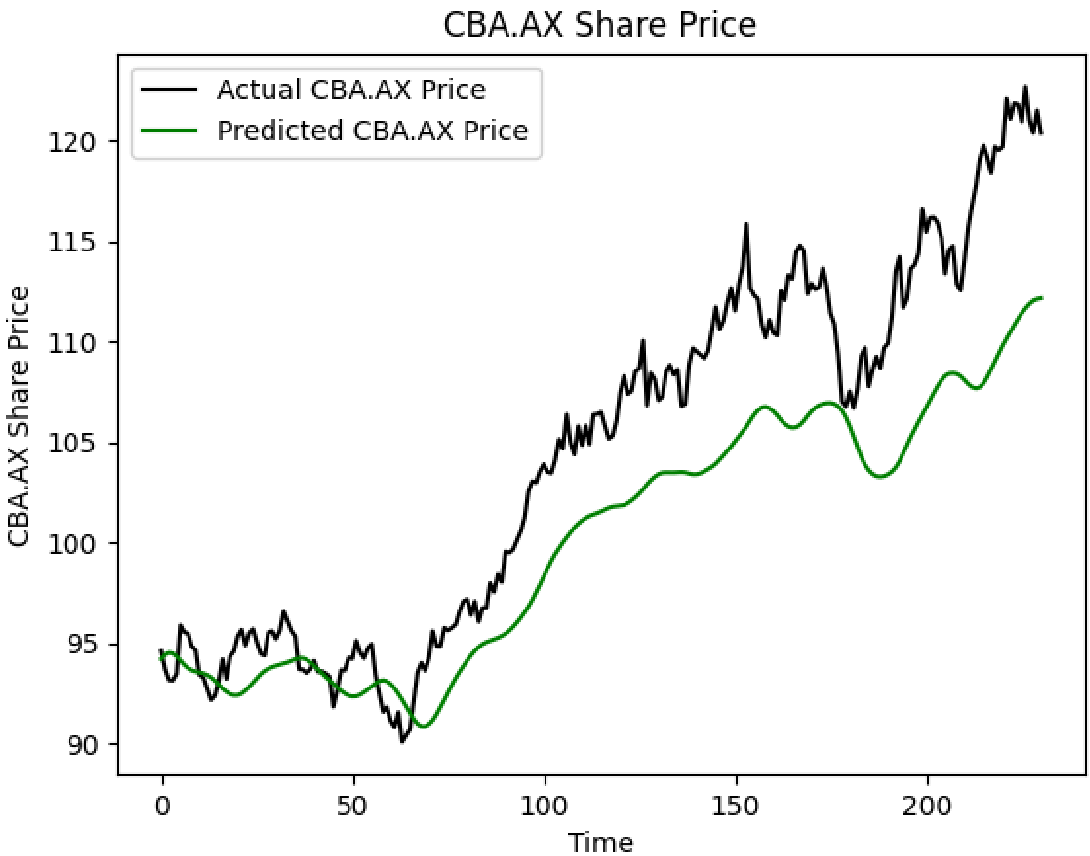
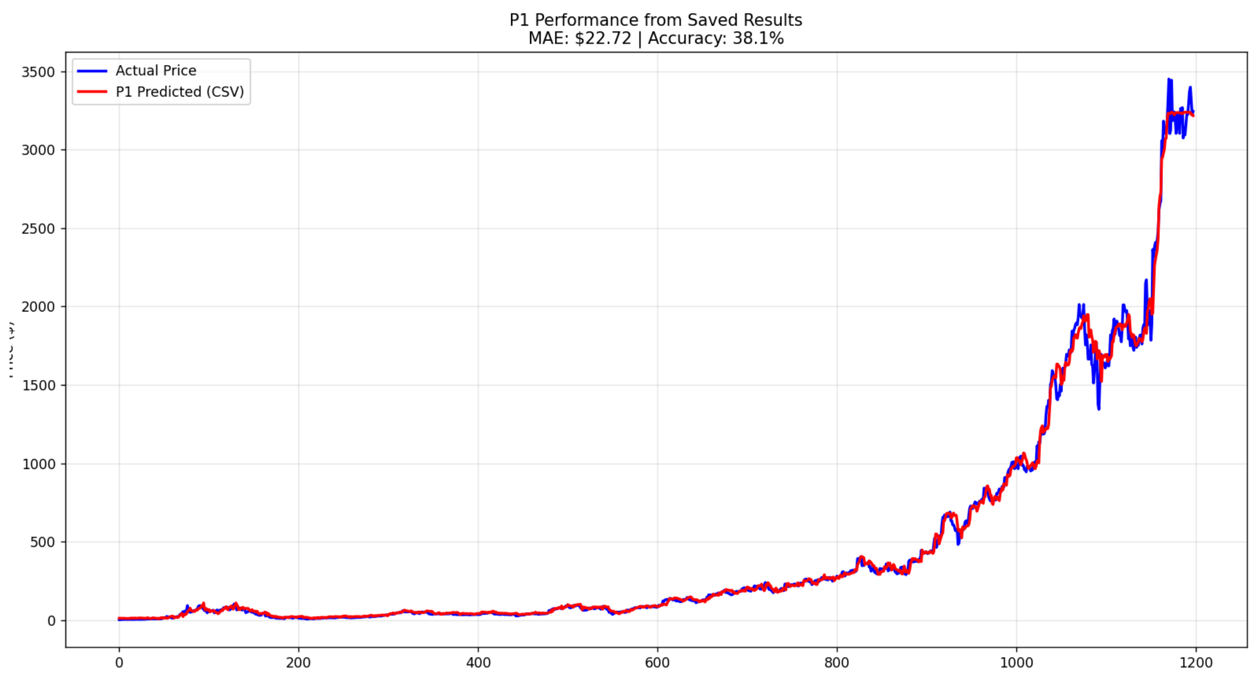
Task 1 Feedback

Script Functions:

- Tests both v0.1 and P1 code bases comprehensively
- Provides detailed comparison between versions
- Generates multiple screenshots showing differences
- Includes RSI, moving averages, volatility analysis for P1
- Creates feature comparison matrix

Screenshots Generated:


CBA.AX Share Price
Actual CBA.AX Price
Predicted CBA.AX Price
CBA.AX Share Price
Time
90
95
100
105
110
115
120
0
50
100
150
200


P1 Performance from Saved Results
MAE: $22.72 | Accuracy: 38.1%
Actual Price
P1 Predicted (CSV)
Days
0
500
1000
1500
2000
2500
3000
3500
0
200
400
600
800
1000
1200

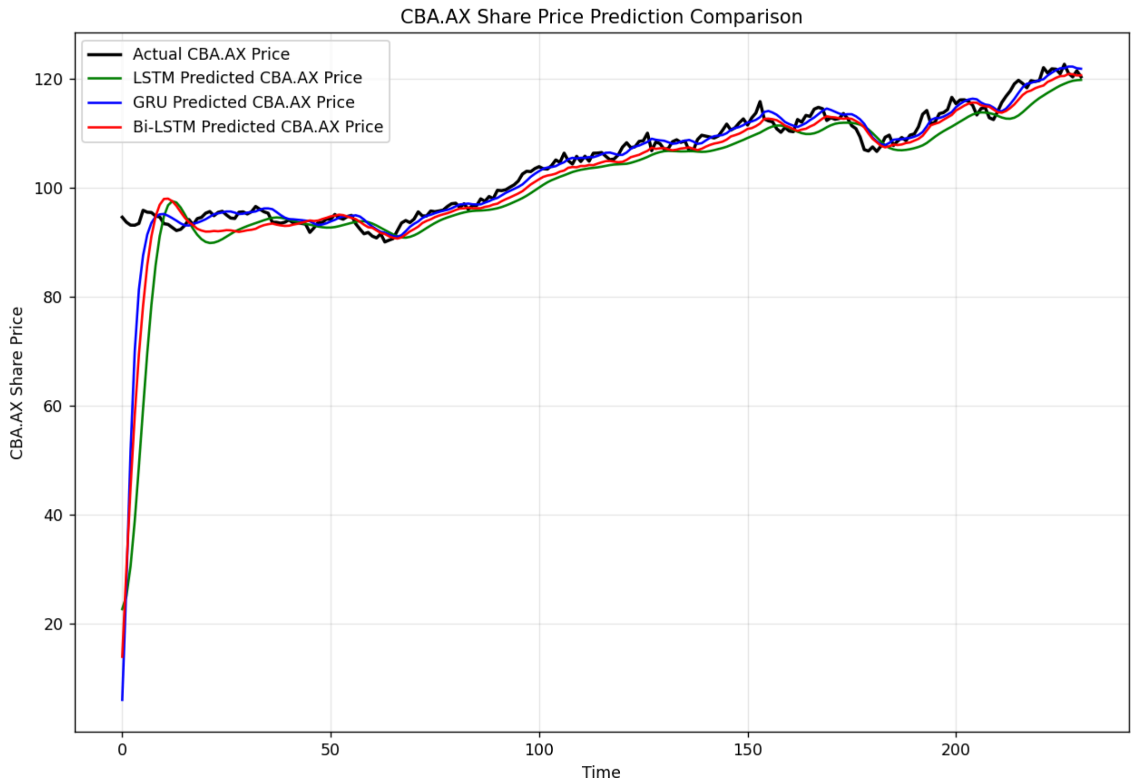
•

- Line-by-line code execution for both versions
- Performance metrics for each version
- Visual proof of enhanced P1 capabilities
- Comprehensive feature comparison table

Task 2 Feedback

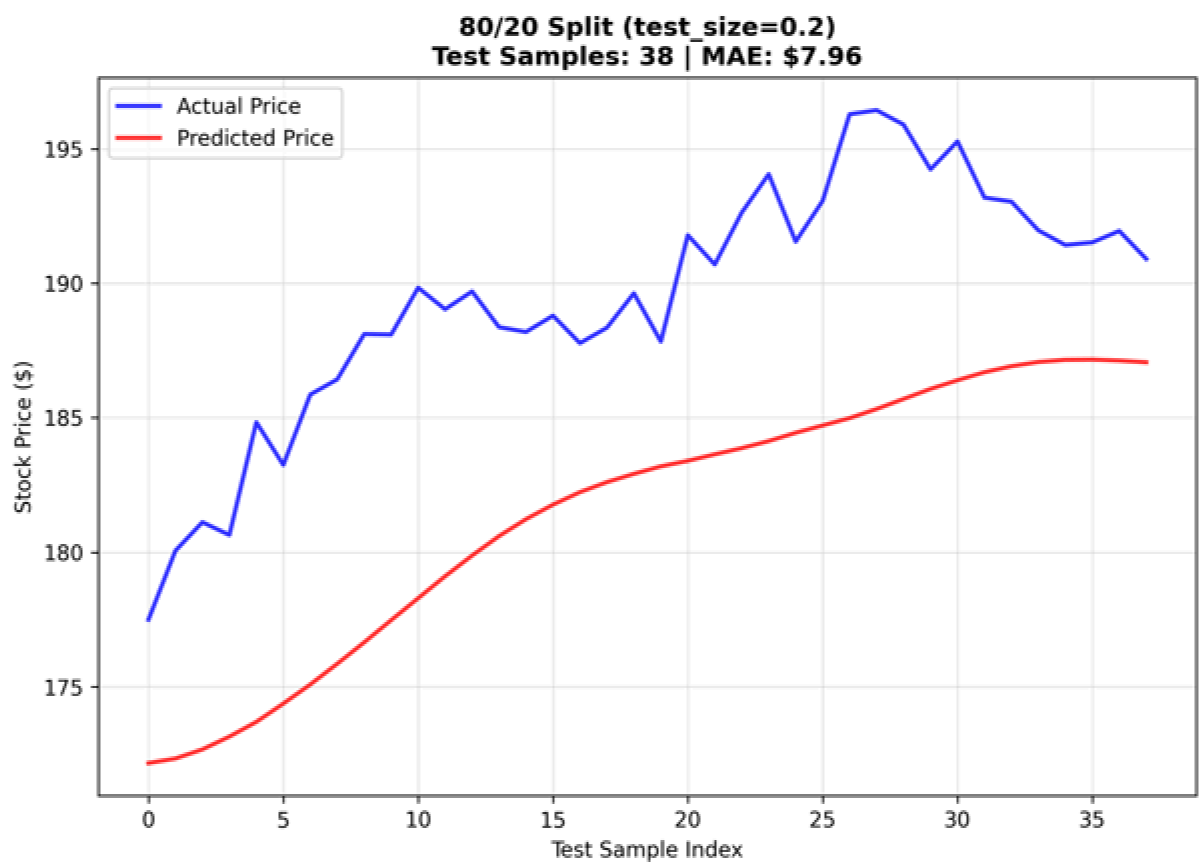
Screenshots Generated:

```
def load_data(company='CBA.AX',            # Stock ticker symbol (default: Commonwealth Bank of Australia)
              start_date='2021-01-01',    # Start date for data collection in YYYY-MM-DD format
              end_date='2025-08-01',      # End date for data collection in YYYY-MM-DD format
              price_column='Close',       # Which price column to use for prediction (Close, Open, High, Low)
              prediction_days=60,         # Number of previous days to use for predicting next day
              split_by_date=True,         # True: split chronologically, False: split randomly
              test_size=0.2,             # Proportion of data for testing (0.2 = 20%)
              scale=True,                # Whether to normalize data to 0-1 range
              save_locally=True,         # Whether to save/load data from local files
              local_path='data',         # Directory to save/load data files
              fill_na_method='ffill',    # Method to handle missing values (ffill/bfill)
              feature_columns=['Close']):# List of columns to scale/normalize
```


CBA.AX Share Price Prediction Comparison
Actual CBA.AX Price
LSTM Predicted CBA.AX Price
GRU Predicted CBA.AX Price
Bi-LSTM Predicted CBA.AX Price
CBA.AX Share Price
Time
120
100
80
60
40
20
0
50
100
150
200

```
FUNCTION LOCATION: Code/stock_prediction.py, Lines 577-646

def create_model(sequence_length, n_features, units=50, cell=LSTM, n_layers=3,
                dropout=0.2, loss="mean_squared_error", optimizer="adam", bidirectional=False):
    """
    Create a deep learning model for stock prediction.

    This function creates different types of RNN models (LSTM, GRU, SimpleRNN) with
    configurable architecture. Based on the example code you provided.

    Parameters:
    -----------
    sequence_length: int        ← Length of input sequences (e.g., 60 days)
    n_features: int            ← Number of features (e.g., 1 for just closing price)
    units: int                 ← Number of neurons in each layer (default: 50)
    cell: keras layer          ← Type of RNN cell (LSTM, GRU, or SimpleRNN)
    n_layers: int              ← Number of RNN layers (default: 3)
    dropout: float             ← Dropout rate for regularization (default: 0.2)
    loss: str                  ← Loss function to use (default: "mean_squared_error")
    optimizer: str             ← Optimizer to use (default: "adam")
    bidirectional: bool        ← Whether to use bidirectional layers (default: False)
    """

    model = Sequential()  ← Create empty model

    # Add layers based on n_layers parameter
    for i in range(n_layers):  ← Loop through each layer
        if i == 0:  ← FIRST LAYER
            # First layer needs input shape specification
            if bidirectional:
                model.add(Bidirectional(cell(units, return_sequences=True),
                                      input_shape=(sequence_length, n_features)))
            else:
                model.add(cell(units, return_sequences=True,
                              input_shape=(sequence_length, n_features)))
        elif i == n_layers - 1:  ← LAST LAYER
            # Last RNN layer doesn't return sequences
            if bidirectional:
                model.add(Bidirectional(cell(units, return_sequences=False)))
            else:
                model.add(cell(units, return_sequences=False))
        else:  ← HIDDEN LAYERS
            # Hidden layers return sequences
            if bidirectional:
                model.add(Bidirectional(cell(units, return_sequences=True)))
            else:
                model.add(cell(units, return_sequences=True))

        # Add dropout after each RNN layer to prevent overfitting
        model.add(Dropout(dropout))

    # Output layer - single neuron for price prediction
    model.add(Dense(1, activation="linear"))

    # Compile the model
    model.compile(loss=loss, metrics=["mean_absolute_error"], optimizer=optimizer)

    return model
```

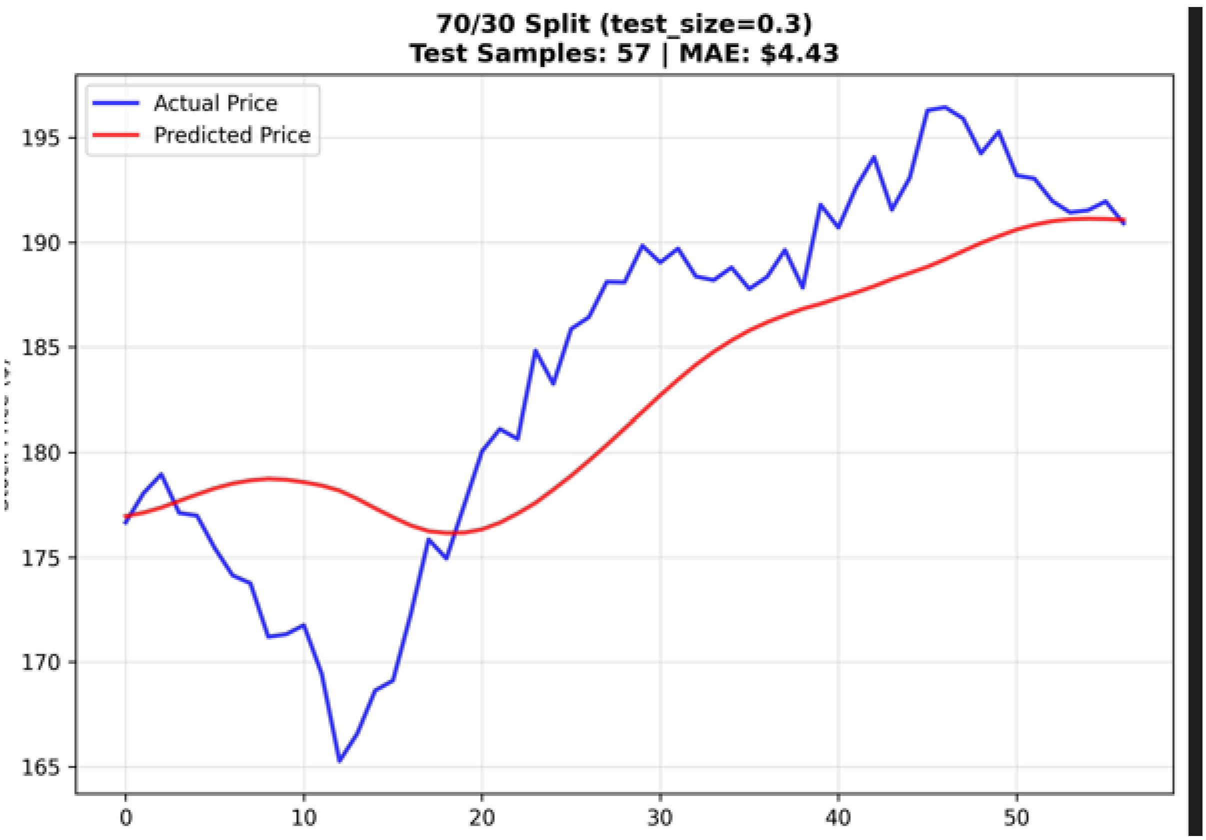
```
HOW TO USE FOR DIFFERENT EXPERIMENTS:

1️⃣ DIFFERENT NETWORK TYPES:
   # LSTM Model
   lstm_model = create_model(60, 1, units=50, cell=LSTM, n_layers=3)

   # GRU Model
   gru_model = create_model(60, 1, units=50, cell=GRU, n_layers=3)

   # SimpleRNN Model
   rnn_model = create_model(60, 1, units=50, cell=SimpleRNN, n_layers=3)

   # Bidirectional LSTM
   bi_model = create_model(60, 1, units=50, cell=LSTM, n_layers=3, bidirectional=True)

2️⃣ DIFFERENT LAYER COUNTS:
   # 2 Layers
   model_2layers = create_model(60, 1, units=50, cell=LSTM, n_layers=2)

   # 4 Layers
   model_4layers = create_model(60, 1, units=50, cell=LSTM, n_layers=4)

   # 5 Layers
   model_5layers = create_model(60, 1, units=50, cell=LSTM, n_layers=5)
```



80/20 Split (with different dates):

Training: Days 1-152 (80%)

Testing: Days 153-190 (20%



70/30 Split (with different dates):

Training: Days 1-133 (70%)

Testing: Days 134-190 (30%)

Task 3 Feedback

3.1 CANDLESTICK CHART IMPLEMENTATION

3.1.1 Overview

I implemented a candlestick chart function to display stock market financial data with support for n-day aggregation.

3.1.2 Function Parameters

Function signature:

Key parameters explained:

- `df` *(pandas.DataFrame)*: Stock data indexed by Date with columns **Open, High, Low, Close** (and optionally **Volume**). Each row represents one trading day.
- `n_days` *(int, default=1)*: ⭐ Core requirement – allows aggregating **n** consecutive trading days into a single candlestick where n ≥ 1.
    - `n_days=1` → each candle shows daily data
    - `n_days=5` → each candle aggregates 5 trading days (weekly candles)
- `style` *(str)*: Visual style theme from mplfinance library ('charles', 'yahoo', 'binance', etc.).
- `volume` *(bool)*: If True, displays volume bars below the price chart.
- `save_path` *(str or None)*: If provided, saves the chart to the specified file path.
- `figsize` *(tuple)*: Chart dimensions as (width, height) in inches.

3.1.3 Core Implementation – N-Day Aggregation

The key innovation is the **n-day aggregation algorithm** located at **lines 366-401**.
This satisfies the requirement to "allow each candlestick to express the data of n trading days (n ≥ 1)".

```
if n_days > 1:

      print(f"Aggregating data into {n_days}-day candles...")
```

```
    # Group data into n-day chunks
    # We'll resample by grouping every n_days rows
    grouped_data = []

    # Iterate through data in chunks of n_days
    for i in range(0, len(data), n_days):
        chunk = data.iloc[i:i+n_days]
        if len(chunk) == 0:
            continue

        # Aggregate the chunk according to OHLC rules
        aggregated_row = {
            'Open': chunk['Open'].iloc[0],         # First day's open
            'High': chunk['High'].max(),           # Highest high
            'Low': chunk['Low'].min(),             # Lowest low
            'Close': chunk['Close'].iloc[-1],      # Last day's close
        }

        # Add volume if present
        if 'Volume' in chunk.columns:
            aggregated_row['Volume'] = chunk['Volume'].sum()  # Sum of
volumes
```

```
            # Use the last date in the chunk as the index

            aggregated_row['Date'] = chunk.index[-1]

            grouped_data.append(aggregated_row)



        # Create new DataFrame from aggregated data

        if grouped_data:

            data = pd.DataFrame(grouped_data)

            data.set_index('Date', inplace=True)

            print(f"Aggregated from {len(df)} daily records to {len(data)}
{n_days}-day candles")

        else:

            raise ValueError("Not enough data to create any n-day
candles")
```

Algorithm explanation:

- **Iterate through data in chunks:** `for i in range(0, len(data), n_days)` steps through data in increments of n_days, creating non-overlapping chunks.
- **Extract chunk:** `chunk = data.iloc[i:i+n_days]` extracts n consecutive trading days.
- **Apply OHLC aggregation rules:**
    - **Open:** `chunk['Open'].iloc[0]` – first day's opening price
    - **High:** `chunk['High'].max()` – max high price across n days
    - **Low:** `chunk['Low'].min()` – min low price across n days
    - **Close:** `chunk['Close'].iloc[-1]` – last day's closing price

    ○ **Volume:** `chunk['Volume'].sum()` – sum of all volumes across n days

- **Create aggregated DataFrame:** new rows combined with last date in each chunk as timestamp.

Example (n_days=3): days 1-3 aggregated into one 3-day candle (Open from day 1, High/Low across all 3 days, Close from day 3).

3.2 BOXPLOT CHART IMPLEMENTATION

3.2.1 Overview

I implemented a boxplot function to display stock price distributions over moving windows of n consecutive trading days.

```
def plot_boxplots_moving_window(
    df,

    price_column="Close",

    window=20,

    stride=1,

    showfliers=False,

    title="Rolling Window Boxplots",

    save_path=None

):



    if price_column not in df.columns:
```

```
        raise ValueError(f"DataFrame must contain '{price_column}'
column.")

    if window < 1:

        raise ValueError("window must be >= 1")

    if stride < 1:

        raise ValueError("stride must be >= 1")



    prices = df[price_column].dropna()

    if len(prices) < window:

        raise ValueError("Not enough data to form one window.")



    # Collect rolling windows and labels

    data_windows = []

    labels = []

    idx = prices.index



    # Build each rolling window by slicing the Series

    for end in range(window, len(prices) + 1, stride):

        start = end - window

        w = prices.iloc[start:end].values

        data_windows.append(w)

        labels.append(idx[end - 1].strftime("%Y-%m-%d"))  # use end date
as label
```

```
    # Plot boxplots
    plt.figure()
    plt.boxplot(
        data_windows,
        showfliers=showfliers,    # whether to draw outliers
        widths=0.6                # width of each box
    )
    plt.title(f"{title} (window={window}, stride={stride})")
    plt.ylabel(price_column)

    # Avoid clutter on the x-axis by showing ~10 evenly spaced labels
    if len(labels) > 10:
        step = max(1, len(labels) // 10)
        xticks = range(1, len(labels) + 1, step)
        xtick_labels = [labels[i - 1] for i in xticks]
        plt.xticks(xticks, xtick_labels, rotation=45, ha="right")
    else:
        plt.xticks(range(1, len(labels) + 1), labels, rotation=45,
ha="right")

    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, dpi=150)
    plt.show()
```

3.2.2 Function Parameters

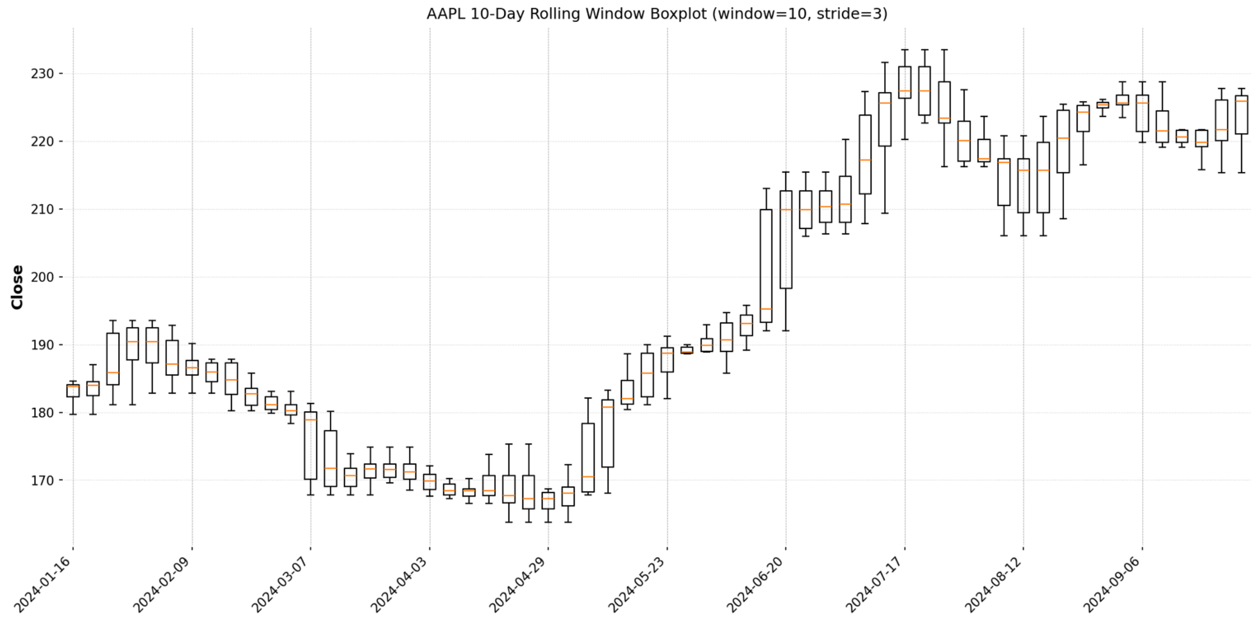
Key parameters explained:

- `df` *(pandas.DataFrame)*: Stock data indexed by Date.
- `price_column` *(str, default='Close')*: Specifies which price column to analyse.
- `window` *(int, default=20)*: ⭐ Core requirement – number of consecutive trading days per box.
- `stride` *(int, default=1)*: Step size between windows.
    - `stride=1` → overlapping windows
    - `stride=5` → less overlap
    - `stride=window` → non-overlapping windows
- `showfliers` *(bool)*: If True, displays outlier points.
- `save_path` *(str or None)*: Optional file path to save the chart.

3.2.5 Results

AAPL 3-Day Candlestick Chart (n_days=3)

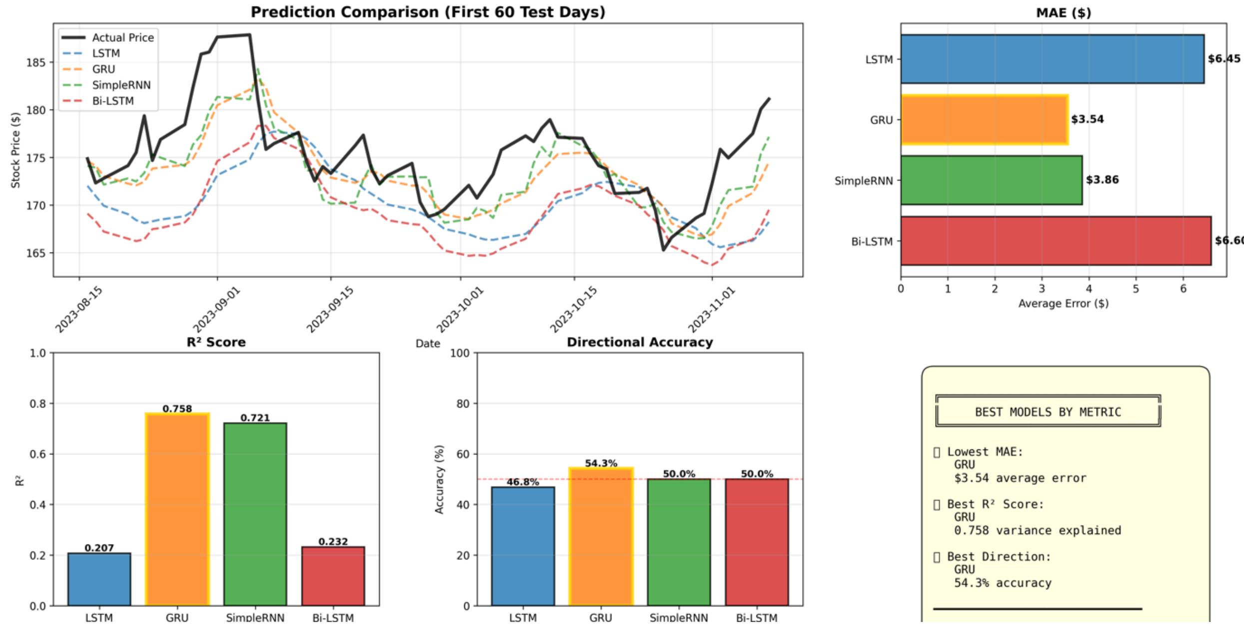
•

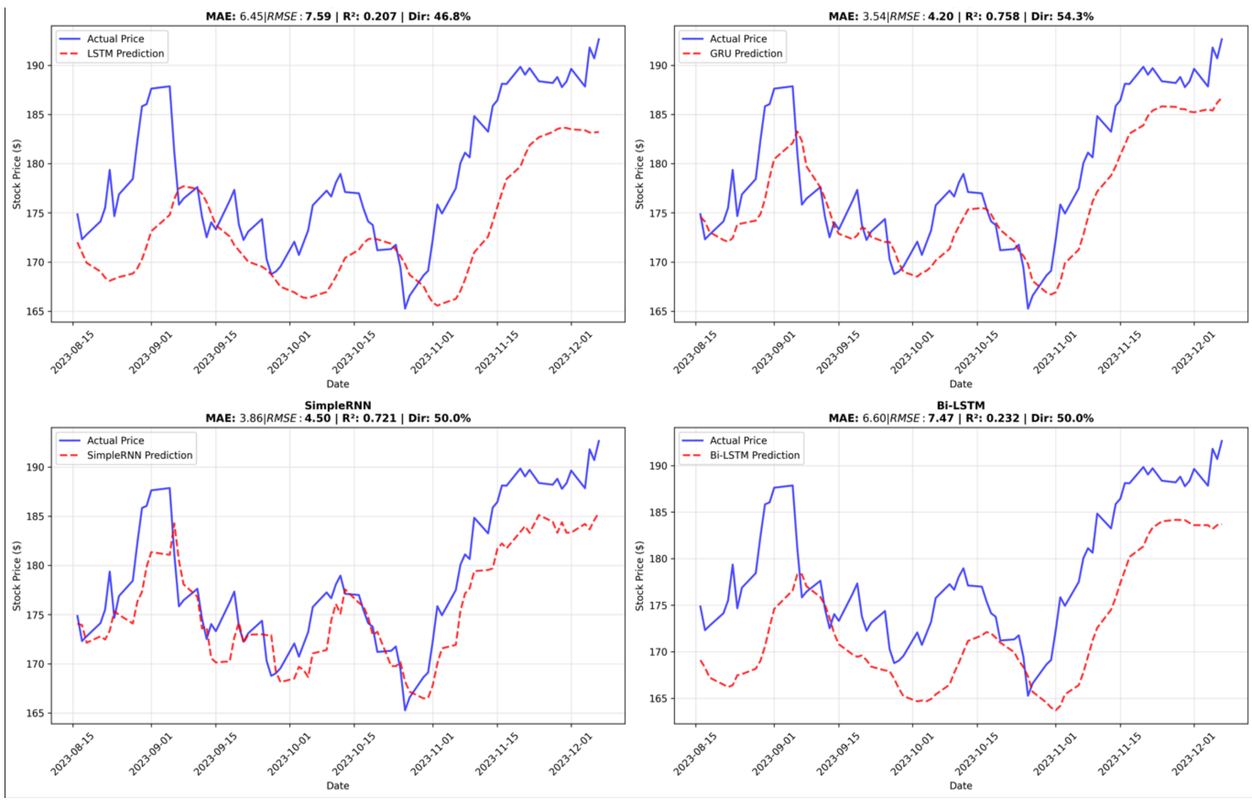


Task 4 Feedback Response

Graph 1 – Network Types

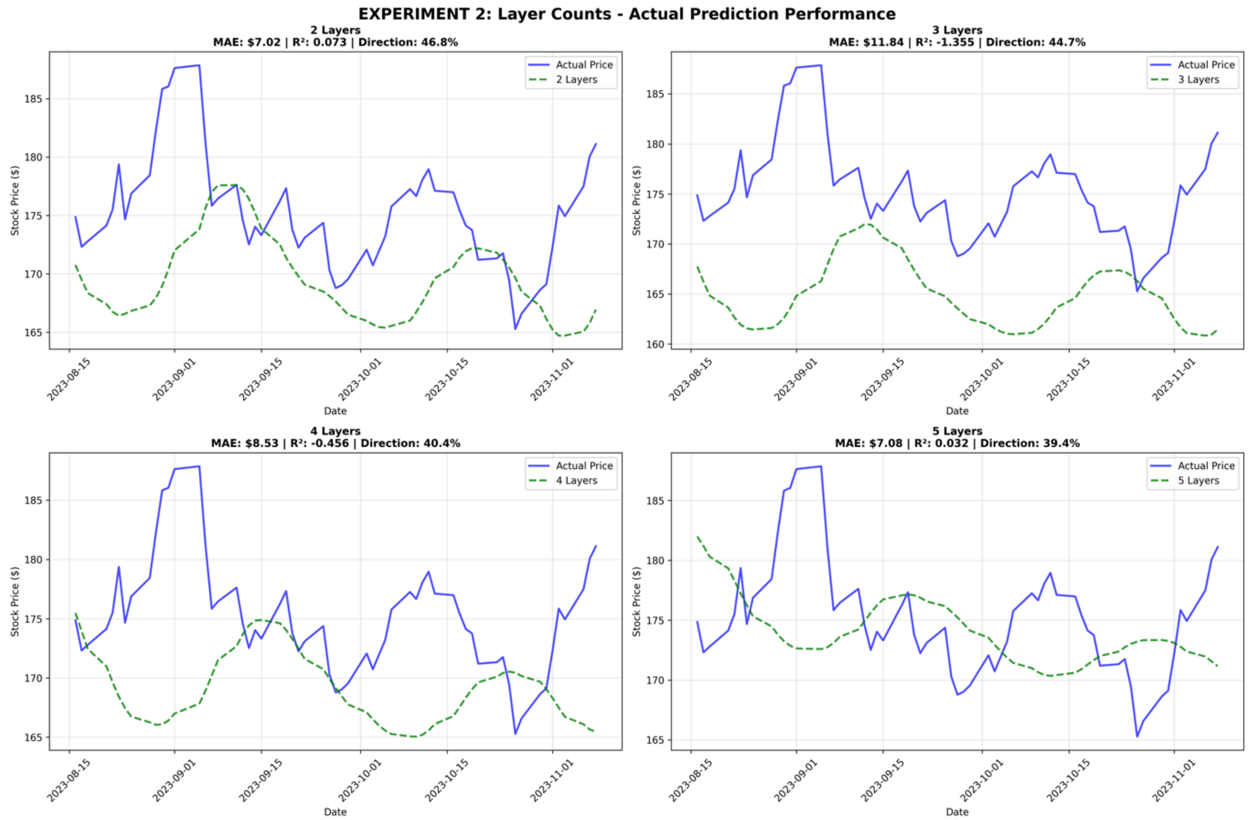
I experimented with 4 different deep learning networks – LSTM, GRU, SimpleRNN, and Bidirectional LSTM.
The results show **GRU performed best** with the lowest test loss.





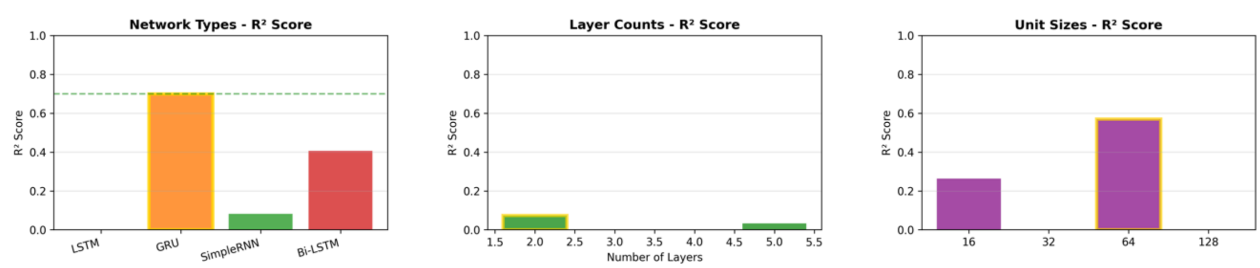
Graph 2 – Hyperparameter Tuning

I tested different hyperparameter configurations varying the number of layers and units per layer.
The results show that **2 layers achieved the best performance**.



Graph 3 – Experimental Results Summary

This summary shows all my experimental results.
GRU was the best network type, and the optimal configuration was 2 layers with 64 units.



📈 Experimental Results Summary Table

Experiment	**Best Config**	**MAE**	**R^2**
Networks	GRU	$3.97	0.701
Layers	2 Layers	$7.02	0.073
Units	64 Units	$4.66	0.570
Batch	Batch 32	$7.22	0.011