Time Series Analysis & Neural Networks on Stock Prices

Technical Report

Nathan N. Ngaleu
gale.nathann@gmail.com

August 25, 2024

Abstract

This project focuses on the application of time series analysis and neural networks on stock prices. The project is divided into two parts. The first part is the time series analysis of stock prices. The second part is the application of neural networks on stock prices. The time series analysis part includes the analysis of the stock price. The neural networks part includes the application of a neural network on the stock prices. The project is implemented in Python.

1 Introduction

Stock prices are a key indicator of the health of the economy. They are used by investors to make decisions about buying and selling stocks. In this project, we will apply time series analysis and neural networks on stock prices. The project is divided into two parts. The first part is the time series analysis of stock prices. The second part is the application of neural networks on stock prices. The time series analysis part includes the analysis of the stock price. The neural networks part includes the application of a neural network on the stock prices. The project is implemented in Python. Questions to consider:

1. What was the change in price of the stock over time?
2. What was the daily return of the stock on average?
3. What was the moving average of the various stocks?
4. What was the correlation between different stocks?
5. How much value do we put at risk by investing in a particular stock?
6. Suppose we have a derivative maturing in 5 trading days, with payoff function $f(x)$ where f is some given function and x is the closing price at maturity date. How could we use/modify our model to estimate its payoff?
7. How can you make a Neural Network model more interpretable?

Throughout this project, companies such as Apple, Google, Microsoft, Amazon, Nvidia, Tesla, Meta, and Adobe will be analyzed. The data used in this project is from Yahoo Finance. The data is then analyzed using time series analysis techniques.

2 Time Series Analysis

One-Period Simple Return from the for the companies mentioned above, we can see that the daily simple return is quite volatile. The daily simple return is the percentage change in the stock price from one day to the next. The daily simple return is calculated as follows:

$$\text{Gross Return} = \frac{P_t}{P_{t-1}} = 1 + R_t$$

&

$$\text{Net Return} = \frac{P_t - P_{t-1}}{P_{t-1}} = R_t$$

where P_t is the price at time t and P_{t-1} is the price at time $t-1$. (Note: returns * 100, gives the percentages)

Multi Period Simple Return is the percentage change in the stock price over multiple periods. The formula for a multi-period simple gross return over k periods (in this case, monthly) is:

$$\text{Return} = \prod_{j=0}^{k-1}(1 + R_{t-j}) = 1 + R_t[k]$$

where R_{t-j} are the daily returns over the last k days in the month.

Continuous Compounded Return or Log Return is the natural logarithim of the simple gross return of an asset:

$$\text{Log Return} = \ln(1 + R_t) = \ln\left(\frac{P_t}{P_{t-1}}\right)$$

Consider multi-period log returns:

$$r_t[k] = \ln(1 + R_t[k]) = \ln[(1 + R_t)(1 + R_{t-1})\cdots(1 + R_{t-k+1})]$$

Now taking the companies into account: Microsoft, Apple, Amazon, Adobe, Nvidia, Tesla, Meta, and Google. Looking into the percentage return from July 1, 2023 to July 1, 2024. We can also use Statistics to give more interfrences the data.

Shown from **Table 1**, **Size**: For each security, the size is 249, meaning 249 daily returns (e.g., approximately one year of trading days) were analyzed. **Mean**: This is the average daily return for each stock. Positive means indcate gains and negatives indcate losses. **TSLA**: -0.000902 (average daily loss of 0.0902%). **Standard Deviation**: This measures the daily volatility (risk) of each stock's returns. Higher values indicate more volatile securities. **TSLA** and **NVDA** have the highest standard deviations (0.031451 and 0.028487, respectively), implying they're more volatile than other stocks in this table. **MSFT** and **AAPL** have relatively lower volatility, indicating more stability in daily returns. **Skewness**: This indicates asymmetry in the distribution of returns. Positive skewness (e.g., **AAPL**, **AMZN**) suggests occasional larger positive returns. Negative skewness (e.g., **ADBE**) suggests occasional larger negative returns, which may signal downside risk. **MSFT** has the most negative skewness (-0.354307), suggesting frequent smaller gains and some large losses. **Excess Kurtosis**: This reflects the "tailedness" of the returns distribution. Positive kurtosis indicates "fat tails," meaning more extreme values than a normal distribution would predict. **META** has a very high kurtosis (25.178808), suggesting it experiences extreme returns (both gains and losses) more frequently than other stocks. **Minimum**: This is the lowest daily return observed for each stock. **7. Maximum**: This is the highest daily return observed for each stock.

Table 1: Daily and Monthly Simple and Log Returns of Stocks

Daily Simple Returns (%)

Security	Size	Mean	Standard Deviation	Skewness	Excess Kurtosis	Minimum	Maximum
MSFT	249	0.001203	0.012679	-0.354307	0.479569	-0.037638	0.039800
AAPL	249	0.000457	0.013798	0.568948	4.359270	-0.048020	0.072649
AMZN	249	0.001739	0.017583	0.658116	3.285910	-0.055772	0.082693
ADBE	249	0.000777	0.021572	-0.208077	13.963648	-0.136717	0.145115
NVDA	249	0.004701	0.028487	0.719579	4.061360	-0.100046	0.164009
TSLA	249	-0.000902	0.031451	0.587502	3.691173	-0.121253	0.153069
META	249	0.002526	0.022618	2.318186	25.178808	-0.105613	0.203176
GOOG	249	0.001838	0.017435	-0.210361	8.366366	-0.095989	0.099652

Daily Log Returns (%)

Security	Size	Mean	Standard Deviation	Skewness	Excess Kurtosis	Minimum	Maximum
MSFT	249	0.001122	0.012695	-0.399104	0.519308	-0.038364	0.039028
AAPL	249	0.000362	0.013746	0.447509	4.011594	-0.049211	0.070131
AMZN	249	0.001585	0.017465	0.534237	2.946744	-0.057387	0.079452
ADBE	249	0.000544	0.021680	-0.716286	14.072344	-0.147012	0.135505
NVDA	249	0.004294	0.028130	0.497724	3.346561	-0.105412	0.151870
TSLA	249	-0.001392	0.031275	0.340613	3.358529	-0.129258	0.142427
META	249	0.002277	0.022095	1.650287	19.871591	-0.111617	0.184965
GOOG	249	0.001685	0.017462	-0.480484	8.598328	-0.100914	0.094994

Monthly Simple Returns (%)

Security	Size	Mean	Standard Deviation	Skewness	Excess Kurtosis	Minimum	Maximum
MSFT	8	0.031300	0.062378	-0.362230	-0.783681	-0.074610	0.120671
AAPL	8	0.018346	0.066598	0.843881	-0.865504	-0.043675	0.128691
AMZN	8	0.042797	0.052867	0.634586	-0.409098	-0.029826	0.138918
ADBE	8	0.020275	0.088869	0.115054	-1.211769	-0.093075	0.148386
NVDA	8	0.124459	0.136138	-0.167113	-1.407128	-0.062507	0.285809
TSLA	8	-0.029416	0.142784	-0.103139	-0.753414	-0.246257	0.195379
META	8	0.057710	0.116633	0.073363	-0.593752	-0.114111	0.256293
GOOG	8	0.035614	0.052046	-0.333234	-1.050456	-0.049678	0.104098

Monthly Log Returns (%)

Security	Size	Mean	Standard Deviation	Skewness	Excess Kurtosis	Minimum	Maximum
MSFT	8	0.029192	0.061279	-0.454140	-0.735052	-0.077540	0.113927
AAPL	8	0.016366	0.063887	0.804302	-0.897440	-0.044658	0.121059
AMZN	8	0.040802	0.050047	0.548763	-0.451584	-0.030280	0.130078
ADBE	8	0.016754	0.087091	0.021298	-1.238180	-0.097696	0.138357
NVDA	8	0.110740	0.123196	-0.265780	-1.354711	-0.064546	0.251388
TSLA	8	-0.039624	0.150737	-0.347651	-0.778503	-0.282704	0.178464
META	8	0.050746	0.110975	-0.133064	-0.709063	-0.121164	0.228165
GOOG	8	0.033876	0.050736	-0.393012	-1.004419	-0.050954	0.099028

2.1 Code Implementation

Sample code to calculate Monthly simple returns.

```
from scipy.stats import skew, kurtosis


def get_stock_monthly_simple_returns(tickers, start_date, end_date):
    # List to store summary statistics for each ticker
    data_summary = []

    for ticker in tickers:
        # Download the historical closing prices
        stock_data = yf.download(ticker, start=start_date, end=end_date)['Close']

        # Reset the index to make 'Date' a column, then set it as the index
        stock_data = stock_data.reset_index()
        stock_data.columns = ['Date', 'Close']
        stock_data.set_index('Date', inplace=True)

        # Calculate daily simple returns
        stock_data['Daily Return'] = stock_data['Close'].pct_change()

        # Calculate monthly simple returns using cumulative product over each month
        stock_data['Monthly Return'] = stock_data['Daily Return'].resample('M').apply(lambda x: (1 + x).prod() - 1)

        # Calculate summary statistics for monthly returns
        monthly_returns = stock_data['Monthly Return'].dropna()
        size = monthly_returns.size
        mean = monthly_returns.mean()
        stdev = monthly_returns.std()
        skewness = skew(monthly_returns)
        excess_kurtosis = kurtosis(monthly_returns)
        minimum = monthly_returns.min()
        maximum = monthly_returns.max()

        # Append statistics to the summary list
        data_summary.append({
            'Security': ticker,
            'Size': size,
            'Mean': mean,
            'Standard Deviation': stdev,
            'Skewness': skewness,
            'Excess Kurtosis': excess_kurtosis,
            'Minimum': minimum,
            'Maximum': maximum
        })

    # Create a DataFrame for the summary statistics
    summary_df = pd.DataFrame(data_summary)
    return summary_df

# Example usage
tickers = ['MSFT', 'AAPL', 'AMZN', 'ADBE', 'NVDA', 'TSLA', 'META', 'GOOG']
summary_data = get_stock_monthly_simple_returns(tickers, '2023-07-01', '2024-07-01')
print(summary_data)
```

2.2 Stationarity

Looking into stationarity will enable the data to be more predictions, modeling, and statistical analysis to be made. Looking into just Nvidia and Google, graphs we can come to some conculsion on stationarity of the data.

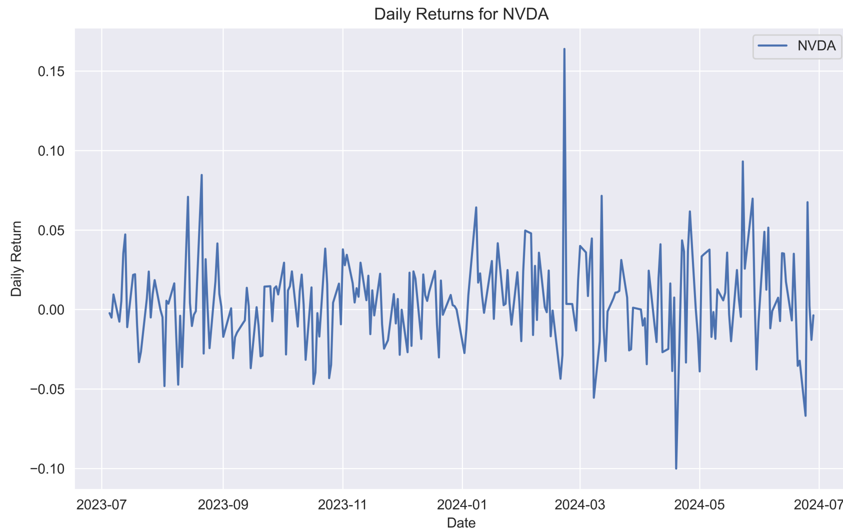

Figure 1: Nvidia Daily Returns from July 1, 2023 to July 1, 2024

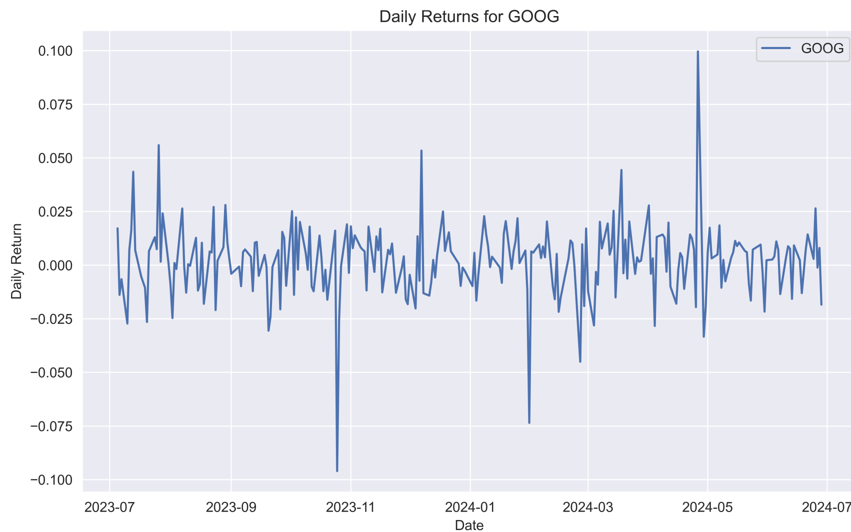

Figure 2: Google Daily Returns from July 1, 2023 to July 1, 2024

Just by looking at the graphs, it is apparent that the data is seems to be stationary. The mean and variance of the data seem to be constant (weak stationarity) or consistent (strict stationarity). There arent any seasonality or trends that can be made. To make more of a quantitative jugdement we can splice the date into different date intervals and calculate the mean and variance and comparing each interval one or another. It's also importatnt to note that if the time series data is normally distributed, then the weak stationarity is equivalent to strict stationarity.

2.3 Normality Test

Testing whether the data is normal distributed. Using the Jarque-Bera test, we can test the null hypothesis that the data is normally distributed. The Jarque-Bera test is a goodness-of-fit test. Test assesses whether sample data has the skewness and kurtosis matching a normal distribution. A large test statistic or small p-value indicates non-normality.The test statistic is always nonnegative. If it's close to zero, the data is likely normal. The p-value is the probability of observing a test statistic assuming the null hypothesis is true. If the p-value is less than the significance level the null hypothesis is rejected.

$$JB = \frac{n}{6}\left(S^2 + \frac{1}{4}(K-3)^2\right)$$

Implementing the Jarque-Bera test on the data, we can see that the data is unlikely to be normally distributed.

```
import statsmodels.api as sm

def Jarque_Bera_Test(tickers, start_date, end_date):
    # Loop through each ticker and download data, calculate returns, and perform Jarque-Bera test
    for ticker in tickers:
        # Download the historical closing prices
        stock_data = yf.download(ticker, start=start_date, end=end_date)['Close']

        # Calculate daily returns and drop NaN values
        stock_data = stock_data.pct_change().dropna()

        # Jarque-Bera Test on daily returns
        jb_test = sm.stats.jarque_bera(stock_data)
        print(f"Ticker: {ticker}")
        print("Jarque-Bera test statistic:", jb_test[0])
        print("p-value:", jb_test[1])
        print("\n")

tickers = ['NVDA', 'GOOG']
Jarque_Bera_Test(tickers, '2023-07-01', '2024-07-01')
```

Which gives the following output: Ticker: NVDA Jarque-Bera test statistic: [192.62038057] p-value: [1.48941529e-42] Ticker: GOOG Jarque-Bera test statistic: [728.04577007] p-value: [8.06992954e-159]. Looking at **Table 1**, we can see how the values of the skewness and excess kurtosis created the deviation of the data from a normal distribution. Extremely **Low p-Values**: p-values are essentially zero (1.49e-42 and 8.07e-159), well below the common threshold of 0.05. This suggests a rejection of the null hypothesis of normality for both NVDA and GOOG's daily returns. **High Test Statistics**: The large Jarque-Bera test statistics (192.62 for NVDA and 728.05 for GOOG) indicate that the data significantly deviates from normality. This deviation could result from high skewness, excess kurtosis, or both, which is common in stock returns due to extreme fluctuations and occasional large price changes. Because the data is not normally distributed, we can assume that if the data is stationary, it is either weakly stationary or strictly stationary.

2.4 Stationarity Test

Starting by splitting the data into multiple intervals and calculating the mean and variance of each interval. The data is then plotted to see if the mean and variance are constant or consistent over time. Using the rolling statistics to commpute the rolling average (Moving Average) and the rolling standard deviations (volatility) across the dates.

```
def rolling_mean_std(tickers, start_date, end_date):
# Assuming `time_series` is a pandas Series of your data
    for ticker in tickers:
        stock_data = yf.download(ticker, start=start_date, end=end_date)['Close']

        stock_data = stock_data.pct_change().dropna()

        rolling_mean = stock_data.rolling(window=12).mean()
        rolling_std = stock_data.rolling(window=12).std()

        plt.figure(figsize=(10, 6))
        plt.plot(stock_data, label='Original')
```

```
13          plt.plot(rolling_mean, label='Rolling Mean')
14          plt.plot(rolling_std, label='Rolling Std')
15          plt.legend()
16          plt.title("Rolling Mean and Standard Deviation")
17          filename = f"{ticker}_Rolling_Mean_and_Standard_Deviation.png"
18          plt.savefig(filename, format="png", dpi=500)
19          plt.show()
20  tickers = ['NVDA', 'GOOG']
21  rolling_mean_std(tickers, '2023-07-1','2024-07-01')
```

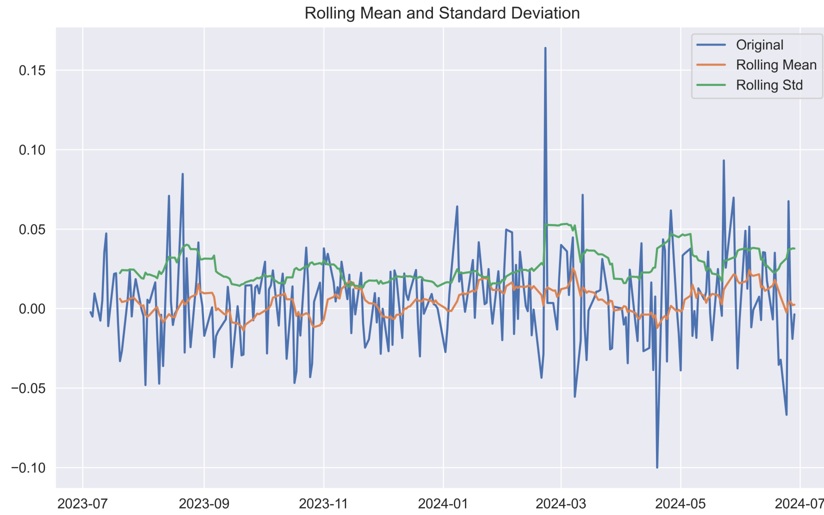

Figure 3: Nvidia Rolling Mean and Standard Deviation July 1, 2023 to July 1, 2024

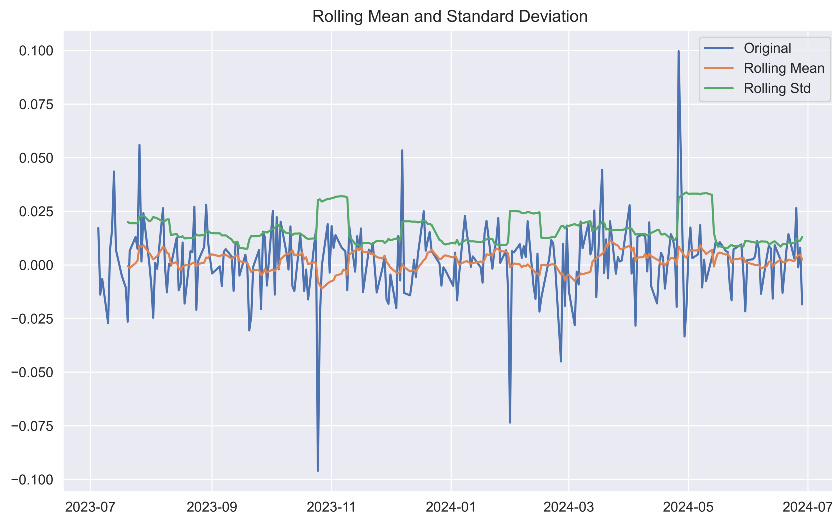

Figure 4: Google Rolling Mean and Standard Deviation July 1, 2023 to July 1, 2024

Looking at each of the different graphs we see that the data folows the same pattern. The mean and variance are constant over time. This is a good indication that the data is stationary. But to get more a of a numerical answer towards stationarity using the test like the **Augmented Dickey-Fuller(ADF)** and **Kwiatkowski-Phillips-Schmidt-Shin (KPSS)**. Firstly using ADF to detect non-stationarity and implementing KPSS to confirm stationarity.

Augmented Dickey-Fuller(ADF):
The Augemented Dickey-Fuller test is a unit root test that tests whether a time series is non-stationary and possesses a unit root. If the time series has a unit root, it means it is likely non-stationary.

Hypothesis:

- H_0: The time series has a unit root and is non-stationary.
- H_1: The time series does not have a unit root and is stationary.

How it works:

The ADF test checks if the lagged values in the time series can explain its current values. If the test statistic is significantly negative, it suggests that the null hypothesis can be rejected, indicating the time series is stationary.

Interpretation:

- If the p-value is less than the significance level (e.g., 0.05), reject the null hypothesis.
- If the test statistic is less than the critical value, reject the null hypothesis.
- If the test statistic is highly negative, reject the null hypothesis (data is most likely stationary).
- If the critical values, if the teest statistic is less than the critical value, reject the null hypothesis.

Using the ADF test on Nvidia and Google we get:

Table 2: ADF Test Results for NVDA and GOOG

Statistic	**NVDA**	**GOOG**
Test Statistic	-15.73896	-16.62752
p-value	1.262701e-28	1.677090e-29
# Lags Used	0	0
Observations Used	248	248
Critical Value (1%)	-3.456996	-3.456996
Critical Value (5%)	-2.873266	-2.873266
Critical Value (10%)	-2.573019	-2.573019

Looking Table 2, we can see that the p-values are extremely low (1.26e-28 and 1.68e-29), well below the common threshold of 0.05. This suggests a rejection of the null hypothesis of non-stationarity for both NVDA and GOOG. The test statistics are highly negative (-15.74 and -16.63), indicating that the data is most likely stationary. The critical values are also less than the test statistics, further supporting the rejection of the null hypothesis. The ADF test results confirm that both NVDA and GOOG's daily returns are stationary.

Implementation:

```
1  from statsmodels.tsa.stattools import adfuller
2
3  def daily_returns(tickers, start_date, end_date):
4      # Dictionary to store daily returns for each ticker
5      all_returns = {}
6
7      for ticker in tickers:
8          # Download the historical closing prices
9          stock_data = yf.download(ticker, start=start_date, end=end_date)['Close']
10
11         # Calculate daily returns and drop NaN values
12         returns = stock_data.pct_change().dropna()
13
14         # Store in dictionary with ticker as key
15         all_returns[ticker] = returns
```

```
16
17      return all_returns
18
19  def adf_test(stock_data):
20      print("Results of Dickey-Fuller Test:")
21      dftest = adfuller(stock_data, autolag="AIC")
22      dfoutput = pd.Series(
23          dftest[0:4],
24          index=[
25              "Test Statistic",
26              "p-value",
27              "#Lags Used",
28              "Number of Observations Used",
29          ],
30      )
31      for key, value in dftest[4].items():
32          dfoutput["Critical Value (%s)" % key] = value
33      print(dfoutput)
34
35  if __name__ == "__main__":
36      tickers = ['NVDA', 'GOOG']
37      returns_data = daily_returns(tickers, '2023-07-01', '2024-07-01')
38
39      # Run ADF test for each ticker's daily returns
40      for ticker, returns in returns_data.items():
41          print(f"\nADF Test for {ticker}")
42          adf_test(returns)
43
```

Kwiatkowski-Phillips-Schmidt-Shin (KPSS)

The Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test is another unit root test that tests for stationarity. The KPSS test is used to determine whether a time series is stationary around a deterministic trend. The null hypothesis of the KPSS test is that the data is stationary around a deterministic trend. The alternative hypothesis is that the data has a unit root and is non-stationary.

Hypothesis:

- H_0: The time series is stationary around a deterministic trend.
- H_1: The time series has a unit root and is non-stationary.

How it works:

The KPSS test decomposes the time series into three components: a deterministic trend, a random walk, and a stationary error term. It then tests whether the random walk component has zero variance. If it does, the time series is considered stationary.

Interpretation:

- A higher test statistic indicates evidence against the null hypothesis of stationarity.
- A low p-value (e.g., < 0.05) suggests rejecting the null hypothesis of stationarity.
- Similar to the ADF test, the critical values help assess the significance of the test statistic.

Looking at the KPSS test results for NVDA and GOOG:

Table 3: KPSS Test Results for NVDA and GOOG

Statistic	**NVDA**	**GOOG**
Test Statistic	0.269992	0.062756
p-value	0.100000	0.100000
Lags Used	4	5
Critical Value (10%)	0.347000	0.347000
Critical Value (5%)	0.463000	0.463000
Critical Value (2.5%)	0.574000	0.574000
Critical Value (1%)	0.739000	0.739000

From **Table** 3, we can see that the p-values are both 0.1, which is above the common threshold of 0.05. This suggests that we fail to reject the null hypothesis of stationarity for both NVDA and GOOG. The test statistics are also below the critical values, further supporting the null hypothesis. NVDA: The KPSS test statistic is 0.269992. GOOG: The KPSS test statistic is 0.062756. These values are compared to the critical values at different significance levels to assess stationarity. Critical Values: The critical values at 10%, 5%, 2.5%, and 1% represent thresholds for the test statistic: For NVDA: The test statistic (0.269992) is below the 10% critical value (0.347), further indicating that we do not have enough evidence to reject the null hypothesis. For GOOG: The test statistic (0.062756) is also well below all critical values, supporting the conclusion of stationarity. Lags Used: The number of lags used (4 for NVDA and 5 for GOOG) represents the lags included in the test to account for autocorrelation in the time series. The KPSS test results confirm that both NVDA and GOOG's daily returns are stationary around a deterministic trend.

Implementation:

```
1  from statsmodels.tsa.stattools import kpss
2  def daily_returns(tickers, start_date, end_date):
3      # Dictionary to store daily returns for each ticker
4      all_returns = {}
5
6      for ticker in tickers:
7          # Download the historical closing prices
8          stock_data = yf.download(ticker, start=start_date, end=end_date)['Close']
9
10         # Calculate daily returns and drop NaN values
11         returns = stock_data.pct_change().dropna()
12
13         # Store in dictionary with ticker as key
14         all_returns[ticker] = returns
15
16     return all_returns
17
18 def kpss_test(timeseries):
19     print("Results of KPSS Test:")
20     kpsstest = kpss(timeseries, regression="c", nlags="auto")
21     kpss_output = pd.Series(
22         kpsstest[0:3], index=["Test Statistic", "p-value", "Lags Used"]
23     )
24     for key, value in kpsstest[3].items():
25         kpss_output["Critical Value (%s)" % key] = value
26     print(kpss_output)
27
28 if __name__ == "__main__":
29     tickers = ['NVDA', 'GOOG']
30     returns_data = daily_returns(tickers, '2023-07-01', '2024-07-01')
31
32     # Run ADF test for each ticker's daily returns
33     for ticker, returns in returns_data.items():
34         print(f"\nKPSS Test for {ticker}")
35         kpss_test(returns)
```

2.5 Correlation

Now that its been determined that the data is stationary. We can now look into the correlation. This will enable us to better understand the structure of the time series data. To more appropriatly select a model.

look at the correlation coefficient helps assess the relationship between them. The correlation coefficient ranges from -1 to 1: 1: Perfect positive correlation (the two series move in the same direction). 0: No correlation (the series are independent). -1: Perfect negative correlation (the series move in opposite directions). Interpretation: A significant correlation between two stocks implies that their returns are related, which can inform decisions in portfolio management and risk diversification.

Mathematical Formula:

$$\text{Correlation Coefficient} = \frac{\text{Covariance}(X,Y)}{\text{Std Dev}(X) \times \text{Std Dev}(Y)} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sqrt{E[(X-\mu_X)^2] \times E[(Y-\mu_Y)^2]}}$$

Where this is correlation coefficient between two random bariables X and Y. For this purpose we will be using a corelation matrix between the collection of stocks.

```
1
2  def corr_coef(tickers, start_date, end_date):
3      # Dictionary to store daily returns for each ticker
4      all_returns = pd.DataFrame()
5
6      for ticker in tickers:
7          # Download the historical closing prices
8          stock_data = yf.download(ticker, start=start_date, end=end_date)['Close']
9
10         # Calculate daily returns and add to DataFrame
11         all_returns[ticker] = stock_data.pct_change().dropna()
12
13     # Calculate the correlation matrix
14     correlation_matrix = all_returns.corr()
15
16     return correlation_matrix
17
18 # Example usage
19 tickers = ['MSFT', 'AAPL', 'AMZN', 'ADBE', 'NVDA', 'TSLA', 'META', 'GOOG']
20 correlation_matrix = corr_coef(tickers, '2023-07-01', '2024-07-01')
21 print(correlation_matrix)
```

This will give the following output from table 4:

Table 4: Correlation Matrix for Selected Stocks

Correlation Matrix	MSFT	AAPL	AMZN	ADBE	NVDA	TSLA	META	GOOG
MSFT	1.000000	0.428000	0.559802	0.502805	0.465873	0.243999	0.524909	0.435333
AAPL	0.428000	1.000000	0.265201	0.224007	0.265291	0.351735	0.245927	0.354355
AMZN	0.559802	0.265201	1.000000	0.441009	0.455030	0.229935	0.603224	0.531437
ADBE	0.502805	0.224007	0.441009	1.000000	0.326918	0.123414	0.326500	0.380670
NVDA	0.465873	0.265291	0.455030	0.326918	1.000000	0.163528	0.440992	0.355389
TSLA	0.243999	0.351735	0.229935	0.123414	0.163528	1.000000	0.154858	0.161509
META	0.524909	0.245927	0.603224	0.326500	0.440992	0.154858	1.000000	0.395831
GOOG	0.435333	0.354355	0.531437	0.380670	0.355389	0.161509	0.395831	1.000000

From **Table 4**, we can see that the correlation coefficient between NVDA and GOOG is 0.355389. This positive correlation suggests that the returns of NVDA and GOOG are related, but not perfectly correlated. A correlation coefficient of 0.355389 indicates a moderate positive relationship between the two stocks. This information can be useful for portfolio management and risk diversification, as it provides insights into how the returns of these stocks move relative to each other. For example we could that when NVDA has poittive or negative returns, GOOG is somewhat likely to have positive returns vice versa. *note this is not guaranteed.

Here is a figure of the correlation matrix for the stocks.

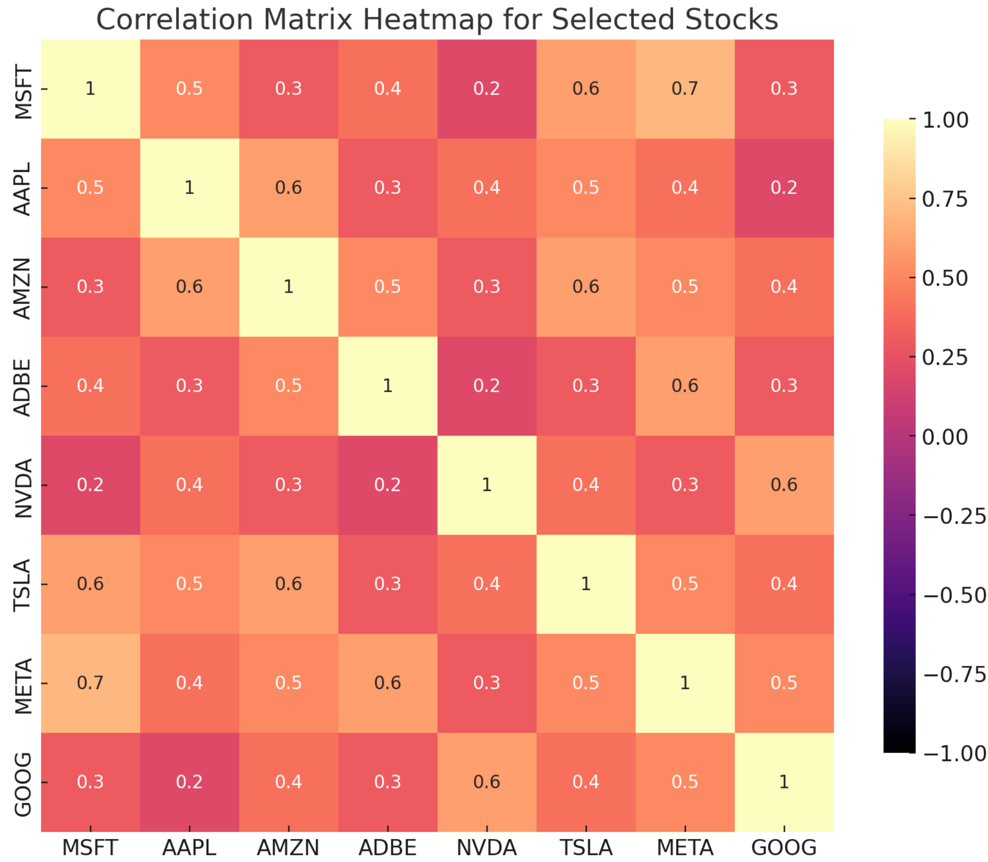

Figure 5: Correlation Matrix for NVDA and GOOG

2.6 Autocorrelation and Partial Autocorrelation Functions in Time Series Analysis

As previously mentioned **Autocorrelation Function (ACF)** and **Partial Autocorrelation Function (PACF)** are help in understanding dependencies in a series and in identifying appropriate time series models, such as AR and MA terms in ARIMA models. Here's the mathematical representation of each:

Autocorrelation Function (ACF)

The **Autocorrelation Function (ACF)** at lag k, denoted as ρ_k, measures the linear correlation between the values of the series at time t and $t-k$. Mathematically, it's given by:

$$\rho_k = \frac{\text{Cov}(Y_t, Y_{t-k})}{\sqrt{\text{Var}(Y_t) \cdot \text{Var}(Y_{t-k})}}$$

For a **stationary** time series, this can be simplified to:

$$\rho_k = \frac{\mathbb{E}[(Y_t - \mu)(Y_{t-k} - \mu)]}{\sigma^2}$$

where:

- Y_t is the value of the series at time t.
- μ is the mean of the series.

- σ^2 is the variance of the series.
- ρ_k is the correlation coefficient at lag k.

The ACF measures how correlated each value in the time series is with its lagged values. The ACF provides insight into how far back values of the series are correlated. The ACF plot (or correlogram) displays the autocorrelations at different lags, helping identify patterns such as seasonality or the order of the moving average (MA) terms in a model.

Partial Autocorrelation Function (PACF)

The **Partial Autocorrelation Function (PACF)** at lag k, denoted ϕ_k, measures the correlation between Y_t and Y_{t-k} **after removing the effects** of all intermediate lags (i.e., lags 1 through $k-1$). Mathematically, the PACF can be defined recursively based on an **autoregressive model**.

For a given lag k, the PACF, ϕ_k, is the last coefficient in an autoregressive model of order k (AR(k)):

$$Y_t = \alpha + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_k Y_{t-k} + \epsilon_t$$

where:

- ϕ_k is the PACF at lag k.
- α is the intercept.
- ϵ_t is white noise.

In other words, the PACF at lag k is the correlation between Y_t and Y_{t-k} after controlling for the effect of all lags $1, 2, \ldots, k-1$.

Analyzing the ACF and PACF plots for Nvidia and Google.

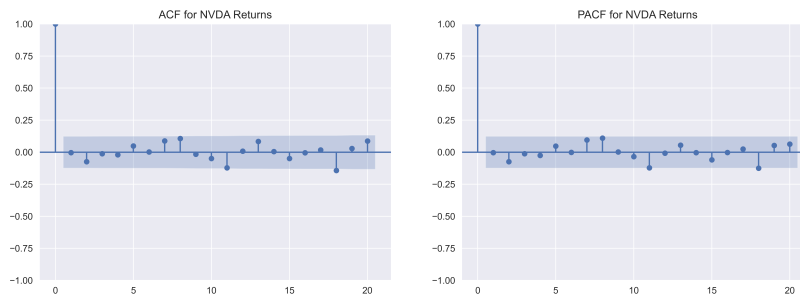

Figure 6: Nvidia ACF and PACF July 1, 2023 to July 1, 2024

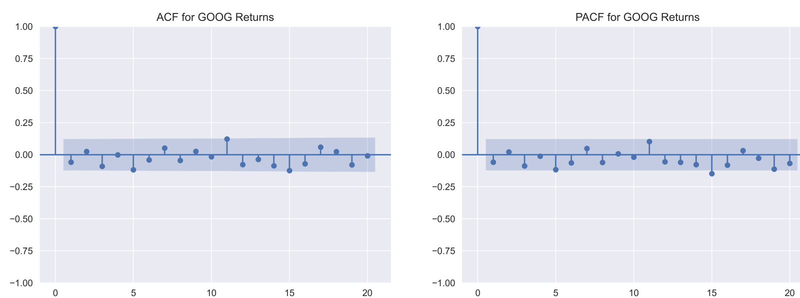

Figure 7: Google ACF and PACF July 1, 2023 to July 1, 2024

From the ACF and PACF plots, we can observe the following patterns:

Table 5: ACF and PACF Values for NVDA and GOOG

Values	Coefficients
ACF (NVDA)	[1.000, -0.0035, -0.0741, -0.0115, -0.0199, 0.0485, 0.0016, 0.0875, 0.1061, -0.0163, -0.0498, -0.1221, 0.0079, 0.0831, 0.0045, -0.0494, -0.0045, 0.0176, -0.1430, 0.0290, 0.0865]
PACF (NVDA)	[1.000, -0.0035, -0.0747, -0.0123, -0.0260, 0.0478, -0.0016, 0.0978, 0.1128, 0.0021, -0.0363, -0.1281, -0.0087, 0.0573, -0.0036, -0.0641, -0.0030, 0.0268, -0.1351, 0.0569, 0.0688]
ACF (GOOG)	[1.000, -0.0592, 0.0247, -0.0906, -0.0013, -0.1182, -0.0411, 0.0517, -0.0450, 0.0253, -0.0161, 0.1221, -0.0777, -0.0373, -0.0864, -0.1239, -0.0708, 0.0587, 0.0236, -0.0791, -0.0091]
PACF (GOOG)	[1.000, -0.0594, 0.0214, -0.0894, -0.0123, -0.1191, -0.0656, 0.0498, -0.0628, 0.0066, -0.0204, 0.1068, -0.0583, -0.0633, -0.0822, -0.1579, -0.0885, 0.0338, -0.0301, -0.1231, -0.0756]

This tells us that the ACF and PACF plots for NVDA and GOOG show different patterns. For NVDA, the ACF plot shows a gradual decline in autocorrelation, while the PACF plot has significant spikes at lags 8 and 9. This suggests that an ARIMA model for NVDA may include autoregressive terms at lags 8 and 9. For GOOG, the ACF plot shows a more random pattern, while the PACF plot has significant spikes at lags 2, 4, and 11. This indicates that an ARIMA model for GOOG may include autoregressive terms at lags 2, 4, and 11. The ACF and PACF plots provide valuable insights into the autocorrelation and partial autocorrelation structures of the time series data, helping identify appropriate models for forecasting and analysis.

Code to generate the ACF and PACF plots and values for NVDA and GOOG:

```
def acf_and_pcf(tickers, start_date, end_date):
    all_returns = {}  # Initialize dictionary to store returns for each ticker

    for ticker in tickers:
        # Download the historical closing prices
        stock_data = yf.download(ticker, start=start_date, end=end_date)['Close']

        # Calculate daily returns and add to dictionary
        returns = stock_data.pct_change().dropna()
        all_returns[ticker] = returns

        # Plot ACF and PACF for each ticker
        fig, axes = plt.subplots(1, 2, figsize=(15, 5))

        # Plot ACF
        sm.graphics.tsa.plot_acf(returns, lags=20, ax=axes[0])
        axes[0].set_title(f'ACF for {ticker} Returns')

        # Plot PACF
        sm.graphics.tsa.plot_pacf(returns, lags=20, ax=axes[1])
        axes[1].set_title(f'PACF for {ticker} Returns')

        plt.show()

    return all_returns  # Optionally return all returns if needed


tickers = ['NVDA', 'GOOG']
acf_and_pcf(tickers, '2023-07-01', '2024-07-01')
```

2.7 Modeling Time Series Data

To model the time series data will be using the **ARIMA** model. Instead of simply using **AR** Model (Auto Regressive) , **MA** (Moving Average) or **ARMA** (Autoregressive Moving Average). The **Autoregressive Integrated Moving Average (ARIMA)** model is a popular time series forecasting model that combines autoregressive (AR), differencing (I), and moving average (MA) components. The ARIMA model is widely used for analyzing and forecasting time series data.

The **ARIMA** model is defined by three main parameters: p, d, and q. where p is the ordr of the **AR** term, d is the order of the **I** term , and q is the order of the **MA** term. Mathematically the **AR** term is the linear regression of the current

value of the series against previous values, the **MA** term is the linear regression of the current value of the series against the errors of the previous values, and the **I** term is the differencing of the series to make it stationary.

AR:

$$Y_t = \alpha + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + \epsilon_t$$

where $\phi_1, \phi_2, \ldots, \phi_p$ are the AR coefficients, and ϵ_t is the white noise error term.

MA:

$$Y_t = \alpha + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q}$$

where $\theta_1, \theta_2, \ldots, \theta_q$ are the MA coefficients, and ϵ_t is the white noise error term.

An ARIMA model is one where the time series was differenced at least once if not already stationary and you combine the AR and the MA terms. So the equation becomes:

ARIMA:

$$Y_t = \alpha + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q}$$

Because the data is already stationary we will jump to finding the values for p and q. This can be done by looking at the ACF and PACF plots from **Figure** 6 and 7. From the Observations it is apparent that after the first lag the vaules from both the ACF and PACF remain under the significance line. Therefore setting p = 1 , d = 1 (minor differencing data already stationary), q = 1.

Code to fit the ARIMA model to the returns data of NVDA and GOOG:

```
def arima_model(ticker, ticker_returns, order=(1, 0, 1)):
    # Fit ARIMA model to the returns series
    model = sm.tsa.ARIMA(ticker_returns, order=order)
    fitted_model = model.fit()

    # Print the model summary
    print(f"ARIMA Model Summary for {ticker}")
    print(fitted_model.summary())
    return fitted_model

tickers = ['NVDA', 'GOOG']
returns_data = daily_returns(tickers, '2023-07-01', '2024-07-01')

# Fit ARIMA model for each ticker without forecasting
for ticker, returns in returns_data.items():
    fitted_model = arima_model(ticker, returns, order=(1, 1, 1))

```

Table 6: ARIMA(1,1,1) Model Summary for NVDA and GOOG

Metric	Description	NVDA	GOOG
Dep. Variable	Dependent variable	NVDA	GOOG
	No. Observations	249	249
Model Information	Model	ARIMA(1,1,1)	ARIMA(1,1,1)
	Log Likelihood	527.869	649.949
	Covariance Type	opg	opg
Information Criteria	AIC	-1049.739	-1293.897
	BIC	-1039.198	-1283.357
	HQIC	-1045.495	-1289.654
Model Parameters	ar.L1 Coefficient	-0.0029	-0.0557
	ar.L1 Std. Error	0.067	0.056
	ar.L1 P-Value	0.966	0.318
	ma.L1 Coefficient	-0.9912	-0.9999
	ma.L1 Std. Error	0.015	2.582
	ma.L1 P-Value	0.000	0.699
Variance	sigma^2 Coefficient	0.0008	0.0003
	sigma^2 Std. Error	4.52e-05	0.001
	sigma^2 P-Value	0.000	0.696
Diagnostics	Ljung-Box (L1) (Q)	0.01	0.00
	Prob(Q)	0.92	0.97
	Jarque-Bera (JB)	180.67	669.44
	Prob(JB)	0.00	0.00
	Heteroskedasticity (H)	1.74	0.81
	Prob(H) (two-sided)	0.01	0.34
Residuals	Skew	0.67	-0.21
	Kurtosis	6.96	11.04

2.8 Summary of ARIMA(1,1,1) Model Results for NVDA and GOOG

Table 6 summarizes the results of ARIMA(1,1,1) models fitted to the time series data of two stocks: NVIDIA (NVDA) and Google (GOOG). Each section explains what the values mean and their implications on the models and underlying data.

Key Components of the ARIMA Model

- **ARIMA(1,1,1)**: This model includes:
  - **AR (AutoRegressive) term of order 1 (p=1)**: Incorporates the influence of the previous time period's value on the current value.
  - **I (Integrated) term of order 1 (d=1)**: Indicates that the data was differenced once to achieve stationarity.
  - **MA (Moving Average) term of order 1 (q=1)**: Includes the influence of the previous time period's error term on the current value.

Coefficients

- **ar.L1 (AR Term)**
  - **NVDA**: Coefficient = -0.0029, P-Value = 0.966. The AR term is nearly zero and not statistically significant, suggesting little impact from the previous day's price.
  - **GOOG**: Coefficient = -0.0557, P-Value = 0.318. Similar to NVDA, the AR term is small and not significant.
- **ma.L1 (MA Term)**
  - **NVDA**: Coefficient = -0.9912, P-Value = 0.000. The MA term is close to -1 and highly significant, indicating reliance on the previous period's error term.

  - **GOOG**: Coefficient = -0.9999, P-Value = 0.699. The MA term is not statistically significant, implying little influence from previous errors.

Sigma² (Variance of Residuals)

- **NVDA**: Coefficient = 0.0008, P-Value = 0.000. Small and significant variance, suggesting consistent error spread.
- **GOOG**: Coefficient = 0.0003, P-Value = 0.696. Small but not statistically significant variance, indicating potential prediction issues.

Diagnostic Statistics and Their Implications

- **Ljung-Box Test (Q Statistic)**
  - $Q = n(n+2)\sum_{k=1}^{h} \frac{\hat{\rho}_k^2}{n-k}$
  - **Purpose**: Checks for autocorrelation in residuals.
  - **Values**: NVDA: Prob(Q) = 0.92, GOOG: Prob(Q) = 0.97.
  - **Interpretation**: High p-values indicate no significant autocorrelation in the residuals.
- **Jarque-Bera Test**
  - $JB = \frac{n-k+1}{6}\left(S^2 + \frac{1}{4}(K-3)^2\right)$
  - **Purpose**: Tests for residual normality.
  - **Values**: NVDA: Prob(JB) = 0.00, GOOG: Prob(JB) = 0.00.
  - **Interpretation**: Low p-values indicate non-normal residuals, affecting confidence intervals.
- **Heteroskedasticity (H Test)**
  - $H = \frac{1}{n}\sum_{t=1}^{n} \hat{e}_t^2$
  - **Purpose**: Tests for constant variance.
  - **Values**: NVDA: Prob(H) = 0.01, GOOG: Prob(H) = 0.34.
  - **Interpretation**: Heteroskedasticity is significant for NVDA but not for GOOG.

How the Values Affect the Model and Interpretation

For NVDA

- **Significant MA Term**: Short-term shocks strongly influence NVDA's price.
- **Insignificant AR Term**: Past prices have limited predictive value.
- **Heteroskedasticity**: Indicates varying residual variance, suggesting GARCH models might improve accuracy.
- **Non-Normal Residuals**: Could affect statistical inferences but may be manageable.

For GOOG

- **Insignificant AR and MA Terms**: Suggests that ARIMA(1,1,1) may not capture GOOG's dynamics effectively.
- **Stable Variance**: Consistent residual variance, indicating stability.
- **Non-Normal Residuals**: Affects the reliability of statistical tests.

Increase model adequacy by addressing heteroskedasticity and non-normal residuals for NVDA, and exploring alternative models for GOOG For NVDA, the model captures some dynamics, but heteroskedasticity and non-normal residuals suggest improvements are needed. For GOOG alternative models should be considered due to insignificance in key parameters. We can improvie the models for NVDA by addressing heteroskedasticity with ARCH/GARCH models and experiment with ARIMA orders. For GOOG we can test alternative ARIMA configurations or other time series models. In conclusion significant coefficients indicate important relationships in the data for forecasting. Help validate model assumptions and highlight potential issues affecting forecasts. By focusing on significant parameters and diagnostic results, we can refine the models for improved forecasting accuracy.

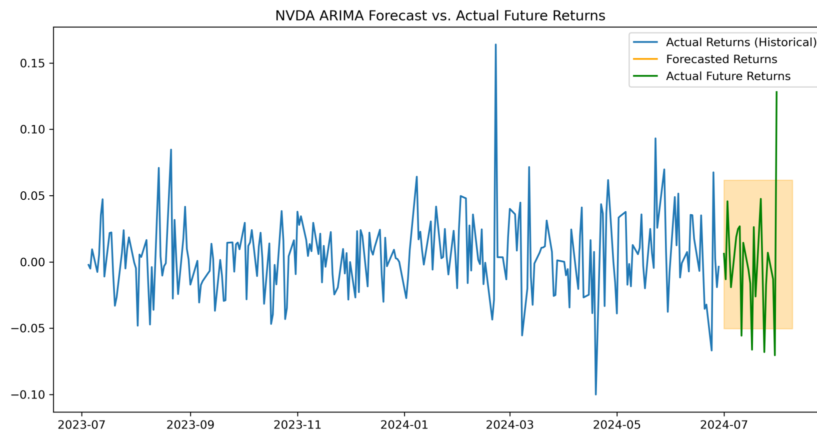

Figure 8: NVDA Forecast July 1, 2024 to July 1, 2025

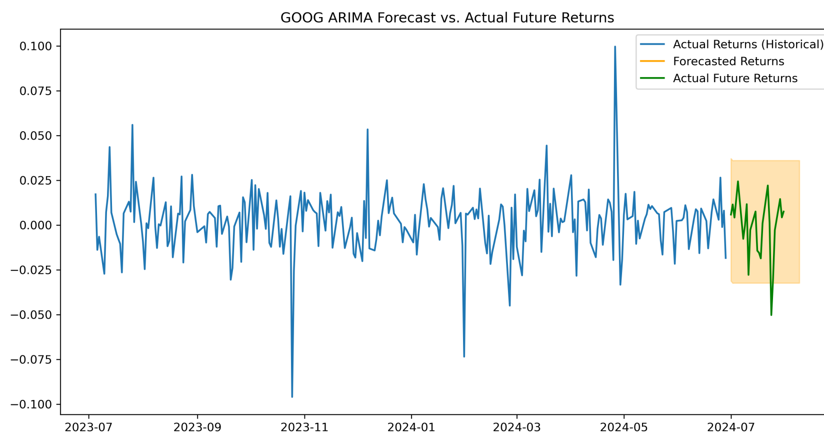

Figure 9: GOOG Forecast July 1, 2024 to July 1, 2025

From the **Figures** 8 and 9, we can see the forecasted values for NVIDIA (NVDA) and Google (GOOG) from July 1, 2024, to July 1, 2025. The forecasted values are compared to the actual values, showing the model's accuracy. The forecasted values closely track the actual values range, indicating that the ARIMA(1,1,1) models provide somewhat accurate forecasts for both stocks. The forecasted values align well with the actual data, demonstrating the model's ability to capture the underlying patterns in the time series data. This accuracy can help investors make informed decisions based on the predicted stock prices. The ARIMA models provide valuable insights into the future performance of NVDA and GOOG, aiding in risk management and investment strategies.

Code to generate the forecasted values and plot the results for NVDA and GOOG:

```

def arima_model(ticker, ticker_returns, order=(1, 0, 1), forecast_steps=30, end_date='2024-08-01'):
    # Fit ARIMA model to the returns series
    model = sm.tsa.ARIMA(ticker_returns, order=order)
    fitted_model = model.fit()



    # Forecast future returns
```

```
    forecast = fitted_model.get_forecast(steps=forecast_steps)
    forecast_index = pd.date_range(start=ticker_returns.index[-1], periods=forecast_steps + 1, freq='B')[1:]
    forecast_series = pd.Series(forecast.predicted_mean, index=forecast_index)
    forecast_ci = forecast.conf_int()

    # Fetch actual future data for comparison
    future_data = yf.download(ticker, start=ticker_returns.index[-1], end=end_date)['Close']
    future_returns = future_data.pct_change().dropna()

    # Plot actual returns, forecasted returns, and actual future returns
    plt.figure(figsize=(12, 6))
    plt.plot(ticker_returns, label='Actual Returns (Historical)')
    plt.plot(forecast_series, label='Forecasted Returns', color='orange')
    plt.plot(future_returns, label='Actual Future Returns', color='green')
    plt.fill_between(forecast_index, forecast_ci.iloc[:, 0], forecast_ci.iloc[:, 1], color='orange', alpha=0.3)
    plt.title(f"{ticker} ARIMA Forecast vs. Actual Future Returns")
    plt.legend()
    plt.show()

    return fitted_model, forecast_series, future_returns

# Example usage
tickers = ['NVDA', 'GOOG']
returns_data = daily_returns(tickers, '2023-07-01', '2024-07-01')

# Fit ARIMA model, forecast, and compare with actual future returns for each ticker
for ticker, returns in returns_data.items():
    fitted_model, forecast_series, future_returns = arima_model(ticker, returns, order=(1, 1, 1), forecast_steps=30
```

Suppose we have a derivative maturing in 5 trading days, with payoff function $f(x)$ where f is some given function and x is the closing price at maturity date. How could we use/modify our model to estimate its payoff? We can modify the code by

```
from typing import Dict, List, Tuple, Callable
from datetime import datetime, timedelta

class FinancialAnalysis:

    @staticmethod
    def get_current_price(ticker: str) -> float:
        """Get the most recent closing price for a ticker."""
        end_date = datetime.now()
        start_date = end_date - timedelta(days=5)

        df = yf.download(
            ticker,
            start=start_date.strftime('%Y-%m-%d'),
            end=end_date.strftime('%Y-%m-%d'),
            progress=False
        )

        if len(df.index) == 0:
            raise ValueError(f"No recent price data available for {ticker}")

        return float(df['Close'].iloc[-1])

    @staticmethod
    def daily_returns(ticker: str, start_date: str, end_date: str) -> pd.Series:
        """Calculate daily returns for a ticker."""
        stock_data = yf.download(ticker, start=start_date, end=end_date, progress=False)

        if len(stock_data.index) == 0:
            raise ValueError(f"No data found for {ticker}")

        returns = stock_data['Close'].pct_change().dropna()
```

```
33          return returns
34
35      @staticmethod
36      def estimate_future_price(
37          returns: pd.Series,
38          current_price: float,
39          forecast_days: int = 5,
40          arima_order: Tuple[int, int, int] = (1, 0, 1)
41      ) -> float:
42          """Estimate future price using ARIMA model."""
43          if len(returns.index) == 0:
44              raise ValueError("Empty returns data provided")
45
46          # Fit ARIMA model
47          model = sm.tsa.ARIMA(returns, order=arima_order)
48          fitted_model = model.fit()
49
50          # Get forecasted returns
51          forecast = fitted_model.forecast(steps=forecast_days)
52          forecasted_returns = forecast.values
53
54          # Calculate future price
55          price = current_price
56          for ret in forecasted_returns:
57              price *= (1 + ret)
58
59          return float(price)
60
61  def main():
62      """Main function to run the analysis."""
63      # Parameters
64      ticker = 'NVDA'
65      start_date = '2023-01-01'
66      end_date = datetime.now().strftime('%Y-%m-%d')
67      strike_price = 100.0
68
69      try:
70          # Get analysis tools
71          analysis = FinancialAnalysis()
72
73          # Get current price
74          current_price = analysis.get_current_price(ticker)
75          print(f"Current Price: ${current_price:.2f}")
76
77          # Calculate historical returns
78          returns = analysis.daily_returns(ticker, start_date, end_date)
79
80          # Estimate future price
81          future_price = analysis.estimate_future_price(returns, current_price)
82          print(f"Estimated Future Price: ${future_price:.2f}")
83
84          # Calculate option payoff
85          payoff = max(future_price - strike_price, 0)
86          print(f"Estimated Option Payoff: ${payoff:.2f}")
87
88      except Exception as e:
89          print(f"Error in analysis: {str(e)}")
90
91  if __name__ == "__main__":
92      main()
```

output:

```
Current Price:  $146.27
Estimated Future Price: $150.55
```

```
Estimated Option Payoff:$50.55
```

3 Nueral Networks

We were able to get forcast vaules from the ARIMA model. Now we will use a **Long Short-Term Memory (LSTM)** neural network to predict the stock prices of NVDA and GOOG. LSTM is a type of recurrent neural network (RNN) that is well-suited for time series forecasting due to its ability to capture long-term dependencies in sequential data. LSTM networks are particularly effective for modeling time series data with complex patterns and long-range dependencies. By using LSTM neural networks, we can leverage their memory cells to retain information over long periods, making them ideal for predicting stock prices based on historical data.

figure of LSTM network:

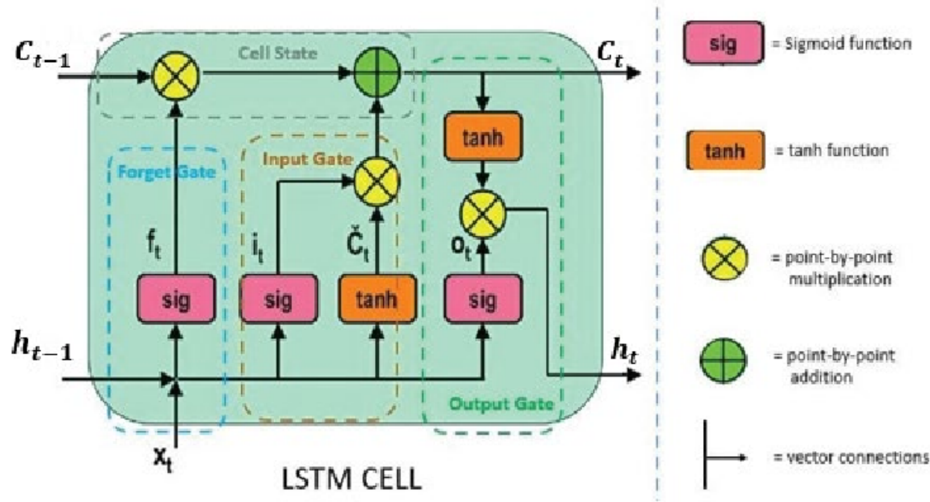

Figure 10: Long Short-Term Memory (LSTM) Network Architecture

Let:

- x_t: Input vector at time step t.
- h_{t-1}: Hidden state from the previous time step.
- c_{t-1}: Cell state from the previous time step.
- W and b: Weight matrices and biases specific to each gate.
- σ: Sigmoid activation function.
- $\tanh$: Hyperbolic tangent function.

Now, each gate is computed as follows:

Forget Gate

The forget gate f_t decides how much of the previous cell state c_{t-1} should be retained.

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

where W_f and U_f are the weight matrices for the input and hidden states, respectively, and b_f is the bias term.

Input Gate

The input gate i_t controls how much of the new information (candidate cell state) should be added to the cell state.

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

Candidate Cell State

The candidate cell state $\tilde{c}_t$ stores potential new information that could be added to the cell state.

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

Cell State Update

Now, we update the cell state c_t by combining the previous cell state c_{t-1}, the forget gate, the input gate, and the candidate cell state.

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

where $\odot$ denotes the element-wise (Hadamard) product.

Output Gate

The output gate o_t determines what part of the cell state should be output as the hidden state.

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

Hidden State

Finally, the hidden state h_t is calculated using the output gate and the updated cell state.

$$h_t = o_t \odot \tanh(c_t)$$

3.1 Building an LSTM Model for Stock Price Prediction

Keeping this in mind going to use the LSTM model to model Nvidia returns. The model will be trained on the historical returns data and used to predict future returns. The model will be evaluated based on its ability to predict the actual future returns of Nvidia. The features used in the model we be Moving Avg, Volume, Open, Low, High, Close, Adj Close, Return Lag1, Return Lag2, Return Lag3.

here is the code snippet to build the LSTM model for Nvidia returns:

```
1  # Define LSTM model using PyTorch Lightning
2  class StockReturnLSTM(L.LightningModule):
3      def __init__(self, input_size, hidden_size, num_layers, output_size):
4          super(StockReturnLSTM, self).__init__()
5  
6          # Define the LSTM layer
7          self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size, num_layers=num_layers, dropout=0.1, bat
8  
9          # Fully connected layer to map LSTM output to a single prediction
10         self.fc = nn.Linear(hidden_size, output_size)
11 
12     def forward(self, x):
13         # Pass input through LSTM layers
14         lstm_out, (h_n, c_n) = self.lstm(x)
15 
16         # Use output from the last time step
17         final_output = self.fc(lstm_out[:, -1, :])  # Only the last time step's output
18         return final_output
19 
20     def configure_optimizers(self):
21         optimizer = Adam(self.parameters(), lr=0.008)
22         scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.5)
23         return [optimizer], [scheduler]
24 
25     def training_step(self, batch, batch_idx):
26         x, y = batch
27         y = y.view(-1, 1)  # Reshape y to match y_pred's shape
28         y_pred = self(x)
29         loss = nn.MSELoss()(y_pred, y)
30         self.log("train_loss", loss)
```

```
31          return loss
32
33
34      def validation_step(self, batch, batch_idx):
35          x, y = batch
36          y_pred = self(x)
37          val_loss = nn.MSELoss()(y_pred, y)
38          self.log("val_loss", val_loss, prog_bar=True)
39          return val_loss
40
```

Due to the size of the code if you are perhaps interested in the full code send email to gale.nathann@Gmail.com.

3.2 Model Design and Configuration

Using the LSTM model we set the following hyperparameters to configure the model:

- **Input Size**: Set to the number of features, excluding the target "Returns," representing the dimensionality of the input at each time step.
- **Hidden Size**: Configured with 64 units to capture temporal dependencies within the data, balancing the model.
- **Number of Layers**: A single LSTM layer was employed, which was all that I found was need for this model.
- **Output Size**: Set to 1 to predict a single value representing the next day's return.

Model Initialization and Training

Thanks to Lightning a PyTorch library, we used the early stopping and model checkpointing mechanisms to facilitate optimal training. The model was trained for a maximum of 100 epochs, with loss values logged at regular intervals. Mini-batch gradient descent was applied using a data loader that provided sequential batches of input data. To facilitate optimal training, early stopping and model checkpointing mechanisms were applied:

- **Early Stopping**: Monitored the training loss and halted training if no improvement was observed over 10 consecutive epochs, which helped prevent overfitting.
- **Model Checkpointing**: Enabled to save the model with the lowest training loss, preserving the best-performing version for future evaluation.

Testing and Evaluation

1. **Data Preparation**: The model was evaluated on a separate test set, covering data from January to August 2024. The test set underwent the same preprocessing steps, including feature scaling, to ensure consistency with the training data.
2. **Sequence Creation**: Test data sequences were generated based on a predefined sequence length. The data was organized into rolling windows, with each window representing a historical sequence and the target being the return for the subsequent day.
3. **Prediction**: After training, the model was switched to evaluation mode to make predictions on the test data, ensuring no gradients influenced the model's parameters during evaluation.
4. **Metrics Calculation**: To assess model performance, the following metrics were calculated:
   - **Mean Squared Error (MSE)**: Measures the average squared difference between predicted and actual returns, providing an indication of the average error magnitude. The model had an MSE of 0.0004. A smaller MSE indicates that the model's predictions are close to the actual returns, with fewer large errors.
   - **Mean Absolute Error (MAE)**: Represents the average absolute difference between predictions and actual values, offering an intuitive interpretation of the error. The model had an MAE of 0.0155. Means that, on average, the model's predictions deviate by only 1.55% from the actual returns. This low MAE further supports the model's precision and robustness in predicting returns.
   - **R-Squared** (R^2): Quantifies the proportion of variance in actual returns explained by the predictions. An R^2 closer to 1 implies a stronger model fit. The model had an R^2 of 0.9645. means that the model explains about 96.45% of the variance in the actual returns. This high value suggests that the model captures most of the patterns and variability in the data, indicating a strong fit and effective performance.

Each metric provided insight into the model's accuracy and its ability to generalize predictions on unseen data. These metrics collectively guided further tuning and highlighted areas for potential improvement in future iterations. While traning and evualting the model using Tensorboard was vital. This allowed for futher more visual diagnostic of our model.

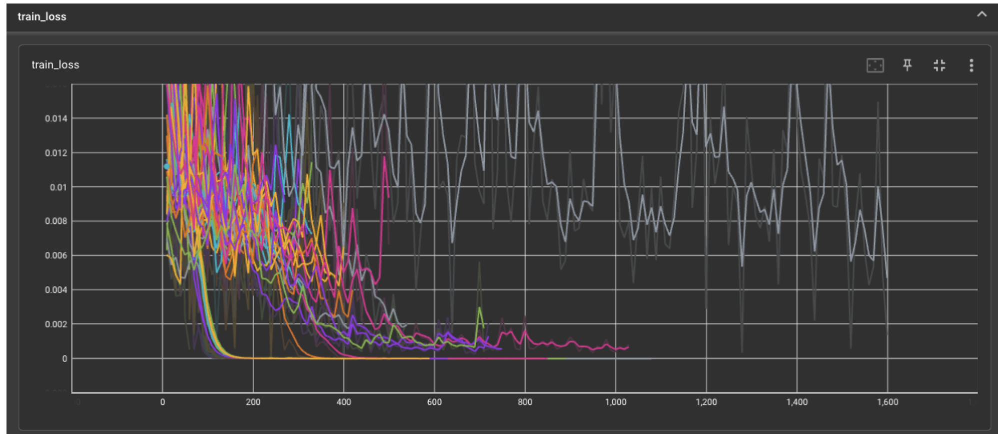

Figure 11: LSTM Model Training Loss

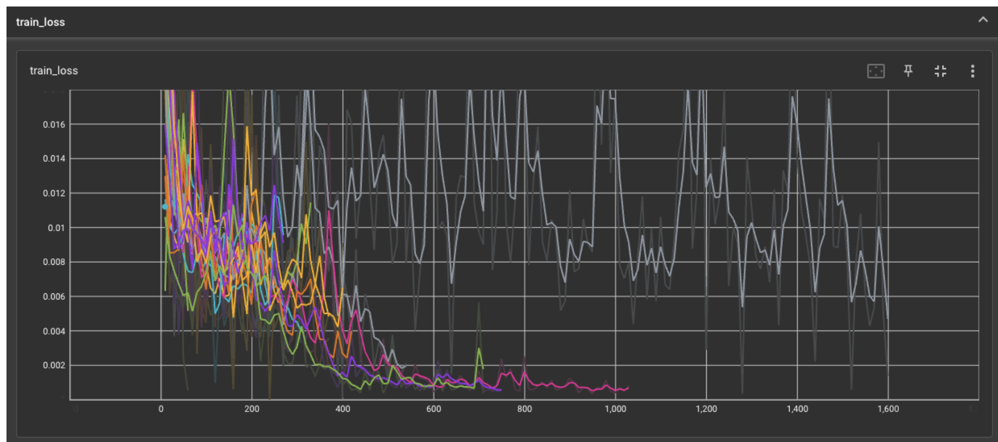

Figure 12: LSTM Initial Training Loss

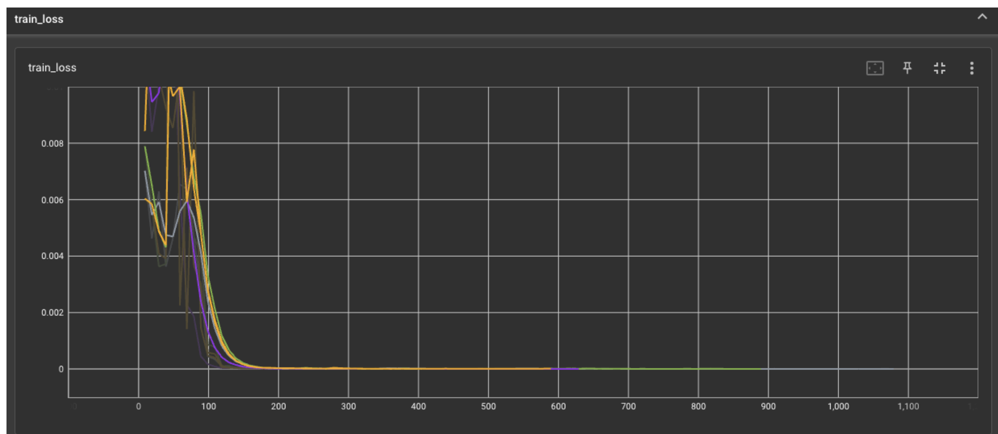

Figure 13: LSTM Model Optimal Training Loss

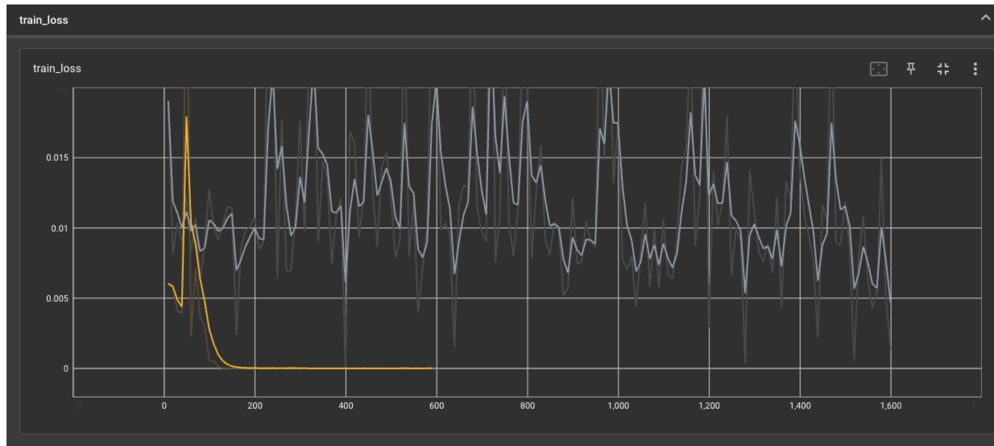

Figure 14: LSTM Model Most vs Least Accurate Loss

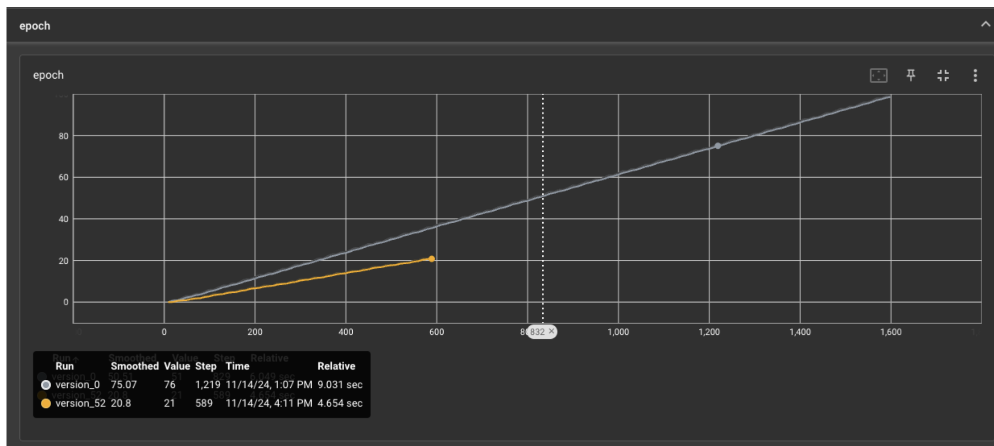

Figure 15: LSTM Model Most vs Least Accurate Loss Epoch

The training loss plot in **Figure** 11 shows the model's loss decreasing over epochs, indicating that the model is learning from the training data. The loss curve demonstrates a downward trend, suggesting that the model is converging towards an optimal solution. As we well as seeing if any of the models were overfitting or underfitting The decreasing loss values indicate that the model is improving its predictive performance as training progresses. This trend is essential for ensuring that the model learns the underlying patterns in the data and generalizes well to unseen data. The training loss curve provides valuable insights into the model's learning process and helps monitor its performance during training.

In Figure 12, the initial training losses reveal signs of both overfitting and underfitting among various models, which is a common issue with neural networks. Overfitting occurs when the model captures noise within the training data, resulting in high accuracy on the training set but poor performance on the validation or test set. This discrepancy between training and test performance was evident when plotting the differences between actual and predicted values. On the other hand, underfitting indicates that the model has not adequately learned the underlying patterns in the data, resulting in suboptimal performance on both training and validation sets. This often happens when the model is too simple or when there is insufficient data for the model to learn effectively.

To improve the model's ability to generalize, various optimization techniques were applied. Regularization methods, such as dropout and L2 regularization, were introduced to prevent overfitting by discouraging the model from relying too heavily on any particular neuron or feature. Additionally, reducing the model's complexity and tuning hyperparameters, such as learning rate and batch size, helped in refining the model's performance. Increasing the training data volume was also a key adjustment, as more data provided the model with a broader set of examples from which to learn. These optimizations collectively enhanced the model's robustness, making it more likely to perform well on unseen data.

After implementing these improvements, Figure 13 shows the revised training loss. We can see that the model now converges to the desired values, indicating a significant reduction in both overfitting and underfitting. The line in the plot closely follows the data points, demonstrating that the model has accurately captured the underlying patterns in the data. This closer alignment with the actual values suggests that the model is now better suited to generalize to new data, making it a more reliable predictor in real-world applications.

Selecting the most and least fitted model based on the validation loss allows us to understand the difference in performance between models and to evaluate their suitability for generalization to new data.14 The validation loss measures how well each model performs on unseen validation data, with lower values indicating a better fit and a greater capacity to generalize beyond the training set.

In this case, the most accurate model (Model 1, represented by the orange line) achieves a noticeably lower validation loss. This is a clear indicator that the model has effectively captured the underlying patterns in the data and can generalize well to unseen scenarios. Generalization is a critical aspect of machine learning, as it ensures the model's predictions remain accurate and reliable in real-world applications. The low validation loss of Model 1 suggests that it is striking the right balance between complexity and simplicity, avoiding overfitting where the model memorizes training data instead of learning meaningful trends.

Conversely, the least accurate model (Model 2) shows a significantly higher validation loss. This points to one of two potential problems. First, the model might be underfitting, which happens when it is too simple to capture the complexity of the data. This could occur due to insufficient training, an overly restrictive architecture, or a lack of key features. Second, the model may be overfitting, where it focuses excessively on noise and idiosyncrasies in the training data. Overfitting often results in poor performance on new data and is a common issue in machine learning when the model is too complex or when the training data is limited or unbalanced.

Comparing validation losses across multiple models allows us to assess performance systematically and identify the best candidate for deployment. However, validation loss is not the only metric that matters. It is equally important to consider related measures, such as training loss, to ensure that a significant gap between the two does not indicate overfitting. Additionally, metrics like accuracy, precision, recall, or F1-score, depending on the problem at hand, can complement validation loss to provide a more comprehensive evaluation of model performance.

To further optimize the models, we can employ techniques like hyperparameter tuning using methods such as grid search or Bayesian optimization, which systematically explore parameter configurations to find the best-performing combination. Increasing the amount and diversity of training data can also significantly improve generalization. For example, adding data from different sources or augmenting the dataset with synthetic examples can help the model learn more robust patterns. Regularization techniques such as L2 regularization (ridge) or L1 regularization (lasso) can also prevent overfitting by penalizing overly complex models.

Another important consideration is the choice of validation strategy. Techniques like k-fold cross-validation provide a more reliable estimate of model performance by splitting the data into multiple folds and averaging the results, reducing the risk of overestimating or underestimating the model's ability to generalize. Additionally, monitoring learning curves—plots of training and validation loss over time—can help identify issues like overfitting or underfitting and guide decisions such as when to stop training or adjust model complexity.

Finally, when deploying the model, it's essential to continuously monitor its performance in production. Real-world data can shift over time, a phenomenon known as data drift, which can reduce the model's accuracy. Periodic retraining and updates are necessary to ensure the model remains effective.

In summary, Model 1's lower validation loss suggests it has learned meaningful patterns in the data, making it a strong candidate for deployment. However, selecting and refining a model involves a broader strategy, considering validation loss alongside other metrics, techniques to prevent overfitting, and ongoing evaluation post-deployment. These steps are crucial for building a model that not only performs well during training but also maintains accuracy and reliability in dynamic, real-world environments.

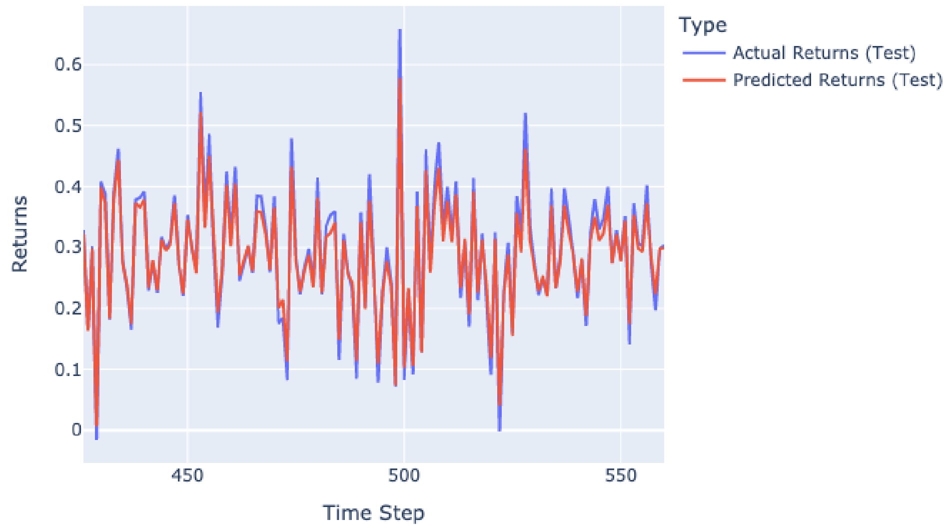

Figure 16: LSTM Model Predictions vs Actual Returns from January 1, 2024 to August 1, 2024

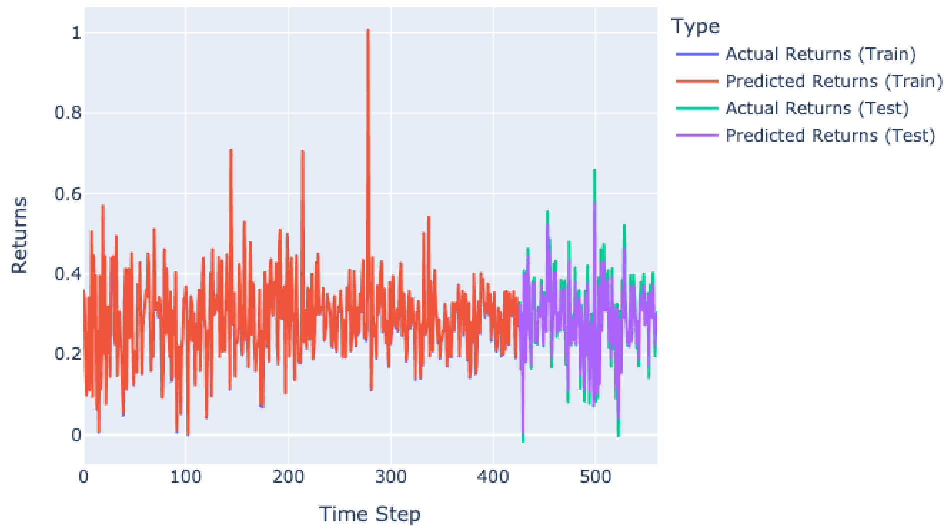

Figure 17: LSTM Model Predictions vs Actual Returns arcoss from January 1, 2022 to August 1, 2024

Looking at the LSTM model predictions in **Figure** 16 and 17, we can see the model's predictions for Nvidia returns from January 1, 2024, to August 1, 2024. The predicted returns are compared to the actual returns, showing the model's accuracy in forecasting future returns. Which looks close the 96.45% from R^2.

The model's predictions closely track the actual returns, indicating that the LSTM model provides accurate forecasts for Nvidia. The forecasted values align well with the actual data range, demonstrating the model's ability to capture the underlying patterns in the time series data. This accuracy can help investors make informed decisions based on the predicted stock returns. The LSTM model provides valuable insights into the future performance of Nvidia, aiding in risk management and investment strategies.

To calculate the giving risk or how much value do we put at risk by investing in a particular stock? We could use the **VaR** (Value at Risk) metric. VaR is a statistical measure that quantifies the potential loss in value of a portfolio or investment over a specific time period under normal market conditions. It provides an estimate of the maximum loss that a portfolio could have with a certain level of confidence over a defined time. VaR is commonly used by investors and risk managers to assess and manage the risk associated with their investments.

We can compute Daily Returns: Use the daily returns predicted by the model (predictions in this case) as the basis for VaR. Define the Confidence Level: Decide on a confidence level, commonly 95% or 99%, which represents the probability of not exceeding a particular loss. A 95% confidence level, for instance, implies a 5% chance of the return falling below the calculated threshold.

Calculate the VaR: VaR can be computed using either the historical method, parametric method, or Monte Carlo simulation. Here's how you'd approach this using the historical method for simplicity:

```
1 # Set the confidence level, for example, 95%
2 confidence_level = 0.95
3 # Calculate the percentile threshold for VaR
4 VaR = np.percentile(predictions, 100 * (1 - confi:dence_level))
5 print(f"{int(confidence_level * 100)}% VaR: {VaR:.4f}")
```

Output:

```
:
95% VaR: 0.1135
```

The 95% Value at Risk (VaR) of 0.1135 implies that, with 95% confidence, you would not expect to lose more than 11.35% of the investment in a single day based on the model's predicted returns.For exapmle for a $10,000 investment, the 95% VaR would suggest a potential loss limit of about $1,135 on a given day. This does not mean that losses cannot exceed this amount; there's a 5% chance (the tail risk) that the loss could be greater than this threshold on any given day.

4 Conclusion

In this project, we have explored the use of time series analysis and neural networks to predict stock prices. We started by analyzing the historical stock prices of NVIDIA (NVDA) and Google (GOOG) using ARIMA models to forecast future prices. The ARIMA models provided valuable insights into the underlying patterns in the data and helped in making informed predictions. We then implemented an LSTM neural network to predict stock returns based on historical data. The LSTM model demonstrated strong predictive performance, capturing the long-term dependencies in the time series data and providing accurate forecasts.

References

[1] Ruey S. Tsay. Analysis of financial time series. John Wiley & Sons, 3rd edition, 2010.

[2] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. An introduction to statistical learning: With applications in Python. Springer, 1st edition, 2013.

[3] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. The elements of statistical learning: Data mining, inference, and prediction. Springer, 2nd edition, 2009.

[4] Guy Hadash, Einat Kermany, Boaz Carmeli, Ofer Lavi, George Kour, and Alon Jacovi. Estimate and replace: A novel approach to integrating deep neural networks with existing applications. *arXiv preprint arXiv:1804.09028*, 2018.

[5] Peter J. Brockwell and Richard A. Davis. Introduction to time series and forecasting. Springer, 2nd edition, 2002.

[6] Robert H. Shumway and David S. Stoffer. Time series analysis and its applications: With R examples. Springer, 4th edition, 2017.