Task C.4 – Machine Learning 1

COS30018 – Intelligent Systems

Rahul Raju

Student ID: 105143065

Introduction

This week I worked on two main things for the stock price prediction project: building a configurable deep learning model function and then testing out different recurrent neural network (RNN) setups with various hyperparameters. The aim wasn't just to get code working, but to make the whole process more flexible so I could try out new ideas without rewriting big chunks of code each time.

Previously, I had a very rigid model setup. If I wanted to test an LSTM versus a GRU, I basically had to hardcode everything again. That made experimenting frustrating and error prone. So my goal this week was to create a create_model() function that could handle different types of RNNs, number of layers, bidirectional options, and so on. After that, I ran some tests to compare the performance of these models on financial time series data.

This report goes through what I implemented, some challenges I ran into, the results from my experiments, and a few key things I learned along the way.

Task 1: Designing and Implementing the Configurable Model Builder

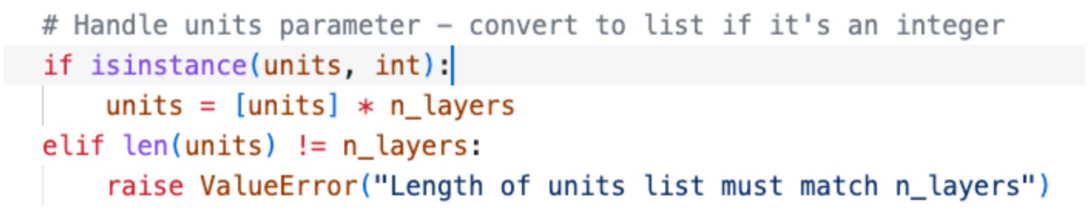
Why I Built It This Way

The idea behind the create_model() function was to avoid repeating the same boilerplate code every time I wanted to try LSTM, GRU, or SimpleRNN layers. Instead of copy-pasting, I wanted a single function where I could just pass parameters and get a model built automatically. This not only saves time but also makes it easier to test different ideas quickly.

Key Challenges and How I Solved Them

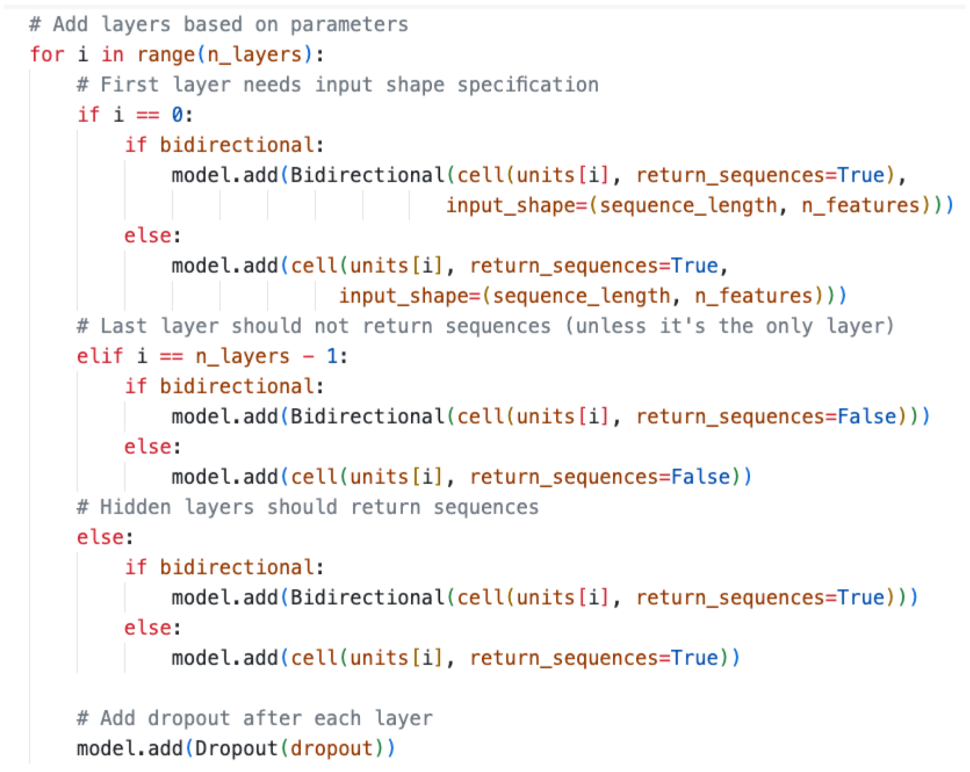
1. **Variable units per layer**

   I wasn't sure at first whether the function should accept a single number of units (same for all layers) or a list (different units per layer). In the end I made it accept both. This meant I had to add type checks and error handling because mismatched list lengths can lead to weird errors. I learned that in ML code, it's better to catch these things early rather than let them blow up halfway through training.



```python
# Handle units parameter - convert to list if it's an integer
if isinstance(units, int):
    units = [units] * n_layers
elif len(units) != n_layers:
    raise ValueError("Length of units list must match n_layers")
```

2. return_sequences confusion

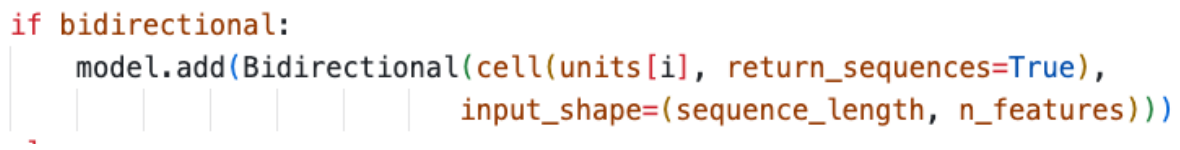


```python
# Add layers based on parameters
for i in range(n_layers):
    # First layer needs input shape specification
    if i == 0:
        if bidirectional:
            model.add(Bidirectional(cell(units[i], return_sequences=True),
                                    input_shape=(sequence_length, n_features)))
        else:
            model.add(cell(units[i], return_sequences=True,
                          input_shape=(sequence_length, n_features)))
    # Last layer should not return sequences (unless it's the only layer)
    elif i == n_layers - 1:
        if bidirectional:
            model.add(Bidirectional(cell(units[i], return_sequences=False)))
        else:
            model.add(cell(units[i], return_sequences=False))
    # Hidden layers should return sequences
    else:
        if bidirectional:
            model.add(Bidirectional(cell(units[i], return_sequences=True)))
        else:
            model.add(cell(units[i], return_sequences=True))

    # Add dropout after each layer
    model.add(Dropout(dropout))
```

One of the trickiest parts was setting return_sequences. If I got it wrong, the next layer would throw shape errors (expecting 3D inputs but getting 2D). After reading more about how recurrent layers work, I realised that intermediate layers need return_sequences=True so they pass the full sequence on. Only the final layer should collapse to a single vector. This gave me a much clearer picture of how RNNs handle sequences.

3. **Integrating Bidirectional Processing**

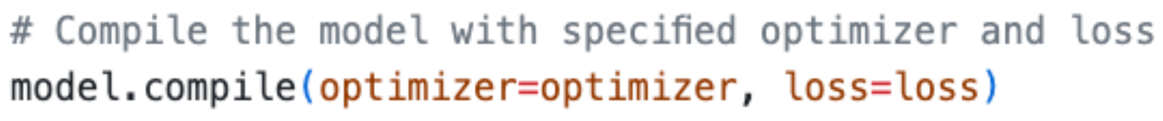


```
if bidirectional:
    model.add(Bidirectional(cell(units[i], return_sequences=True),
                            input_shape=(sequence_length, n_features)))
```

I added support for bidirectional models too. The main issue here was that the output dimension doubles, which I had to make sure later layers could handle. Training also took roughly twice as long, so it's not always worth it unless the accuracy gain is significant.

4. **Configurable optimizers and loss functions**

Finally, I made the optimizer and loss function configurable. This might seem like a small detail, but it's actually very useful because it allows me to experiment with training strategies without editing the core model code.



```
# Compile the model with specified optimizer and loss
model.compile(optimizer=optimizer, loss=loss)
```

This may seem trivial, but it is profoundly important. It allows us to experiment with different optimization strategies (Adam, RMSprop, SGD) and loss functions (MSE, MAE, Huber) without touching the architecture. In future work, this will enable us to conduct ablation studies on optimization effects independently of structural changes.

Task 2: Experimental Evaluation of Model Variants

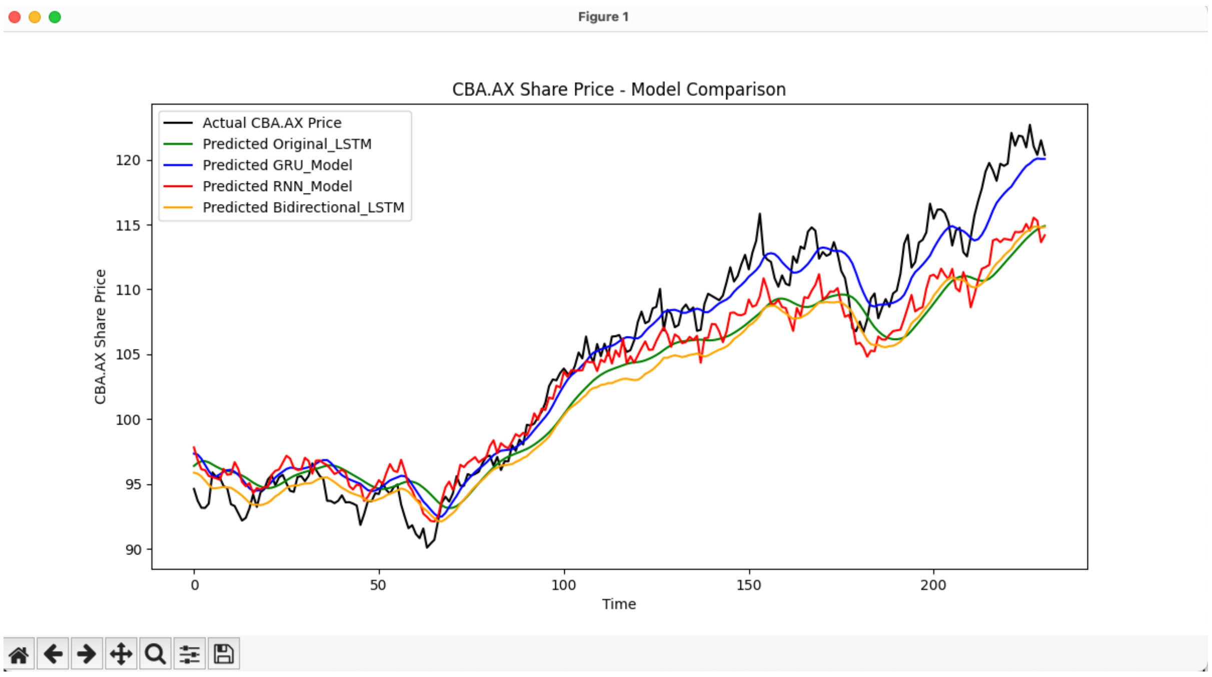
Experimental Design

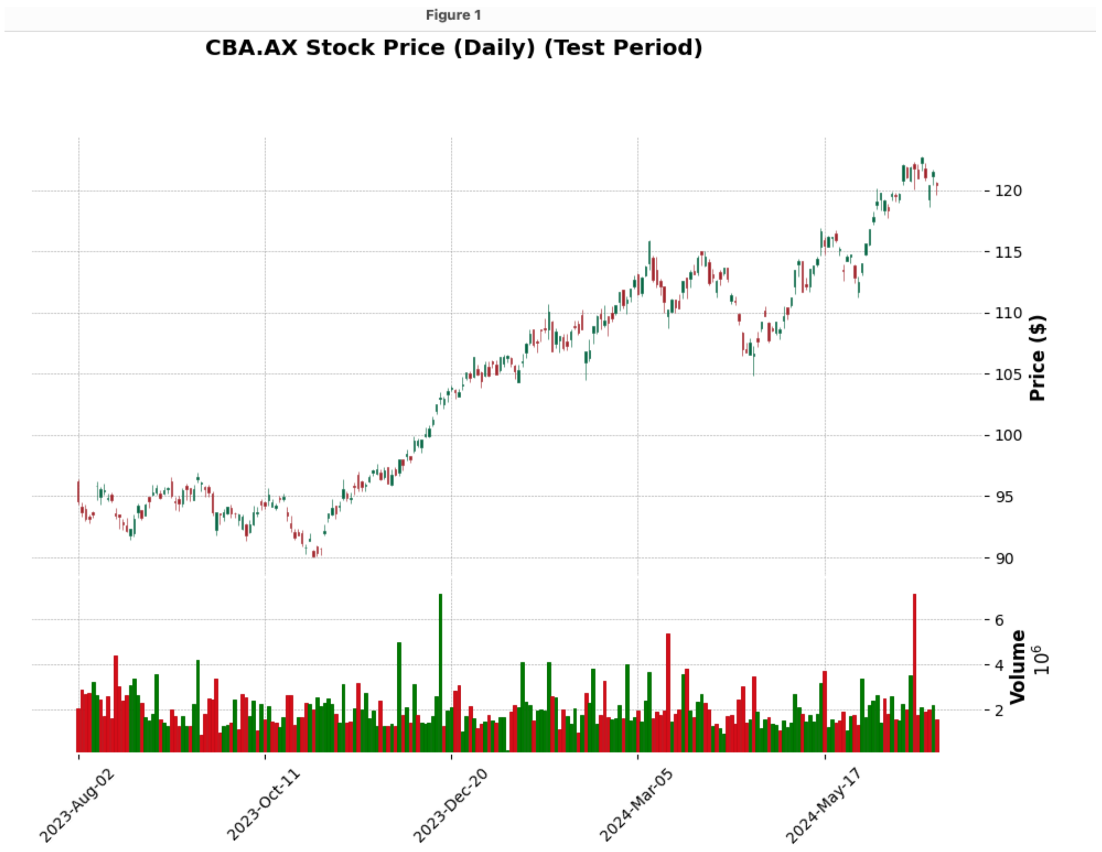
I tested four different models. To keep things consistent, all models used the same input sequence length (60 days) and one feature (scaled closing price).

MODEL NAME	CELL TYPE	LAYERS	UNITS	DROPOUT	BIDIRECTIONAL	EPOCHS	BATCH SIZE
Original_LSTM	LSTM	3	[50,50,50]	0.2	No	25	32
GRU_Model	GRU	3	[50,50,50]	0.2	No	15	32
RNN_Model	SimpleRNN	2	[50,50]	0.3	No	15	32
Bidirectional_LSTM	LSTM	2	[50,50]	0.2	Yes	15	32

I didn’t train all of them for the same number of epochs because I mainly wanted to see how they behaved early in training.

Results and Analysis





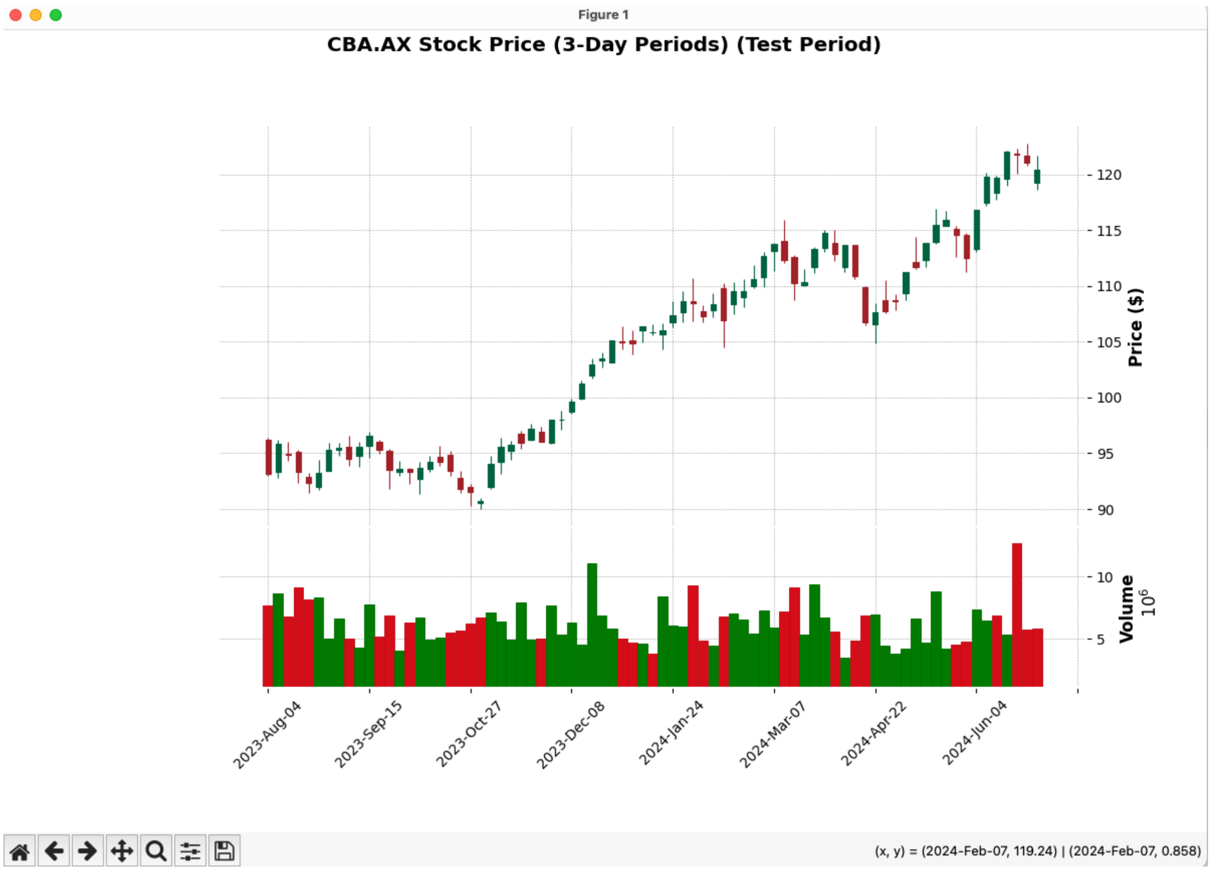


Figure 1
CBA.AX Stock Price (3-Day Periods) (Test Period)
Price ($)
Volume
10^6
2023-Aug-04
2023-Sep-15
2023-Oct-27
2023-Dec-08
2024-Jan-24
2024-Mar-07
2024-Apr-22
2024-Jun-04
(x, y) = (2024-Feb-07, 119.24) | (2024-Feb-07, 0.858)

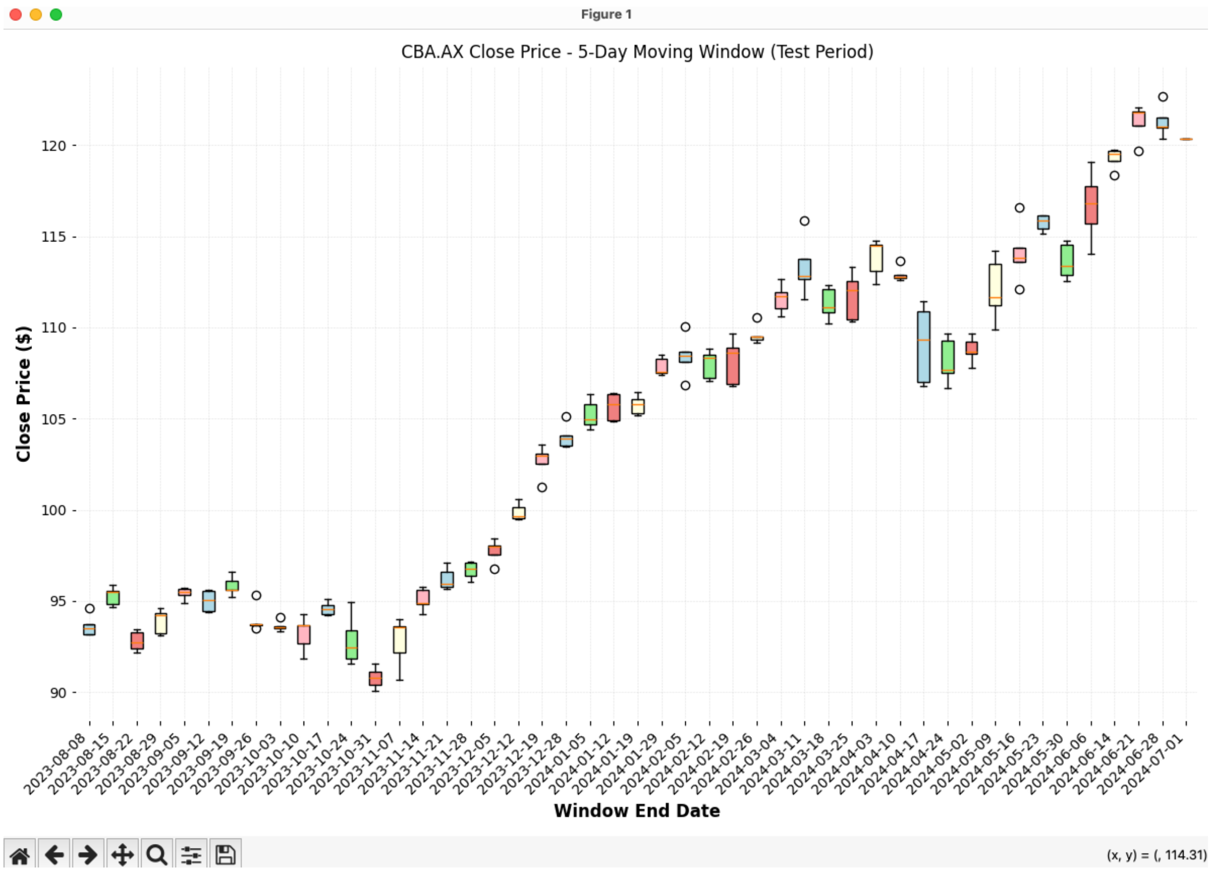
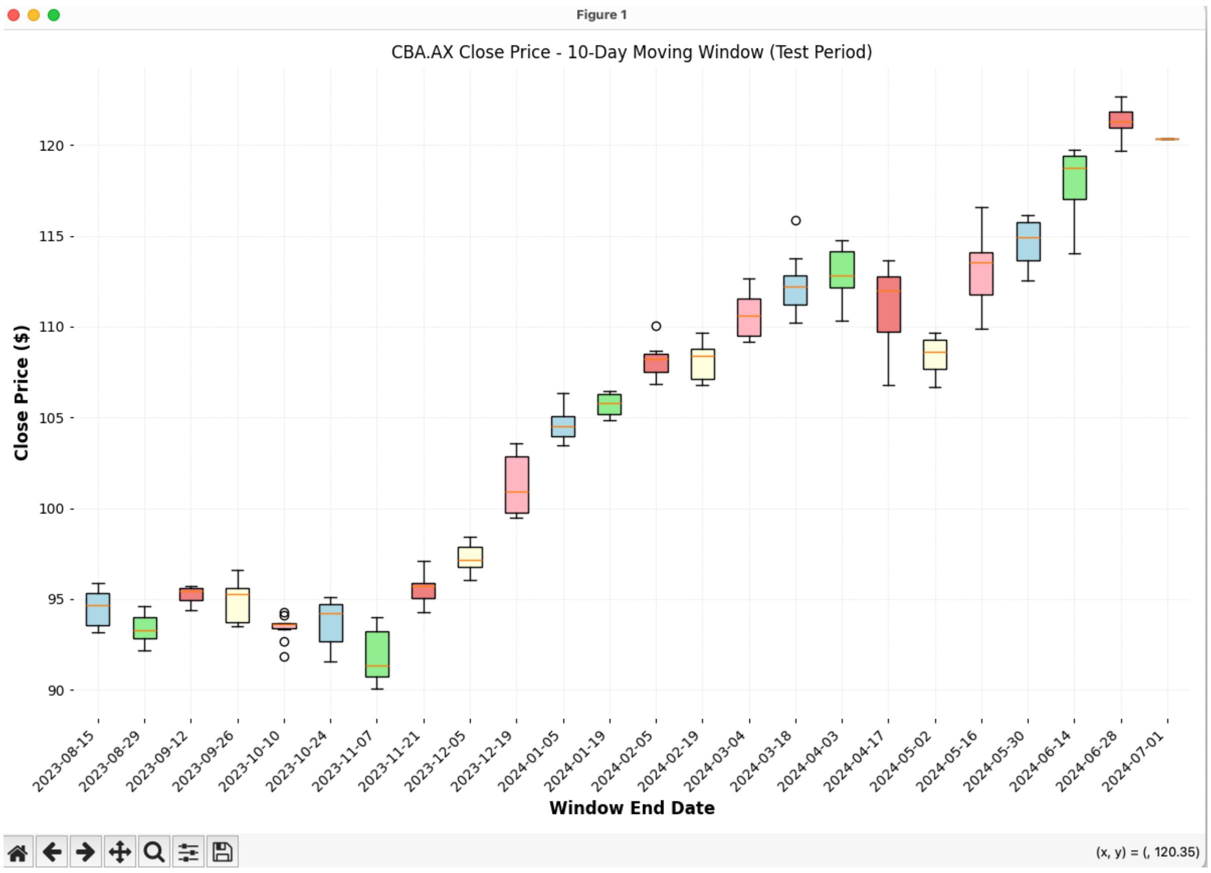
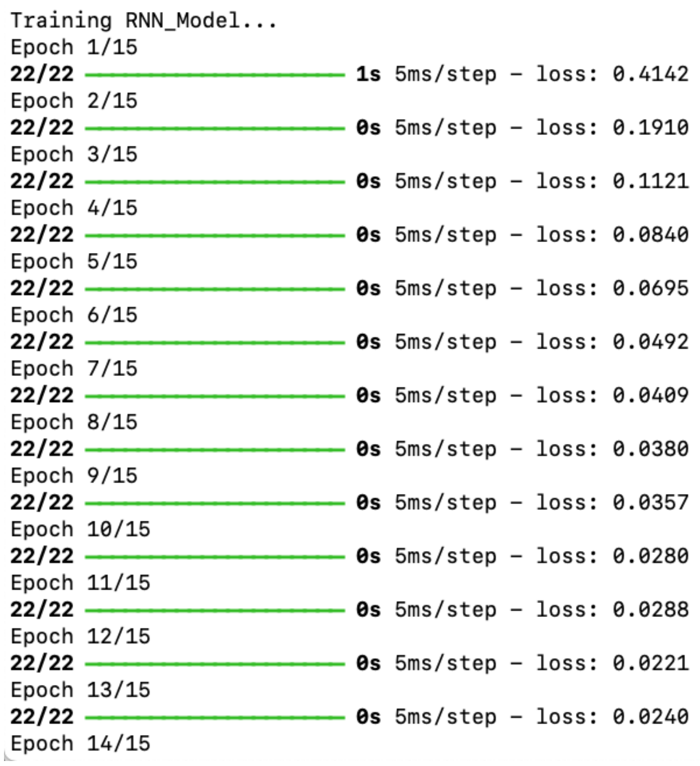


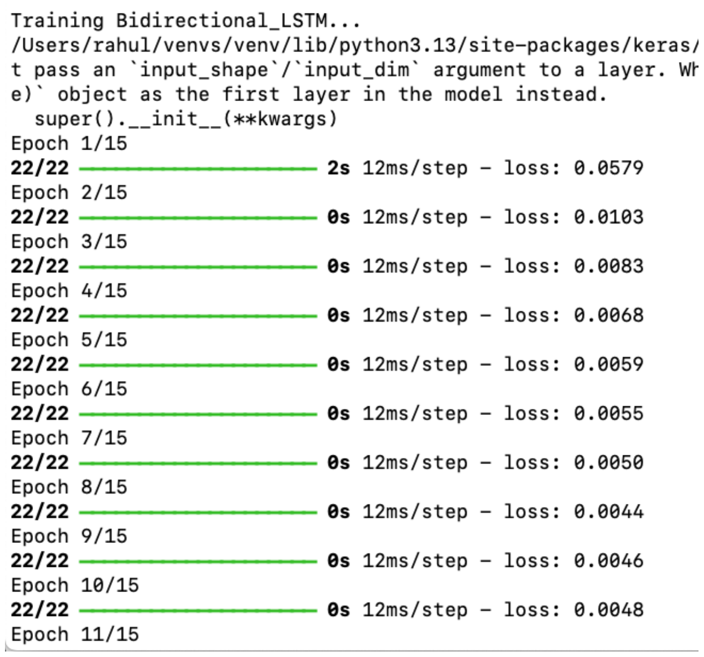
Figure 1
CBA.AX Close Price - 5-Day Moving Window (Test Period)
Close Price ($)
Window End Date
(x, y) = (, 114.31)



```
Training GRU_Model...
Epoch 1/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - loss: 0.0617
Epoch 2/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0119
Epoch 3/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0087
Epoch 4/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0070
Epoch 5/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0065
Epoch 6/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0058
Epoch 7/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0060
Epoch 8/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0052
Epoch 9/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0066
Epoch 10/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0067
Epoch 11/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0054
Epoch 12/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0050
Epoch 13/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.0053
```



```
Training RNN_Model...
Epoch 1/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - loss: 0.4142
Epoch 2/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.1910
Epoch 3/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.1121
Epoch 4/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0840
Epoch 5/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0695
Epoch 6/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0492
Epoch 7/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0409
Epoch 8/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0380
Epoch 9/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0357
Epoch 10/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0280
Epoch 11/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0288
Epoch 12/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0221
Epoch 13/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0240
Epoch 14/15
```



```
Training Bidirectional_LSTM...
/Users/rahul/venvs/venv/lib/python3.13/site-packages/keras/
t pass an `input_shape`/`input_dim` argument to a layer. Wh
e)` object as the first layer in the model instead.
  super().__init__(**kwargs)
Epoch 1/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - loss: 0.0579
Epoch 2/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0103
Epoch 3/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0083
Epoch 4/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0068
Epoch 5/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0059
Epoch 6/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0055
Epoch 7/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0050
Epoch 8/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0044
Epoch 9/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0046
Epoch 10/15
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0048
Epoch 11/15
```

Observations

- **Original LSTM:** This was my baseline. It tracked trends fairly well but had a slight lag compared to actual prices. That seems pretty normal for this type of model.
- **GRU:** I was surprised at how well it did, even with fewer epochs. It sometimes performed just as well as the LSTM. This matches what I've read — GRUs are often just as good but faster.
- **SimpleRNN:** This one really struggled. Predictions were erratic and it underfit badly. Increasing dropout didn't help much. This confirmed what I've learned about SimpleRNNs not handling long sequences well.
- **Bidirectional LSTM:** The most interesting result. It didn't have a lower overall error than the baseline LSTM, but it seemed better at picking up turning points in the trend. The tradeoff is that it took twice as long to train.

Training Insights

- GRUs trained the fastest per epoch.
- SimpleRNN had the steepest drop in loss early on but then plateaued quickly.
- Both LSTM models kept improving steadily and probably would have benefited from more epochs.

Reflections and Lessons Learned

This week was less about getting the "best" model and more about improving the way I approach building them. Creating the create_model() function made me think about design choices in a more systematic way instead of just hacking something together.

A big lesson was learning to change only one variable at a time when experimenting. At first, I made the mistake of altering too many things (like units, dropout, and epochs all at once), which made it impossible to know why performance changed. Keeping experiments controlled made the results easier to interpret.

Debugging was also a huge part of the process. For example, the ndim=3, found ndim=2 error had me stuck for a while, but going back to the docs and forums helped me understand the cause. I feel more confident now in handling these errors without panicking.

Overall, I feel like I've shifted from just coding to actually engineering ML systems, even if only at a small scale.

Resources and References

The implementation and experimentation phases relied on several key resources:

- Official Keras and TensorFlow Documentation. Essential for understanding layer parameters, shape requirements, and compilation options. Particularly useful was the guide on working with RNNs, which clarified the role of return_sequences.
- Academic Papers and Blog Posts. I consulted several articles comparing LSTM, GRU, and SimpleRNN performance on time series tasks. One particularly helpful post on Towards Data Science explained the gating mechanisms in intuitive terms, helping me understand why GRUs might perform comparably to LSTMs with fewer parameters.
- Stack Overflow and GitHub Issues. Invaluable for debugging. A common error, “expected ndim=3, found ndim=2,” led me to several threads explaining the importance of return_sequences in stacked RNNs.
- The P1 Reference Codebase. Provided the initial template for the create_model() function. I extended it significantly, adding support for variable units, bidirectional layers, and improved error handling.