LSTM Neural Network

An **LSTM (Long Short-Term Memory**) is a type of recurrent neural network tailored for sequences, especially time series, by using a memory cell and three gates (**forget, input, output**) to control information flow. This architecture lets it learn long-range dependencies without suffering from vanishing or exploding gradients. In practice, we first:

- Window our series into input–target pairs and normalize them. Then,
- Stack one or more LSTM layers (possibly bidirectional) and add dense output layers. And finally, we
- Train (e.g., with Adam) on a regression loss (like MSE) for forecasting.

Their strength lies in capturing both short-term fluctuations and long-term trends, making LSTMs a go-to for time-series prediction.

Comparing Forecasting through ARMA-GARCH with LSTM on Time-Series Data

Financial time series, such as daily closing prices of publicly traded stocks, exhibit complex patterns including autocorrelation, volatility clustering, and non-stationarity. Classical approaches like ARMA (AutoRegressive Moving Average) augmented with a GARCH (Generalized AutoRegressive Conditional Heteroskedasticity) error model are well-established for capturing linear dependencies and time-varying volatility. More recently, deep learning models, particularly Long Short-Term Memory (LSTM) networks, have shown promise in learning nonlinear patterns and long-range dependencies directly from raw data.

In this section, we compare the forecasting performance of an LSTM network against an ARMA(1,1)–GARCH(1,1) pipeline on daily closing prices of Yahoo Inc. from November 23, 2015 to November 20, 2020. Our objectives are to:

1. Assess whether the LSTM's ability to model complex, non-linear dynamics translates into more accurate short-term forecasts.
2. Evaluate how well a classical ARMA-GARCH approach handles linear trends and volatility clusters compared to the LSTM.
3. Understand the trade-offs between model interpretability, computational cost, and predictive accuracy.

We begin by outlining our data preprocessing pipeline then detail the architecture and training regimen for our LSTM model. Next, we describe the selection

of ARMA and the fitting of a GARCH error structure to capture conditional heteroskedasticity. Forecasts from both models are evaluated over an out-of-sample test set using standard error metrics (MSE, MAE, RMSE, and MAPE). Finally, we discuss the empirical results and their implications for financial forecasting practice.

LSTM Model

Data Preprocessing

To prepare the daily Yahoo stock data for our forecasting comparison, we first imported the raw CSV into a time-indexed DataFrame and focused on five core series—High, Low, Open, Volume, and Close. In the first approach, we experimented with different look-back windows (time steps of 5, 10, 20, 40, and 80 days) to assess how varying historical context affects predictive performance. After converting the Date column to a `DateTimeIndex`, we extracted the feature matrix and target vector (the next-day Close price). All feature columns were scaled to the [0, 1] range using a Min-Max transformation to ensure uniformity across inputs. We then slid a window of length t (for each chosen time-step) over the normalized data, constructing sequences of t consecutive feature vectors and pairing each sequence with the corresponding next-day Close value. Finally, the resulting dataset was split chronologically into an 80 percent training set and a 20 percent hold-out test set.

```
{python}
time_step = [5, 10, 20, 40, 80]
# Load data ONCE outside the loop
df = pd.read_csv('yahoo_stock.csv')
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)
features = df[['High', 'Low', 'Open', 'Volume', 'Close']]
target = df['Close'].values.reshape(-1, 1)

# Normalization
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(features)

# Create sequences
X, y = [], []
for i in range(len(scaled_data) - time_steps):
  X.append(scaled_data[i:i+time_steps])
  y.append(scaled_data[i+time_steps, -1])

X, y = np.array(X), np.array(y)

# Train-test split
split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]
```

In the second preprocessing scheme, we fixed the look-back window at 5 days and instead varied the richness of input information by incrementally adding new technical indicators. Starting again with the basic five features, we computed a suite of momentum and volatility measures: a 20-day simple moving

average, 12- and 26-day exponential moving averages (for MACD), a 14-day Relative Strength Index, upper and lower Bollinger Bands (±2 standard deviations around the 20-day SMA), and a five-day momentum series. We dropped any rows containing NaNs arising from indicator calculations to maintain data integrity. From this enhanced feature set, we selected subsets of increasing cardinality (8, 9, 11, and 14 features) to investigate how additional derived variables influence model accuracy. Each subset was normalized via Min-Max scaling, then converted into five-day input sequences paired with the next-day Close price. As before, we partitioned the sequences into 80 percent training and 20 percent test splits.

```
{python}
# Calculate technical indicators
df['SMA_20'] = df['Close'].rolling(20).mean()
df['EMA_12'] = df['Close'].ewm(span=12, adjust=False).mean()
df['EMA_26'] = df['Close'].ewm(span=26, adjust=False).mean()
# RSI Calculation
delta = df['Close'].diff()
gain = delta.where(delta > 0, 0)
loss = -delta.where(delta < 0, 0)
avg_gain = gain.rolling(14).mean()
avg_loss = loss.rolling(14).mean()
rs = avg_gain / avg_loss
df['RSI'] = 100 - (100 / (1 + rs))
df['MACD'] = df['EMA_12'] - df['EMA_26']
df['Signal_Line'] = df['MACD'].ewm(span=9, adjust=False).mean()
df['Upper_BB'] = df['SMA_20'] + 2*df['Close'].rolling(20).std()
df['Lower_BB'] = df['SMA_20'] - 2*df['Close'].rolling(20).std()
# Momentum
df['Momentum_5'] = df['Close'] - df['Close'].shift(5)
df = df.dropna()
totalfeatures = df[['High', 'Low', 'Open', 'Volume', 'Close', 'SMA_20','EMA_12',
'EMA_26', 'RSI', 'MACD', 'Signal_Line','Upper_BB', 'Lower_BB','Momentum_5']]
time_steps = 5
selected_columns = totalfeatures.columns.tolist()[:num_features] #num_features will
be features we want to use
features = totalfeatures[selected_columns]
close_idx = selected_columns.index('Close')
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(features)
X, y = [], []
for j in range(len(scaled_data) - time_steps):
  X.append(scaled_data[j:j + time_steps])
  y.append(scaled_data[j + time_steps, close_idx])  # Use correct Close index
X, y = np.array(X), np.array(y)
split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]
```

This two-pronged preprocessing strategy allows us both to gauge the effect of memory length and to evaluate the incremental value of technical indicators in forecasting performance.

Figure 2: Yahoo Stock data

Model Training and Forecasting

For both preprocessing schemes, we employed an identical LSTM architecture to ensure a fair comparison. The network begins with a 50-unit LSTM layer that returns the full sequence of hidden states, allowing the model to capture temporal patterns across the entire look-back window. A dropout layer with a 20 percent rate follows to reduce overfitting by randomly omitting a fraction of the LSTM's outputs during training. A second 50-unit LSTM layer then distills the sequence into a single vector representation, which again passes through a 20 percent dropout. Finally, a dense layer with a single neuron produces the one-step-ahead forecast for the closing price. We optimized the model using the Adam algorithm and minimized mean squared error (MSE) as our loss function.

Training proceeded for up to 200 epochs with a batch size of 32, but we incorporated early stopping based on validation loss: if the validation MSE did not improve for ten consecutive epochs, training halted and the model reverted to the weights that achieved the lowest validation error. We reserved 20 percent of the training data for validation at each epoch, enabling us to monitor generalization performance and guard against overfitting. Throughout training, we recorded the loss curves for both training and validation sets to visualize convergence behavior and to confirm that neither underfitting nor overfitting dominated.

Once training completed, we generated out-of-sample forecasts by feeding the test sequences into the trained LSTM and obtaining predicted scaled closing values. To interpret these predictions on the original price scale, we applied the inverse of the Min-Max transformation using the same scaler fitted during preprocessing to both the predicted values and the actual test targets. We

then plotted the reconstructed forecasted and true closing prices over the test period. Finally, we overlaid the training and validation loss curves to provide a clear picture of the model's learning trajectory and to support our discussion of forecasting accuracy in the subsequent results section.

```{python}
model = Sequential()
model.add(LSTM(50, return_sequences=True, input_shape=(time_steps, num_features)))
model.add(Dropout(0.2))
model.add(LSTM(50))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')

early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights
=True)
history = model.fit(
    X_train, y_train,
    epochs=200,
    batch_size=32,
    validation_split=0.2,
    callbacks=[early_stop],
    verbose=0
)
y_pred_actual = inverse_scale(scaler, X_test, model.predict(X_test))
y_test_actual = inverse_scale(scaler, X_test, y_test.reshape(-1, 1))
```

Visual Analysis of Actual vs Predicted (Forecasted) Closing Prices

The forecasting performance of the LSTM model was visually assessed across varying look-back windows (5, 10, 20, and 40 days) by plotting actual and predicted closing prices over the test period.

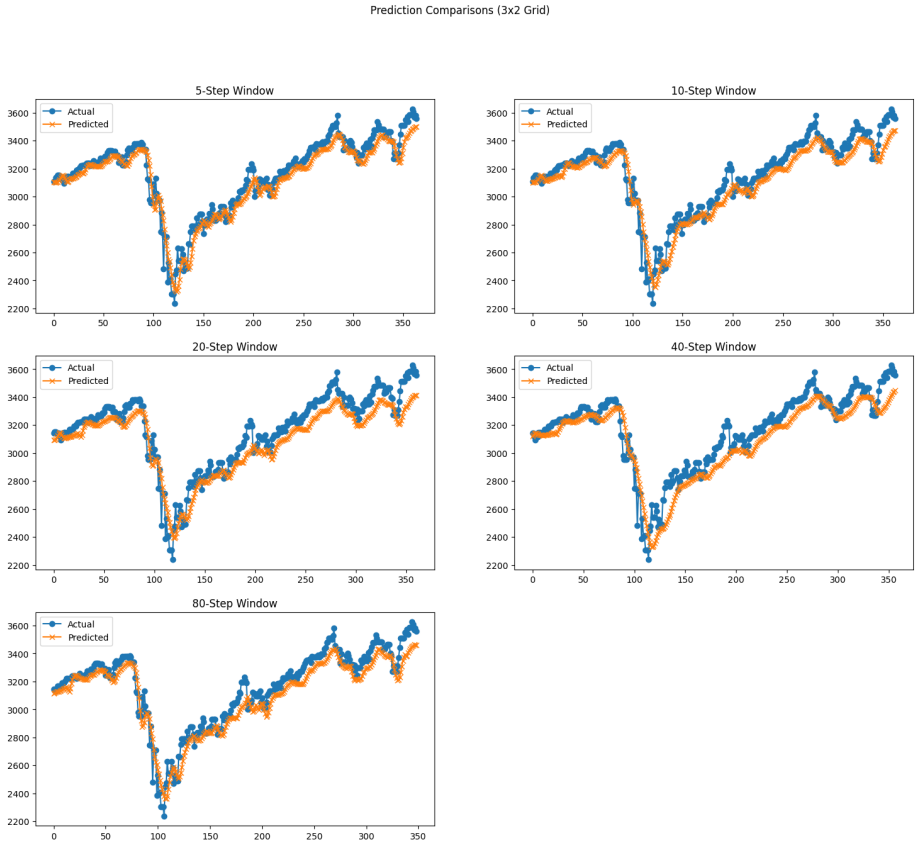


Figure 3: Plot: Actual Close vs Predicted Close by LSTM for different time step window

We can see if we translate the curve of predicted closing prices, it could actually approximate the curve of actual closing price. It means there is a lag, but when we zoom-in these plot, we can see that our model is not really predicting nicely for immediate gains / loss:

Figure 4: Plot: Actual Close vs Predicted Close by LSTM for different time step window (zoomed in)

Evaluation Metrics:

Time-step Window	RMSE	MAE
5	82.07	64.22
10	95.00	75.95
20	107.57	91.17
40	116.92	93.59
80	90.82	75.09

However, our model is performing well if we just look at the performance on the basis of training and validation loss:

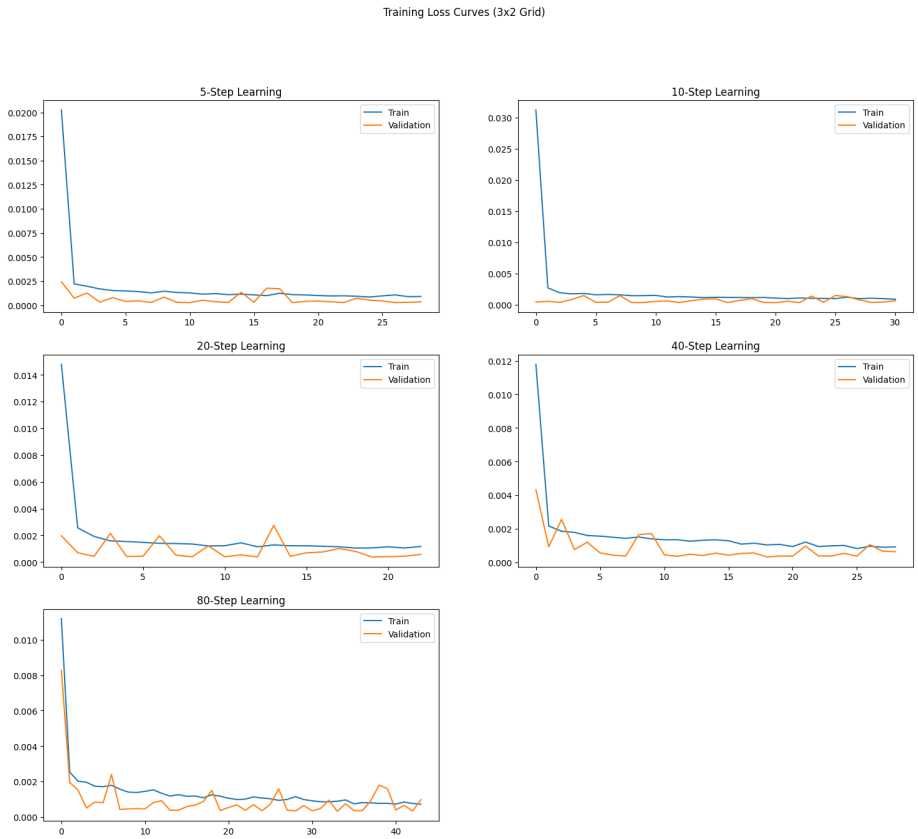


Figure 5: Plot: Training Loss and Validation Loss while Training for different time step window

While the training and validation loss curves (e.g., for 5-, 10-, 20-, and 40-step windows) suggest successful convergence, the visual misalignment between predicted and actual closing prices reveals a critical limitation. The loss functions (MSE, MAE) measure average error magnitudes but are insensitive to systematic timing errors, such as lag. For instance, the LSTM may learn to forecast smoothed approximations of the true series, prioritizing overall trend adherence over precise temporal alignment. This is particularly evident in volatile regimes, where the model's predictions trail abrupt market reversals or spikes, as seen in the attached plots.

The lag arises because financial time series often exhibit stochastic trends and noise that are challenging to disentangle. While the model minimizes average error effectively, it inherently averages out high-frequency fluctuations, resulting in delayed responses to turning points. Thus, the loss curves, while indicative of technical training success, mask the model's struggle to resolve the precise

timing of market movements, a limitation inherent to both the data's complexity and the loss metrics' design.

We will plot the similar results for different number of features we can take into account in our model:

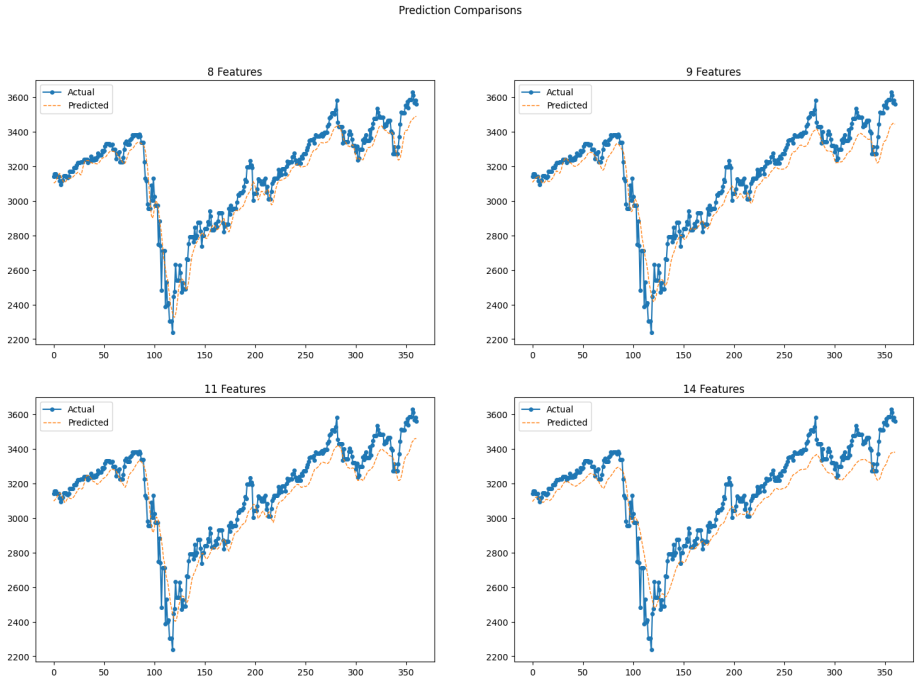


Figure 6: Plot: Actual Close vs Predicted Close by LSTM on different set of features

On zooming it, we have:

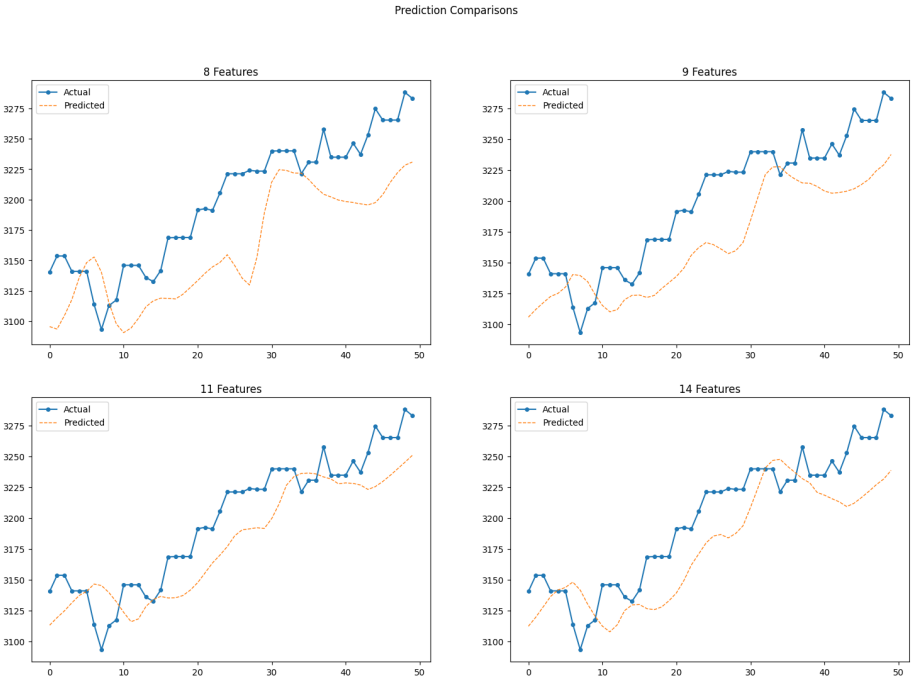


Figure 7: Plot: Actual Close vs Predicted Close by LSTM on different set of features (zoomed-in)

And the model performance plot we got:

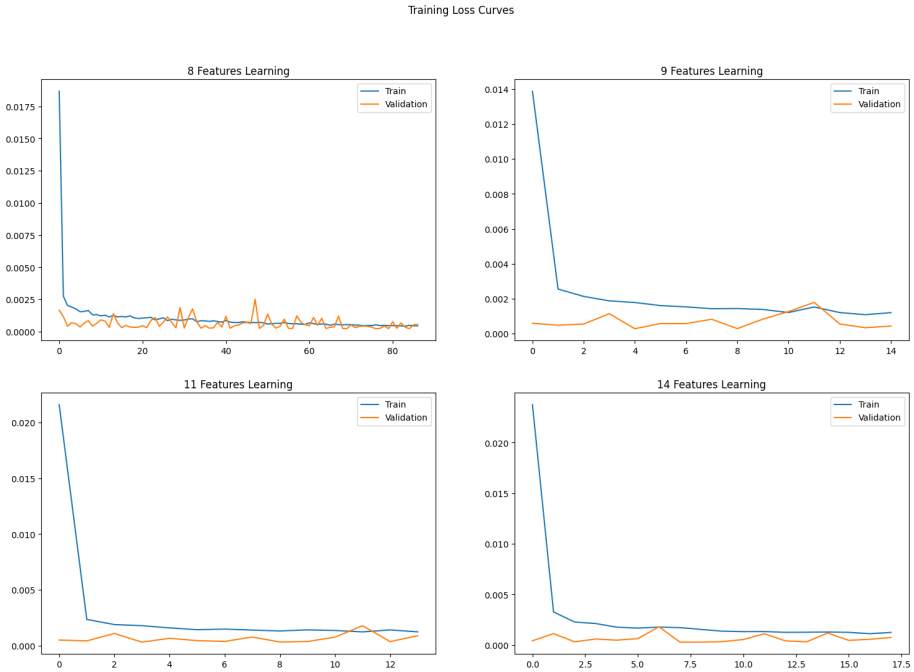
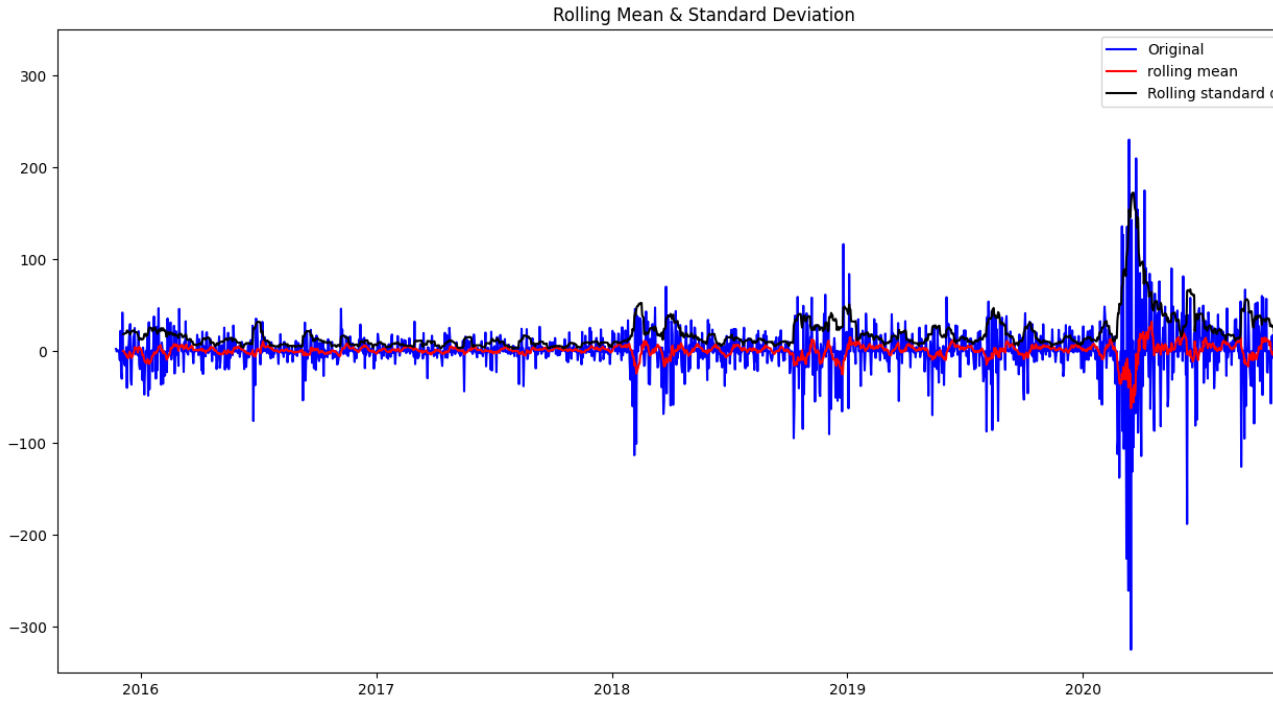


Figure 8: Plot: Training Loss and Validation Loss on different set of features

ARMA Method

To forecast Yahoo stock prices using the ARMA approach, we first ensured stationarity by differencing the log-transformed closing prices. The Dickey-Fuller test (test statistic: -8.52, p-value: 1.09×10^{-13}) and stable rolling statistics confirm stationarity in the differenced series, a prerequisite for ARMA modeling. The absence of trends in the rolling mean and bounded volatility (rolling std) validate consistent statistical properties.

```python
{python}
def test_stationarity(timeseries):
    rolmean = pd.Series.rolling(timeseries,window=12).mean()
    rolstd = pd.Series.rolling(timeseries, window=12).std()
    fig = plt.figure(figsize=(16,8))
    fig.add_subplot()
    orig = plt.plot(timeseries, color = 'blue',label='Original')
    mean = plt.plot(rolmean , color = 'red',label = 'rolling mean')
    std = plt.plot(rolstd, color = 'black', label= 'Rolling standard deviation')
    plt.ylim([-350,350])
    plt.legend(loc = 'best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)
    print( 'Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries,autolag = 'AIC')
    dfoutput = pd.Series(dftest[0:4],index = ['Test Statistic','p-value','#Lags
Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical value (%s)' %key] = value
    print(dfoutput)
ts_log = df['Close']
ts_log_diff = ts_log - ts_log.shift()
ts_log_diff.dropna(inplace=True)
test_stationarity(ts_log_diff)
```



Results of Dickey-Fuller Test: Test Statistic -8.522188e+00 p-value 1.093086e-13 #Lags Used 2.200000e+01 Number of Observations Used 1.801000e+03 Critical value (1%) -3.433986e+00 Critical value (5%) -2.863146e+00 Critical value (10%) -2.567625e+00 dtype: float64

```python
{python}
lag_acf = acf(ts_log_diff, nlags=20)
lag_pacf = pacf(ts_log_diff, nlags=20, method='ols')
plt.subplot(121)
plt.plot(lag_acf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(ts_log_diff)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(ts_log_diff)),linestyle='--',color='gray')
plt.title('Autocorrelation Function')
plt.subplot(122)
plt.plot(lag_pacf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(ts_log_diff)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(ts_log_diff)),linestyle='--',color='gray')
plt.title('Partial Autocorrelation Function')
plt.show()
```

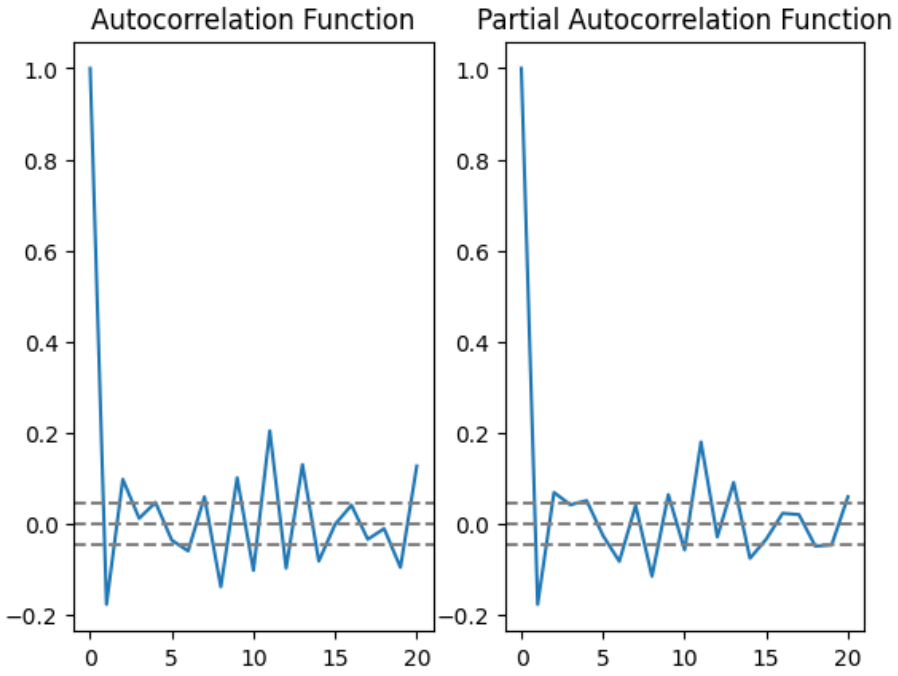


Figure 9: Plot: ACF and PACF

This stationary behavior, coupled with decaying ACF/PACF patterns (not sharp cutoffs), aligns with an ARMA(1,1) structure, which captures short-term autocorrelation via both autoregressive (AR) and moving average (MA) terms. The model's simplicity and alignment with the data's linear dependencies make it a pragmatic choice for forecasting the transformed series. We then implemented an ARIMA(1,0,1) model (equivalent to ARMA(1,1) on the differenced series) and performed walk-forward validation on the test set. At each step, the model was re-fitted to the expanding training window, generating one-step-ahead forecasts.

```python
{python}
train_arma = train_data['Close']
test_arma = test_data['Close']
history = [x for x in train_arma]
y = test_arma
predictions = list()
model_fit = ARIMA(history, order=(1,0,1)).fit()
yhat = model_fit.forecast()[0]
predictions.append(yhat)
history.append(y[0])
for i in range(1, len(y)):
    model = ARIMA(history, order=(1,0,1))
    model_fit = model.fit()
    yhat = model_fit.forecast()[0]
    predictions.append(yhat)
    obs = y[i]
    history.append(obs)
plt.figure(figsize=(14,8))
plt.plot(df.index, df['Open'], color='green', label = 'Train Stock Price')
plt.plot(test_data.index, y, color = 'red', label = 'Real Stock Price')
plt.plot(test_data.index, predictions, color = 'blue', label = 'Predicted Stock
Price')
plt.legend()
plt.grid(True)
plt.show()
plt.figure(figsize=(14,8))
plt.plot(df.index[-100:], df['Open'].tail(100), color='green', label = 'Train Stock
Price')
plt.plot(test_data.index, y, color = 'red', label = 'Real Stock Price')
plt.plot(test_data.index, predictions, color = 'blue', label = 'Predicted Stock
Price')
plt.legend()
plt.grid(True)
plt.show()
print('MSE: '+str(mean_squared_error(y, predictions)))
print('MAE: '+str(mean_absolute_error(y, predictions)))
print('RMSE: '+str(sqrt(mean_squared_error(y, predictions))))
```

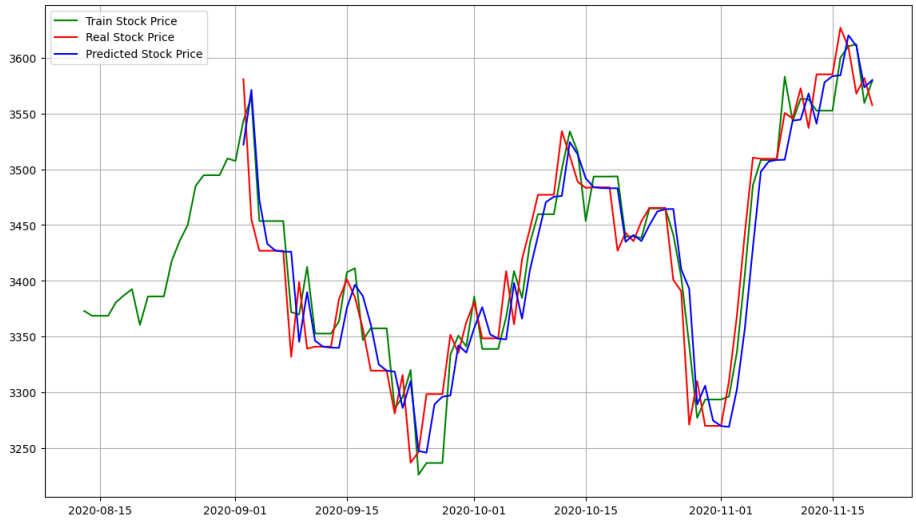


Figure 10: Plot: Actual Close vs Predicted Close vs Trained after Prediction using ARMA (zoomed in)

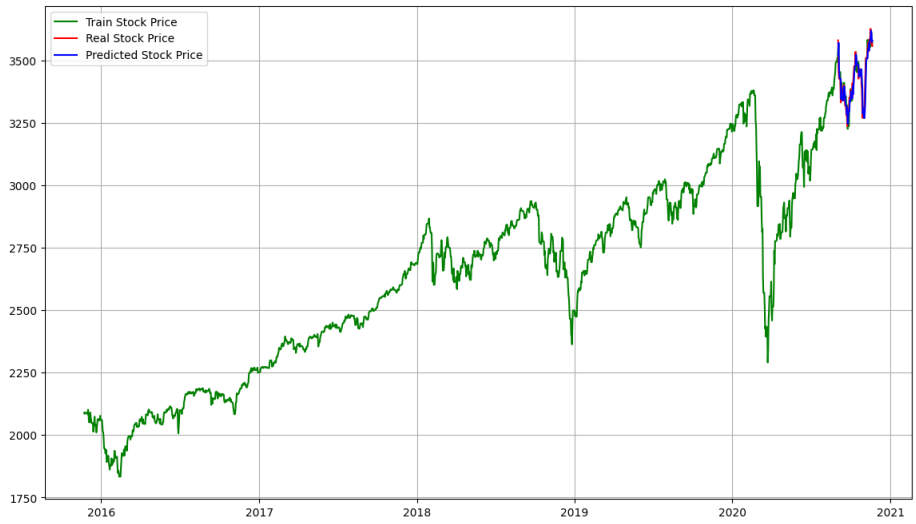


Figure 11: Plot: Actual Close vs Predicted Close vs Trained using ARMA

MSE: 1548.6888350240904 MAE: 27.768974448514733 RMSE: 39.35338403522739

The predicted prices (blue line) in the resulting plots closely followed the actual test series (red line) during stable market periods, demonstrating the model's ability to capture linear dependencies. However, during volatile phases, the ARMA forecasts exhibited noticeable lag, particularly in reacting to sudden price spikes or drops. This aligns with the model's inherent limitation of relying on linear historical relationships, which struggle to adapt to abrupt nonlinear shifts. The iterative retraining approach ensured adaptability to recent trends but introduced computational overhead compared to the LSTM's batch forecasting.

Overall, the ARMA model provided interpretable and stable predictions but lagged in capturing rapid market dynamics, highlighting the trade-off between classical and machine learning methods.

Volatility Prediction

While closing price forecasts provide directional insights, financial markets are inherently noisy and dominated by stochastic shocks, rendering precise point predictions unreliable for risk-sensitive decisions. **Volatility**, the variability of price movements, serves as a critical measure of market uncertainty and risk. Unlike deterministic price trends, volatility exhibits clustering (e.g., periods of high instability followed by calm) and persistence, making it more systematically predictable. By modeling volatility, we shift focus from predicting exact

price levels to quantifying the *magnitude* of expected fluctuations, which is indispensable for portfolio optimization, derivative pricing, and Value-at-Risk (VaR) calculations.

This transition aligns with the limitations observed in our earlier LSTM and ARMA forecasts: both models struggled to anticipate abrupt price swings, as their loss functions prioritized average error reduction over volatility dynamics. Integrating a **GARCH** framework with ARMA explicitly addresses this gap by modeling time-varying variance, where volatility depends on past squared residuals and lagged variances. This approach not only complements price forecasts but also quantifies the confidence intervals around predictions, offering a more holistic view of market behavior. Thus, volatility prediction becomes a pragmatic pivot, bridging the gap between directional accuracy and actionable risk assessment.

Volatility Prediction through ARMA-GARCH:

To model time-varying volatility in Yahoo stock returns, we first computed daily percentage returns from adjusted closing prices, which exhibited characteristic volatility clustering. An ARMA(1,1) model was fitted to the returns to capture linear dependencies in the mean equation, with residuals extracted to isolate the stochastic noise component. The PACF plot of squared residuals confirmed persistent autocorrelation in volatility, justifying the need for a GARCH structure. We then integrated a GARCH(1,2) model to the ARMA residuals, where volatility was modeled as a function of one lagged conditional variance (GARCH term) and two lagged squared residuals (ARCH terms).

```{python}
df = pd.read_csv('yahoo_stock.csv',
                 parse_dates=['Date'],    # parse “Date” into Timestamps
                 index_col='Date')
returns = 100 * df['Close'].pct_change().dropna()
plt.figure(figsize=(10,4))
plt.plot(returns.index, returns)    # x = dates, y = returns
plt.ylabel('Pct Return', fontsize=16)
plt.title('Yahoo Stock Returns', fontsize=20)
plt.tight_layout()
plt.show()
```

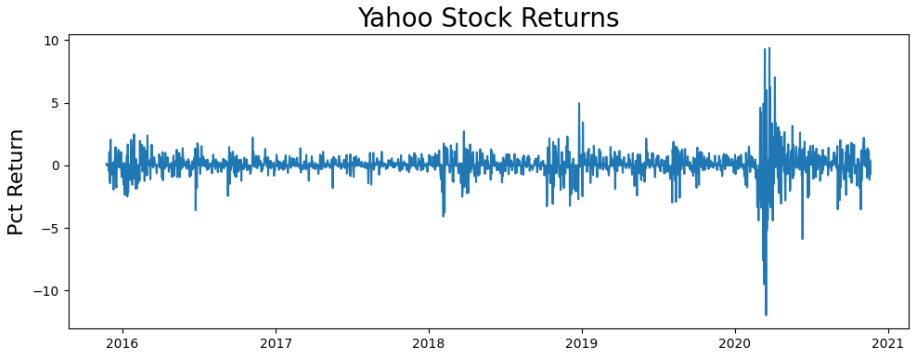


Figure 12: Plot: Percentage Return

```{python}
plot_pacf(returns**2)
plt.show()
```

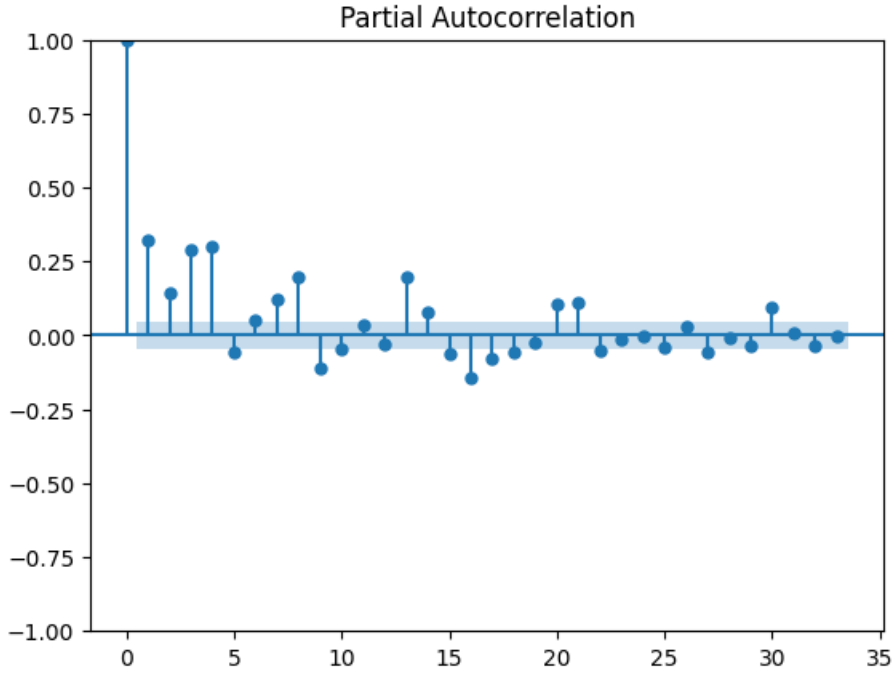


Figure 13: Plot: PACF for Pcnt Change

From the above figure, we can assume that ARMA(4,4) may work and so we compute p-value of coefficients, we realised that choosing p=1 and q=1 will be

sufficient. Similarly, we found using p=1, q=2 for GARCH would have meaning as one coefficient was insignificant when q=3.

```{python}
arma_model = ARIMA(returns, order=(1, 0, 1))
arma_fit = arma_model.fit()
print(arma_fit.summary())
```

self._init_dates(dates, freq) SARIMAX Results
==========
Dep. Variable: Close No. Observations: 1824 Model: ARIMA(1, 0, 1) Log Likelihood -2574.094 Date: Tue, 20 May 2025 AIC 5156.188 Time: 06:09:12 BIC 5178.223 Sample: 11-24-2015 HQIC 5164.316 - 11-20-2020
Covariance Type: opg
==========
coef std err z P>|z| [0.025 0.975] ———— const 0.0344 0.022 1.562 0.118 -0.009 0.078 ar.L1 -0.4325 0.031 -13.857 0.000 -0.494 -0.371 ma.L1 0.2537 0.033 7.594 0.000 0.188 0.319 sigma2 0.9847 0.008 118.005 0.000 0.968 1.001
==========
Ljung-Box (L1) (Q): 0.00 Jarque-Bera (JB): 72120.30 Prob(Q): 0.95 Prob(JB): 0.00 Heteroskedasticity (H): 5.33 Skew: -1.34 Prob(H) (two-sided): 0.00 Kurtosis: 33.69 ==========

```{python}
resid = arma_fit.resid
model = arch_model(resid, p=1, q=2)
model_fit = model.fit()
model_fit.summary()
```

Constant Mean - GARCH Model Results

Dep. Variable:	None	R-squared:	0.000
Mean Model:	Constant Mean	Adj. R-squared:	0.000
Vol Model:	GARCH	Log-Likelihood:	-1881.80
Distribution:	Normal	AIC:	3773.61
Method:	Maximum Likelihood	BIC:	3801.15
		No. Observations:	1824
Date:	Tue, May 20 2025	Df Residuals:	1823
Time:	06:13:40	Df Model:	1

Mean Model

	coef	std err	t	P>|t|	95.0% Conf. Int.
mu	0.0401	1.445e-02	2.775	5.521e-03	[1.177e-02,6.840e-02]

Volatility Model

	coef	std err	t	P>|t|	95.0% Conf. Int.
omega	0.0234	7.236e-03	3.228	1.248e-03	[9.174e-03,3.754e-02]
alpha[1]	0.2265	4.628e-02	4.895	9.849e-07	[0.136, 0.317]
beta[1]	0.2025	7.475e-02	2.709	6.755e-03	[5.597e-02, 0.349]
beta[2]	0.5553	6.531e-02	8.503	1.850e-17	[0.427, 0.683]

For robustness, we performed a rolling 1-step volatility forecast over the final 365 days: at each iteration, the ARMA-GARCH pipeline was re-estimated on an expanding window, and the conditional variance was forecasted. This adaptive approach ensured the model dynamically incorporated recent market shocks.

```python
{python}
arma = ARIMA(returns, order=(1,0,1)).fit()
resid = arma.resid  # get the ARMA residuals
# Fit GARCH(1,2) to the residuals
garch = arch_model(resid, vol='GARCH', p=1, q=2, dist='normal')
garch_fit = garch.fit(disp='off')
# -- Rolling 1-step volatility forecast over last 365 days ---
rolling_vol = []
test_size = 365

for i in range(test_size):
    # rolling window of ARMA
    train_ret = returns[:-(test_size - i)]
    arma_i = ARIMA(train_ret, order=(1,0,1)).fit()
    resid_i = arma_i.resid

    # fit GARCH to that window
    garch_i = arch_model(resid_i, vol='GARCH', p=1, q=2)
    garch_fit_i = garch_i.fit(disp='off')

    fcast = garch_fit_i.forecast(horizon=1)
    var1  = fcast.variance.values[-1, 0]
    rolling_vol.append(np.sqrt(var1))

rolling_vol = pd.Series(rolling_vol, index=returns.index[-test_size:])
plt.figure(figsize=(10,4))
plt.plot(returns[-test_size:], label='True Returns')
plt.plot(rolling_vol, label='Predicted Volatility')
plt.title('Rolling Forecast: ARMA(1,1)-GARCH(1,2)')
plt.legend()
plt.tight_layout()
plt.show()
```

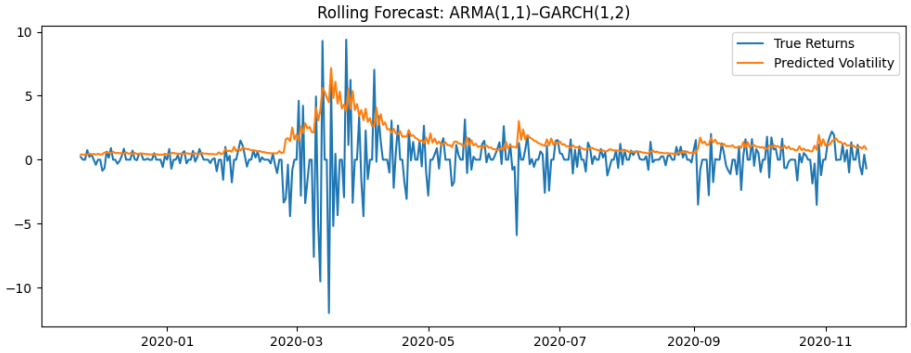


Figure 14: Plot: True Returns vs Volatility using ARMA(1,1)-GARCH(1,2)

Additionally, we generate 7-day-ahead volatility forecast from the full-sample ARMA-GARCH fit, projecting future uncertainty. The results revealed that volatility spikes during market downturns were systematically captured, with the GARCH(1,2) structure effectively reflecting the "memory" of past turbulence.

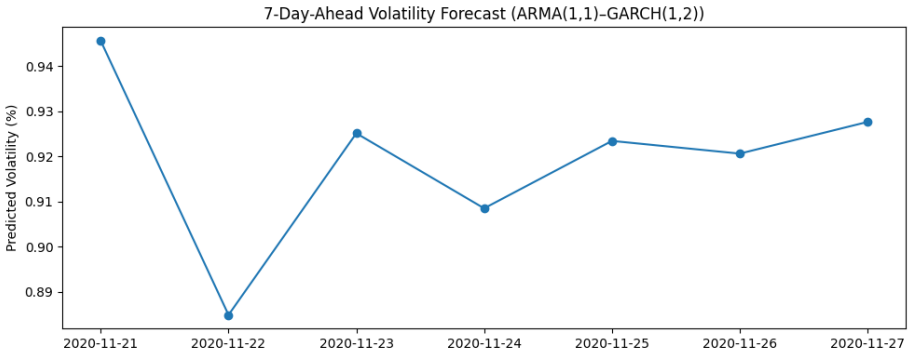
```python
{python}
arma_full = ARIMA(returns, order=(1,0,1)).fit()
resid_full = arma_full.resid

garch_full = arch_model(resid_full, vol='GARCH', p=1, q=2)
garch_full_fit = garch_full.fit(disp='off')

fcast7 = garch_full_fit.forecast(horizon=7)
var7 = fcast7.variance.values[-1, :]
vol7 = np.sqrt(var7)

future_dates = [returns.index[-1] + timedelta(days=i) for i in range(1,8)]
vol7_series = pd.Series(vol7, index=future_dates)

plt.figure(figsize=(10,4))
plt.plot(vol7_series, marker='o')
plt.title('7-Day-Ahead Volatility Forecast (ARMA(1,1)-GARCH(1,2))')
plt.ylabel('Predicted Volatility (%)')
plt.tight_layout()
plt.show()
```



By decoupling mean (ARMA) and variance (GARCH) dynamics, this hybrid approach quantifies both expected returns and their confidence intervals, addressing a critical limitation of standalone price prediction models.

Volatility Prediction through LSTM-GARCH

To forecast volatility via LSTM, we first derived daily volatility from Yahoo stock returns using a 5-day rolling standard deviation (annualized by scaling with $\sqrt{252}$). The LSTM model incorporated five features: High, Low, Open, Volume, and lagged Volatility—normalized to [0,1] to ensure uniform scaling. Sequences of varying historical windows (5, 10, 20, 40, and 80 days) were constructed to evaluate how temporal context influences predictive accuracy. We need to pay attention that this set of windows (5, 10, 20, 40, 80) is the time step used while training the model. Each sequence paired a window of scaled features with the subsequent volatility value, enabling the LSTM to learn temporal dependencies.

The architecture included two LSTM layers (50 units each) with 20% dropout to mitigate overfitting, followed by a dense output layer. Training employed early

stopping (patience=10) on validation loss, with an 80-20 train-test split. Predictions were inverse-transformed to the original volatility scale by reconstructing the feature matrix, ensuring interpretable outputs.

```
{python}
df['Return'] = df['Close'].pct_change()
window = 5
df['Volatility'] = df['Return'].rolling(window).std() * np.sqrt(252)
df.dropna(inplace=True)
features = df[['High','Low','Open','Volume','Volatility']].values
target = df['Volatility'].values.reshape(-1,1)
scaler = MinMaxScaler()
scaled = scaler.fit_transform(features)

X, y = [], []
for i in range(len(scaled) - time_steps):
  X.append(scaled[i:i+time_steps])
  y.append(scaled[i+time_steps, -1])
X, y = np.array(X), np.array(y)

split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(time_steps, X.shape[2])),
    Dropout(0.2),
    LSTM(50),
    Dropout(0.2),
    Dense(1)
])
model.compile(optimizer='adam', loss='mean_squared_error')
early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights
=True)
history = model.fit(X_train, y_train,
                    epochs=200, batch_size=32,
                    validation_split=0.2,
                    callbacks=[early_stop], verbose=1)
y_pred = model.predict(X_test)
y_test_vol = inverse_vol(scaler, X_test, y_test.reshape(-1,1))
y_pred_vol = inverse_vol(scaler, X_test, y_pred)
```

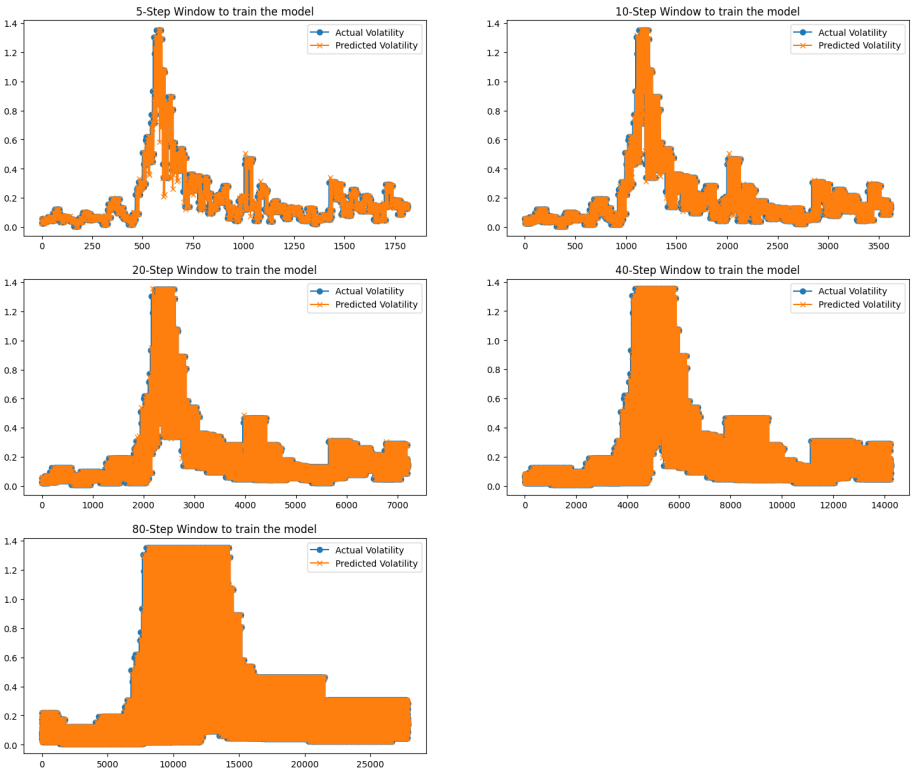


Figure 15: Plot: Rolling forecast using LSTM

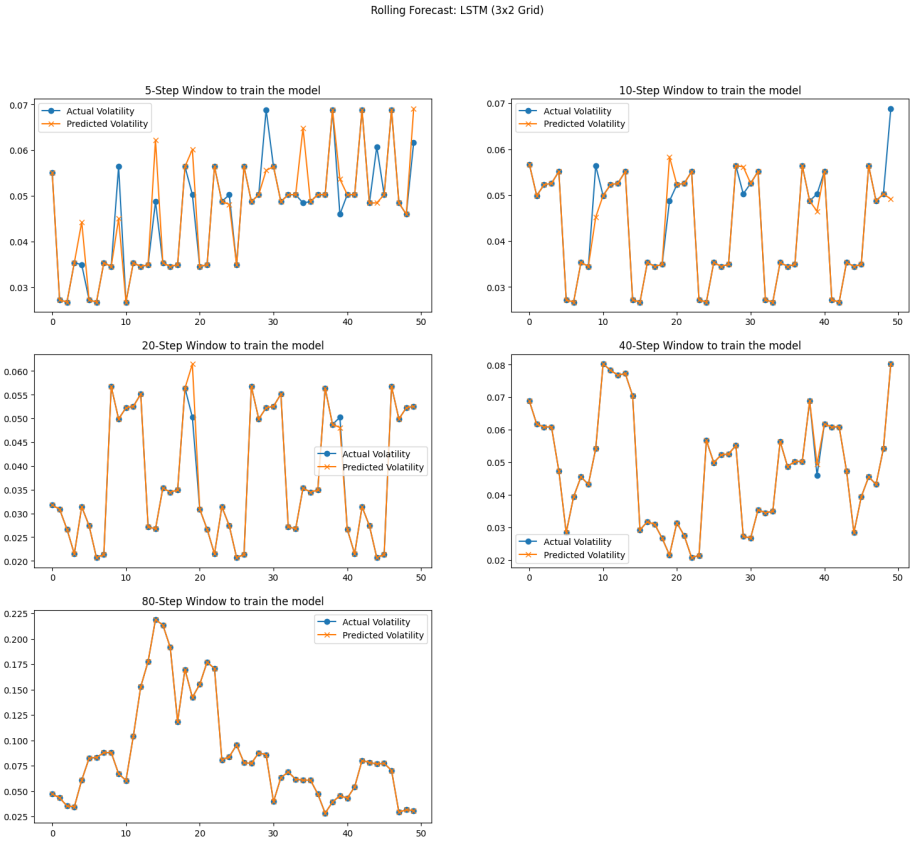


Figure 16: Plot: Rolling forecast using LSTM (zoomed in)

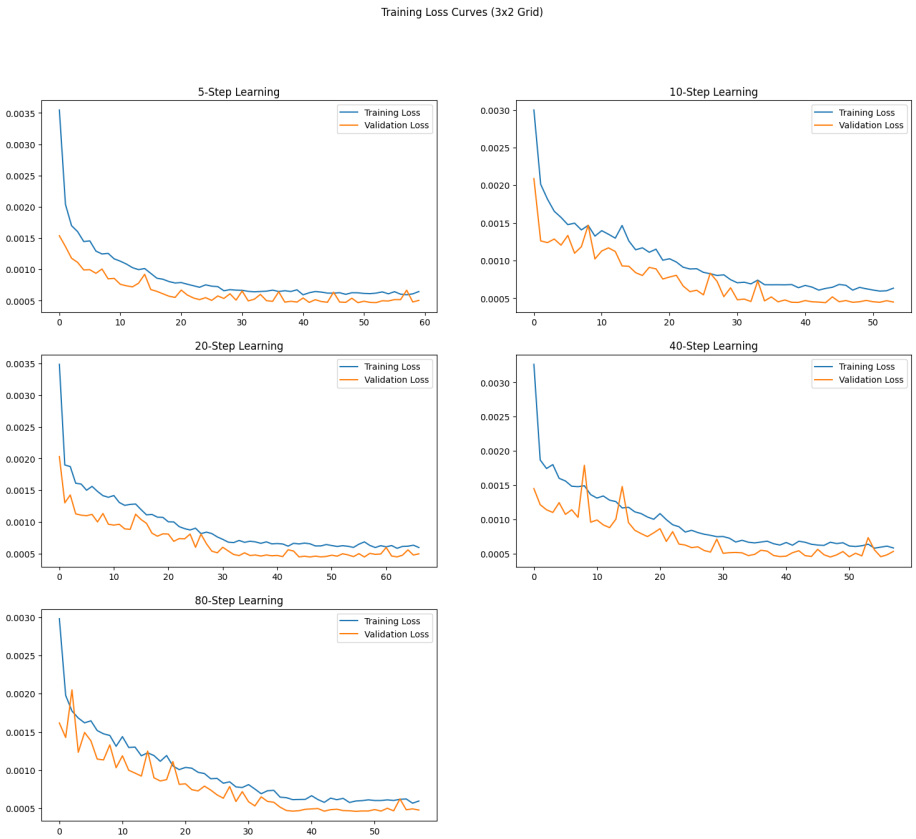


Figure 17: Plot: Training vs Validation Loss in LSTM

Time-Step Window	Volatility RMSE	MAE
5	0.0395	0.0089
10	0.0245	0.0042
20	0.0163	0.0021
40	0.0125	0.0011
80	0.0089	0.0005

Performance was quantified via RMSE and MAE, while prediction plots compared actual vs. forecasted volatility for the test points. We can conclude as we increase the window size while training, the volatility will have spikes at nearly all date. Training and validation loss curves were visualized in a 3×2 grid to assess convergence across window sizes.