The Stock Picker, fully annotated and explained

Ethan Lin and Ayan Bhardwaj

Link:
https://docs.google.com/document/d/1fxYNXhhcR8C7DAETdtO7fXcMMqNip5RFBgUCFUb6oIU/edit?usp=sharing

Before we begin, we must ensure that all required libraries are installed. We'll use y-finance for data fetching and PyTorch for building the LSTM model. Additionally, we'll use scikit-learn for data preprocessing and evaluation.

```
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.preprocessing import RobustScaler

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

from scipy.optimize import minimize

# For reproducibility
import random
import os
```

Why are libraries important?

Short definition: A coding library is a reusable collection of prewritten code (functions, classes, constants, utilities) that a developer can call from their own program to avoid rewriting common logic.

Libraries sit between low-level language primitives and full applications. They package tested functionality into a convenient API so you can focus on your problem rather than reinventing basic building blocks.

What are the libraries we've used?

“yfinance” is the library for Yahoo Finance. It is this library that allows us to pull data about certain tickers, though it tends to break whenever too many inputs are requested. There are several other libraries that have this function, but we decided to use this one because it is free.

“Pandas” is a python library which allows us to clean, organize, and manipulate the data we pull from finance. It offers several functions which allow us to put data into matrices, and other operations which allow us to draw useful conclusions about said data.
“Numpy” is similar to pandas. It provides support for large, multi-dimensional arrays and matrices, along with a vast collection of high-level mathematical functions to operate on these arrays efficiently. Its ndarray object is an NxN matrix that can store a variety of data, useful for storing the variety of ticker data we are going to pull.

“Matplotlib.pyplot” is a library that allows us to visualize said data. We use a different visualization method which gets uploaded to our frontend, but whilst building the code this was very helpful to see if our code was working, and we left it in there.

“Scikit-learn” and “pytorch” allows us to do said machine learning, the backbone of our model. scikit-learn helps us to prepare our data for machine learning algorithms, and implement said algorithms. Pytorch is a machine learning framework that facilitates the development and training of deep learning models. We will cover their role more in-depth as we explain the machine learning model.

“Random” is the library that allows us to generate random numbers, and “os” allows us to perform file system operations.

For reproducibility, we'll add this code.

```
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
set_seed()
```

What does this do?

We're going to use an ML model to determine the first weights we use in the network we build. Weights are the strength of connection between neurons, and neurons are the cells that do the calculations. The weights help determine the strength of signals that appear throughout the network, which may affect our predictions greatly as we run the network.

If our seed is always different, we may get different responses from the exact same code every time it runs. By setting a seed, the randomness of our initial weights (and other factors such as train–test splits) remains the same, allowing us to tweak our model as needed, and produce the same results every time.

Now, we'll fetch historical stock data for selected tickers using the yfinance package.

```
def get_stock_data(ticker, start_date='2010-01-01'):
    stock = yf.Ticker(ticker)
    stock_data = stock.history(start=start_date)
    return stock_data

tickers = [ticker]

data = {ticker: get_stock_data(ticker) for ticker in tickers}
```

What does this do?

These lines of code essentially get us the stock data we need in order to input into our model. The input that the user types in our website gets fed to our model, which pulls the data (open, close) of said ticker into the code. Later, this data will be enhanced.

Now, we'll enhance the dataset by adding key financial ratios and technical indicators:

```
def calculate_ratios(stock_data, ticker):

    stock = yf.Ticker(ticker)
    info = stock.info



    stock_data['P/E'] = info.get("trailingPE", pd.NA)  # trailing P/E
    stock_data['ROE'] = info.get("returnOnEquity", pd.NA)  # ROE as
decimal
    stock_data['Debt-to-Equity'] = info.get("debtToEquity", pd.NA)  #
Debt-to-Equity



    stock_data['P/E'] = stock_data['P/E'].clip(lower=0, upper=50)
    stock_data['ROE'] = stock_data['ROE'] * 100  # convert to percentage

    return stock_data

def add_technical_indicators(stock_data):

    stock_data['12-day EMA'] = stock_data['Close'].ewm(span=12,
adjust=False).mean()
    stock_data['26-day EMA'] = stock_data['Close'].ewm(span=26,
adjust=False).mean()
    stock_data['MACD'] = stock_data['12-day EMA'] - stock_data['26-day
EMA']



    delta = stock_data['Close'].diff()
    gain = delta.where(delta > 0, 0)
    loss = -delta.where(delta < 0, 0)
    avg_gain = gain.rolling(window=14, min_periods=1).mean()
    avg_loss = loss.rolling(window=14, min_periods=1).mean()
    rs = avg_gain / avg_loss
    stock_data['RSI'] = 100 - (100 / (1 + rs))
    stock_data['RSI'] = stock_data['RSI'].clip(lower=0, upper=100)

    return stock_data
```

```
for ticker in tickers:
    stock_data = data[ticker]
    stock_data = calculate_ratios(stock_data, ticker)
    data[ticker] = add_technical_indicators(stock_data)
```

Feature Engineering, Ratios, and more

Feature engineering is the process of creating and preparing data from existing data to help our algorithms make more accurate and detailed predictions. The structured, meaningful format helps it learn quicker. This can involve getting BMI from "height" and "weight", or price per square foot from "price" and "square footage"

IBM actually has an amazing explanation on this, and they've been extremely helpful in our journey of understanding (and for Ayan, coding) ML processes.

https://www.ibm.com/think/topics/feature-engineering

Here are some fundamental ratios we pull. We will use feature engineering for technical indicators.

P/E is calculated by taking the price of the targeted share, and dividing it by the Earnings Per Share. EPS is calculated by taking the difference between Net Income and Dividends, and dividing that by the number of shares outstanding.

Earnings per share tells us basically, how much money is actually being earned by each share someone holds. And by taking the ratio of the price of the stock with EPS, we get a good idea of how much the market prices the value of each stock relative to its actual earnings. This ratio helps us determine if a stock is being overvalued, undervalued, or fairly priced.

In order to help train our model more effectively, we cap extreme values to 50. Our ML model trains itself based on the data it sees, and extreme values tend to not be representative of the average P/E, even during market shifts where it would normally be higher. If we were to let these values through, the gradients (the functions that tell the model how to change their weights) we would have may shift unpredictably, and hinder the accuracy of our model.

ROE (Return on Equity) is calculated by taking the company's Net Income and dividing it by Shareholders' Equity. This helps us understand how well a company's capital investments from shareholders are being invested, or how much money every dollar invested is generating.

A higher ROE generally means the company is good at turning invested money into earnings, while a lower ROE could suggest inefficiency. This ratio helps investors understand the company's ability to create value for its owners.

Debt-to-Equity (D/E) is calculated by dividing the company's total debt by its Shareholders' Equity. The Debt-to-Equity ratio gives a sense of how much a company is relying on borrowed money versus its own funds to finance operations. A higher ratio indicates more leverage, meaning the company is taking on more risk, whereas a lower ratio suggests a more conservative financial structure. This helps investors assess the financial stability and riskiness of a company.

These ratios will help us train our model by giving it more data to work with, and by allowing our model to make decisions based on a company's leverage, efficiency, and market valuation.

Technical Indicators

Whilst fundamental ratios can help give insight into the operations of a company, technical indicators tell us the timing we can use to trade stocks.

Stocks also tend to fluctuate very, very frequently. A tool that could smooth out these fluctuations are called moving averages, and we simply calculate it using the functions above. More specifically, we are using an exponential moving average, which is a moving average which adds more bias to recent price fluctuations. This allows our ML model to respond quickly to recent changes if there are any.

By using both a 12 day and a 26 day EMA, our model can get an idea of long-term and short-term trends. The "adjust=false" parameter ensures that our EMAs are calculated recursively.

From our EMAs, we can get the Moving Average Convergence Divergence, which is calculated by taking the difference between the 12 day EMA and the 26 day EMA. Whenever MACD is positive (or in order words, 12-day EMA is higher than 26-day EMA), it means that the short term momentum is higher than the long term momentum, which may imply bullish conditions. Whenever MACD is negative, the long term momentum is smaller than the short term momentum, which may mean a stock's momentum is weakening.

Price changes (and later RSI) are the last feature we engineer. First, we find the day to day change in the price of the stock, using the line: "delta = stock_data['Close'].diff()". From this, we get a range of values that indicate losses and gains.

Then, we split it into components using the next two lines, and take their averages over a 14-day rolling window with the following two. This is necessary for calculating the Relative Strength Index, and smoothing out volatility.

Relative Strength is calculated by dividing the gains and losses in the line: "rs = avg_gain / avg_loss". Higher values imply strong upwards momentum, and lower values imply weak performance.

Finally, we calculate our relative strength index with the line: "stock_data['RSI'] = 100 - (100 / (1 + rs))". The function we have created is an oscillator bounded between 0 and 100. Values above 70 tend to imply the stock is overbought, and values below 30 mean that it is oversold. RSI is mainly used to evaluate whether price movements are becoming unsustainable.

Taken together, these engineered features give us (and the model) a balanced and information-rich view of the market. By combining long-term structural signals with short-term timing signals, the model can learn not only what companies may be attractive, but also when market conditions are favorable.

From here on, it's all Ayan talking. Ethan out!

This code prepares stock market data for a machine-learning model that predicts the next day's closing price. It engineers features, handles missing values, and scales the data so it's ready for trainingticker = ticker

```
Ticker = ticker
stock_data = data[ticker].copy()


stock_data['Target'] = stock_data['Close'].shift(-1)


for lag in range(1, 4):
    stock_data[f'Close_lag_{lag}'] = stock_data['Close'].shift(lag)


stock_data['Rolling_mean_7'] =
stock_data['Close'].rolling(window=7).mean()
stock_data['Rolling_std_7'] = stock_data['Close'].rolling(window=7).std()


stock_data.replace([np.inf, -np.inf], np.nan, inplace=True)


stock_data.fillna(method='ffill', inplace=True)  # Forward fill
stock_data.dropna(inplace=True)  # If any NaNs remain at the start


features = ['P/E', 'ROE', 'Debt-to-Equity', 'MACD', 'RSI',
            'Close_lag_1', 'Close_lag_2', 'Close_lag_3',
            'Rolling_mean_7', 'Rolling_std_7']
target = 'Target'


stock_data[features] = stock_data[features].apply(pd.to_numeric,
errors='coerce')

stock_data.fillna(method='ffill', inplace=True)
stock_data.dropna(inplace=True)
```

```
feature_scaler = RobustScaler()
target_scaler = RobustScaler()



stock_data[features] = feature_scaler.fit_transform(stock_data[features])
stock_data['Target'] = target_scaler.fit_transform(stock_data[['Target']])



train_size = int(len(stock_data) * 0.7)
val_size = int(len(stock_data) * 0.1)
test_size = len(stock_data) - train_size - val_size

train_data = stock_data.iloc[:train_size]
val_data = stock_data.iloc[train_size:train_size + val_size]
test_data = stock_data.iloc[train_size + val_size:]

train_data.head()

print("NaNs in training features:",
train_data[features].isnull().sum().sum())
print("NaNs in training target:", train_data[target].isnull().sum())
```

This code section prepares historical stock data for a machine-learning model that predicts the **next trading day's closing price**. It begins by isolating the data for a single stock ticker and creating a new target variable by shifting the closing price forward one day, ensuring that today's features correspond to tomorrow's outcome. To capture short-term price behavior, the code engineers lagged features using the closing prices from the previous one, two, and three days. It also computes rolling statistics, a 7-day moving average and a 7-day rolling standard deviation—to represent recent trend and volatility. Any infinite values generated during these calculations are replaced with missing values, which are then forward-filled to preserve time continuity, and any remaining invalid rows are dropped to maintain data integrity.

Next, the code defines a comprehensive feature set that combines **fundamental indicators** (such as P/E ratio, return on equity, and debt-to-equity) with **technical indicators** (MACD and RSI) and the engineered time-series features (price lags and rolling statistics). All features are explicitly coerced into numeric form to prevent hidden data-type errors, and missing values introduced during this process are again handled through forward filling and row removal. To make the data suitable for machine learning, both the input features and the prediction target are

scaled using `RobustScaler`, which normalizes values based on medians and interquartile ranges and is therefore resistant to outliers common in financial data.

Finally, the dataset is split chronologically into training, validation, and testing subsets—70% for training, 10% for validation, and the remaining 20% for testing—preserving the time-series structure and avoiding data leakage. The code then performs a final sanity check by printing the head of the training data and confirming that no missing values remain in either the features or the target. Overall, this pipeline transforms raw stock data into a clean, scaled, and time-aware dataset that is ready for use in predictive financial models.

```
from torch.utils.data import Dataset, DataLoader
class StockDataset(Dataset):
    def __init__(self, data, features, target, sequence_length):
        self.data = data
        self.features = features
        self.target = target
        self.sequence_length = sequence_length
    def __len__(self):
        return len(self.data) - self.sequence_length
    def __getitem__(self, idx):
        x =
self.data[self.features].iloc[idx:idx+self.sequence_length].values
        y = self.data[self.target].iloc[idx+self.sequence_length]
        return torch.tensor(x, dtype=torch.float32), torch.tensor(y,
dtype=torch.float32)
sequence_length = 60  # You can adjust this based on experimentation
batch_size = 64       # Adjusted batch size for hyperparameter tuning
train_size = int(len(stock_data) * 0.7)
val_size = int(len(stock_data) * 0.1)
test_size = len(stock_data) - train_size - val_size
train_data = stock_data.iloc[:train_size]
val_data = stock_data.iloc[train_size:train_size + val_size]
test_data = stock_data.iloc[train_size + val_size:]
train_dataset = StockDataset(train_data, features, target,
sequence_length)
val_dataset = StockDataset(val_data, features, target, sequence_length)
test_dataset = StockDataset(test_data, features, target, sequence_length)
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True, drop_last=False)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,
drop_last=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False, drop_last=False)
print(f"Number of training samples: {len(train_dataset)}")
print(f"Number of validation samples: {len(val_dataset)}")
print(f"Number of testing samples: {len(test_dataset)}")
```

```
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True, drop_last=False)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,
drop_last=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False, drop_last=False)
for X_batch, y_batch in train_loader:
    print(f"X_batch shape: {X_batch.shape}")  # Expected: (batch_size,
sequence_length, num_features)
    print(f"y_batch shape: {y_batch.shape}")  # Expected: (batch_size,)
    break
```

This section of code converts the preprocessed stock DataFrame into **time-series tensors suitable for a PyTorch deep learning model**, such as an LSTM or GRU. It begins by importing Dataset and DataLoader from torch.utils.data and defining a custom StockDataset class that inherits from Dataset. Inside __init__, the dataset stores the full DataFrame (data), the list of feature column names (features), the target column (target), and a fixed sequence_length, which controls how many consecutive trading days are used as one input sample. The __len__ method returns len(self.data) - self.sequence_length, ensuring that each sample has enough future data to form a complete sequence without indexing out of bounds. The __getitem__ method implements the core time-series logic: it slices the feature columns using self.data[self.features].iloc[idx : idx + self.sequence_length].values to produce an input window X, and extracts the corresponding target value at the end of the window using self.data[self.target].iloc[idx + self.sequence_length]. Both are converted into torch.tensor objects with dtype=torch.float32, yielding tensors shaped (sequence_length, num_features) for X and (1,) for y.

After defining the dataset class, the code sets key hyperparameters such as sequence_length = 60, which means each training example represents 60 consecutive days of market data, and batch_size = 64, which controls how many sequences are processed in parallel during training. The dataset is then split chronologically using integer indices: train_size = int(len(stock_data) * 0.7), val_size = int(len(stock_data) * 0.1), and the remainder for testing. This preserves temporal ordering and avoids look-ahead bias. Three separate DataFrame slices—train_data, val_data, and test_data—are created using .iloc, and each slice is wrapped in a StockDataset instance with the same features, target, and sequence_length.

Next, the code constructs PyTorch DataLoader objects for each dataset. The training loader is created with shuffle=True to randomize batches and improve gradient stability, while the

validation and test loaders use shuffle=False to preserve time order during evaluation. All loaders use drop_last=False, ensuring that incomplete final batches are still included. Diagnostic print statements report the number of samples in each split using len(train_dataset), len(val_dataset), and len(test_dataset), confirming that the dataset slicing worked as intended. Finally, the code iterates once over train_loader and prints X_batch.shape and y_batch.shape, explicitly verifying that the batch tensors match the expected dimensions—(batch_size, sequence_length, num_features) for inputs and (batch_size, 1) for targets. Overall, this code bridges the gap between pandas-based financial feature engineering and sequence-based neural network training by packaging the data into well-structured, time-aware PyTorch batches.

```
import torch.nn as nn
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size,
dropout=0.3):
        super(LSTMModel, 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, dropout=dropout,
bidirectional=True)
        self.batch_norm = nn.BatchNorm1d(hidden_size * 2)  # Multiply by 2
for bidirectional
        self.dropout = nn.Dropout(p=dropout)
        self.fc = nn.Linear(hidden_size * 2, output_size)  # Multiply by 2
for bidirectional
    def forward(self, x):
        h0 = torch.zeros(self.num_layers * 2, x.size(0),
self.hidden_size).to(x.device)  # 2 for bidirectional
        c0 = torch.zeros(self.num_layers * 2, x.size(0),
self.hidden_size).to(x.device)
        out, _ = self.lstm(x, (h0, c0))  # out: tensor of shape
(batch_size, seq_length, hidden_size*2)
        out = out[:, -1, :]  # Shape: (batch_size, hidden_size*2)
        out = self.batch_norm(out)
        out = self.dropout(out)
        out = self.fc(out)  # Shape: (batch_size, output_size)
        return out
input_size = len(features)  # Number of features
hidden_size = 50           # Reduced from 50 for hyperparameter tuning
num_layers = 2             # Reduced from 2 for hyperparameter tuning
output_size = 1
dropout = 0.3              # Increased from 0.2 for hyperparameter tuning
model = LSTMModel(input_size, hidden_size, num_layers, output_size,
dropout)
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = model.to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)  #
Added weight_decay
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.1)
```

```
print(model)
```

This part of the code defines, configures, and initializes the **neural network that actually makes the stock price prediction**, building directly on the time-series batches created earlier. It introduces a custom LSTMModel class that inherits from nn.Module, which is PyTorch's base class for all trainable models. Inside the __init__ method, the model stores key hyperparameters like hidden_size and num_layers, then constructs a multi-layer LSTM using nn.LSTM. The LSTM is set with batch_first=True, matching the (batch_size, sequence_length, num_features) input shape produced by the DataLoader, and includes dropout to reduce overfitting. The model is explicitly configured as **bidirectional**, meaning it processes each time window both forward and backward in time, allowing it to learn more nuanced temporal relationships within the sequence.

Because bidirectional LSTMs output twice the hidden dimension, the code accounts for this throughout the rest of the architecture. It applies nn.BatchNorm1d(hidden_size * 2) to stabilize training by normalizing the LSTM's output activations, and defines a final fully connected layer (nn.Linear(hidden_size * 2, output_size)) to map the learned representation down to a single numerical prediction. In the forward method, the model initializes the hidden and cell states with zeros, runs the input sequence through the LSTM, and then selects only the output from the **last time step**. This design choice treats the final timestep as a compressed summary of the entire input window, which is common in sequence-to-one prediction problems like next-day price forecasting. The output then flows through batch normalization, dropout, and the linear layer before being returned.

After defining the model class, the code instantiates it using the number of input features, a tuned hidden size, a reduced number of LSTM layers for stability, and a dropout rate chosen to balance bias and variance. The model is moved to GPU if one is available, ensuring faster training on compatible hardware. The training setup is finalized by specifying nn.MSELoss() as the loss function, which is appropriate for continuous regression targets, and using the Adam optimizer with both a learning rate and weight decay to encourage smoother generalization. A StepLR scheduler is also added to gradually reduce the learning rate over time, helping the model converge more reliably. Printing the model at the end serves as a final verification step, confirming the architecture and parameter configuration before training begins.

```
import torch
patience = 75
best_val_loss = float('inf')
epochs_no_improve = 0
early_stop = False
num_epochs = 1000
max_grad_norm = 1.0  # For gradient clipping
for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0
    for inputs, targets in train_loader:
        inputs = inputs.to(device)
        targets = targets.to(device).unsqueeze(1)  # Shape: (batch_size,
1)
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)
        optimizer.step()
        epoch_loss += loss.item()
    avg_train_loss = epoch_loss / len(train_loader)
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for inputs, targets in val_loader:
            inputs = inputs.to(device)
            targets = targets.to(device).unsqueeze(1)
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            val_loss += loss.item()
    avg_val_loss = val_loss / len(val_loader)
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        epochs_no_improve = 0
        torch.save(model.state_dict(), f'best_model_{ticker}.pth')
    else:
        epochs_no_improve += 1
    if epochs_no_improve == patience:
        print(f'Early stopping at epoch {epoch+1}')
```

```
        break
    scheduler.step()
    if (epoch+1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss:
{avg_train_loss:.6f}, Val Loss: {avg_val_loss:.6f}')
        with torch.no_grad():
            sample_pred = model(inputs[:5]).cpu().numpy()
            sample_pred = target_scaler.inverse_transform(sample_pred)  #
Invert scaling
            print(f"Sample Predictions at Epoch {epoch + 1}:
{sample_pred}")
```

This section of the code implements the **training and validation loop** for the LSTM model, along with several safeguards to keep training stable and prevent overfitting. It begins by defining training control parameters such as patience for early stopping, num_epochs for the maximum number of training iterations, and max_grad_norm for gradient clipping. The training loop runs over epochs, and at the start of each epoch the model is set to training mode using model.train(). For every batch from train_loader, the inputs and targets are moved to the selected device, the model generates predictions, and the loss is computed using the predefined regression loss function. Gradients are then calculated with loss.backward(), clipped using torch.nn.utils.clip_grad_norm_ to avoid exploding gradients (a common issue with LSTMs), and applied using optimizer.step(). The code accumulates batch losses to compute an average training loss for the entire epoch.

After each training phase, the model switches to evaluation mode with model.eval() and runs through the validation dataset inside a torch.no_grad() block to ensure no gradients are tracked. Validation loss is computed in the same way as training loss but without updating model weights, and an average validation loss is calculated across all validation batches. This validation metric is then compared against the best loss seen so far. If the validation loss improves, the model's state is saved using torch.save, and the early-stopping counter is reset. If it does not improve, the counter increases, and once it reaches the specified patience, training stops early to avoid overfitting.

The code also integrates a learning-rate scheduler (scheduler.step()), which gradually reduces the learning rate over time to help the optimizer converge more smoothly. Periodically (every 10 epochs), the code prints both training and validation losses to track progress. At the same time, it generates a small batch of sample predictions, reverses the scaling applied earlier using target_scaler.inverse_transform, and prints these values. This provides a human-readable check on whether the model's predictions are becoming more realistic as training progresses. Overall,

this section controls how the model learns, monitors its performance, and ensures training stops at an optimal point rather than blindly running for all epochs.

```
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error
import torch
model.load_state_dict(torch.load(f'best_model_{ticker}.pth'))
model.eval()
predictions = []
actuals = []
with torch.no_grad():
    for inputs, targets in test_loader:
        inputs = inputs.to(device)
        targets = targets.to(device).unsqueeze(1)
        outputs = model(inputs)
        predictions.append(outputs.cpu().numpy())
        actuals.append(targets.cpu().numpy())
predictions = np.concatenate(predictions, axis=0)
actuals = np.concatenate(actuals, axis=0)
predictions = target_scaler.inverse_transform(predictions)
actuals = target_scaler.inverse_transform(actuals)
mae = mean_absolute_error(actuals, predictions)
print(f'Mean Absolute Error on Test Set: {mae:.4f}')
plt.figure(figsize=(14,7))
plt.plot(actuals, label='Actual Prices')
plt.plot(predictions, label='Predicted Prices')
plt.title(f"{ticker} Actual vs Predicted Prices")
plt.xlabel("Time")
plt.ylabel("Price")
plt.legend()
plt.show()
```

```
num_samples = 10  # Number of samples to inspect
indices = np.random.choice(len(actuals), num_samples, replace=False)
print("Index\tActual\tPredicted\tDifference")
for idx in indices:
    actual = actuals[idx][0]
    predicted = predictions[idx][0]
    difference = actual - predicted
    print(f"{idx}\t{actual:.2f}\t{predicted:.2f}\t\t{difference:.2f}")
```

```
test_data_aligned = test_data.iloc[sequence_length:].copy()
test_data_aligned['Predicted'] = predictions.flatten()
test_data_aligned['Return'] = test_data_aligned['Close'].pct_change()
test_data_aligned['Strategy'] = np.where(test_data_aligned['Predicted'] >
test_data_aligned['Close'], test_data_aligned['Return'],
-test_data_aligned['Return'])
test_data_aligned['Cumulative Market Return'] = (1 +
test_data_aligned['Return']).cumprod() - 1
test_data_aligned['Cumulative Strategy Return'] = (1 +
test_data_aligned['Strategy']).cumprod() - 1
plt.figure(figsize=(14,7))
plt.plot(test_data_aligned['Cumulative Market Return'], label='Cumulative
Market Return')
plt.plot(test_data_aligned['Cumulative Strategy Return'],
label='Cumulative Strategy Return')
plt.title(f"{ticker} Backtest Strategy vs Market Performance")
plt.xlabel("Date")
plt.ylabel("Cumulative Return")
plt.legend()
plt.show()
```

This section of the code handles **model evaluation, visualization, and strategy backtesting** to understand how well the trained LSTM performs in practice. It begins by loading the saved best-performing model weights and switching the model into evaluation mode so predictions are generated without updating parameters. Using the test dataset, the code iterates through the test loader inside a no-gradient context, collecting both predicted values and true target values. These are concatenated across batches and then inverse-transformed using the previously fitted scaler so the predictions and actuals are expressed in real price units rather than normalized values. A quantitative performance metric is computed using mean absolute error (MAE), giving a clear numerical measure of how far predictions deviate from true prices on average.

To complement the numeric metric, the code produces a visualization comparing actual prices against predicted prices over time. This plot helps visually assess whether the model captures overall trends, direction changes, and volatility rather than just minimizing error. The code then prints a small random sample of individual predictions alongside their true values and

differences, providing an intuitive sense of model accuracy on specific days rather than only aggregate statistics.

The portion translates model predictions into a **simple trading strategy**. Predicted prices are aligned with the correct dates in the test set, daily market returns are computed from closing prices, and a strategy signal is generated based on whether the model predicts a positive or negative next-day movement. Using these signals, the code calculates both cumulative market returns and cumulative strategy returns over the test period. These are plotted together to directly compare a buy-and-hold baseline against the model-driven strategy. This step reframes the machine-learning output in financial terms, showing not just whether the model predicts prices accurately, but whether those predictions could plausibly translate into improved trading performance.

```
def calculate_var(returns, confidence_level=0.95):
    var = np.percentile(returns.dropna(), (1 - confidence_level) * 100)  #
Calculate percentile for VaR
    return var
var_strategy = calculate_var(test_data_aligned['Strategy'],
confidence_level=0.95)
print(f"Value at Risk (VaR) for the Strategy: {var_strategy:.4f}")
var_market = calculate_var(test_data_aligned['Return'],
confidence_level=0.95)
print(f"Value at Risk (VaR) for the Market: {var_market:.4f}")

def portfolio_optimization(expected_returns, covariance_matrix,
target_return):
    def portfolio_volatility(weights, covariance_matrix):
        return np.sqrt(np.dot(weights.T, np.dot(covariance_matrix,
weights)))  # Portfolio volatility
    n_assets = len(expected_returns)
    args = (covariance_matrix,)
    constraints = (
        {'type': 'eq', 'fun': lambda weights: np.sum(weights) - 1},  #
Weights must sum to 1
        {'type': 'eq', 'fun': lambda weights: np.dot(weights,
expected_returns) - target_return}  # Target return
    )
    bounds = tuple((0, 1) for _ in range(n_assets))  # Weights between 0
and 1
    initial_weights = n_assets * [1. / n_assets,]
    result = minimize(portfolio_volatility, initial_weights, args=args,
method='SLSQP', bounds=bounds, constraints=constraints)
    return result
expected_returns = np.array([0.10, 0.12, 0.15])  # Example expected
returns
cov_matrix = np.array([
    [0.2, 0.1, 0.1],
    [0.1, 0.3, 0.15],
    [0.1, 0.15, 0.25]
])  # Example covariance matrix
target_return = 0.12
portfolio = portfolio_optimization(expected_returns, cov_matrix,
target_return)
```

```
print("Optimized Portfolio Weights:", portfolio.x)

torch.save(model.state_dict(), f"{ticker}_lstm_model.pth")
print(f"Model saved as {ticker}_lstm_model.pth")
```

This section extends the project beyond prediction and into **risk analysis and portfolio construction**. It first defines a helper function to calculate **Value at Risk (VaR)** using historical returns, where the risk metric is computed as a percentile of the return distribution based on a chosen confidence level. The code applies this function separately to the strategy returns and to the overall market returns, allowing a direct comparison of downside risk between the model-driven strategy and a simple buy-and-hold approach. Printing both VaR values reframes model performance in risk terms, not just returns, which is critical in finance.

The next part introduces a **mean–variance portfolio optimization routine**. Using expected returns and a covariance matrix, the code defines portfolio volatility as the objective to minimize, subject to realistic constraints: portfolio weights must sum to one and achieve a target expected return, with bounds ensuring no negative weights. The optimization is solved numerically, producing a set of asset weights that balance risk and return under the given assumptions. Example expected returns and a covariance matrix are included to demonstrate how the optimization framework works in practice.

Finally, the optimized portfolio weights are printed, and the trained LSTM model is saved to disk. This wraps up the pipeline by noting that the model and its outputs are not just evaluated once, but are reusable for future inference or deployment. Overall, this section shows how the predictive model feeds into **risk management and portfolio decision-making**, completing the transition from raw forecasts to actionable financial insights.