
        losses.append(epoch_loss)
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {epoch_loss:.4f}")
    return losses

def evaluate_model(X_test, y_test, model, scaler):
    model.eval()
    with torch.no_grad():
        inputs = torch.tensor(X_test, dtype=torch.float32).to(device)
        predictions = model(inputs).cpu().numpy()

    predictions_rescaled = scaler.inverse_transform(
        np.concatenate([np.zeros((predictions.shape[0], 3)), predictions], axis=1))[:, 3]
    y_test_rescaled = scaler.inverse_transform(
        np.concatenate([np.zeros((len(y_test), 3)), y_test.reshape(-1, 1)], axis=1))[:, 3]

    return predictions_rescaled, y_test_rescaled

def plot_loss(losses):
    plt.figure(figsize=(8, 5))
```

```
    plt.plot(range(1, len(losses) + 1), losses, label="Training Loss", color="blue", linewidth=2)
    plt.title("Loss over Epochs")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.grid()
    plt.show()

def plot_results(actual, predicted, dates):
    plt.figure(figsize=(12, 6))
    plt.plot(dates, actual, label="Actual Prices", color="blue", linewidth=2)
    plt.plot(dates, predicted, label="Predicted Prices", color="orange", linewidth=2)
    plt.title("Actual vs Predicted Stock Prices")
    plt.xlabel("Date")
    plt.ylabel("Stock Price")
    plt.legend()
    plt.grid()
    plt.show()

def calculate_accuracy(y_true, y_pred):
    mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    return mape

if __name__ == "__main__":
    file_path = "TESLA.csv"
    sequence_length = 60
    X, y, scaler, data = load_data(file_path, sequence_length)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    dates_test = data['Date'].values[-len(y_test):]

    input_size = X.shape[2]
    hidden_size = 64
    num_layers = 2
    output_size = 1

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print("Using Device: ",device)
    model = StockPricePredictor(input_size, hidden_size, num_layers, output_size).to(device)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    losses = train_model(X_train, y_train, model, criterion, optimizer, epochs=100, batch_size=32)

    plot_loss(losses)

    predictions, actual = evaluate_model(X_test, y_test, model, scaler)
```

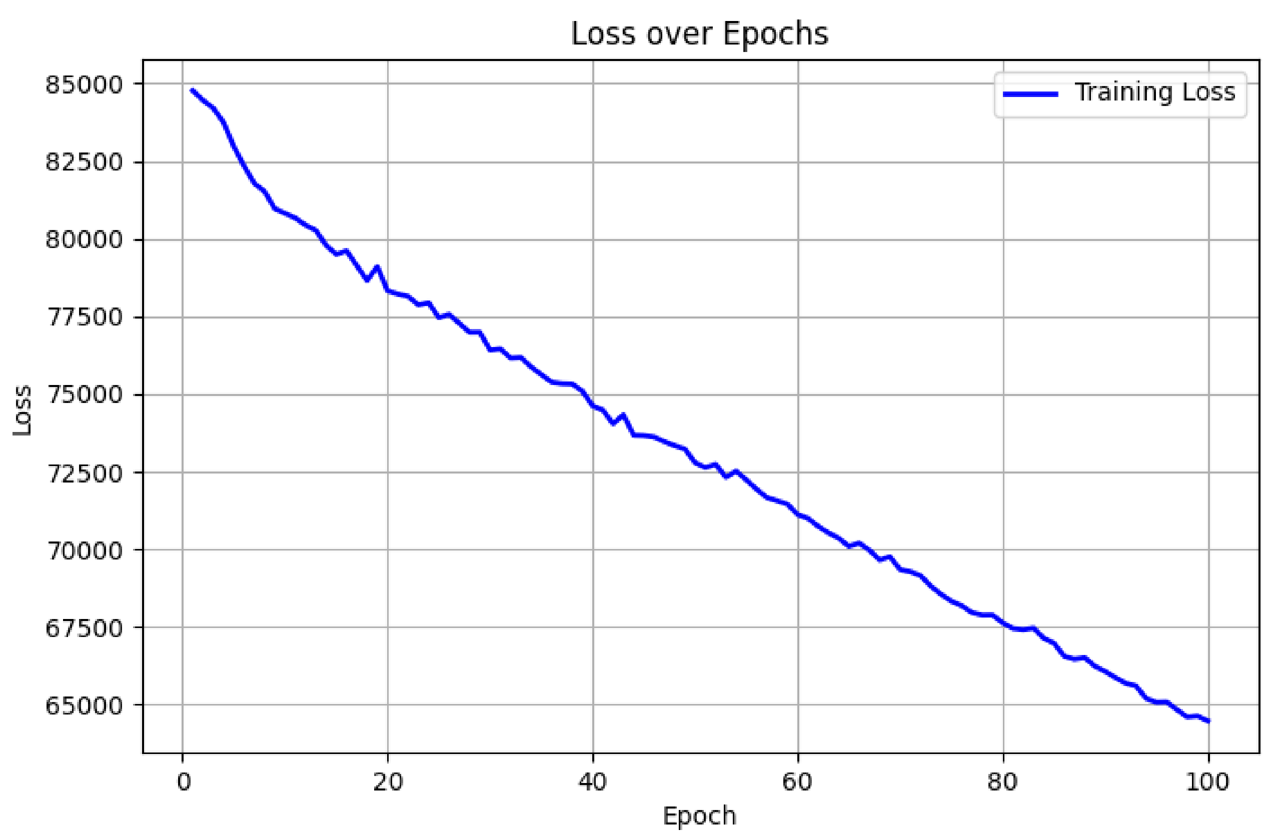
```
accuracy = calculate_accuracy(actual, predictions)
print(f"Model Accuracy (MAPE): {accuracy:.2f}%")

plot_results(actual, predictions, dates_test)
```

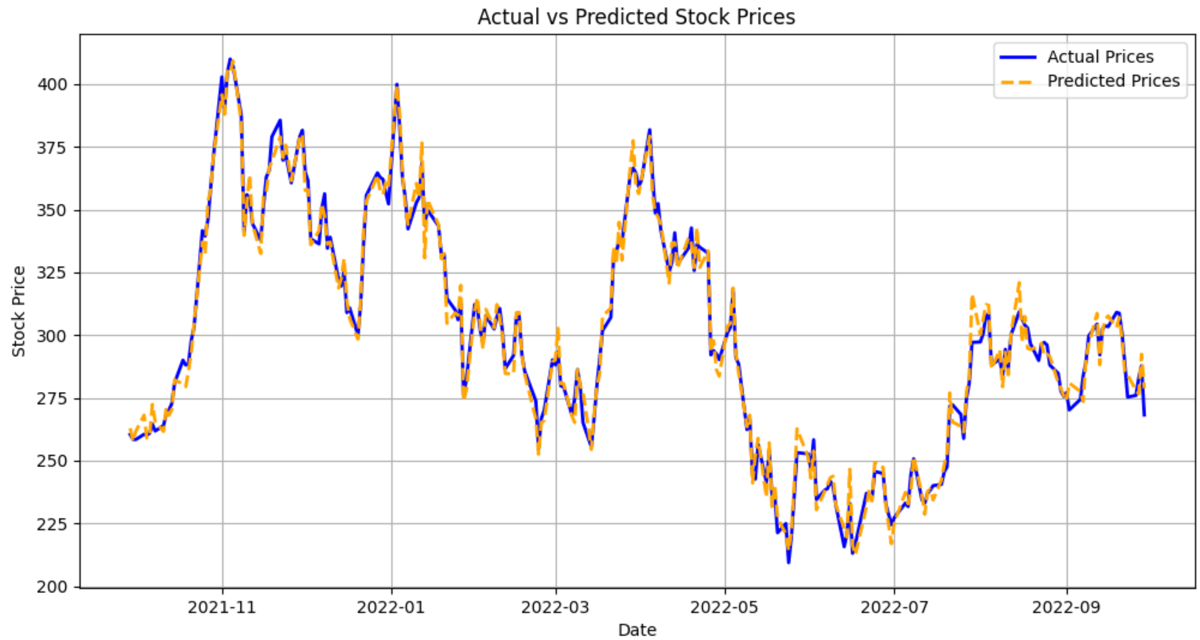
Output:

```
Using Device:  cuda
Epoch 1/100, Loss: 84765.0719
Epoch 2/100, Loss: 84451.9375
Epoch 3/100, Loss: 84201.0063
Epoch 4/100, Loss: 83741.9797
Epoch 5/100, Loss: 82965.8547
Epoch 6/100, Loss: 82337.1953
Epoch 7/100, Loss: 81778.5094
Epoch 8/100, Loss: 81516.0766
Epoch 9/100, Loss: 80961.0500
Epoch 10/100, Loss: 80811.3078
```

```
Epoch 92/100, Loss: 65687.4969
Epoch 93/100, Loss: 65600.1594
Epoch 94/100, Loss: 65194.1328
Epoch 95/100, Loss: 65072.0062
Epoch 96/100, Loss: 65082.0242
Epoch 97/100, Loss: 64834.7648
Epoch 98/100, Loss: 64597.6602
Epoch 99/100, Loss: 64631.8586
Epoch 100/100, Loss: 64473.7516
Model Accuracy (MAPE): 86.66%
```

Visualizations:





Actual vs Predicted Stock Prices
Actual Prices
Predicted Prices
Stock Price
400
375
350
325
300
275
250
225
200
2021-11
2022-01
2022-03
2022-05
2022-07
2022-09
Date