Stock Price Forecasting Using Neural Networks

## Abhigya Sinha, Aniket Iyer, Gautham Ande, Priyangshu Bhowmik

# Table of Contents *1. Motivation 2. Data Collection and Cleaning 3. Feature Engineering 4. Exploratory Data Analysis 5. Modeling: Single-Layer Neural Network 6. Modeling: Multi-Layer Neural Network 7. Comparison 8. Conclusion & Source Code*

# Motivation

The financial sector is characterized by its dynamic and often volatile nature. Forecasting stock prices has immense applications, from informing investment strategies to algorithmic trading systems. While classical statistical models have been widely used, recent advances in machine learningparticularly deep learningoffer promising alternatives.

This project explores how feedforward neural networks, both shallow and deep, can model next-day closing prices across companies from three major sectors: Technology (AAPL, MSFT), Finance (JPM, BAC), and Energy (XOM, CVX). By comparing single-layer and multi-layer architectures, we aim to investigate model performance differences and analyze generalization capabilities across domains.

# Data Collection and Cleaning

We utilized Yahoo Finance via the yfinance Python API to obtain historical stock prices from 2018 to 2024. The raw data was cleaned by removing null values and formatting the datetime index.

## Code: 01\_data\_collection\_and\_cleaning.ipynb

import yfinance as yf  
import pandas as pd  
  
stocks = ['AAPL', 'MSFT', 'JPM', 'BAC', 'XOM', 'CVX']  
data\_dict = {}  
  
for stock in stocks:  
df = yf.download(stock, start="2018-01-01", end="2024-01-01")  
df['Ticker'] = stock  
data\_dict[stock] = df  
  
full\_data = pd.concat(data\_dict.values())  
full\_data.reset\_index(inplace=True)  
full\_data.dropna(inplace=True)  
full\_data.to\_csv("data/processed/merged\_data.csv", index=False)

# Feature Engineering

To enrich the raw dataset with time-dependent information, we engineered several features:

* - Lagged returns (1, 2, 3 days)
* - Moving averages (5-day and 10-day SMA)
* - 10-day rolling standard deviation as a volatility measure

## Code: 02\_feature\_engineering.ipynb

def create\_features(df):  
df['Return\_1'] = df['Close'].pct\_change(1)  
df['Return\_2'] = df['Close'].pct\_change(2)  
df['Return\_3'] = df['Close'].pct\_change(3)  
df['SMA\_5'] = df['Close'].rolling(window=5).mean()  
df['SMA\_10'] = df['Close'].rolling(window=10).mean()  
df['Volatility'] = df['Close'].rolling(window=10).std()  
return df.dropna()  
  
feature\_data = full\_data.groupby('Ticker').apply(create\_features)  
feature\_data.to\_csv("data/processed/feature\_data.csv", index=False)

# Exploratory Data Analysis

We conducted visual analyses to understand sector behavior over time.

## Average Returns by Sector

A graph showing a sound wave

AI-generated content may be incorrect.

Interpretation: Technology, Finance, and Energy sectors all show high volatility early in the time period, particularly around 2020 (likely due to COVID-19 market disruption). The volatility stabilizes over time with similar mean-reversion behavior, suggesting comparable patterns for short-term forecasting.

## Normalized Stock Prices

A graph of stock prices

AI-generated content may be incorrect.

Interpretation: AAPL and MSFT demonstrate the most growth, consistent with tech-sector outperformance. Financial stocks (JPM, BAC) recovered post-2020 but remain more cyclic. Energy (XOM, CVX) rebounded after 2021 with strong late-stage performance, likely tied to oil prices.

## Rolling Volatility Comparison

A graph showing a number of data

AI-generated content may be incorrect.

Interpretation: All stocks experienced peak volatility around 2020. Since then, 10-day rolling volatility has normalized across sectors. Slightly elevated volatility in CVX and BAC post-2022 suggests these may require more robust model regularization.

# Modeling: Single-Layer Neural Network

## Source Code

***import tensorflow as tf***

***from sklearn.preprocessing import StandardScaler***

***from sklearn.metrics import mean\_squared\_error, r2\_score***

***import numpy as np***

***import matplotlib.pyplot as plt***

***# Prepare features and target***

***features = ['Return\_1', 'Return\_2', 'Return\_3', 'SMA\_5', 'SMA\_10', 'Volatility']***

***results = {}***

***for ticker in feature\_data['Ticker'].unique():***

***df = feature\_data[feature\_data['Ticker'] == ticker].copy()***

***X = df[features].values***

***y = df['Return\_1'].shift(-1).dropna().values***

***X = X[:-1]***

***# Split into train and test***

***split = int(len(X) \* 0.8)***

***X\_train, X\_test = X[:split], X[split:]***

***y\_train, y\_test = y[:split], y[split:]***

***# Normalize features***

***scaler = StandardScaler()***

***X\_train\_scaled = scaler.fit\_transform(X\_train)***

***X\_test\_scaled = scaler.transform(X\_test)***

***# Define model***

***model = tf.keras.Sequential([***

***tf.keras.layers.Dense(1, input\_shape=(X\_train.shape[1],))***

***])***

***model.compile(optimizer='adam', loss='mse')***

***# Train***

***model.fit(X\_train\_scaled, y\_train, epochs=100, verbose=0)***

***# Predict***

***y\_pred = model.predict(X\_test\_scaled).flatten()***

***# Evaluation***

***mse = mean\_squared\_error(y\_test, y\_pred)***

***rmse = np.sqrt(mse)***

***r2 = r2\_score(y\_test, y\_pred)***

***results[ticker] = (mse, rmse, r2)***

***# Plot***

***plt.figure(figsize=(10,4))***

***plt.plot(y\_test[:250], label='Actual')***

***plt.plot(y\_pred[:250], label='Predicted')***

***plt.title(f"{ticker} - Single Layer NN (R = {r2:.4f})")***

***plt.legend()***

***plt.savefig(f"plots/model\_plots/{ticker}\_single\_layer\_nn.png")***

***plt.close()***

def create\_features(df):  
df['Return\_1'] = df['Close'].pct\_change(1)  
df['Return\_2'] = df['Close'].pct\_change(2)  
df['Return\_3'] = df['Close'].pct\_change(3)  
df['SMA\_5'] = df['Close'].rolling(window=5).mean()  
df['SMA\_10'] = df['Close'].rolling(window=10).mean()  
df['Volatility'] = df['Close'].rolling(window=10).std()  
return df.dropna()  
  
feature\_data = full\_data.groupby('Ticker').apply(create\_features)  
feature\_data.to\_csv("data/processed/feature\_data.csv", index=False)

## Explanation of Code

The modeling pipeline for the single-layer neural network involves several core steps:

* Feature Selection: Six engineered features are used: three lagged returns, two SMAs, and rolling volatility.
* Target Variable: The next day's return (shifted return\_1) is used as the label.
* Data Splitting: We used a chronological 80/20 train-test split to prevent data leakage.
* Normalization: StandardScaler standardizes features to have zero mean and unit variance.
* Model Architecture: A single dense layer with one neuron and no activation is usedthis acts as a linear regressor optimized using MSE loss.
* Training: The model is trained for 100 epochs silently.
* Evaluation: RMSE, MSE, and are calculated and predictions are plotted for the first 250 time steps.

## Understanding Single-Layer Neural Networks

A single-layer neural network is essentially a linear model with weights and bias. It assumes linear relationships between the input features and the output. In our use case, it serves as a baseline to test whether simple mappings from technical indicators to price returns can provide meaningful forecasts.

Given the stochastic, non-stationary, and noisy nature of financial markets, especially at short prediction horizons, even small values indicate some predictive capability. Forecasting daily stock returns is exceptionally challenging due to market efficiency and random walk behavior.

## Prediction Plots and Observations

* AAPL: The model tracks the trend well, though with dampened amplitudes. RMSE is low (0.0052), and of 0.6110 suggests moderate fit.
* A graph with blue and orange lines

  AI-generated content may be incorrect.BAC: Closely mirrors the actual returns; of 0.6703 is one of the highest among the group.
* A graph with blue and orange lines

  AI-generated content may be incorrect.CVX: Shows strong alignment; spikes are captured well. High of 0.6990 reflects this.
* A graph with blue and orange lines

  AI-generated content may be incorrect. JPM: Lowest (0.4575). Captures broad movements but misses sharp shifts.
* A graph with blue and orange lines

  AI-generated content may be incorrect. MSFT: Mixed results; predictions sometimes lag or miss high frequency changes.
* A graph with orange and blue lines

  AI-generated content may be incorrect. XOM: Smooth and reasonably accurate fit, especially in low-volatility regions.

A graph with blue and orange lines

AI-generated content may be incorrect.

## Summary of Results

|  |  |  |  |
| --- | --- | --- | --- |
| Stock | MSE | RMSE | R² |
| AAPL | 0.000027 | 0.005182 | 0.6110 |
| MSFT | 0.000020 | 0.004518 | 0.5868 |
| JPM | 0.000034 | 0.005841 | 0.4575 |
| BAC | 0.000024 | 0.004901 | 0.6703 |
| XOM | 0.000016 | 0.004048 | 0.6776 |
| CVX | 0.000017 | 0.004159 | 0.6990 |

While values may seem low compared to traditional regression benchmarks, they are reasonable for stock prediction where daily returns can appear random. Even minor predictive power, when consistent, is valuable in finance. The lowest performance on JPM suggests sector-specific volatility and noise may hinder shallow models.

# Modeling: Multi-Layer Neural Network

## Source Code

import tensorflow as tf  
from sklearn.preprocessing import StandardScaler  
from sklearn.metrics import mean\_squared\_error, r2\_score  
import numpy as np  
import matplotlib.pyplot as plt  
  
# Prepare features and target  
features = ['Return\_1', 'Return\_2', 'Return\_3', 'SMA\_5', 'SMA\_10', 'Volatility']  
results = {}  
  
for ticker in feature\_data['Ticker'].unique():  
df = feature\_data[feature\_data['Ticker'] == ticker].copy()  
X = df[features].values  
y = df['Return\_1'].shift(-1).dropna().values  
X = X[:-1]  
  
# Split into train and test  
split = int(len(X) \* 0.8)  
X\_train, X\_test = X[:split], X[split:]  
y\_train, y\_test = y[:split], y[split:]  
  
# Normalize features  
scaler = StandardScaler()  
X\_train\_scaled = scaler.fit\_transform(X\_train)  
X\_test\_scaled = scaler.transform(X\_test)  
  
# Define multi-layer model  
model = tf.keras.Sequential([  
tf.keras.layers.Dense(64, activation='relu', input\_shape=(X\_train.shape[1],)),  
tf.keras.layers.Dense(32, activation='relu'),  
tf.keras.layers.Dense(1)  
])  
model.compile(optimizer='adam', loss='mse')  
  
# Train  
model.fit(X\_train\_scaled, y\_train, epochs=100, verbose=0)  
  
# Predict  
y\_pred = model.predict(X\_test\_scaled).flatten()  
  
# Evaluation  
mse = mean\_squared\_error(y\_test, y\_pred)  
rmse = np.sqrt(mse)  
r2 = r2\_score(y\_test, y\_pred)  
results[ticker] = (mse, rmse, r2)  
  
# Plot  
plt.figure(figsize=(10,4))  
plt.plot(y\_test[:250], label='Actual')  
plt.plot(y\_pred[:250], label='Predicted')  
plt.title(f"{ticker} - Multi-layer NN (R² = {r2:.4f})")  
plt.legend()  
plt.savefig(f"plots/model\_plots/{ticker}\_multi\_layer\_nn.png")  
plt.close()

## Explanation of Code

In this multi-layer approach:

* We retained the same six input features and data preparation steps as the single-layer network.
* The key difference is the model architecture: two hidden layers with 64 and 32 ReLU-activated neurons were added, enabling the network to learn more complex patterns.
* Adam optimizer and MSE loss remain standard choices for regression.
* Predictions and evaluation metrics follow the same procedure.

## Understanding Multi-Layer Neural Networks

Multi-layer neural networks (MLPs) can approximate nonlinear relationships by stacking layers and applying non-linear activation functions (ReLU). Each layer transforms the feature space, enabling the network to model more intricate patterns.

This makes MLPs well-suited for tasks like stock prediction, where the relationship between indicators and future returns may be nonlinear. However, they also introduce risk of overfitting and increased sensitivity to noisy inputs.

## Prediction Plots and Observations

* AAPL: Slight improvement over single-layer, with better alignment on high-magnitude return spikes. .
* A graph with blue and orange lines

  AI-generated content may be incorrect. BAC: Strong performance, tracking the actual curve closely even during sharp downturns. Highest of 0.7554.
* A graph with blue and orange lines

  AI-generated content may be incorrect.CVX: Improved sharpness in response; shows high explanatory power.
* A graph with blue and orange lines

  AI-generated content may be incorrect. JPM: Performance dropped with deeper networkpossibly due to overfitting or excess sensitivity to noise. .
* A graph with blue and orange lines

  AI-generated content may be incorrect. MSFT: More responsive to small fluctuations than single-layer, achieving .
* A graph with orange and blue lines

  AI-generated content may be incorrect. XOM: Smoother but accurate predictions, although high-frequency deviations remain. .

## A graph with blue and orange lines AI-generated content may be incorrect.Summary of Results

|  |  |  |  |
| --- | --- | --- | --- |
| Stock | MSE | RMSE | R² |
| AAPL | 0.000015 | 0.003825 | 0.6270 |
| MSFT | 0.000010 | 0.003084 | 0.6778 |
| JPM | 0.000025 | 0.004982 | 0.3381 |
| BAC | 0.000010 | 0.003213 | 0.7554 |
| XOM | 0.000011 | 0.003315 | 0.6257 |
| CVX | 0.000011 | 0.003311 | 0.6768 |

While some stocks such as JPM underperformed, the multi-layer network offered improved generalization for BAC, MSFT, and CVX. This confirms that deeper architectures can learn better representationsprovided the underlying signal-to-noise ratio is favorable.

Daily stock returns remain notoriously hard to predict due to market randomness, but even moderate increases in can provide useful trading signals.

# Comparison: Single-Layer vs Multi-Layer Neural Networks

In this section, we compare the performance of the single-layer and multi-layer neural network models across the six selected stocks. The goal is to understand the trade-offs in accuracy, generalization, and model complexity.

## Quantitative Performance Summary

| Stock | Single-Layer MSE | RMSE | R² | Multi-Layer MSE | RMSE | R² |
| --- | --- | --- | --- | --- | --- | --- |
| AAPL | 0.000027 | 0.00518 | 0.6110 | 0.000015 | 0.00383 | 0.6270 |
| MSFT | 0.000020 | 0.00452 | 0.5868 | 0.000010 | 0.00308 | 0.6778 |
| JPM | 0.000034 | 0.00584 | 0.4575 | 0.000025 | 0.00498 | 0.3381 |
| BAC | 0.000024 | 0.00490 | 0.6703 | 0.000010 | 0.00321 | 0.7554 |
| XOM | 0.000016 | 0.00405 | 0.6776 | 0.000011 | 0.00332 | 0.6257 |
| CVX | 0.000017 | 0.00416 | 0.6990 | 0.000011 | 0.00331 | 0.6768 |

## Model Behavior and Insights

Overall Improvement: Across most stocks, the multi-layer model consistently achieved lower MSE and RMSE values and higher scores compared to the single-layer counterpart. This is expected as deeper networks capture non-linear patterns better.

Best Performers: BAC and MSFT stand out in the multi-layer setup with values exceeding 0.75 and 0.67 respectively, up from already decent baselines. These stocks may exhibit smoother trends and stronger signal-to-feature relationships.

JPM Exception: Unlike others, JPM showed reduced performance with the deeper model (dropped from 0.4575 to 0.3381). This suggests that either the additional layers overfit noise or failed to extract useful patterns. Financials may carry unpredictable or lagged responses to market information.

Complexity vs Simplicity: The single-layer model, though linear, performed surprisingly well for stocks like CVX and XOM. This shows that in certain contexts, simpler models can approximate returns effectively without risk of overfitting.

## Trade-Off Analysis

* Accuracy: Multi-layer networks outperform on most metrics but not universally. The marginal gains are most valuable when rises significantly (e.g., BAC, MSFT).
* Robustness: Single-layer networks are less prone to overfitting due to lower capacity. They performed relatively consistently across all stocks.
* Interpretability: Linear models offer more transparent behavior, aiding financial interpretability, which is often desirable in regulated environments.
* Computation: Multi-layer models require more resources and time, both in training and hyperparameter tuning.

## Conclusion of Comparison

While the multi-layer model improves predictive performance across most stocks, its effectiveness is not uniform. Stock-specific volatility, noise, and nonlinearities play a significant role. Therefore, model selection in stock forecasting must balance complexity with generalizability.

In practice, a hybrid ensemble approachor sector-specific model designmight yield the best results, leveraging strengths of both architectures based on the data characteristics.

# Conclusion

This project investigated the viability of using feedforward neural networks to forecast next-day stock returns across multiple sectorsTechnology, Finance, and Energy. We implemented two architectures: a simple single-layer neural network and a more expressive multi-layer neural network with hidden layers and nonlinear activations. Each model was evaluated on six prominent stocks (AAPL, MSFT, JPM, BAC, XOM, CVX) using regression metrics such as MSE, RMSE, and .

Through detailed preprocessing and feature engineering, we incorporated technical indicators like lagged returns, moving averages, and volatility into our model inputs. The results showed that while both models capture some predictive patterns, multi-layer networks typically outperform single-layer ones, particularly for stocks with less erratic behavior. Nevertheless, the performance was stock-dependentJPM, for instance, suffered from reduced performance in the deeper model, highlighting the non-trivial challenge of generalization in financial time series.

Importantly, even the best scores in this project hovered around 0.75. This underscores the inherent difficulty of stock prediction: markets are noisy, dynamic, and prone to sudden changes. Despite this, the fact that neural networksespecially multi-layer onescan extract structure from such data is promising. Our findings suggest that while no single model is universally optimal, neural networks can be a valuable tool in a broader predictive analytics pipeline.

Future work could explore more sophisticated architectures like LSTMs or Transformers, incorporate macroeconomic indicators, and utilize ensemble approaches to balance variance and bias. Another promising direction is the integration of sentiment analysis and natural language processing (NLP). Since stock prices are often influenced by investor emotions and public sentimentespecially in reaction to breaking news, earnings announcements, or social media trendsmodels that account for emotional and textual signals from news articles, Twitter feeds, and Reddit forums could add valuable predictive power. Additionally, incorporating alternative data sources such as Google Trends, options flow, and institutional sentiment could further enhance prediction robustness. The potential for improvement remains significant, especially with richer datasets and advanced regularization techniques.

## Code: 01\_data\_collection\_and\_cleaning.ipynb (Extended Version)

import pandas as pd # For data manipulation and analysis  
import os # For file system operations like creating directories  
  
def generate\_features(df):  
"""  
Generate technical indicators and features from raw price data.  
  
Parameters:  
-----------  
df : pandas.DataFrame  
Raw stock data with at least 'Close' price column  
  
Returns:  
--------  
pandas.DataFrame  
Processed dataframe with additional technical features  
"""  
df = df.copy()  
df['Close'] = pd.to\_numeric(df['Close'], errors='coerce')  
df['Return'] = df['Close'].pct\_change(fill\_method=None)  
df['Lag\_1'] = df['Return'].shift(1)  
df['Lag\_2'] = df['Return'].shift(2)  
df['SMA\_5'] = df['Close'].rolling(window=5).mean()  
df['SMA\_10'] = df['Close'].rolling(window=10).mean()  
df['Volatility'] = df['Return'].rolling(window=10).std()  
return df.dropna()  
  
os.makedirs('../data/processed', exist\_ok=True)  
tickers = ['AAPL', 'MSFT', 'JPM', 'BAC', 'XOM', 'CVX']  
  
for ticker in tickers:  
df = pd.read\_csv(f'../data/raw/{ticker}\_raw.csv', index\_col=0, parse\_dates=True)  
df.index.name = 'Date'  
processed = generate\_features(df)  
processed.to\_csv(f'../data/processed/{ticker}\_processed.csv')  
print(f"{ticker} processed.")

## Code: 02\_feature\_engineering.ipynb Feature Generation

import pandas as pd # For data manipulation and analysis  
import os # For file system operations like creating directories  
  
def generate\_features(df):  
"""  
Generate technical indicators and features from raw price data.  
  
Parameters:  
-----------  
df : pandas.DataFrame  
Raw stock data with at least 'Close' price column  
  
Returns:  
--------  
pandas.DataFrame  
Processed dataframe with additional technical features  
"""  
df = df.copy()  
df['Close'] = pd.to\_numeric(df['Close'], errors='coerce')  
df['Return'] = df['Close'].pct\_change(fill\_method=None)  
df['Lag\_1'] = df['Return'].shift(1)  
df['Lag\_2'] = df['Return'].shift(2)  
df['SMA\_5'] = df['Close'].rolling(window=5).mean()  
df['SMA\_10'] = df['Close'].rolling(window=10).mean()  
df['Volatility'] = df['Return'].rolling(window=10).std()  
return df.dropna()  
  
os.makedirs('../data/processed', exist\_ok=True)  
tickers = ['AAPL', 'MSFT', 'JPM', 'BAC', 'XOM', 'CVX']  
  
for ticker in tickers:  
df = pd.read\_csv(f'../data/raw/{ticker}\_raw.csv', index\_col=0, parse\_dates=True)  
df.index.name = 'Date'  
processed = generate\_features(df)  
processed.to\_csv(f'../data/processed/{ticker}\_processed.csv')  
print(f"{ticker} processed.")

## Code: 02\_feature\_engineering.ipynb DataFrame Analysis

import pandas as pd  
import matplotlib.pyplot as plt  
import os  
import seaborn as sns  
  
tickers = ['AAPL', 'MSFT', 'JPM', 'BAC', 'XOM', 'CVX']  
processed\_dfs = {}  
  
for ticker in tickers:  
file\_path = f'../data/processed/{ticker}\_processed.csv'  
processed\_dfs[ticker] = pd.read\_csv(file\_path, index\_col='Date', parse\_dates=True)  
print(f"Loaded {ticker} data with {processed\_dfs[ticker].shape[0]} rows and {processed\_dfs[ticker].shape[1]} columns")  
  
print("\nApple (AAPL) processed data sample:")  
print(processed\_dfs['AAPL'].head())  
  
print("\nApple (AAPL) summary statistics:")  
print(processed\_dfs['AAPL'].describe())  
  
print("\nCorrelation between AAPL features:")  
plt.figure(figsize=(12, 10))  
sns.heatmap(processed\_dfs['AAPL'].corr(), annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)  
plt.title('AAPL Feature Correlations')  
plt.tight\_layout()  
plt.show()  
  
plt.figure(figsize=(14, 7))  
for ticker in tickers:  
normalized\_price = processed\_dfs[ticker]['Close'] / processed\_dfs[ticker]['Close'].iloc[0] \* 100  
plt.plot(processed\_dfs[ticker].index, normalized\_price, label=ticker)  
plt.title('Normalized Stock Prices (Starting Price = 100)')  
plt.xlabel('Date')  
plt.ylabel('Normalized Price')  
plt.legend()  
plt.grid(True)  
plt.show()  
  
plt.figure(figsize=(14, 7))  
for ticker in tickers:  
plt.plot(processed\_dfs[ticker].index, processed\_dfs[ticker]['Volatility'], label=f"{ticker} Volatility")  
plt.title('Volatility Comparison Across Stocks')  
plt.xlabel('Date')  
plt.ylabel('Volatility (10-day rolling std)')  
plt.legend()  
plt.grid(True)  
plt.show()  
  
tech\_stocks = ['AAPL', 'MSFT']  
finance\_stocks = ['JPM', 'BAC']  
energy\_stocks = ['XOM', 'CVX']  
  
sectors = {  
'Technology': tech\_stocks,  
'Finance': finance\_stocks,  
'Energy': energy\_stocks  
}  
  
sector\_stats = pd.DataFrame()  
  
for sector\_name, sector\_tickers in sectors.items():  
sector\_returns = pd.concat([processed\_dfs[ticker]['Return'] for ticker in sector\_tickers], axis=1)  
sector\_stats[f'{sector\_name}\_Avg\_Return'] = sector\_returns.mean(axis=1)  
  
sector\_volatility = pd.concat([processed\_dfs[ticker]['Volatility'] for ticker in sector\_tickers], axis=1)  
sector\_stats[f'{sector\_name}\_Avg\_Volatility'] = sector\_volatility.mean(axis=1)  
  
print("\nSector average statistics:")  
print(sector\_stats.describe())  
  
plt.figure(figsize=(14, 7))  
for sector in sectors.keys():  
plt.plot(sector\_stats.index, sector\_stats[f'{sector}\_Avg\_Return'], label=f"{sector} Avg Return")  
plt.title('Average Returns by Sector')  
plt.xlabel('Date')  
plt.ylabel('Average Return')  
plt.legend()  
plt.grid(True)  
plt.show()

## Code: 03\_modeling\_single\_layer\_nn.py Full Modeling Script (Single-Layer NN)

import sys  
sys.path.append('..') # Add parent directory to Python path to import modules from parent directory  
import json # For saving feature names to JSON files  
import pandas as pd # For data manipulation and analysis  
import numpy as np # For numerical operations  
import os # For file system operations  
import random # For setting random seed  
import tensorflow as tf # Deep learning framework  
from sklearn.model\_selection import train\_test\_split # For splitting data into train and test sets  
from sklearn.preprocessing import StandardScaler # For standardizing features to zero mean and unit variance  
from sklearn.feature\_selection import mutual\_info\_regression # For feature selection using information theory  
from sklearn.metrics import mean\_squared\_error, r2\_score # For model evaluation  
import matplotlib.pyplot as plt # For data visualization  
from tensorflow.keras.models import Sequential # For creating sequential neural network models  
from tensorflow.keras.layers import Dense # For creating fully connected neural network layers  
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau # For optimizing training process  
import logging # For controlling log output  
  
# Suppress TensorFlow warnings to keep notebook output clean  
# Level 2 hides info and warnings but shows errors  
os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'  
logging.getLogger('tensorflow').setLevel(logging.ERROR)  
  
# Set random seeds for reproducibility across runs  
# This ensures that random operations like weight initialization and train/test splits are consistent  
np.random.seed(42)  
tf.random.set\_seed(42)  
random.seed(42)  
  
# List of stock tickers representing different market sectors  
# AAPL, MSFT: Technology  
# JPM, BAC: Banking/Finance  
# XOM, CVX: Energy/Oil  
tickers = ['AAPL', 'MSFT', 'JPM', 'BAC', 'XOM', 'CVX']  
results = {} # Dictionary to store performance metrics for each ticker  
  
# Iterate through each ticker to build individual models  
for ticker in tickers:  
print(f"Training {ticker}...")  
# Load processed price data for current ticker  
# Parse\_dates ensures that the date index is properly formatted  
df = pd.read\_csv(f'../data/processed/{ticker}\_processed.csv', index\_col='Date', parse\_dates=True)  
df['Close'] = pd.to\_numeric(df['Close'], errors='coerce') # Convert 'Close' column to numeric, handling any errors  
  
# ------ Feature Engineering: Technical Indicators ------  
# Calculate daily percentage change in closing price  
df['Return'] = df['Close'].pct\_change()  
  
# Create lagged returns (previous 1-5 days) to capture short-term price patterns  
# These features help the model understand recent market momentum  
for i in range(1, 6):  
df[f'Lag\_{i}'] = df['Return'].shift(i)  
  
# 10-day price momentum: difference between current price and price 10 days ago  
# Captures medium-term trend direction  
df['Momentum\_10'] = df['Close'] - df['Close'].shift(10)  
  
# Calculate RSI (Relative Strength Index) - popular technical indicator  
# RSI measures the magnitude of recent price changes to evaluate overbought/oversold conditions  
delta = df['Close'].diff() # Daily price difference  
gain = delta.clip(lower=0).rolling(14).mean() # Average gains over 14 days (standard RSI period)  
loss = -delta.clip(upper=0).rolling(14).mean() # Average losses over 14 days  
rs = gain / (loss.replace(0, np.finfo(float).eps)) # Relative strength (gain/loss ratio), avoiding division by zero  
df['RSI'] = 100 - (100 / (1 + rs)) # RSI formula: scales result to 0-100 range  
  
# Calculate MACD (Moving Average Convergence Divergence)  
# MACD is a trend-following momentum indicator showing relationship between two moving averages  
ema12 = df['Close'].ewm(span=12, adjust=False).mean() # 12-day exponential moving average  
ema26 = df['Close'].ewm(span=26, adjust=False).mean() # 26-day exponential moving average  
df['MACD'] = ema12 - ema26 # MACD line is difference between these EMAs  
df['MACD\_signal'] = df['MACD'].ewm(span=9, adjust=False).mean() # 9-day EMA of MACD (signal line)  
  
# ------ Target Variable Creation ------  
# Log returns are preferred in finance as they're additive over time and more normally distributed  
df['LogReturn'] = np.log(df['Close'] / df['Close'].shift(1))  
# Target: 3-day smoothed forward log return  
# Shift(-1) looks one day ahead, rolling(3).mean() smooths over 3 days to reduce noise  
df['Target'] = df['LogReturn'].shift(-1).rolling(3).mean()  
  
# Remove rows with NaN values resulting from the lag and rolling operations  
df.dropna(inplace=True)  
  
# Feature Selection using Mutual Information  
# Mutual information measures how much information one variable provides about another  
# It can capture non-linear relationships unlike correlation  
all\_features = df.drop(columns=['Target']) # All columns except target  
target = df['Target']  
  
# Calculate mutual information between each feature and target  
mi\_scores = mutual\_info\_regression(all\_features, target)  
# Convert to Series, sort by MI score, and select top 10 most informative features  
top\_features = pd.Series(mi\_scores, index=all\_features.columns).sort\_values(ascending=False).head(10).index.tolist()  
  
# Create feature matrix X and target vector y using only selected features  
X = all\_features[top\_features]  
y = target  
  
# ------ Target Normalization ------  
# Normalize target to zero mean and unit standard deviation  
# This improves neural network training stability and convergence speed  
y\_mean, y\_std = y.mean(), y.std()  
y = (y - y\_mean) / y\_std  
  
# ------ Data Splitting ------  
# Split data chronologically (no shuffle) to maintain time-series integrity  
# Use 80% for training and 20% for testing  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, shuffle=False, test\_size=0.2)  
  
# ------ Feature Scaling ------  
# Standardize features to zero mean and unit variance  
# Neural networks perform better with standardized inputs  
scaler = StandardScaler()  
X\_train\_scaled = scaler.fit\_transform(X\_train) # Fit scaler on training data only to prevent data leakage  
X\_test\_scaled = scaler.transform(X\_test) # Apply same transformation to test data  
  
# ------ Model Architecture ------  
# Create a single-layer neural network (linear model)  
# Single dense layer with one output neuron makes this equivalent to linear regression  
model = Sequential([  
Dense(1, input\_shape=(X\_train\_scaled.shape[1],))  
])  
model.compile(optimizer='adam', loss='mse') # Adam optimizer and mean squared error loss function  
  
# ------ Training Callbacks ------  
# Early stopping to prevent overfitting  
# Stops training when validation loss doesn't improve for 15 epochs  
early\_stop = EarlyStopping(patience=15, restore\_best\_weights=True, monitor='val\_loss')  
  
# Learning rate reduction  
# Reduces learning rate by half when validation loss plateaus for 5 epochs  
# Helps fine-tune model training when approaching optimal weights  
reduce\_lr = ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=5, min\_lr=1e-5, verbose=0)  
  
# ------ Model Training ------  
# Train for up to 200 epochs with 20% of training data used for validation  
# verbose=0 suppresses training output for cleaner notebook  
model.fit(X\_train\_scaled, y\_train, validation\_split=0.2, epochs=200, verbose=0,  
callbacks=[early\_stop, reduce\_lr])  
  
# ------ Save Model and Feature List ------  
# Create directory if it doesn't exist  
os.makedirs('../models/saved\_model', exist\_ok=True)  
# Save trained model in Keras format  
model.save(f'../models/saved\_model/{ticker}\_single\_layer\_nn.keras')  
# Save list of selected features for later use in prediction  
with open(f'../models/saved\_model/{ticker}\_single\_layer\_features.json', 'w') as f:  
json.dump(top\_features, f)  
  
# ------ Generate Predictions ------  
# Make predictions on test set  
y\_pred = model.predict(X\_test\_scaled, verbose=0).flatten() # Suppress prediction output  
  
# De-normalize Predictions and Actual Values   
# Convert standardized values back to original scale for meaningful evaluation  
y\_pred = y\_pred \* y\_std + y\_mean  
y\_test\_actual = y\_test \* y\_std + y\_mean  
  
# ------ Calculate Performance Metrics ------  
mse = mean\_squared\_error(y\_test\_actual, y\_pred) # Mean Squared Error  
rmse = np.sqrt(mse) # Root Mean Squared Error - more interpretable in original units  
r2 = r2\_score(y\_test\_actual, y\_pred) # R-squared - proportion of variance explained by model  
results[ticker] = (mse, rmse, r2) # Store metrics for later comparison  
  
# ------ Visualize Results ------  
# Plot actual vs predicted values for visual assessment  
plt.figure(figsize=(10, 4))  
plt.plot(y\_test\_actual.values, label='Actual')  
plt.plot(y\_pred, label='Predicted')  
plt.title(f"{ticker} - Single Layer NN (R² = {r2:.4f})")  
plt.legend()  
plt.grid(True)  
plt.show()  
  
# ------ Summary Output ------  
# Print performance metrics for all tickers for comparison  
print("\nSUMMARY")  
for ticker, (mse, rmse, r2) in results.items():  
print(f"{ticker}: MSE = {mse:.6f}, RMSE = {rmse:.6f}, R² = {r2:.4f}")

## Code: 04\_modeling\_multi\_layer\_nn.ipynb Full Modeling Script (Multi-Layer NN)

import pandas as pd # For data manipulation and analysis  
import json # For saving feature lists  
import numpy as np # For numerical operations  
from sklearn.model\_selection import train\_test\_split # For splitting data into train and test sets  
from sklearn.preprocessing import StandardScaler, RobustScaler # For feature scaling  
from tensorflow.keras.models import Sequential # For building neural network architecture  
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization # For network layers  
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint # For optimizing training  
from tensorflow.keras.regularizers import l2 # For adding regularization to reduce overfitting  
from tensorflow.keras.optimizers import Adam # For adaptive learning rate optimization  
from sklearn.metrics import mean\_squared\_error, r2\_score # For model evaluation  
import matplotlib.pyplot as plt # For visualizing results  
import os # For file system operations  
from sklearn.feature\_selection import mutual\_info\_regression, SelectKBest # For information-based feature selection  
import logging # For controlling log output  
  
# Set random seeds for reproducibility across runs  
# This ensures consistent results for weight initialization and data splitting  
import random  
import tensorflow as tf  
np.random.seed(42)  
tf.random.set\_seed(42)  
random.seed(42)  
  
# Suppress TensorFlow warnings for cleaner output  
os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2' # 0=all, 1=info, 2=warning, 3=error  
logging.getLogger('tensorflow').setLevel(logging.ERROR)  
  
# Define stock tickers from different market sectors for analysis  
tickers = ['AAPL', 'MSFT', 'JPM', 'BAC', 'XOM', 'CVX']  
results = {} # Dictionary to store evaluation metrics for each ticker  
  
# Process each ticker individually with its own model  
for ticker in tickers:  
print(f"\nTraining {ticker}...")  
# Load processed price data from CSV with dates as index  
df = pd.read\_csv(f'../data/processed/{ticker}\_processed.csv', index\_col='Date', parse\_dates=True)  
df['Close'] = pd.to\_numeric(df['Close'], errors='coerce') # Ensure Close prices are numeric  
  
# === Enhanced Feature Engineering ===  
# Basic price-based features: returns and transformations  
df['Return'] = df['Close'].pct\_change() # Daily percentage change in price  
df['Return\_Abs'] = np.abs(df['Return']) # Magnitude of price changes (volatility indicator)  
df['Log\_Return'] = np.log(df['Close'] / df['Close'].shift(1)) # Log returns for statistical properties  
  
# Return lags - capture autocorrelation patterns in price movements  
# Using up to 9 days of history to identify recurring patterns  
for i in range(1, 10):  
df[f'Lag\_{i}'] = df['Return'].shift(i) # Past returns as predictors  
if i <= 5: # Create non-linear transformations for recent days  
df[f'Lag\_{i}\_squared'] = df[f'Lag\_{i}']\*\*2 # Squared terms capture magnitude effects  
  
# Momentum indicators at different timeframes  
# Price differences across various windows to identify trends  
for window in [5, 10, 20, 30]:  
df[f'Momentum\_{window}'] = df['Close'] - df['Close'].shift(window) # Absolute change  
df[f'Momentum\_Ratio\_{window}'] = df['Close'] / df['Close'].shift(window) # Relative change  
  
# Moving averages - common smoothing technique in technical analysis  
# Longer windows identify longer-term trends  
for window in [5, 10, 20, 50, 100]:  
df[f'MA\_{window}'] = df['Close'].rolling(window=window).mean() # Simple moving average  
# Distance between current price and moving average (normalized)  
df[f'Close\_MA\_{window}\_Ratio'] = df['Close'] / df[f'MA\_{window}'] # Relative position to trend  
  
# Volatility features at different timeframes  
# Standard deviation of returns measures price variability  
for window in [5, 10, 20, 30]:  
df[f'Volatility\_{window}'] = df['Return'].rolling(window=window).std() # Historical volatility  
# Normalize short-term volatility relative to recent volatility trend  
if window in [5, 10]:  
df[f'Volatility\_Ratio\_{window}'] = df[f'Volatility\_{window}'] / df[f'Volatility\_{window}'].rolling(window=5).mean()  
  
# RSI (Relative Strength Index) at multiple timeframes  
# Momentum oscillator measuring speed and change of price movements  
for window in [6, 14, 21]: # 14 is standard, also include shorter and longer periods  
delta = df['Close'].diff() # Daily price changes  
gain = delta.clip(lower=0).rolling(window).mean() # Average gains over period  
loss = -delta.clip(upper=0).rolling(window).mean() # Average losses over period  
rs = gain / (loss.replace(0, np.finfo(float).eps)) # Relative strength ratio  
df[f'RSI\_{window}'] = 100 - (100 / (1 + rs)) # RSI formula (0-100 scale)  
  
# MACD (Moving Average Convergence Divergence) - trend-following momentum indicator  
# Shows relationship between two moving averages of price  
ema12 = df['Close'].ewm(span=12, adjust=False).mean() # 12-day exponential moving average  
ema26 = df['Close'].ewm(span=26, adjust=False).mean() # 26-day exponential moving average  
df['MACD'] = ema12 - ema26 # MACD line  
df['MACD\_signal'] = df['MACD'].ewm(span=9, adjust=False).mean() # 9-day EMA of MACD  
df['MACD\_hist'] = df['MACD'] - df['MACD\_signal'] # Histogram (difference between MACD and signal line)  
df['MACD\_hist\_diff'] = df['MACD\_hist'].diff() # Change in histogram (acceleration/deceleration)  
  
# Bollinger Bands - volatility bands placed above and below moving average  
# Wider bands indicate higher volatility, narrower bands indicate lower volatility  
for window in [20]: # 20 is standard period  
middle\_band = df['Close'].rolling(window=window).mean() # Center band (20-day SMA)  
std\_dev = df['Close'].rolling(window=window).std() # Standard deviation of price  
df[f'BB\_upper\_{window}'] = middle\_band + (std\_dev \* 2) # Upper band (2 std devs above)  
df[f'BB\_lower\_{window}'] = middle\_band - (std\_dev \* 2) # Lower band (2 std devs below)  
df[f'BB\_width\_{window}'] = (df[f'BB\_upper\_{window}'] - df[f'BB\_lower\_{window}']) / middle\_band # Band width (volatility)  
df[f'BB\_position\_{window}'] = (df['Close'] - df[f'BB\_lower\_{window}']) / (df[f'BB\_upper\_{window}'] - df[f'BB\_lower\_{window}']) # Position within bands  
  
# Rate of Change - percentage change in price over specified periods  
# Measures momentum by comparing current price to price n periods ago  
for window in [5, 10, 21]:  
df[f'ROC\_{window}'] = df['Close'].pct\_change(window) \* 100 # Expressed as percentage  
  
# Volume features (if available) - provide insight into strength of price movements  
if 'Volume' in df.columns:  
df['Volume'] = pd.to\_numeric(df['Volume'], errors='coerce')  
df['Log\_Volume'] = np.log(df['Volume'] + 1) # Log transformation for statistical properties  
df['Volume\_Change'] = df['Volume'].pct\_change() # Day-over-day volume change  
  
# Volume moving averages - identify abnormal volume levels  
for window in [5, 10, 20]:  
df[f'Volume\_SMA\_{window}'] = df['Volume'].rolling(window).mean()  
df[f'Volume\_Ratio\_{window}'] = df['Volume'] / df[f'Volume\_SMA\_{window}'] # Current volume relative to average  
  
# Price-volume relationships - volume often confirms price movements  
df['Volume\_Return\_Ratio'] = df['Volume'] / (df['Return\_Abs'] + 0.001) # Avoid division by zero  
df['Volume\_Close\_Ratio'] = df['Volume'] / df['Close'] # Volume relative to price level  
  
# Define target variable: 5-day smoothed forward log return  
# Using 5-day smoothing reduces noise in the prediction target  
df['LogReturn'] = np.log(df['Close'] / df['Close'].shift(1))  
df['Target'] = df['LogReturn'].shift(-1).rolling(5).mean() # Looking ahead 1 day, smoothing over 5 days  
  
# Remove rows with NaN values from lag operations and rolling windows  
df = df.dropna()  
  
# Handle outliers by clipping extreme values  
# This prevents rare events from dominating model training  
for col in df.columns:  
if col != 'Close' and col != 'Target': # Preserve original price and target  
lower\_bound = df[col].quantile(0.005) # 0.5% percentile  
upper\_bound = df[col].quantile(0.995) # 99.5% percentile  
df[col] = df[col].clip(lower=lower\_bound, upper=upper\_bound) # Trim outliers  
# Feature selection using mutual information  
# This measures the non-linear relationship between each feature and target  
if df.shape[1] > 30: # Only perform selection if we have many features  
X\_temp = df.drop(columns=['Close', 'Target', 'LogReturn', 'Return']) # Candidate features  
y\_temp = df['Target'] # Target variable  
  
# Calculate mutual information between each feature and target  
mi\_scores = mutual\_info\_regression(X\_temp, y\_temp)  
mi\_df = pd.DataFrame({'Feature': X\_temp.columns, 'MI\_Score': mi\_scores})  
mi\_df = mi\_df.sort\_values('MI\_Score', ascending=False) # Rank features by information content  
  
# Select top 25 features plus essential technical indicators  
top\_features = mi\_df.head(25)['Feature'].tolist() # Top 25 by mutual information  
essential\_features = [col for col in X\_temp.columns if 'RSI' in col or 'MACD' in col  
or 'Momentum' in col or 'Volatility' in col] # Key technical indicators  
selected\_features = list(set(top\_features + essential\_features)) # Combine lists, remove duplicates  
  
# Create feature matrix with only selected features  
X = df[selected\_features]  
else:  
# If few features, just drop redundant ones  
X = df.drop(columns=['Return', 'Close', 'Target', 'LogReturn', 'Log\_Return'])  
  
# Define target vector  
y = df['Target']  
  
# Normalize target variable to improve model convergence  
y\_mean, y\_std = y.mean(), y.std() # Calculate statistics  
# Split chronologically without shuffling to preserve time-series nature  
y\_train, y\_test = train\_test\_split(y, shuffle=False, test\_size=0.2)  
y\_train\_norm = (y\_train - y\_mean) / y\_std # Standardize training targets  
y\_test\_norm = (y\_test - y\_mean) / y\_std # Standardize test targets using same parameters  
  
# Split features chronologically (train-test split)  
X\_train, X\_test = train\_test\_split(X, shuffle=False, test\_size=0.2)  
  
# Scale features with RobustScaler which is less influenced by outliers than StandardScaler  
scaler = RobustScaler() # Uses median and interquartile range instead of mean and variance  
X\_train\_scaled = scaler.fit\_transform(X\_train) # Fit on training data only  
X\_test\_scaled = scaler.transform(X\_test) # Apply same transformation to test data  
  
# === Build a deep neural network with regularization ===  
# Architecture designed to balance complexity, overfitting prevention, and learning capacity  
  
model = Sequential([  
# First layer - largest to capture complex patterns  
Dense(192, activation='relu', input\_shape=(X\_train\_scaled.shape[1],),  
kernel\_regularizer=l2(0.0001)), # L2 regularization prevents large weights  
BatchNormalization(), # Normalizes layer inputs for more stable training  
Dropout(0.3), # Randomly drops 30% of neurons during training to prevent overfitting  
  
# Second layer - halve the neurons  
Dense(96, activation='relu', kernel\_regularizer=l2(0.0001)),  
BatchNormalization(),  
Dropout(0.2), # Gradually decrease dropout rate in deeper layers  
  
# Third layer - further reduction in complexity  
Dense(48, activation='relu', kernel\_regularizer=l2(0.0001)),  
BatchNormalization(),  
Dropout(0.2),  
  
# Fourth layer - final hidden layer for fine-tuning  
Dense(24, activation='relu', kernel\_regularizer=l2(0.0001)),  
BatchNormalization(),  
Dropout(0.1),  
  
# Output layer - single neuron for regression prediction  
Dense(1) # No activation function for unbounded regression output  
])  
  
# Compile with Adam optimizer - adaptive learning rate method  
optimizer = Adam(learning\_rate=0.001) # Initial learning rate  
model.compile(optimizer=optimizer, loss='mse') # Mean squared error loss for regression  
  
# === Training Strategy with Advanced Callbacks ===  
  
# Create a weight folder for this ticker to save checkpoints  
model\_folder = f'../models/saved\_model/{ticker}\_weights'  
os.makedirs(model\_folder, exist\_ok=True)  
  
# Model checkpoint to save the best model during training  
checkpoint = ModelCheckpoint(  
filepath=f'{model\_folder}/best\_checkpoint\_v4.keras', # Use .keras format instead of .h5  
monitor='val\_loss', # Track validation loss  
save\_best\_only=True, # Only save when model improves  
mode='min', # Lower validation loss is better  
verbose=0 # Suppress output messages  
)  
  
# Early stopping to prevent overfitting  
# Stops training when validation loss stops improving  
early\_stop = EarlyStopping(  
monitor='val\_loss',  
patience=25, # Wait 25 epochs before stopping  
restore\_best\_weights=True, # Restore weights from epoch with best validation loss  
mode='min',  
verbose=0 # Suppress output messages  
)  
  
# Learning rate reduction when training plateaus  
# Reduces learning rate when validation loss stops improving  
reduce\_lr = ReduceLROnPlateau(  
monitor='val\_loss',  
factor=0.5, # Multiply learning rate by 0.5 when triggered  
patience=8, # Wait 8 epochs before reducing  
min\_lr=1e-6, # Minimum learning rate  
mode='min',  
verbose=0 # Suppress output messages  
)  
  
# Train model with validation split for monitoring  
print(f"Training {ticker} model...")  
history = model.fit(  
X\_train\_scaled, y\_train\_norm, # Training data  
validation\_split=0.2, # Use 20% of training data for validation  
epochs=100, # Maximum number of epochs  
batch\_size=64, # Process 64 samples per gradient update  
verbose=0, # Suppress epoch-by-epoch output  
callbacks=[early\_stop, reduce\_lr, checkpoint] # Use all three callbacks  
)  
  
# Save final model and selected features  
os.makedirs('../models/saved\_model', exist\_ok=True)  
model.save(f'../models/saved\_model/{ticker}\_multi\_layer\_nn.keras') # Use .keras format  
with open(f'../models/saved\_model/{ticker}\_multi\_layer\_features.json', 'w') as f:  
json.dump(selected\_features, f) # Save feature names for later use  
  
# === Generate predictions and evaluate model performance ===  
y\_pred\_norm = model.predict(X\_test\_scaled, verbose=0).flatten() # Suppress prediction output  
# Convert standardized predictions back to original scale  
y\_pred = y\_pred\_norm \* y\_std + y\_mean # De-normalize predictions  
y\_test\_actual = y\_test.values # Actual values for comparison  
  
# Calculate performance metrics  
mse = mean\_squared\_error(y\_test\_actual, y\_pred) # Mean Squared Error  
rmse = np.sqrt(mse) # Root Mean Squared Error (in original units)  
r2 = r2\_score(y\_test\_actual, y\_pred) # R-squared (proportion of variance explained)  
results[ticker] = (mse, rmse, r2) # Store metrics for comparison  
  
# Visualize predictions vs actual values  
plt.figure(figsize=(10, 4))  
plt.plot(y\_test\_actual, label='Actual')  
plt.plot(y\_pred, label='Predicted')  
plt.title(f"{ticker} - Multi-layer NN (R² = {r2:.4f})")  
plt.legend()  
plt.grid(True)  
plt.show()  
  
# Print summary of results for all tickers  
print("\nSUMMARY OF MULTI-LAYER NN MODEL")  
for ticker, (mse, rmse, r2) in results.items():  
print(f"{ticker}: MSE = {mse:.6f}, RMSE = {rmse:.6f}, R² = {r2:.4f}")