


```

import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler

#dataset
df = pd.read_csv("dataset13.csv")

#Fix column names
df.columns = df.columns.str.strip().str.replace('\t', '')

#Convert types
df['Return on Equity'] = pd.to_numeric(df['Return on Equity'], errors='coerce')

#Sort by company and time
df = df.sort_values(by=['CO_ID', 'Year', 'Quarter'])

#Select features and target
features = [
    'Return on Assets', 'Return on Capital', 'Return on Equity',
    'Return on Common Equity', 'Gross Profit Margin', 'EBITA Margin',
    'EBIT Margin', 'Net Income Margin', 'Asset Turnover', 'Current Ratio',
    'Debt / Equity', 'Long Term Debt / Captial', 'Book Value Per Share'
]
target = 'Stock Return'

#Fill missing values per company
df[features] = df.groupby('CO_ID')[features].transform(lambda g: g.ffill().bfill())

# Normalize
scaler = StandardScaler()
df[features] = scaler.fit_transform(df[features])

# Check if missing values remain
df[features].isnull().sum()

```

```

Return on Assets      0
Return on Capital     0
Return on Equity      0
Return on Common Equitv  0

```

```

X_all, y_all = [], []
window_size = 16
target_offset = 1

for _, group in df.groupby('CO_ID'):
    X = group[features].values
    y = group[target].values
    for i in range(len(X) - window_size - target_offset + 1):
        X_all.append(X[i:i+window_size])
        y_all.append(y[i+window_size+target_offset-1])

import torch
X_tensor = torch.tensor(np.array(X_all), dtype=torch.float32)
y_tensor = torch.tensor(np.array(y_all), dtype=torch.float32).view(-1, 1)

```

```

from torch.utils.data import Dataset, DataLoader

class FinancialDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

dataset = FinancialDataset(X_tensor, y_tensor)
train_loader = DataLoader(dataset, batch_size=16, shuffle=True)

```

```

from torch.utils.data import random_split, DataLoader

# Step 1: Split into training and validation sets
total_size = len(dataset)
train_size = int(0.8 * total_size)
val_size = total_size - train_size

train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

# Step 2: Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)

```

```

# -----
# HYPERPARAMETERS
# -----
input_dim = 13          # Number of input features
hidden_dim = 64         # LSTM hidden size
num_layers = 2          # Number of LSTM layers
dropout = 0.2           # Dropout rate
learning_rate = 0.0005  # ← Try reducing from 0.001
batch_size = 16         # Batch size for training
num_epochs = 30         # Number of training epochs

```

```

from torch.utils.data import Dataset, DataLoader

class FinancialDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

dataset = FinancialDataset(X_tensor, y_tensor)
train_loader = DataLoader(dataset, batch_size=16, shuffle=True)

```

```

from torch.utils.data import random_split, DataLoader

# Step 1: Split into training and validation sets
total_size = len(dataset)
train_size = int(0.8 * total_size)
val_size = total_size - train_size

train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

# Step 2: Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)

```

```

from torch.utils.data import Dataset, DataLoader

class FinancialDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

dataset = FinancialDataset(X_tensor, y_tensor)
train_loader = DataLoader(dataset, batch_size=16, shuffle=True)

```

```

from torch.utils.data import random_split, DataLoader

#training and validation sets
total_size = len(dataset)
train_size = int(0.8 * total_size)
val_size = total_size - train_size

train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

#Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)

```

```
# -----
# HYPERPARAMETERS
# -----
input_dim = 13      # Number of input features
hidden_dim = 64     # LSTM hidden size
num_layers = 2      # Number of LSTM layers
dropout = 0.2       # Dropout rate
learning_rate = 0.0005 # ← Try reducing from 0.001
batch_size = 16     # Batch size for training
num_epochs = 30     # Number of training epochs
```

```
import torch.nn as nn

class LSTMRegressor(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, dropout):
        super().__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, dropout=dropout, batch_first=True)
        self.fc = nn.Linear(hidden_dim, 1)

    def forward(self, x):
        out, _ = self.lstm(x)
        return self.fc(out[:, -1, :]) # last time step
```

```
total_size = len(dataset)
train_size = int(0.8 * total_size)
val_size = total_size - train_size

train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

```
model = LSTMRegressor(input_dim, hidden_dim, num_layers, dropout)
criterion = nn.MSELoss()
mae_criterion = nn.L1Loss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5)

train_losses, val_losses = [], []
```

```
# Training and validation
for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        y_pred = model(X_batch)
        loss = criterion(y_pred, y_batch)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    # Validation
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for X_val, y_val in val_loader:
            y_val_pred = model(X_val)
            val_loss += criterion(y_val_pred, y_val).item()

    #Average Losses
    train_loss /= len(train_loader)
    val_loss /= len(val_loader)

    val_mae = 0
    with torch.no_grad():
        for X_val, y_val in val_loader:
            y_val_pred = model(X_val)
            val_loss += criterion(y_val_pred, y_val).item()

            val_mae += nn.L1Loss()(y_val_pred, y_val).item()

    # Average both
    val_loss /= len(val_loader)
    val_mae /= len(val_loader)

    print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Val Loss = {val_loss:.4f}, Val MAE = {val_mae:.4f}")

#print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Val Loss = {val_loss:.4f}")
```

Epoch 30: Train Loss = 314.9690, Val Loss = 417.7370, Val MAE = 13.7248

```

import numpy as np
import matplotlib.pyplot as plt

# Switch model to evaluation mode
model.eval()
actuals, predictions = [], []

# Get predictions on the validation set
with torch.no_grad():
    for X_val, y_val in val_loader:
        y_pred = model(X_val)
        actuals.extend(y_val.view(-1).tolist())
        predictions.extend(y_pred.view(-1).tolist())

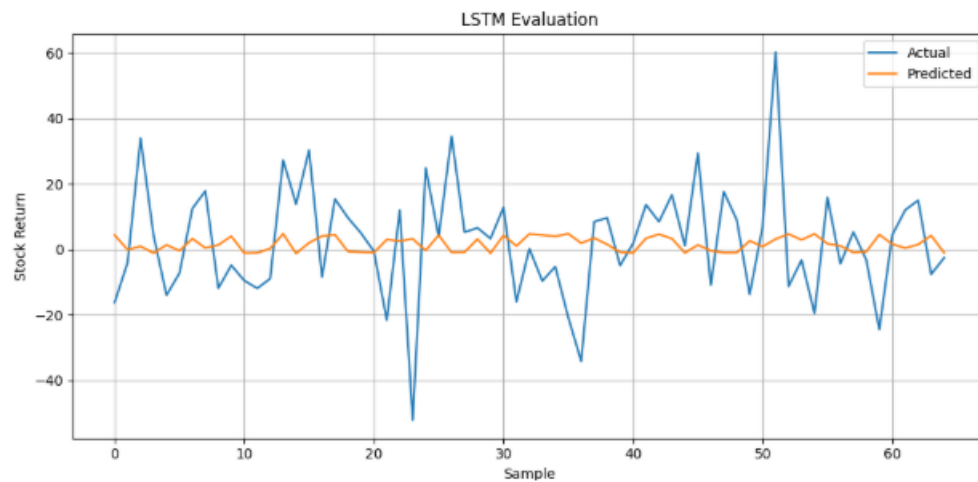
# Convert to NumPy arrays
actuals = np.array(actuals)
predictions = np.array(predictions)

# Compute MAE
mae = np.mean(np.abs(actuals - predictions))

# Compute MAPE with epsilon to avoid division by 0
mape = np.mean(np.abs((actuals - predictions) / (actuals + 1e-8))) * 100

# Optional: plot actual vs predicted
plt.figure(figsize=(10, 5))
plt.plot(actuals, label='Actual')
plt.plot(predictions, label='Predicted')
plt.title('LSTM Evaluation')
plt.xlabel('Sample')
plt.ylabel('Stock Return')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

```



```

X_all, y_all = [], []
window_size = 8
target_offset = 1

for _, group in df.groupby('CO_ID'):
    X = group[features].values
    y = group[target].values
    for i in range(len(X) - window_size - target_offset + 1):
        X_all.append(X[i:i+window_size])
        y_all.append(y[i+window_size+target_offset-1])

import torch
X_tensor = torch.tensor(np.array(X_all), dtype=torch.float32)
y_tensor = torch.tensor(np.array(y_all), dtype=torch.float32).view(-1, 1)

y_tensor = (y_tensor > 0).float()

```

```
#binary classification method
```

```
input_dim = 13  
hidden_dim = 64  
num_layers = 2  
dropout = 0.2  
learning_rate = 0.0005  
batch_size = 16  
num_epochs = 30
```

```
#Dataset
```

```
class FinancialDataset(torch.utils.data.Dataset):  
    def __init__(self, X, y):  
        self.X = X  
        self.y = y  
    def __len__(self):  
        return len(self.X)  
    def __getitem__(self, idx):  
        return self.X[idx], self.y[idx]
```

```
dataset = FinancialDataset(X_tensor, y_tensor)
```

```
#Train/Val Split
```

```
total_size = len(dataset)  
train_size = int(0.8 * total_size)  
val_size = total_size - train_size
```

```
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])  
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)  
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

```

#LSTM Classifier Model

class LSTMClassifier(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, dropout):
        super().__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, dropout=dropout, batch_first=True)
        self.fc = nn.Sequential(
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        out, _ = self.lstm(x)
        return self.fc(out[:, -1, :]) # output shape: [batch_size, 1]

model = LSTMClassifier(input_dim, hidden_dim, num_layers, dropout)

# Loss, Optimizer, Scheduler

criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5)

train_losses, val_losses, val_accuracies = [], [], []

#Training

for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        y_pred = model(X_batch)
        loss = criterion(y_pred, y_batch.view(-1, 1))
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    train_loss /= len(train_loader)

    # Validation
    model.eval()
    val_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for X_val, y_val in val_loader:
            y_val_pred = model(X_val) # [B, 1]
            val_loss += criterion(y_val_pred, y_val.view(-1, 1)).item()
            preds = (y_val_pred > 0.5).float()
            correct += (preds == y_val.view(-1, 1)).sum().item()
            total += y_val.size(0)

    val_loss /= len(val_loader)
    val_acc = correct / total

    train_losses.append(train_loss)
    val_losses.append(val_loss)
    val_accuracies.append(val_acc)
    scheduler.step()

print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Val Loss = {val_loss:.4f}, Val Accuracy = {val_acc:.2%}")

plt.figure(figsize=(12, 5))

```

```

print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Val Loss = {val_loss:.4f}, Val Accuracy = {val_acc:.2%}")

plt.figure(figsize=(12, 5))

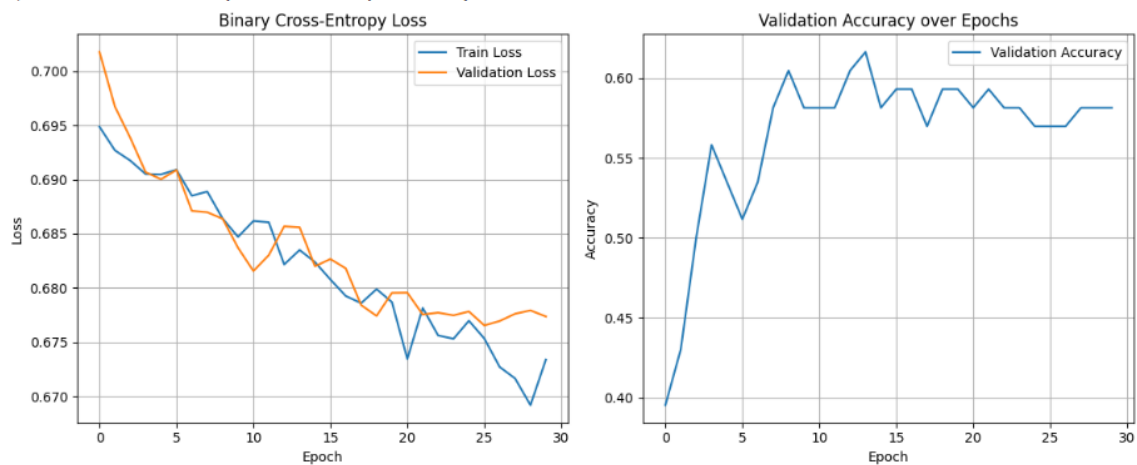
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Binary Cross-Entropy Loss')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(val_accuaries, label='validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Validation Accuracy over Epochs')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

```

Epoch 30: Train Loss = 0.6734, Val Loss = 0.6774, Val Accuracy = 58.14%



```

input_dim = 13
hidden_dim = 64
num_layers = 2
dropout = 0.4
learning_rate = 0.0005
batch_size = 16
num_epochs = 30

```

```

class GRURegressor(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, dropout):
        super().__init__()
        self.gru = nn.GRU(input_dim, hidden_dim, num_layers,
                           dropout=dropout, batch_first=True)
        self.fc = nn.Linear(hidden_dim, 1) # map to output

    def forward(self, x):
        out, _ = self.gru(x)
        return self.fc(out[:, -1, :])

```

```

input_dim = 13
hidden_dim = 64
num_layers = 2
dropout = 0.4
learning_rate = 0.0005
batch_size = 16
num_epochs = 30

```

```

class GRURegressor(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, dropout):
        super().__init__()
        self.gru = nn.GRU(input_dim, hidden_dim, num_layers,
                           dropout=dropout, batch_first=True)
        self.fc = nn.Linear(hidden_dim, 1) # map to output

    def forward(self, x):
        out, _ = self.gru(x)
        return self.fc(out[:, -1, :])

```

```

for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        y_pred = model(X_batch)
        loss = criterion(y_pred, y_batch)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    model.eval()
    val_loss = 0
    with torch.no_grad():
        for X_val, y_val in val_loader:
            y_val_pred = model(X_val)
            val_loss += criterion(y_val_pred, y_val).item()

    train_loss /= len(train_loader)
    val_loss /= len(val_loader)

    model.eval()
    actuals, predictions = [], []

    with torch.no_grad():
        for X_val, y_val in val_loader:
            y_pred = model(X_val)
            actuals.extend(y_val.view(-1).tolist())
            predictions.extend(y_pred.view(-1).tolist())

    actuals = np.array(actuals)
    predictions = np.array(predictions)

    mae = np.mean(np.abs(actuals - predictions))
    mape = np.mean(np.abs((actuals - predictions) / (actuals + 1e-8))) * 100

    print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Val Loss = {val_loss:.4f}")
    print(f"MAE: {mae:.4f}")
    print(f"MAPE: {mape:.2f}%")

```

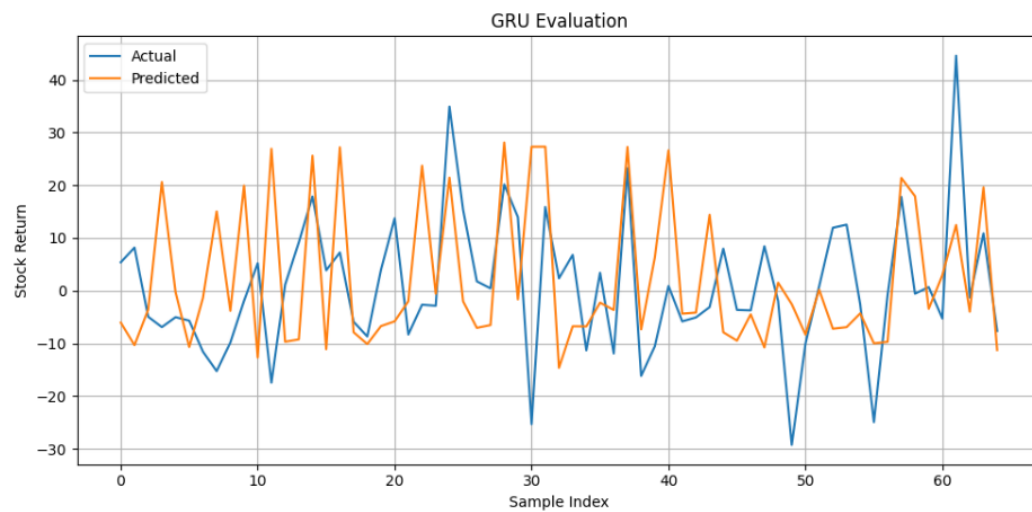
```

print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Val Loss = {val_loss:.4f}")
print(f"MAE: {mae:.4f}")
print(f"MAPE: {mape:.2f}%")

plt.figure(figsize=(10, 5))
plt.plot(actuals, label='Actual')
plt.plot(predictions, label='Predicted')
plt.title("GRU Evaluation")
plt.xlabel("Sample Index")
plt.ylabel("Stock Return")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

```

Epoch 30: Train Loss = 203.9855, Val Loss = 218.0814
MAE: 12.5897
MAPE: 326.03%



```

X_all, y_all = [], []
window_size = 8
target_offset = 1

for _, group in df.groupby('CO_ID'):
    X = group[features].values
    y = group[target].values
    for i in range(len(X) - window_size - target_offset + 1):
        X_all.append(X[i:i+window_size])
        y_all.append(y[i+window_size+target_offset-1])

import torch
X_tensor = torch.tensor(np.array(X_all), dtype=torch.float32)
y_tensor = torch.tensor(np.array(y_all), dtype=torch.float32).view(-1, 1)

```

```

from torch.utils.data import Dataset, DataLoader

class FinancialDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

dataset = FinancialDataset(X_tensor, y_tensor)
train_loader = DataLoader(dataset, batch_size=16, shuffle=True)

```

```

from torch.utils.data import random_split, DataLoader

# Step 1: Split into training and validation sets
total_size = len(dataset)
train_size = int(0.8 * total_size)
val_size = total_size - train_size

train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

# Step 2: Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)

```

```

input_dim = 13
hidden_dim = 64
num_layers = 2
dropout = 0.4
learning_rate = 0.0001
batch_size = 16
num_epochs = 30

```

```

class GRURegressor(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, dropout):
        super().__init__()
        self.gru = nn.GRU(input_dim, hidden_dim, num_layers,
                           dropout=dropout, batch_first=True)
        self.fc = nn.Linear(hidden_dim, 1) # map to output

    def forward(self, x):
        out, _ = self.gru(x)
        return self.fc(out[:, -1, :])

```

Epoch 30: Train Loss = 301.4486, Val Loss = 307.9990
MAE: 12.3167
MAPE: 137.27%

