Time Series & Recurrent Neural Networks

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Patrick Laub

Lecture Outline

- **Time Series**
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- Predicting Multiple Time Series

Tabular data vs time series data

Tabular data

We have a dataset $\{\boldsymbol{x}_i, y_i\}_{i=1}^{n}$ which we assume are i.i.d. observations.

Brand	Mileage	# Claims
BMW	101 km	1
Audi	432 km	0
Volvo	3 km	5
⋮	⋮	⋮

The goal is to *predict* the y for some covariates $\boldsymbol{x}$.

Time series data

Have a sequence $\{\boldsymbol{x}_t, y_t\}_{t=1}^{T}$ of observations taken at regular time intervals.

Date	Humidity	Temp.
Jan 1	60%	20 °C
Jan 2	65%	22 °C
Jan 3	70%	21 °C
⋮	⋮	⋮

The task is to *forecast* future values based on the past.

Attributes of time series data

- **Temporal ordering**: The order of the observations matters.
- **Trend**: The general direction of the data.
- **Noise**: Random fluctuations in the data.
- **Seasonality**: Patterns that repeat at regular intervals.

> **Question**
>
> What will be the temperature in Berlin tomorrow? What information would you use to make a prediction?

Australian financial stocks

```python
# First, install yfinance if you haven't already:
# pip install yfinance
import yfinance as yf
import pandas as pd
from datetime import date

# 1. Define the ASX tickers (Yahoo uses “.AX” for Australian stocks
#    and “^AXJO” for the S&P/ASX 200 index)
tickers = {
    "ANZ":    "ANZ.AX",
    "BOQ":    "BOQ.AX",
    "CBA":    "CBA.AX",
    "NAB":    "NAB.AX",
    "QBE":    "QBE.AX",
    "SUN":    "SUN.AX",
    "WBC":    "WBC.AX",
    "ASX200": "^AXJO"
}

# 2. Choose your date range
start = "1999-01-01"
end   = date.today().isoformat()  # e.g. "2025-06-29"

# 3. Download the data
#    This returns a Panel-like DataFrame: columns are tickers, rows are trading dates.
raw_data = yf.download(
    tickers=list(tickers.values()),
    start=start,
    end=end,
    progress=False
)

raw_data.to_csv("asx_raw_data.csv")  # Optional: Save raw data for inspection

```

Australian financial stocks

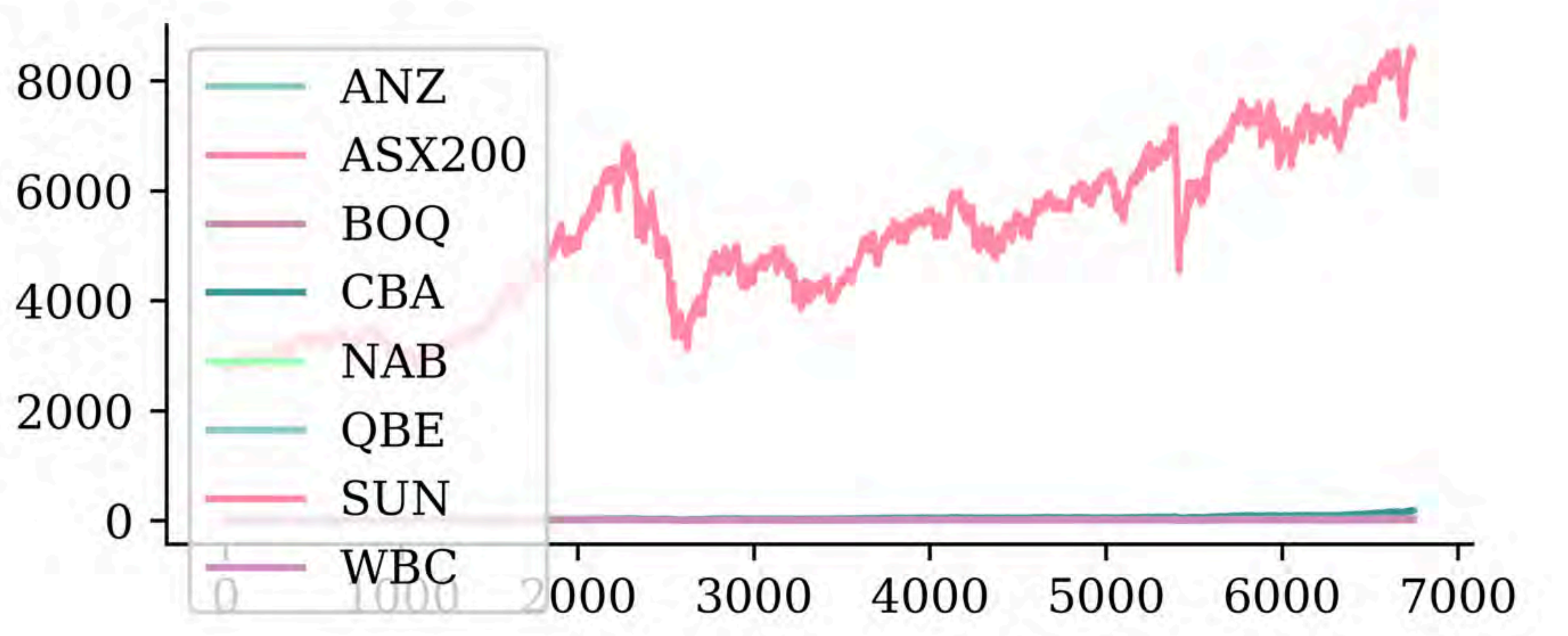
```
stocks = pd.read_csv("../data/aus_fin_stocks.csv")
stocks
```

	Date	ANZ	ASX200	BOQ	CBA	NAB	QBE	SUN	WBC
0	1999-01-01	8.413491	NaN	NaN	14.560169	NaN	NaN	7.965791	NaN
1	1999-01-04	8.476515	2732.199951	NaN	14.402927	12.995510	2.648447	7.965791	6.370629
2	1999-01-05	8.452881	2716.600098	NaN	14.409224	13.169948	2.564247	7.965791	6.485921
...	...	...	...	...	...	...	...	...	...
6742	2025-06-25	29.100000	8559.200195	7.86	191.399994	40.049999	23.480000	21.750000	34.540001
6743	2025-06-26	29.740000	8550.799805	7.86	190.710007	39.889999	23.330000	21.459999	34.570000
6744	2025-06-27	29.200001	8514.200195	7.77	185.360001	39.259998	23.219999	21.330000	33.900002

6745 rows × 9 columns

Plot

```python
stocks.plot()
```



Data types and NA values

```python
stocks.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 6745 entries, 0 to 6744
Data columns (total 9 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date    6745 non-null   str
 1   ANZ     6745 non-null   float64
 2   ASX200  6691 non-null   float64
 3   BOQ     6599 non-null   float64
 4   CBA     6745 non-null   float64
 5   NAB     6744 non-null   float64
 6   QBE     6744 non-null   float64
 7   SUN     6745 non-null   float64
 8   WBC     6744 non-null   float64
dtypes: float64(8), str(1)
memory usage: 540.3 KB
```

```python
for col in stocks.columns:
    print(f"{col}: {stocks[col].isna().su
```

```
Date: 0
ANZ: 0
ASX200: 54
BOQ: 146
CBA: 0
NAB: 1
QBE: 1
SUN: 0
WBC: 1
```

```python
asx200 = stocks.pop("ASX200")
```

Set the index to the date

```
1 stocks["Date"] = pd.to_datetime(stocks["Date"])
2 stocks = stocks.set_index("Date") # or `stocks.set_index("Date", inplace=True)`
3 stocks
```

	ANZ	BOQ	CBA	NAB	QBE	SUN	WBC
Date							
1999-01-01	8.413491	NaN	14.560169	NaN	NaN	7.965791	NaN
1999-01-04	8.476515	NaN	14.402927	12.995510	2.648447	7.965791	6.370629
1999-01-05	8.452881	NaN	14.409224	13.169948	2.564247	7.965791	6.485921
...	...	...	...	...	...	...	...
2025-06-25	29.100000	7.86	191.399994	40.049999	23.480000	21.750000	34.540001
2025-06-26	29.740000	7.86	190.710007	39.889999	23.330000	21.459999	34.570000
2025-06-27	29.200001	7.77	185.360001	39.259998	23.219999	21.330000	33.900002

6745 rows × 7 columns

Plot II

```python
stocks.plot()
plt.ylabel("Stock Price ($)")
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.5), ncol=4);
```

Can index using dates I

```
stocks.loc["2010-1-4":"2010-01-8"]
```

Date	ANZ	BOQ	CBA	NAB	QBE	SUN	WBC
2010-01-04	18.874071	8.412694	34.510429	14.440043	13.831507	10.478957	14.958415
2010-01-05	18.964771	8.520640	35.032455	14.624498	13.902833	10.636262	15.082577
2010-01-06	18.684425	8.477461	35.208561	14.339908	13.688861	10.406354	15.011630
2010-01-07	18.239164	8.218388	34.868923	14.223969	13.441965	10.551559	14.810603
2010-01-08	18.346357	8.419890	35.321777	14.176538	13.590102	10.612061	14.869730

Note, these ranges are *inclusive*, not like Python’s normal slicing.

Can index using dates II

So to get 2019's December and all of 2020 for CBA:

```
1 stocks.loc["2019-12":"2020", ["CBA"]]
```

	CBA
Date	
2019-12-02	66.193298
2019-12-03	64.494362
2019-12-04	63.250660
...	...
2020-12-29	70.822792
2020-12-30	70.468712
2020-12-31	69.221031

275 rows × 1 columns

Can look at the first differences

```python
stocks.diff().plot()
plt.ylabel("Daily Price Changes ($)")
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.5), ncol=4);
```

Daily Price Changes ($)
5
0
−5
−10
2000 2004 2008 2012 2016 2020 2024
Date
ANZ CBA QBE WBC
BOQ NAB SUN

Can look at the percentage changes

```python
stocks.pct_change().plot()
plt.ylabel("Daily Returns (%)")
plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.5), ncol=4);
```

Daily Returns (%)
0.5
0.0
2000
2004
2008
2012
2016
2020
2024
Date
ANZ
CBA
QBE
WBC
BOQ
NAB
SUN

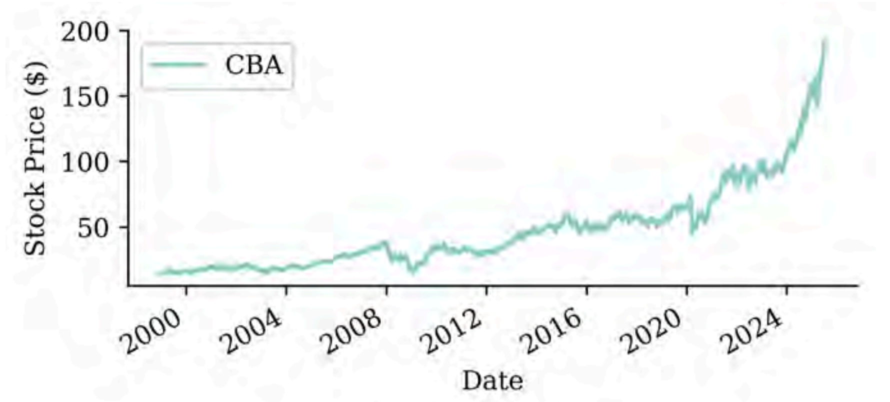
Focus on one stock

```
1  stock = stocks[["CBA"]].copy()
2  stock
```

	CBA
Date	
1999-01-01	14.560169
1999-01-04	14.402927
1999-01-05	14.409224
...	...
2025-06-25	191.399994
2025-06-26	190.710007
2025-06-27	185.360001

6745 rows × 1 columns

```
1  stock.plot()
2  plt.ylabel("Stock Price ($)");
```




```
1  stock.isna().sum()
```

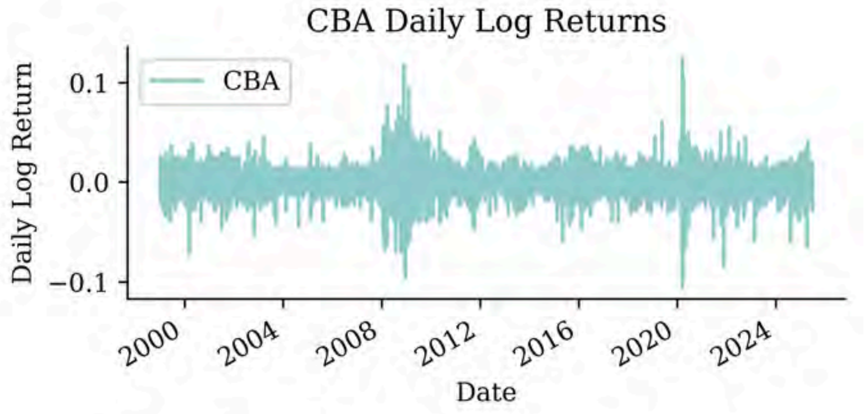
```
CBA      0
dtype: int64
```

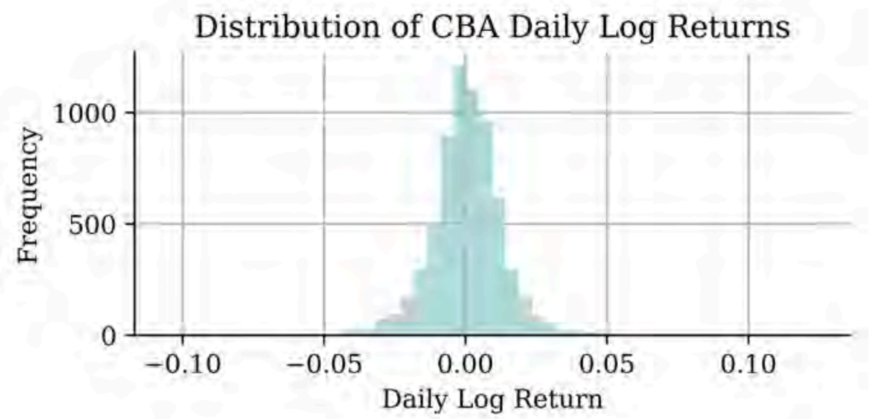
Convert to log returns

Instead of working with raw prices, we'll work with daily log returns:

```
1  # Calculate log returns
2  stock_log = np.log(stock / stock.shift(1)
3  stock_log.head()
```

	CBA
Date	
1999-01-04	-0.010858
1999-01-05	0.000437
1999-01-06	0.008042
1999-01-07	0.025859
1999-01-08	-0.021323





Fill in the missing values

```
asx200 = pd.DataFrame(asx200).set_index(stocks.index)
missing_day = asx200.index[asx200["ASX200"].isna()][1]
prev_day = missing_day - pd.Timedelta(days=1)
after = missing_day + pd.Timedelta(days=3)
```

```
asx200.loc[prev_day:after]
```

	ASX200
Date	
1999-01-25	2713.100098
1999-01-26	NaN
1999-01-27	2738.800049
1999-01-28	2765.100098
1999-01-29	2781.699951

```
asx200 = asx200.ffill()
asx200.loc[prev_day:after]
```

	ASX200
Date	
1999-01-25	2713.100098
1999-01-26	2713.100098
1999-01-27	2738.800049
1999-01-28	2765.100098
1999-01-29	2781.699951

Lecture Outline

- Time Series
- **Baseline forecasts**
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- Predicting Multiple Time Series

Helper functions

```python
def log_to_price(log_returns, initial_price):
    """Convert log returns to raw prices given an initial price."""
    # Use cumulative sum of log returns for numerical stability
    # P_t = P_0 * exp(sum of log returns from 1 to t)
    cumulative_log_returns = log_returns.cumsum()
    prices = initial_price * np.exp(cumulative_log_returns)

    return prices

def get_last_price(stock_df, cutoff_date):
    """Get the last known price before the forecast period starts."""
    last_known_date = stock_df.loc[:cutoff_date].index[-1]
    return stock_df.loc[last_known_date, "CBA"]
```

Persistence forecast

Predict the next value to be the same as the current value.

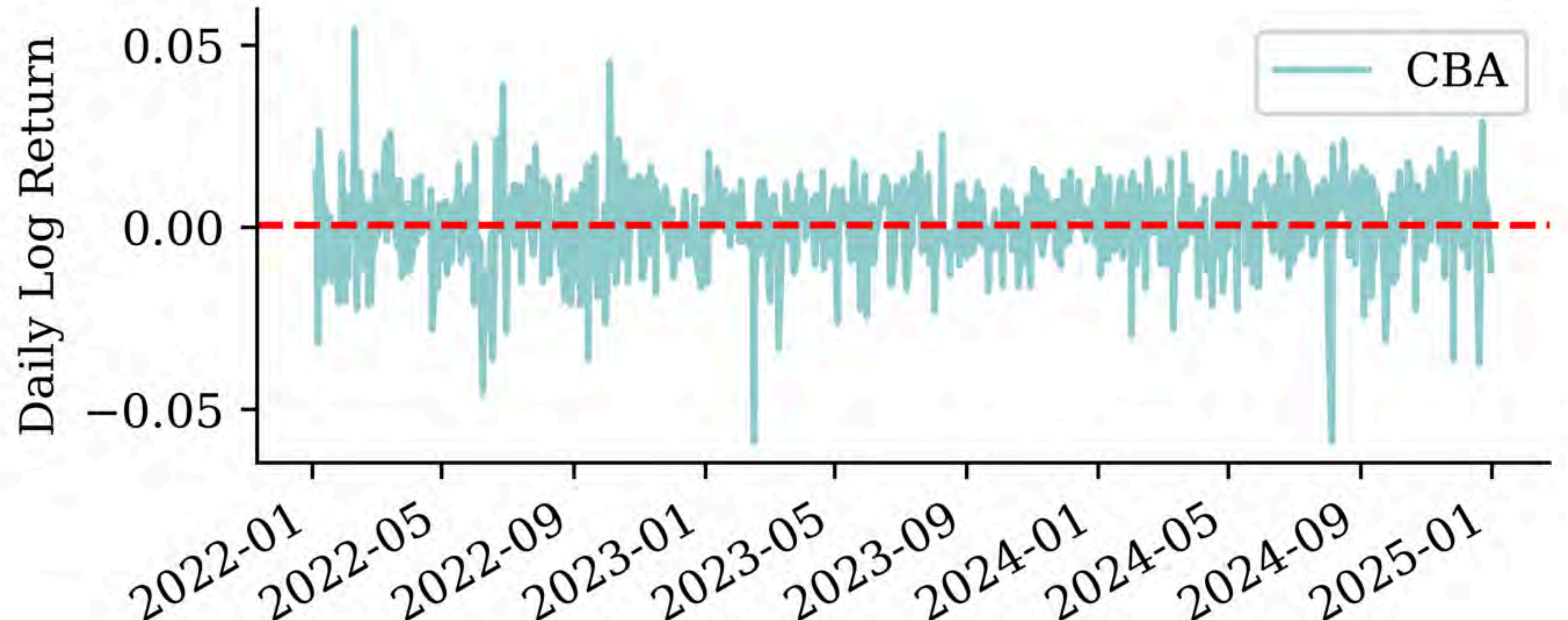
```python
last_price = get_last_price(stock, cutoff_date="2024-12")
log_persistence = pd.Series(0.0, index=stock_log.loc["2025":].index)
persistence_prices = log_to_price(log_persistence, last_price)
```

Extrapolate the trend

```python
recent_log_returns = stock_log.loc["2022-01":"2024-12"]
trend_log = recent_log_returns.mean().values[0]
print(f"Average daily log return: {trend_log:.6f}")
```

```
Average daily log return: 0.000709
```

```python
log_trend = pd.Series(trend_log, index=stock_log.loc["2025":].index)
trend_prices = log_to_price(log_trend, last_price)
```



Trend fitted

```python
# Create trend forecast for the recent period to show fitted trend
recent_trend_log = pd.Series(trend_log, index=recent_log_returns.index)
trend_start_price = get_last_price(stock, cutoff_date=recent_log_returns.index[0].strftim
recent_trend_prices = log_to_price(recent_trend_log, trend_start_price)
```

Trend forecasts

CBA
Persistence
Trend
Stock Price ($)
180
160
140
2024-12
2025-01
2025-02
2025-03
2025-04
2025-05
2025-06
2025-07
Date

Which is better?

If we look at the mean squared error (MSE) of the two models:

```
# Calculate MSE using the actual forecasts we computed
actual_prices = stock.loc["2025":, "CBA"]
persistence_mse = mean_squared_error(actual_prices, persistence_prices)
trend_mse = mean_squared_error(actual_prices, trend_prices)
persistence_mse, trend_mse
```

```
(254.04075100411256, 99.28717915684173)
```

Use the history

```python
cba_log_shifted = stock_log["CBA"].head().shift(1)
both = pd.concat([stock_log["CBA"].head(), cba_log_shifted], axis=1, keys=["Today", "Yest
both
```

	Today	Yesterday
Date		
1999-01-04	-0.010858	NaN
1999-01-05	0.000437	-0.010858
1999-01-06	0.008042	0.000437
1999-01-07	0.025859	0.008042
1999-01-08	-0.021323	0.025859

```python
def lagged_timeseries(df, target, window=40):
    lagged = pd.DataFrame()
    for i in range(window, 0, -1):
        lagged[f"T-{i}"] = df[target].shift(i)
    lagged["T"] = df[target].values
    return lagged
```

Lagged time series

```
df_lags = lagged_timeseries(stock_log, "CBA", 40)
df_lags
```

	T-40	T-39	T-38	T-37	T-36	T-35	T-34	T-33	T-32	T-31
Date										
1999-01-04	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-01-05	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...
2025-06-26	0.021968	0.003894	0.014307	-0.016222	-0.001139	-0.004689	-0.002715	0.009202	0.000838	-0.006240
2025-06-27	0.003894	0.014307	-0.016222	-0.001139	-0.004689	-0.002715	0.009202	0.000838	-0.006240	0.008153

6744 rows × 41 columns

Split into training and testing

```python
# Split the data in time
X_train = df_lags.loc[:"2021"]
X_val = df_lags.loc["2022-01":"2024-12"]  # 2022-2024
X_test = df_lags.loc["2025":]  # 2025

# Remove any with NAs and split into X and y
X_train = X_train.dropna()
X_val = X_val.dropna()
X_test = X_test.dropna()

y_train = X_train.pop("T")
y_val = X_val.pop("T")
y_test = X_test.pop("T")
```

```python
X_train.shape, y_train.shape, X_val.shape, y_val.shape, X_test.shape, y_test.shape
```

```
((5825, 40), (5825,), (757, 40), (757,), (122, 40), (122,))
```

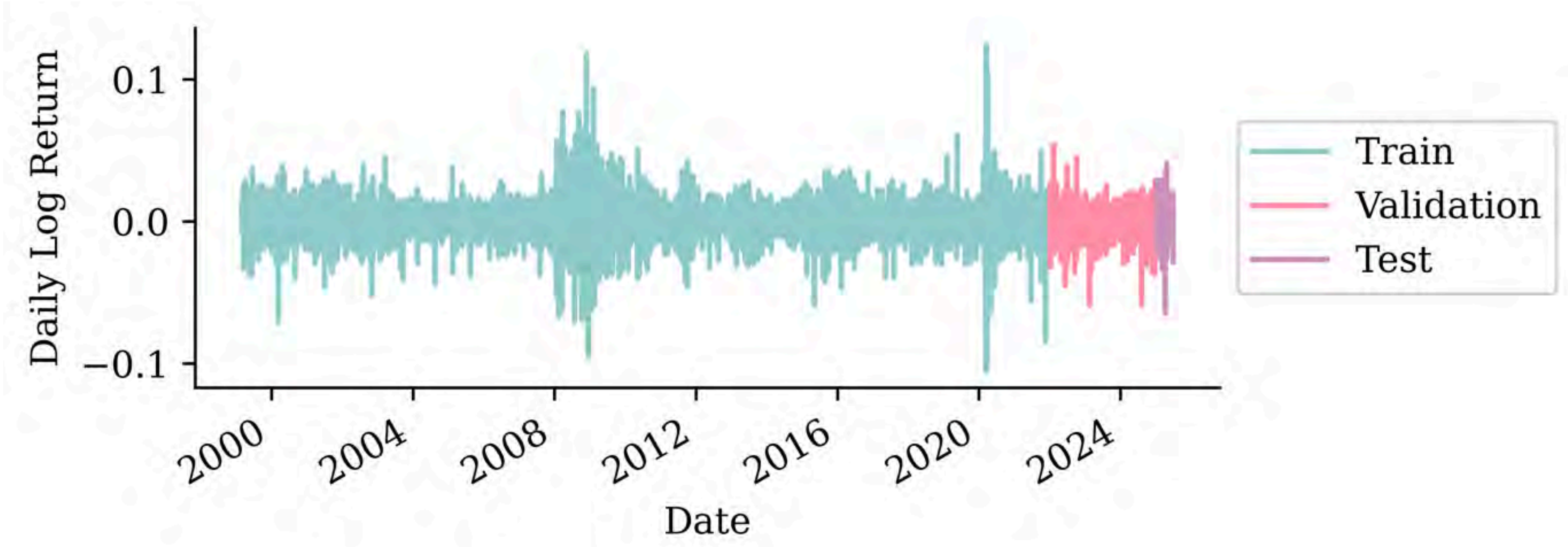
Inspect the split data

```
1  X_train
```

	T-40	T-39	T-38	T-37	T-36	T-35	T-34	T-33	T-32	T-3
Date										
1999-03-01	-0.010858	0.000437	0.008042	0.025859	-0.021323	-0.010573	-0.011214	-0.014823	-0.009704	0.018119
1999-03-02	0.000437	0.008042	0.025859	-0.021323	-0.010573	-0.011214	-0.014823	-0.009704	0.018119	0.012994
1999-03-03	0.008042	0.025859	-0.021323	-0.010573	-0.011214	-0.014823	-0.009704	0.018119	0.012994	0.010881
...	...	...	...	...	...	...	...	...	...	...
2021-12-29	0.015263	-0.004999	0.011656	0.013921	0.011090	0.003821	-0.012058	0.000919	-0.016106	0.008825
2021-12-30	-0.004999	0.011656	0.013921	0.011090	0.003821	-0.012058	0.000919	-0.016106	0.008825	-0.00101
2021-12-31	0.011656	0.013921	0.011090	0.003821	-0.012058	0.000919	-0.016106	0.008825	-0.001018	-0.00306

5825 rows × 40 columns

Plot the split



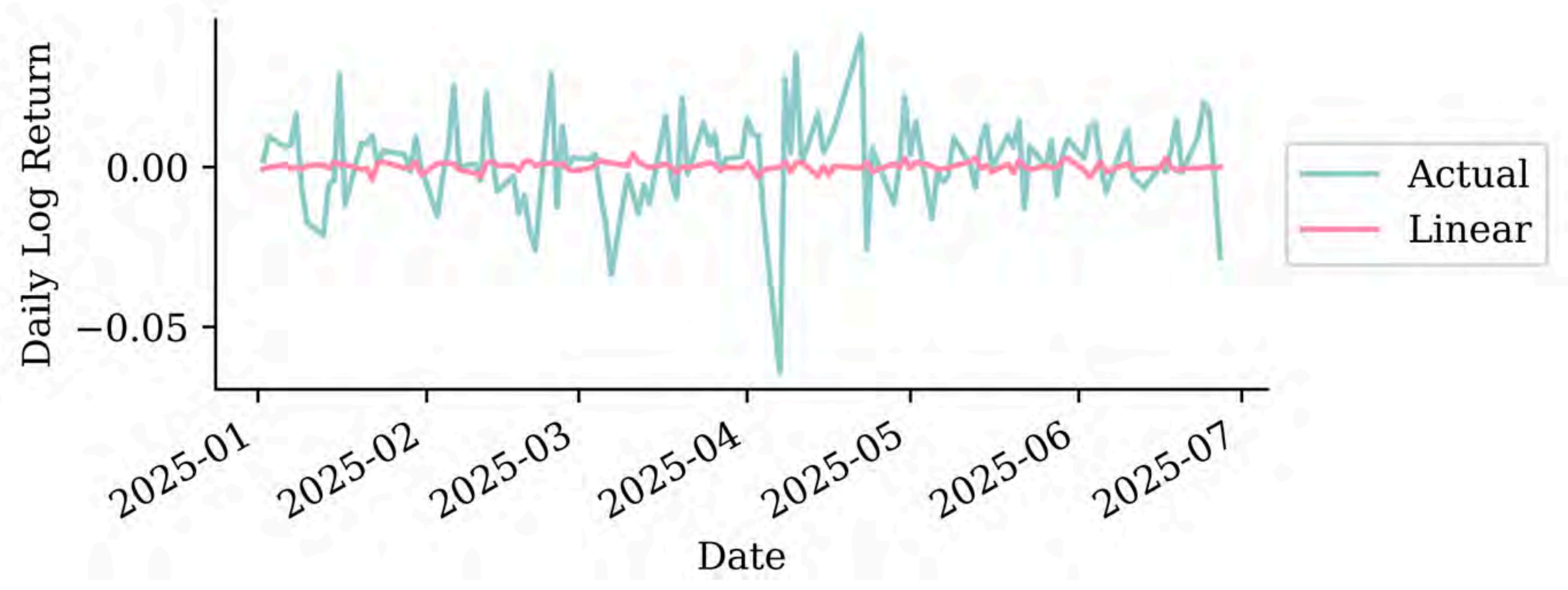
Fit a linear model

```
1  lr = LinearRegression()
2  lr.fit(X_train, y_train);
```

Make a forecast for the test data:

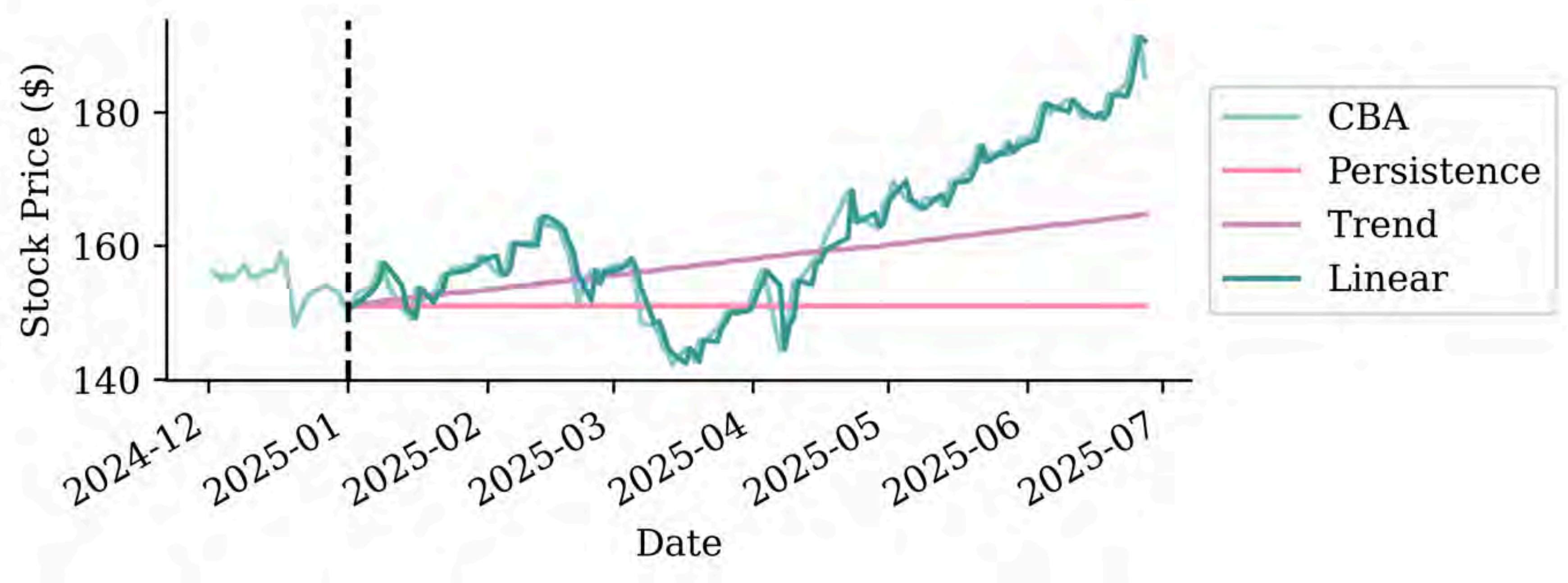
```
1  y_pred = lr.predict(X_test)
```

Plot predictions in log return space

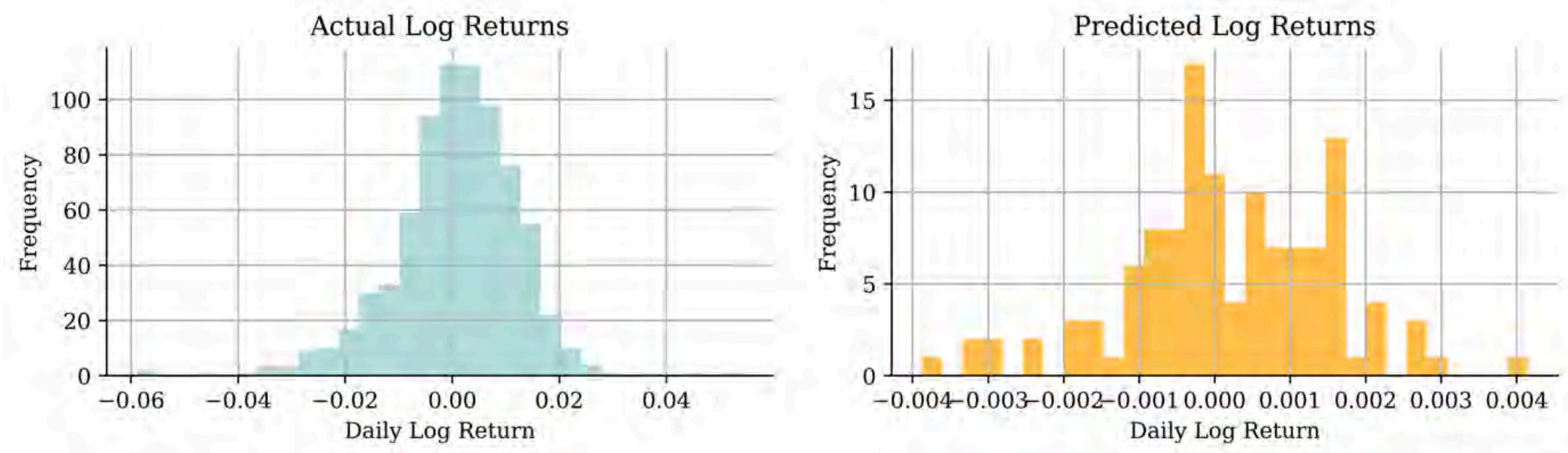


Convert to raw prices and plot

```python
linear_log_forecasts = pd.Series(y_pred, index=y_test.index)
prev_price = stock.shift(1).loc["2025", "CBA"]
linear_price_forecasts = prev_price * np.exp(linear_log_forecasts)
```



Distribution of log return predictions




```
1  # Calculate MSE on raw prices using explicit forecast calculation
2  test_actual_prices = stock.loc["2025":, "CBA"]
3  test_linear_prices = linear_price_forecasts.loc["2025":]
4  linear_mse = mean_squared_error(test_actual_prices, test_linear_prices)
5  linear_mse
```

```
5.1366640383449536
```

```
1  persistence_mse, trend_mse, linear_mse
```

```
(254.04075100411256, 99.28717915684173, 5.1366640383449536)
```

Lecture Outline

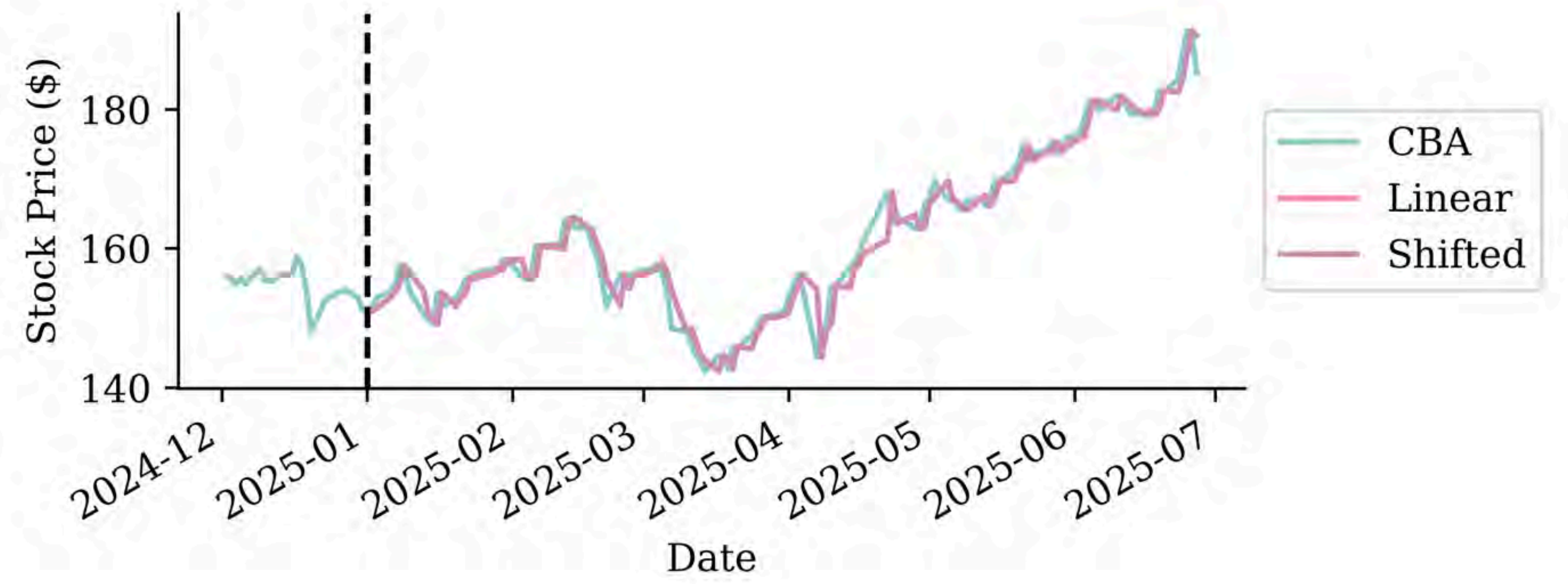
- Time Series
- Baseline forecasts
- **Multi-step forecasts**
- Neural network forecasts
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- Predicting Multiple Time Series

Comparing apples to apples

The linear model is only producing *one-step-ahead* forecasts.

The other models are producing *multi-step-ahead* forecasts.

```
shifted_forecast = stock["CBA"].shift(1).loc["2025":]
```




```
shifted_mse = mean_squared_error(stock.loc["2025":, "CBA"], shifted_forecast)
persistence_mse, trend_mse, linear_mse, shifted_mse
```

```
(254.04075100411256, 99.28717915684173, 5.1366640383449536, 5.06190042634739)
```

Autoregressive forecasts

The linear model needs the last 90 days to make a forecast.

Idea: Make the first forecast, then use that to make the next forecast, and so on.

$$\begin{aligned}
\hat{y}_t &= \beta_0 + \beta_1 y_{t-1} + \beta_2 y_{t-2} + \ldots + \beta_n y_{t-n} \\
\hat{y}_{t+1} &= \beta_0 + \beta_1 \hat{y}_t + \beta_2 y_{t-1} + \ldots + \beta_n y_{t-n+1} \\
\hat{y}_{t+2} &= \beta_0 + \beta_1 \hat{y}_{t+1} + \beta_2 \hat{y}_t + \ldots + \beta_n y_{t-n+2}
\end{aligned}$$

$$\vdots$$

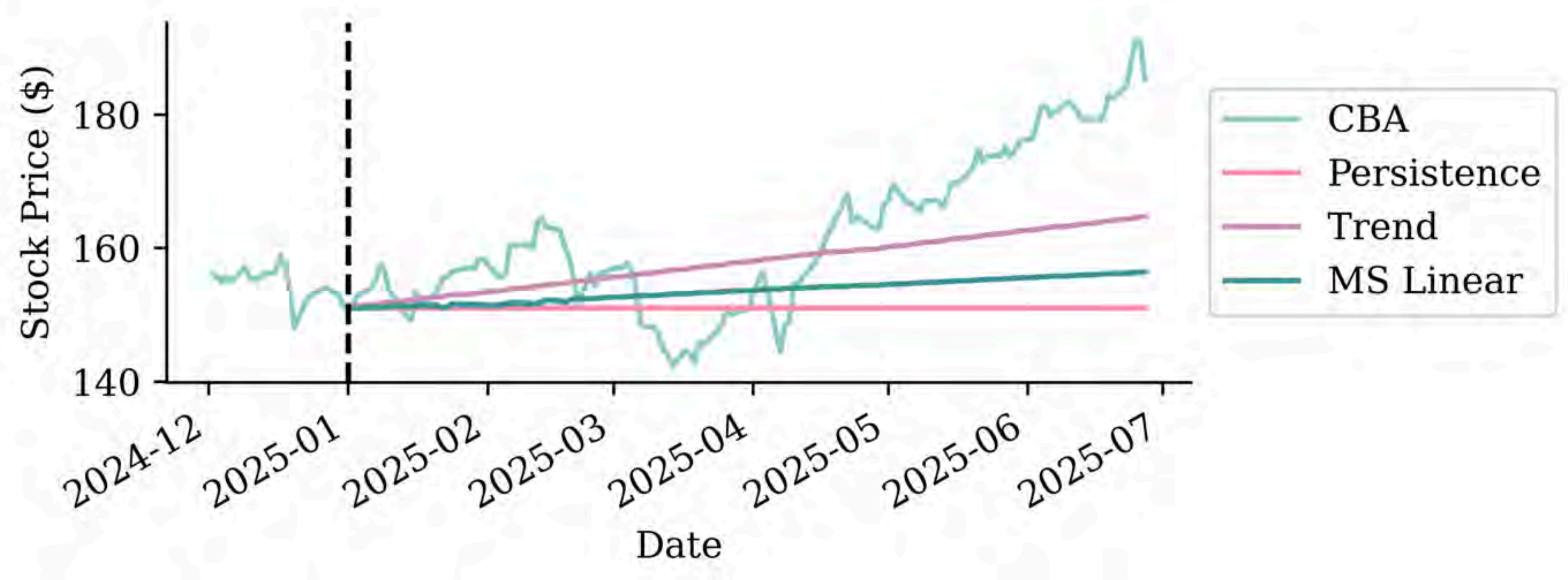
$$\hat{y}_{t+k} = \beta_0 + \beta_1 \hat{y}_{t+k-1} + \beta_2 \hat{y}_{t+k-2} + \ldots + \beta_n \hat{y}_{t+k-n}$$

Autoregressive forecasting function

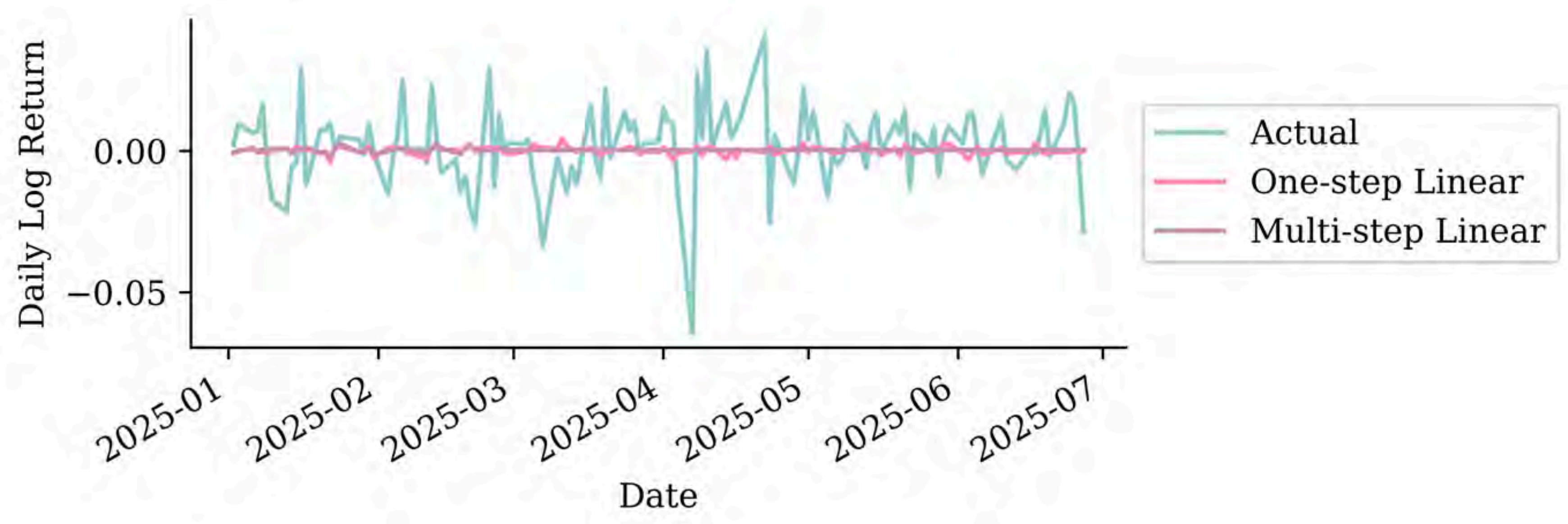
```python
def autoregressive_forecast(model, X_val, suppress=False):
    """
    Generate a multi-step forecast using the given model.
    """
    multi_step = pd.Series(index=X_val.index, name="Multi Step")

    # Initialize the input data for forecasting
    input_data = X_val.iloc[0].values.reshape(1, -1)

    for i in range(len(multi_step)):
        # Ensure input_data has the correct feature names
        input_df = pd.DataFrame(input_data, columns=X_val.columns)
        if suppress:
            next_value = model.predict(input_df, verbose=0)
        else:
            next_value = model.predict(input_df)

        multi_step.iloc[i] = next_value.flatten()[0]

        # Append that prediction to the input for the next forecast
        if i + 1 < len(multi_step):
            input_data = np.append(input_data[:, 1:], next_value).reshape(1, -1)

    return multi_step
```

Look at the autoregressive linear forecasts

```python
lr_forecast = autoregressive_forecast(lr, X_test)
lr_multi_step_prices = log_to_price(lr_forecast, last_price)
```



Compare log return forecasts



Metrics

One-step-ahead forecasts:

```
linear_mse, shifted_mse
```

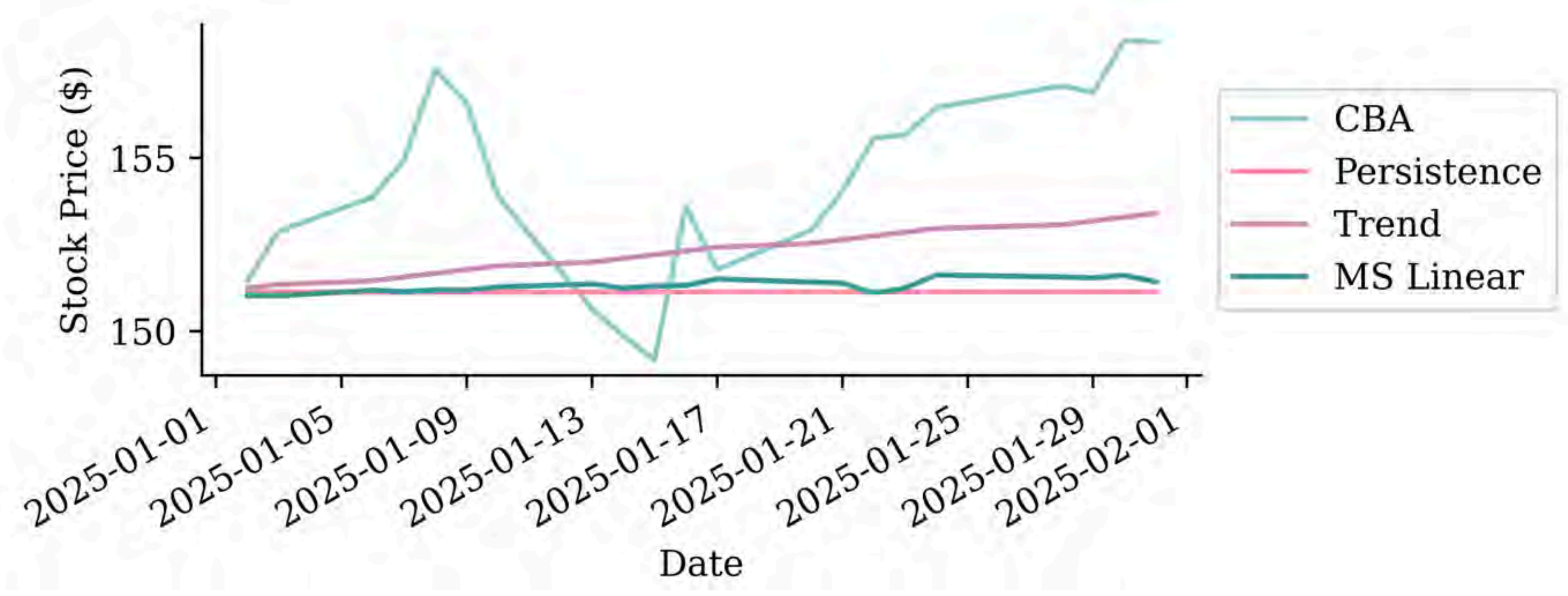
```
(5.1366640383449536, 5.06190042634739)
```

Multi-step-ahead forecasts:

```
multi_step_linear_mse = mean_squared_error(stock.loc["2025":, "CBA"], lr_multi_step_price
persistence_mse, trend_mse, multi_step_linear_mse
```

```
(254.04075100411256, 99.28717915684173, 181.11397713761005)
```

Prefer only short windows



> “It’s tough to make predictions, especially about the future.”

Yogi Berra

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- **Neural network forecasts**
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- Predicting Multiple Time Series

Simple feedforward neural network

```python
model = Sequential([
        Rescaling(1/0.02),
        Dense(32, activation="leaky_relu"),
        Dense(1)])  # Linear activation for log returns
model.compile(optimizer="adam", loss="mean_absolute_error")
```

```python
if Path("models/aus_fin_fnn_model.keras").exists():
    model = keras.models.load_model("models/aus_fin_fnn_model.keras")
else:
    es = EarlyStopping(patience=15, restore_best_weights=True)
    model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=500,
        callbacks=[es], verbose=0)
    model.save("models/aus_fin_fnn_model.keras")

model.summary()
```

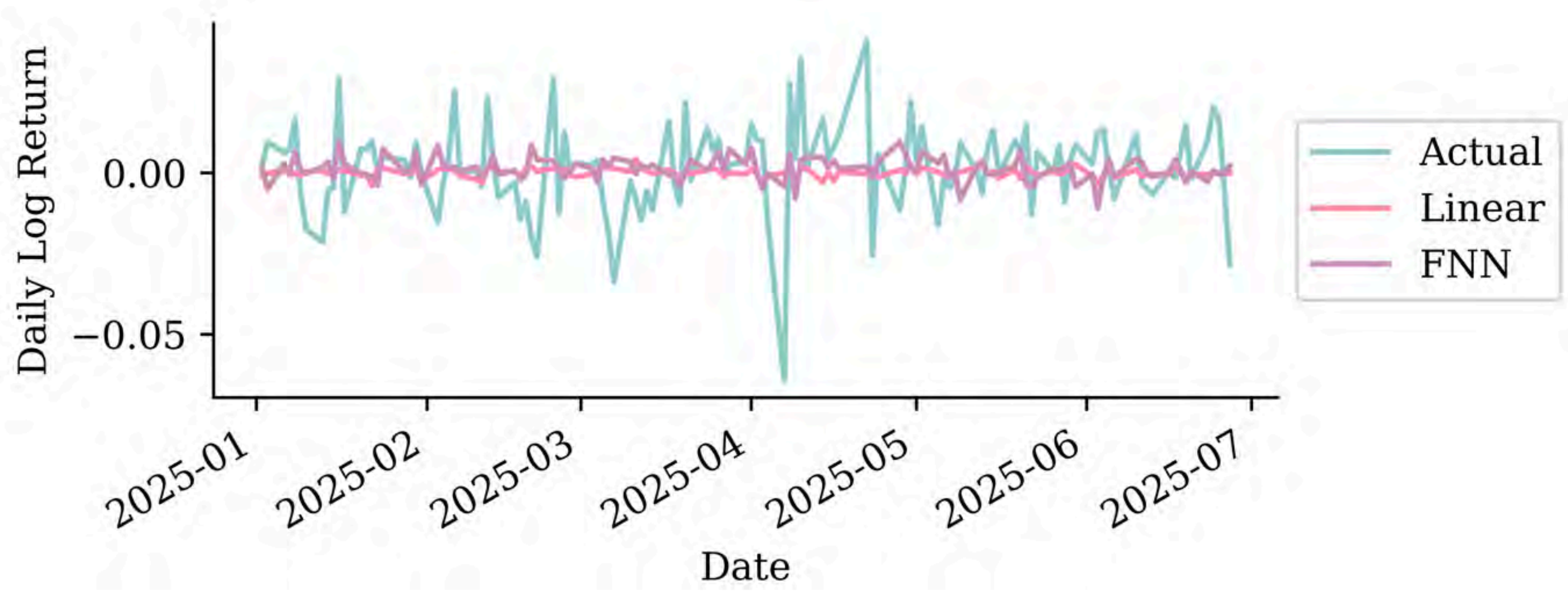
Model: "sequential"

Layer (type)	Output Shape	Param #
rescaling (Rescaling)	(32, 40)	0
dense (Dense)	(32, 32)	1,312
dense_1 (Dense)	(32, 1)	33

Total params: 4,037 (15.77 KB)
Trainable params: 1,345 (5.25 KB)

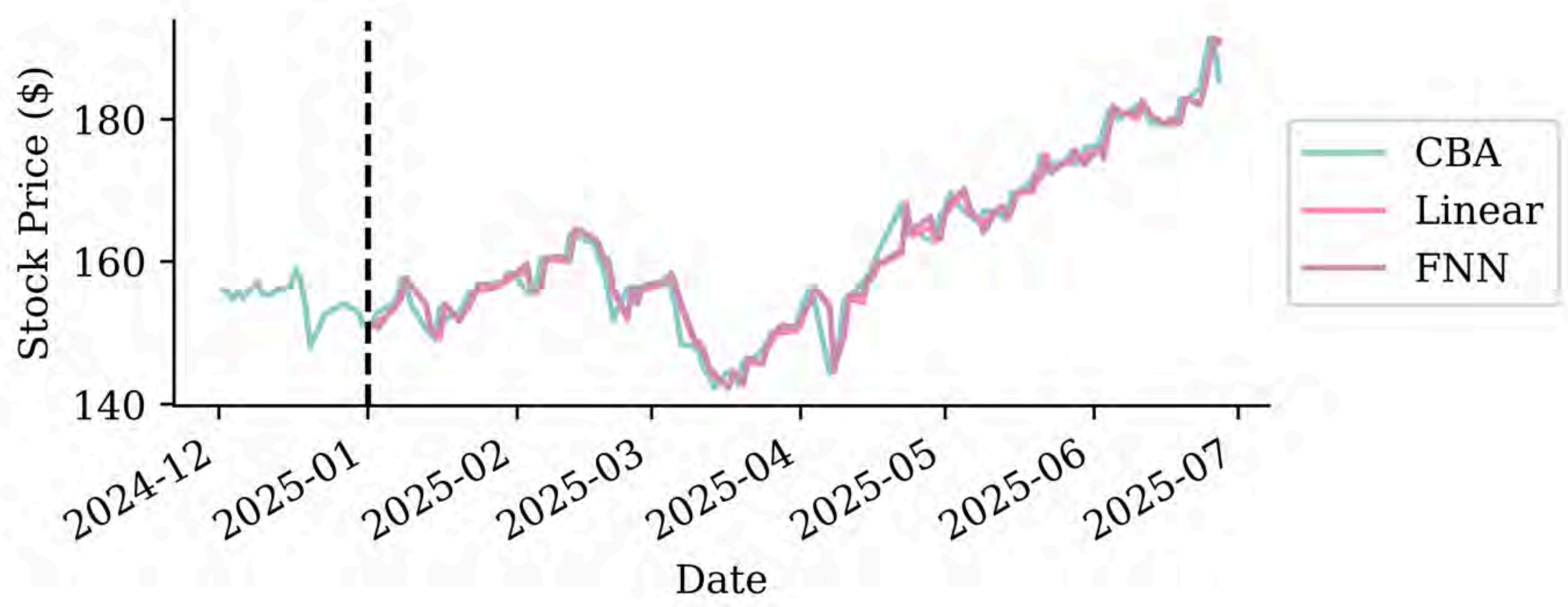
Forecast and plot

```python
y_pred = model.predict(X_test, verbose=0)
```

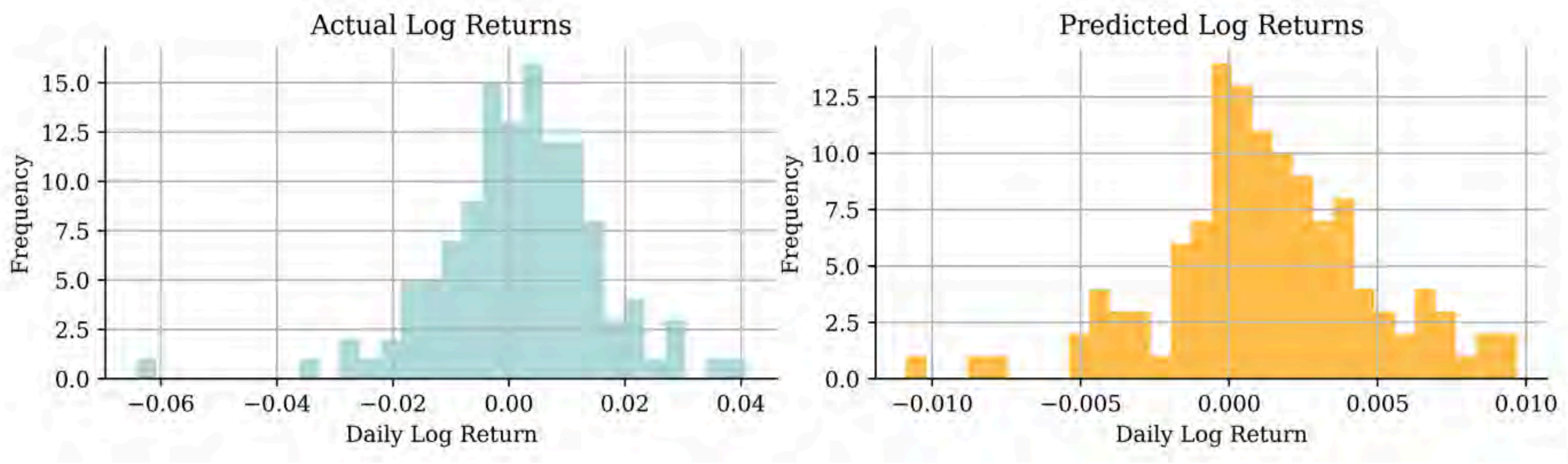


Plot forecasts in raw price space

```python
fnn_log_forecasts = pd.Series(y_pred.flatten(), index=y_test.index)
fnn_price_forecasts = prev_price * np.exp(fnn_log_forecasts)
```

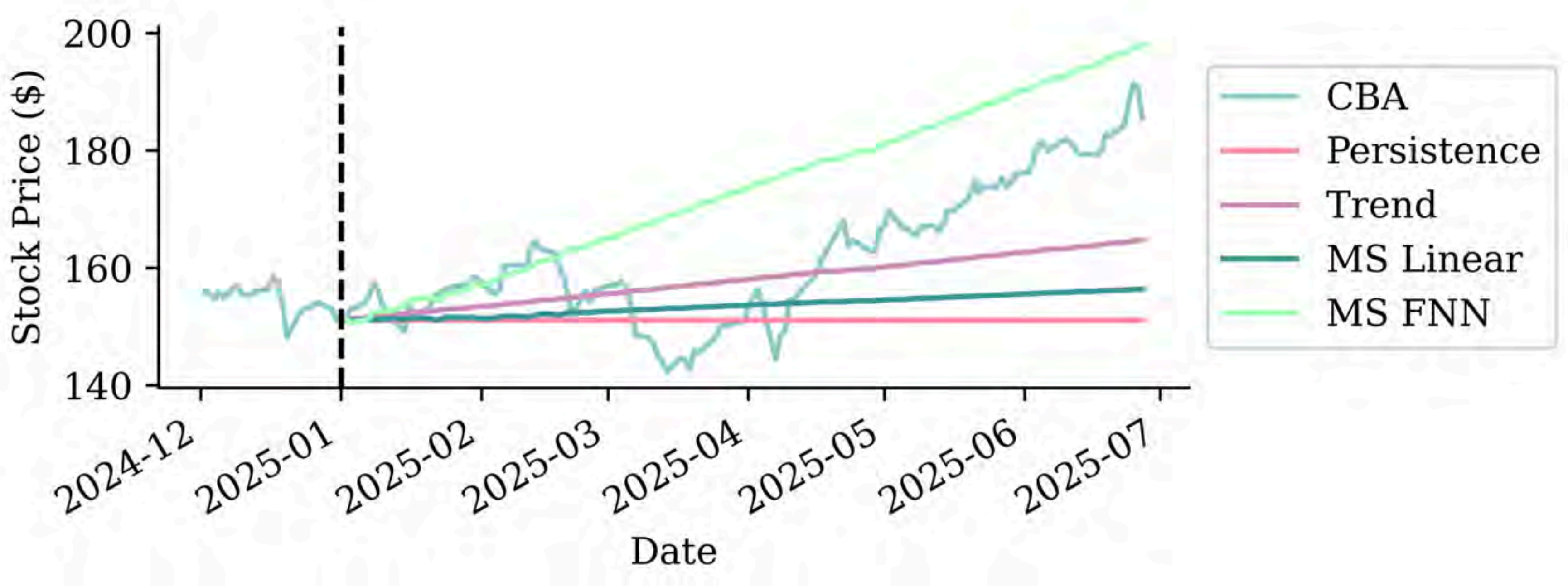


Distribution of log return predictions

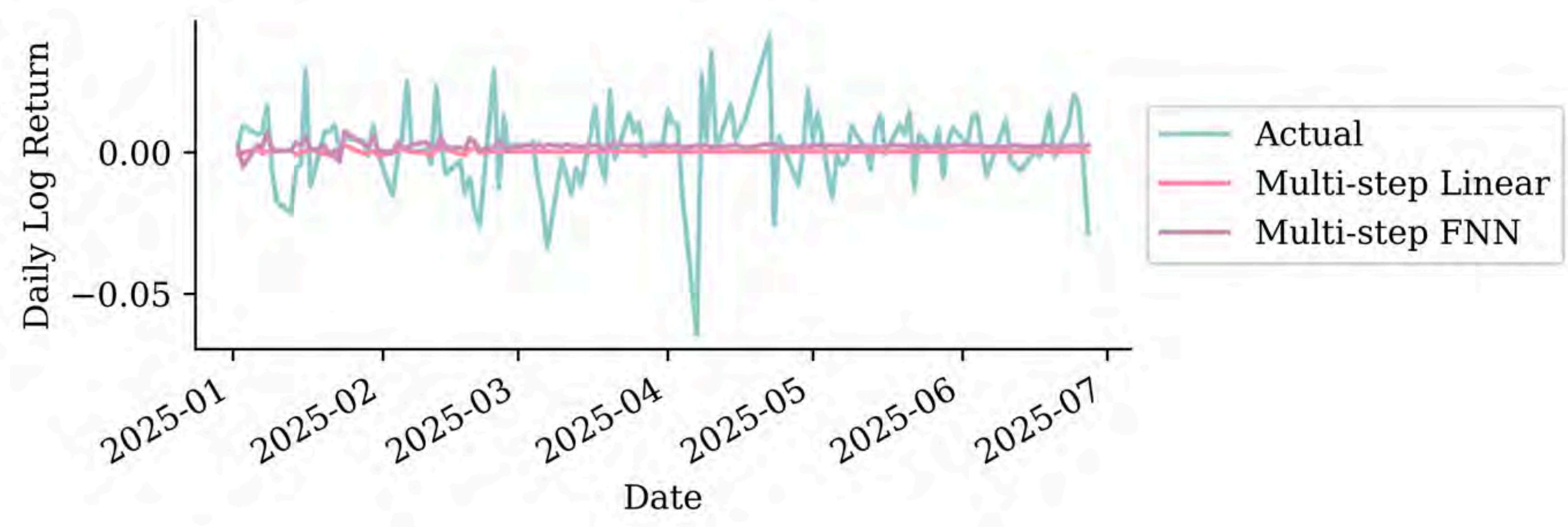


Autoregressive forecasts

```
fnn_forecast = autoregressive_forecast(model, X_test, True)
fnn_multi_step_prices = log_to_price(fnn_forecast, last_price)
```



Compare multi-step log return forecasts



Metrics

One-step-ahead forecasts (MSE on raw prices):

```
1  nn_mse = mean_squared_error(stock.loc["2025":, "CBA"], fnn_price_forecasts.loc["2025":])
2  linear_mse, nn_mse
```

```
(5.136664038344953б, 5.249813642619517)
```

Multi-step-ahead forecasts (MSE on raw prices):

```
1  multi_step_fnn_mse = mean_squared_error(stock.loc["2025":, "CBA"], fnn_multi_step_prices.
2  persistence_mse, trend_mse, multi_step_linear_mse, multi_step_fnn_mse
```

```
(254.04075100411256, 99.28717915684173, 181.11397713761005, 213.40476751283З)
```

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- **Recurrent Neural Networks**
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- Predicting Multiple Time Series

Basic facts of RNNs

- A recurrent neural network is a type of neural network that is designed to process sequences of data (e.g. time series, sentences).
- A recurrent neural network is any network that contains a recurrent layer.
- A recurrent layer is a layer that processes data in a sequence.
- An RNN can have one or more recurrent layers.
- Weights are shared over time; this allows the model to be used on arbitrary-length sequences.

Applications

- *Forecasting*: revenue forecast, weather forecast, predict disease rate from medical history, etc.
- *Classification*: given a time series of the activities of a visitor on a website, classify whether the visitor is a bot or a human.
- *Event detection*: given a continuous data stream, identify the occurrence of a specific event. Example: Detect utterances like "Hey Alexa" from an audio stream.
- *Anomaly detection*: given a continuous data stream, detect anything unusual happening. Example: Detect unusual activity on the corporate network.

Origin of the name of RNNs

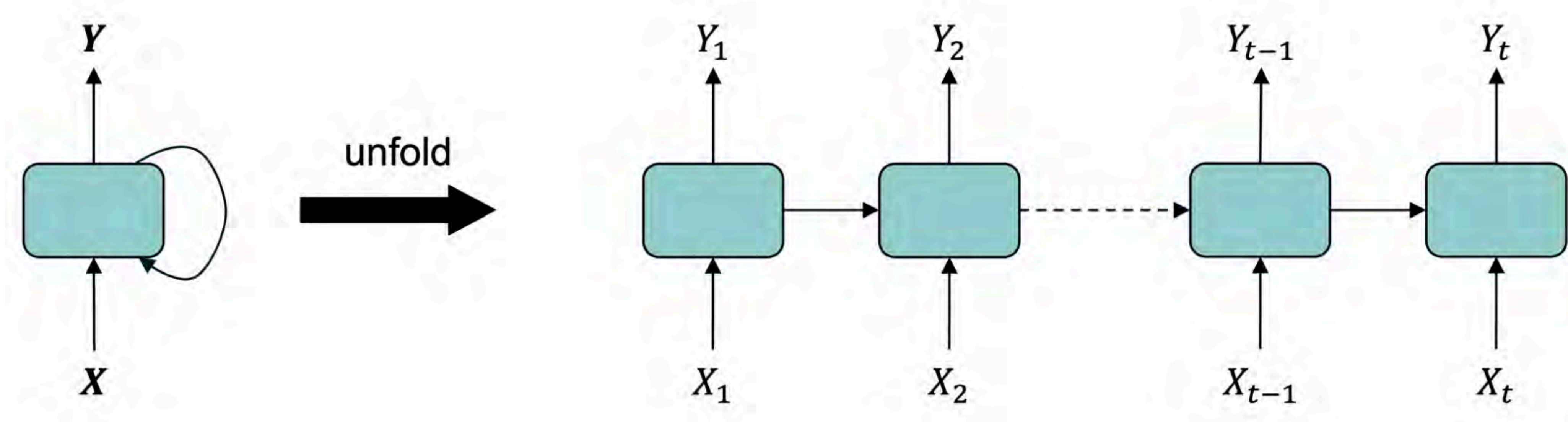
> A recurrence relation is an equation that expresses each element of a sequence as a function of the preceding ones. More precisely, in the case where only the immediately preceding element is involved, a recurrence relation has the form
>
> $$u_n = \psi(n, u_{n-1}) \quad \text{for} \quad n > 0.$$

Example: Factorial $n! = n(n-1)!$ for $n > 0$ given $0! = 1$.

Source: Wikipedia, Recurrence relation.

Diagram of an RNN cell

The RNN processes each data in the sequence one by one, while keeping memory of what came before.



Schematic of a recurrent neural network. E.g. SimpleRNN, LSTM, or GRU.

Source: Melissa Renard (2025)

A SimpleRNN cell

Diagram of a SimpleRNN cell.

All the outputs before the final one are often discarded.

Source: Christopher Olah (2015), Understanding LSTM Networks, Colah's Blog.

LSTM internals

h_{t-1} h_t h_{t+1}

A

x + tanh x x

σ σ tanh σ

A

X_{t-1} X_t X_{t+1}

Neural Network Layer

Pointwise Operation

Vector Transfer

Concatenate

Copy

Source: Christopher Olah (2015), Understanding LSTM Networks, Colah's Blog.

GRU internals

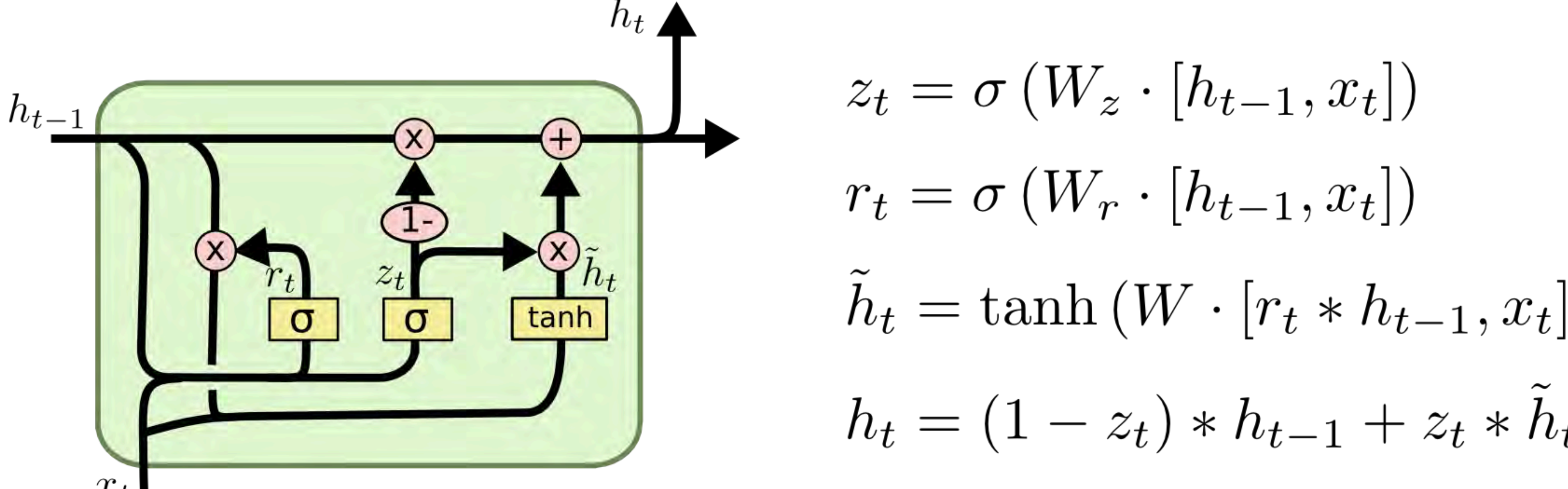


Diagram of a GRU cell.

Source: Christopher Olah (2015), Understanding LSTM Networks, Colah's Blog.

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- **Stock prediction with recurrent networks**
- Internals of the SimpleRNN
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- Predicting Multiple Time Series

SimpleRNN

```python
from keras.layers import SimpleRNN, Reshape
model = Sequential([
        Rescaling(1/0.02),
        Reshape((-1, 1)),
        SimpleRNN(64, activation="tanh"),
        Dense(1)])  # Linear activation for log returns
model.compile(optimizer="adam", loss="mean_absolute_error")
```

```python
es = EarlyStopping(patience=15, restore_best_weights=True)
model.fit(X_train, y_train, validation_data=(X_val, y_val),
    epochs=500, callbacks=[es], verbose=0)
model.summary()
```

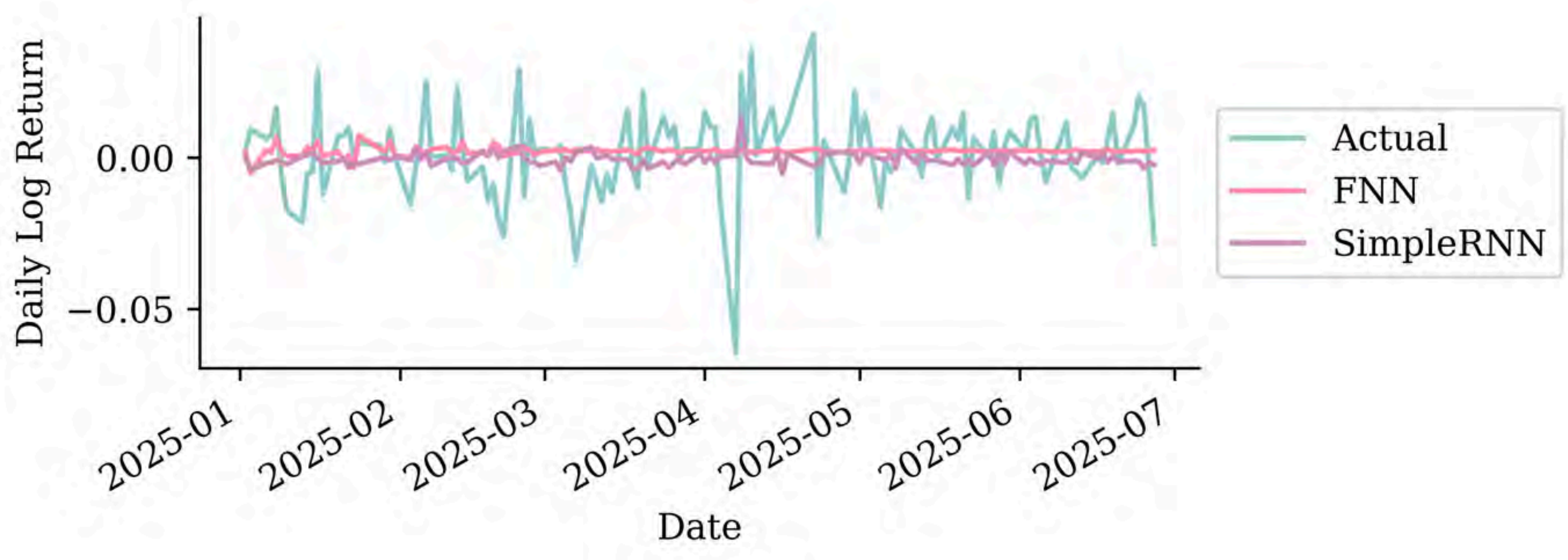
Model: "sequential_1"

Layer (type)	Output Shape	Param #
rescaling_1 (Rescaling)	(32, 40)	0
reshape (Reshape)	(32, 40, 1)	0
simple_rnn (SimpleRNN)	(32, 64)	4,224
dense_2 (Dense)	(32, 1)	65

Total params: 12,869 (50.27 KB)
Trainable params: 4,289 (16.75 KB)
Non-trainable params: 0 (0.00 B)

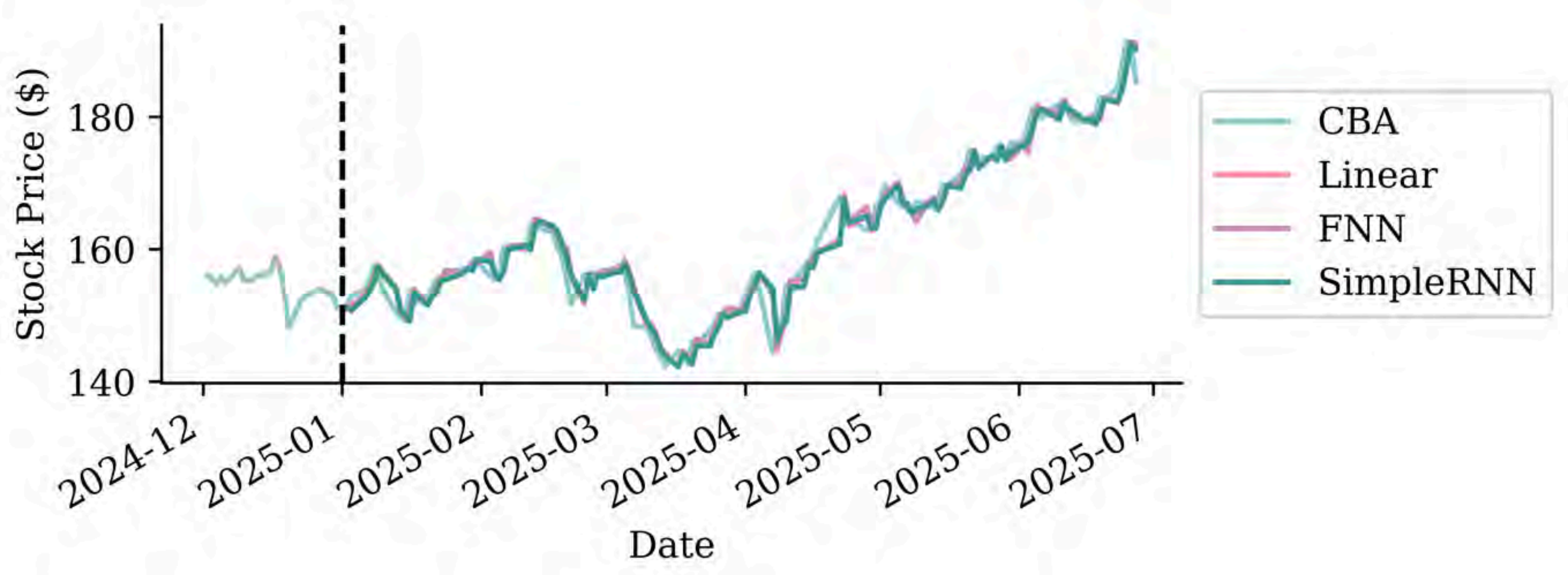
Forecast and plot

```
1 y_pred = model.predict(X_test.to_numpy(), verbose=0)
```



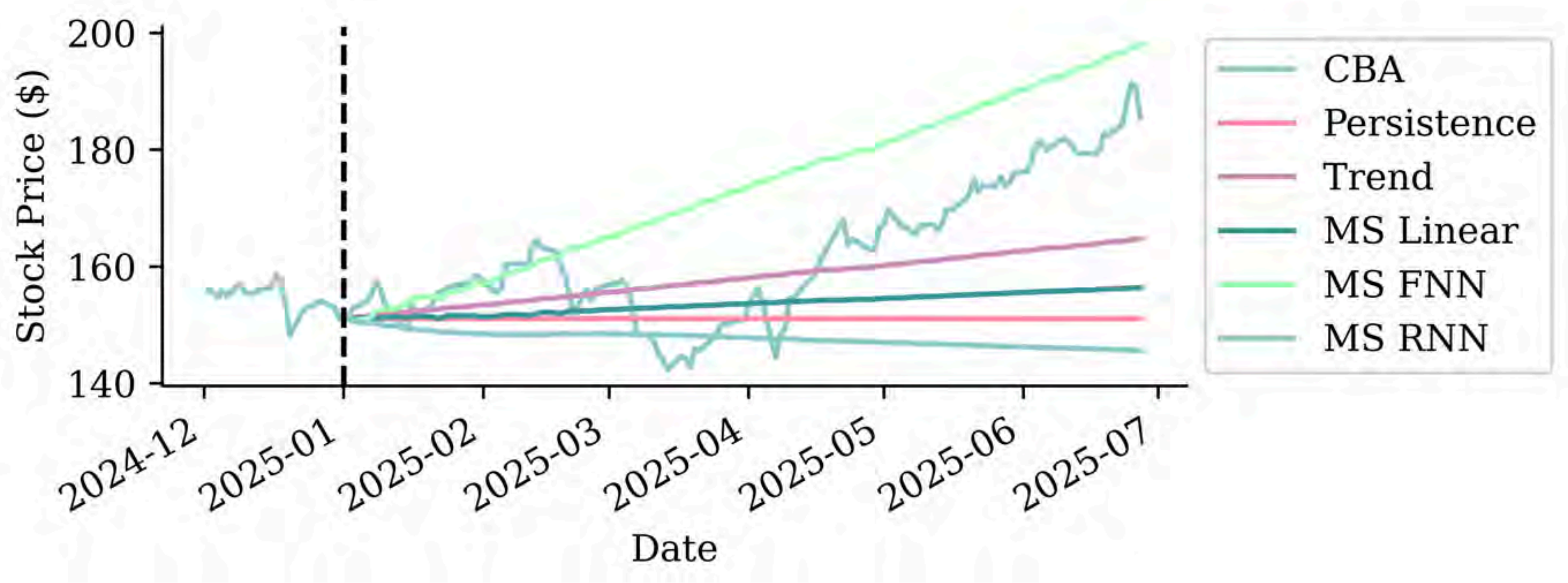
Plot forecasts in raw price space

```python
rnn_log_forecasts = pd.Series(y_pred.flatten(), index=y_test.index)
rnn_price_forecasts = prev_price * np.exp(rnn_log_forecasts)
```



Multi-step forecasts

```python
rnn_forecast = autoregressive_forecast(model, X_test, True)
rnn_multi_step_prices = log_to_price(rnn_forecast, last_price)
```



Metrics

One-step-ahead forecasts (MSE on raw prices):

```
1  rnn_mse = mean_squared_error(stock.loc["2025":, "CBA"], rnn_price_forecasts.loc["2025":])
2  linear_mse, nn_mse, rnn_mse
```

```
(5.1366640383449536, 5.249813642619517, 5.279663252855173)
```

Multi-step-ahead forecasts (MSE on raw prices):

```
1  multi_step_rnn_mse = mean_squared_error(stock.loc["2025":, "CBA"], rnn_multi_step_prices.
2  persistence_mse, trend_mse, multi_step_linear_mse, multi_step_fnn_mse, multi_step_rnn_mse
```

```
(254.04075100411256,
 99.28717915684173,
 181.11397713761005,
 213.404767512833,
 363.6276114811913)
```

GRU

```python
from keras.layers import GRU

model = Sequential([
        Rescaling(1/0.02),
        Reshape((-1, 1)),
        GRU(16, activation="tanh"),
        Dense(1)])  # Linear activation for log returns
model.compile(optimizer="adam", loss="mean_absolute_error")
```

```python
es = EarlyStopping(patience=15, restore_best_weights=True)
model.fit(X_train, y_train, validation_data=(X_val, y_val),
    epochs=500, callbacks=[es], verbose=0)
model.summary()
```

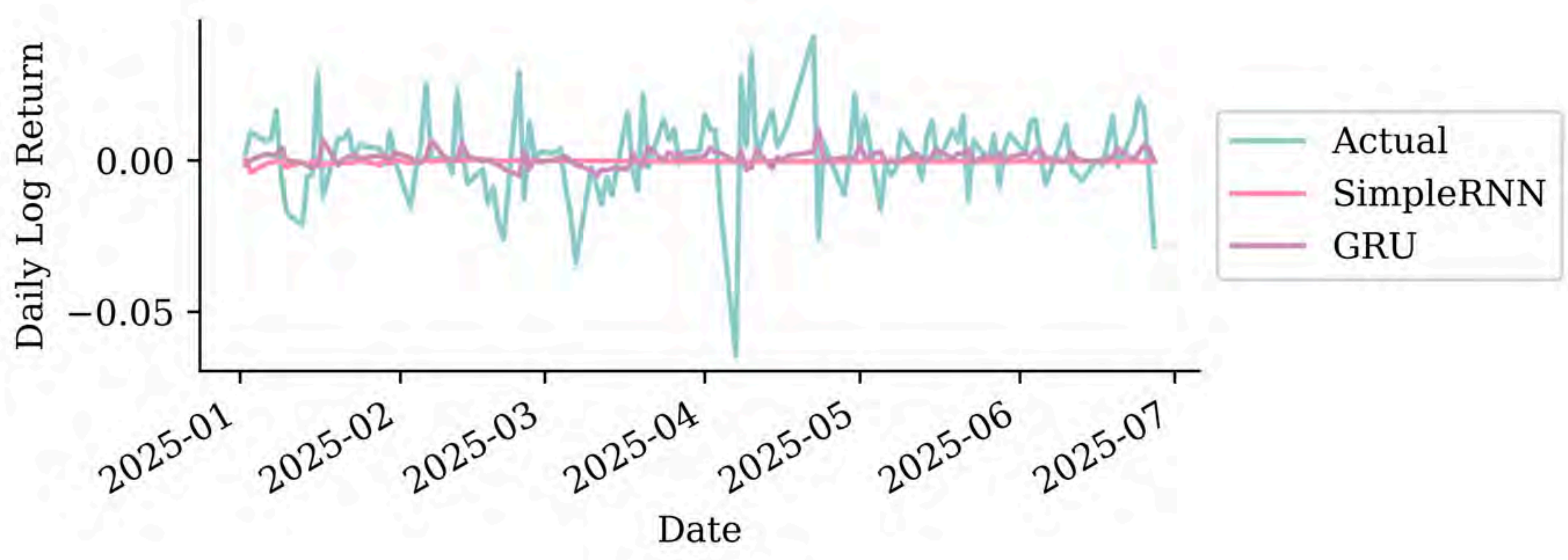
Model: "sequential_2"

Layer (type)	Output Shape	Param #
rescaling_2 (Rescaling)	(32, 40)	0
reshape_1 (Reshape)	(32, 40, 1)	0
gru (GRU)	(32, 16)	912
dense_3 (Dense)	(32, 1)	17

Total params: 2,789 (10.89 KB)
Trainable params: 929 (3.63 KB)

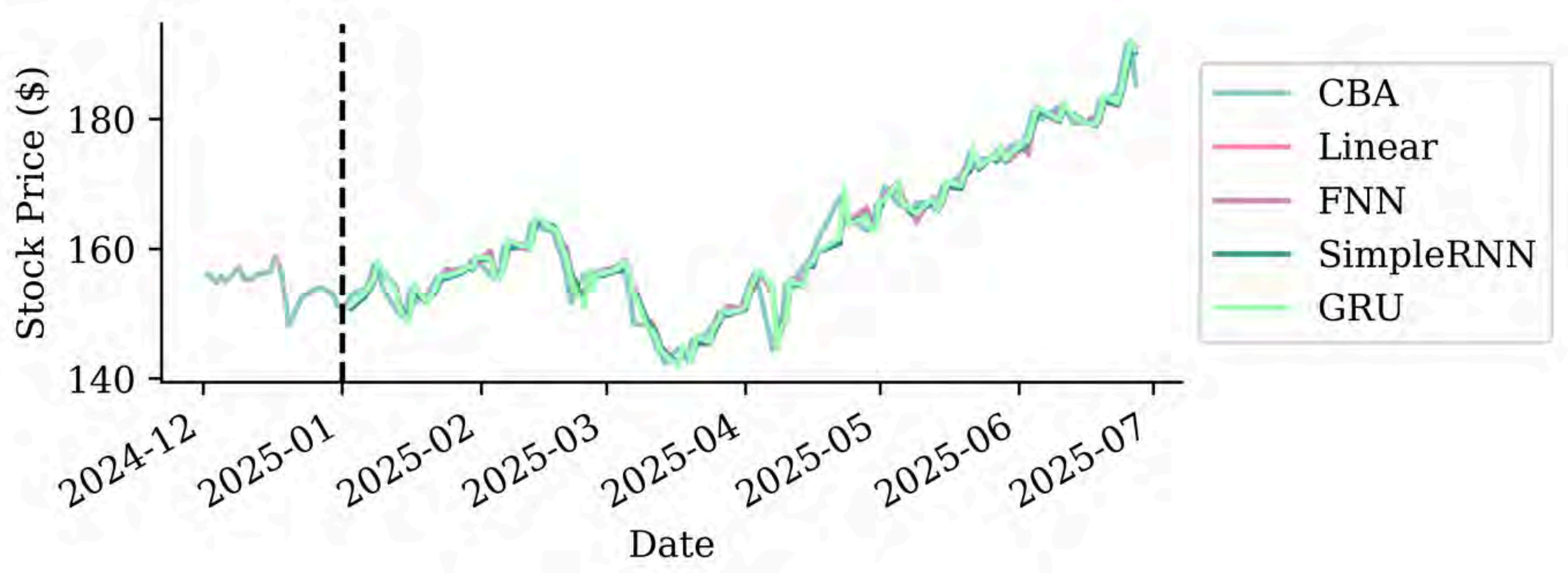
Forecast and plot

```python
y_pred = model.predict(X_test, verbose=0)
```



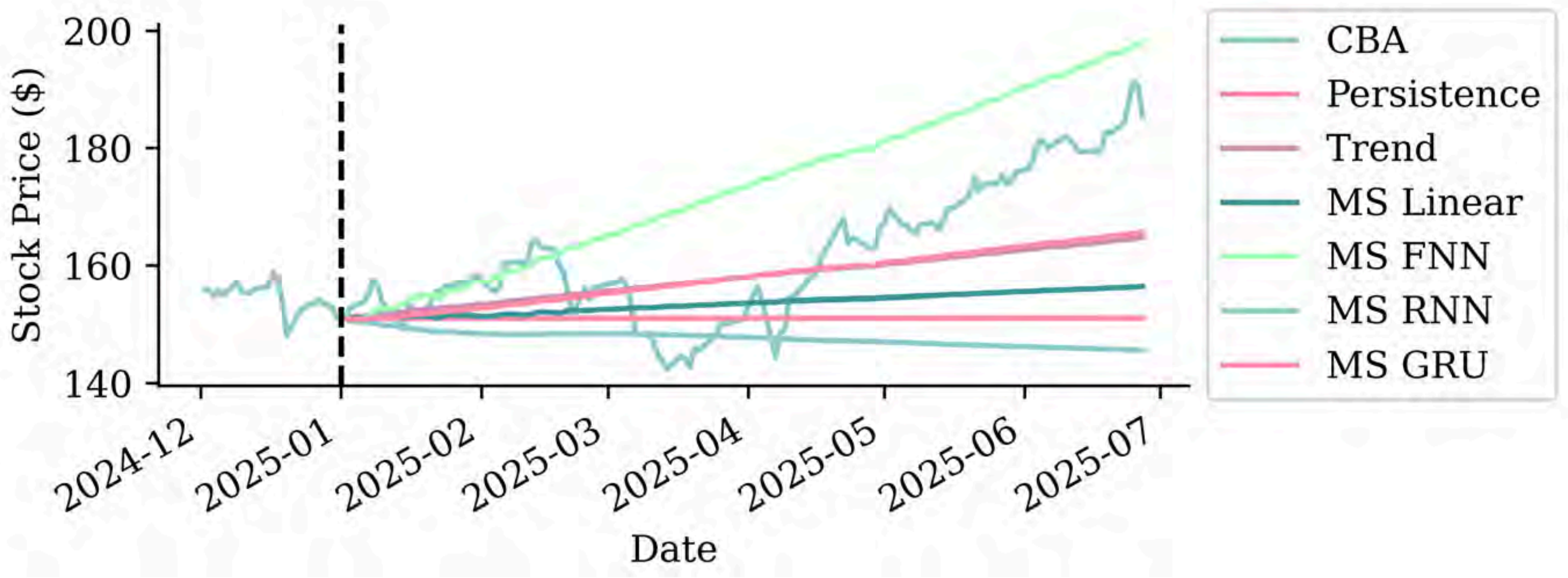
Plot forecasts in raw price space

```python
gru_log_forecasts = pd.Series(y_pred.flatten(), index=y_test.index)
gru_price_forecasts = prev_price * np.exp(gru_log_forecasts)
```

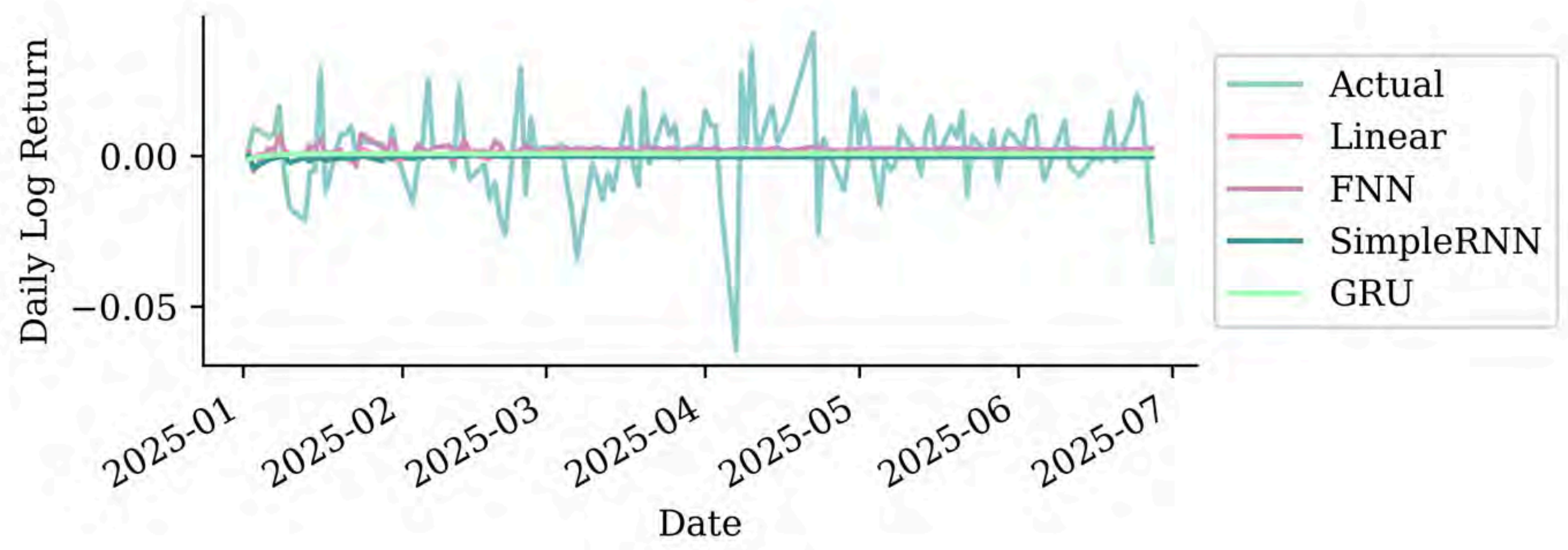


Multi-step forecasts

```python
gru_forecast = autoregressive_forecast(model, X_test, True)
gru_multi_step_prices = log_to_price(gru_forecast, last_price)
```



All multi-step log return forecasts



Metrics

One-step-ahead forecasts (MSE on raw prices):

```
1  gru_mse = mean_squared_error(stock.loc["2025":, "CBA"], gru_price_forecasts.loc["2025":])
2  linear_mse, nn_mse, rnn_mse, gru_mse
```

```
(5.1366640383449536, 5.249813642619517, 5.279663252855173, 5.167950295726733)
```

Multi-step-ahead forecasts (MSE on raw prices):

```
1  multi_step_gru_mse = mean_squared_error(stock.loc["2025":, "CBA"], gru_multi_step_prices.
2  persistence_mse, trend_mse, multi_step_linear_mse, multi_step_fnn_mse, multi_step_rnn_mse
```

```
(254.04075100411256,
 99.28717915684173,
 181.11397713761005,
 213.40476751283,
 363.6276114811913,
 94.02476396370041)
```

Summary of all model performance

One-step forecasts

	MSE
Shifted	5.061900
Linear	5.136664
FNN	5.249814
SimpleRNN	5.279663
GRU	5.167950

Multi-step forecasts

	MSE
Persistence	254.040751
Trend	99.287179
Linear	181.113977
FNN	213.404768
SimpleRNN	363.627611
GRU	94.024764

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- **Internals of the SimpleRNN**
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- Predicting Multiple Time Series

The rank of a time series

Say we had n observations of a time series $x_1, x_2, \ldots, x_n$.

This $\boldsymbol{x} = (x_1, \ldots, x_n)$ would have shape $(n,)$ & rank 1.

If instead we had a batch of b time series'

$$\boldsymbol{X} = \begin{pmatrix} x_7 & x_8 & \cdots & x_{7+n-1} \\ x_2 & x_3 & \cdots & x_{2+n-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_3 & x_4 & \cdots & x_{3+n-1} \end{pmatrix},$$

the batch $\boldsymbol{X}$ would have shape (b, n) & rank 2.

Multivariate time series

t	x	y
0	x_0	y_0
1	x_1	y_1
2	x_2	y_2
3	x_3	y_3

Say n observations of the m time series, would be a shape (n, m) matrix of rank 2.

In Keras, a batch of b of these time series has shape (b, n, m) and has rank 3.

Note

Use $\boldsymbol{x}_t \in \mathbb{R}^{1 \times m}$ to denote the vector of all time series at time t. Here, $\boldsymbol{x}_t = (x_t, y_t)$.

SimpleRNN

Say each prediction is a vector of size d, so $\boldsymbol{y}_t \in \mathbb{R}^{1 \times d}$.

Then the main equation of a SimpleRNN, given $\boldsymbol{y}_0 = \boldsymbol{0}$, is

$$\boldsymbol{y}_t = \psi\big(\boldsymbol{x}_t \boldsymbol{W}_x + \boldsymbol{y}_{t-1} \boldsymbol{W}_y + \boldsymbol{b}\big).$$

Here,

$$\boldsymbol{x}_t \in \mathbb{R}^{1 \times m}, \boldsymbol{W}_x \in \mathbb{R}^{m \times d},$$
$$\boldsymbol{y}_{t-1} \in \mathbb{R}^{1 \times d}, \boldsymbol{W}_y \in \mathbb{R}^{d \times d}, \text{ and } \boldsymbol{b} \in \mathbb{R}^{d}.$$

SimpleRNN (in batches)

Say we operate on batches of size b, then $\boldsymbol{Y}_t \in \mathbb{R}^{b \times d}$.

The main equation of a SimpleRNN, given $\boldsymbol{Y}_0 = \boldsymbol{0}$, is

$$\boldsymbol{Y}_t = \psi\big(\boldsymbol{X}_t \boldsymbol{W}_x + \boldsymbol{Y}_{t-1} \boldsymbol{W}_y + \boldsymbol{b}\big).$$

Here,

$$\boldsymbol{X}_t \in \mathbb{R}^{b \times m}, \boldsymbol{W}_x \in \mathbb{R}^{m \times d},$$
$$\boldsymbol{Y}_{t-1} \in \mathbb{R}^{b \times d}, \boldsymbol{W}_y \in \mathbb{R}^{d \times d}, \text{ and } \boldsymbol{b} \in \mathbb{R}^{d}.$$

Remember, $\boldsymbol{X} \in \mathbb{R}^{b \times n \times m}$, $\boldsymbol{Y} \in \mathbb{R}^{b \times d}$, and $\boldsymbol{X}_t$ is equivalent to `X[:, t, :]`.

Simple Keras demo

```
num_obs = 4
num_time_steps = 3
num_time_series = 2

X = (
    np.arange(num_obs * num_time_steps * num_time_series)
    .astype(np.float32)
    .reshape([num_obs, num_time_steps, num_time_series])
)

output_size = 1
y = np.array([0, 0, 1, 1])
```

```
X[:2]
```

```
array([[[ 0.,  1.],
        [ 2.,  3.],
        [ 4.,  5.]],

       [[ 6.,  7.],
        [ 8.,  9.],
        [10., 11.]]], dtype=float32)
```

```
X[2:]
```

```
array([[[12., 13.],
        [14., 15.],
        [16., 17.]],

       [[18., 19.],
        [20., 21.],
        [22., 23.]]], dtype=float32)
```

Keras' SimpleRNN

As usual, the `SimpleRNN` is just a layer in Keras.

```
from keras.layers import SimpleRNN

random.seed(1234)
model = Sequential([SimpleRNN(output_size, activation="sigmoid")])
model.compile(loss="binary_crossentropy", metrics=["accuracy"])

hist = model.fit(X, y, epochs=500, verbose=False)
model.evaluate(X, y, verbose=False)
```

```
[8.05906867980957, 0.5]
```

The predicted probabilities on the training set are:

```
model.predict(X, verbose=0)
```

```
array([[8.56e-05],
       [2.25e-10],
       [5.98e-16],
       [1.59e-21]], dtype=float32)
```

SimpleRNN weights

```
model.get_weights()
```

```
[array([[-1.47],
        [-0.67]], dtype=float32),
 array([[0.99]], dtype=float32),
 array([-0.14], dtype=float32)]
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


W_x, W_y, b = model.get_weights()

Y = np.zeros((num_obs, output_size), dtype=np.float32)
for t in range(num_time_steps):
    X_t = X[:, t, :]
    z = X_t @ W_x + Y @ W_y + b
    Y = sigmoid(z)

Y
```

```
array([[8.56e-05],
       [2.25e-10],
       [5.98e-16],
       [1.59e-21]], dtype=float32)
```

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- **Other recurrent network variants**
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- Predicting Multiple Time Series

Input and output sequences

(a) Many to many

(b) Many to one

(c) One to Many

(d) Encoder-decoder

Categories of recurrent neural networks: sequence to sequence, sequence to vector, vector to sequence, encoder-decoder network.

Source: Melissa Renard (2025)

Input and output sequences

- Sequence to sequence: Useful for predicting time series such as using prices over the last N days to output the prices shifted one day into the future (i.e. from $N - 1$ days ago to tomorrow.)
- Sequence to vector: ignore all outputs in the previous time steps except for the last one. Example: give a sentiment score to a sequence of words corresponding to a movie review.

Input and output sequences

- Vector to sequence: feed the network the same input vector over and over at each time step and let it output a sequence. Example: given that the input is an image, find a caption for it. The image is treated as an input vector (pixels in an image do not follow a sequence). The caption is a sequence of textual description of the image. A dataset containing images and their descriptions is the input of the RNN.
- The Encoder-Decoder: The encoder is a sequence-to-vector network. The decoder is a vector-to-sequence network. Example: Feed the network a sequence in one language. Use the encoder to convert the sentence into a single vector representation. The decoder decodes this vector into the translation of the sentence in another language.

Recurrent layers can be stacked.

Y

Stack 3

Stack 2

Stack 1

X

unfold

Y_1 Y_2 Y_{t-1} Y_t

X_1 X_2 X_{t-1} X_t

Deep RNN unrolled through time.

Source: Melissa Renard (2025)

Beyond RNNs

RNNs are **foundational** and the concepts transfer everywhere, but in practice they have been largely overtaken by newer architectures:

- **Attention** / **Transformers** (Vaswani et al., 2017): instead of processing a sequence step-by-step, attention lets every position look at every other position simultaneously. This avoids the vanishing gradient problem over long sequences and enables massive parallelism during training.
- **State-space models** (e.g. Mamba, 2023): process sequences in linear time rather than the quadratic cost of full attention, while still capturing long-range dependencies.

Why learn RNNs then?

- The **hidden state** concept carries directly into transformers (which use it differently) and state-space models
- **Gating** (LSTM/GRU) was a key insight that influenced later architectures
- RNNs remain a strong, simple baseline for short sequences and small datasets
- Understanding RNNs makes it easier to appreciate *why* attention was such a breakthrough

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- Other recurrent network variants
- **CoreLogic Hedonic Home Value Index**
- Predicting Sydney House Prices
- Predicting Multiple Time Series

Australian House Price Indices

200
150
100
50
1994 1999 2004 2009 2014 2019
Date

Note

I apologise in advance for not being able to share this dataset with anyone (it is not mine to share).

UNSW
SYDNEY

Percentage changes

```python
changes = house_prices.pct_change().dropna()
changes.round(2)
```

Date	Brisbane	East_Bris	North_Bris	West_Bris	Melbourne	North_Syd	Sydney
1990-02-28	0.03	-0.01	0.01	0.01	0.00	-0.00	-0.02
1990-03-31	0.01	0.03	0.01	0.01	0.02	-0.00	0.03
1990-04-30	0.02	0.02	0.01	-0.00	0.01	0.03	0.04
...	...	...	...	...	...	...	...
2021-03-31	0.04	0.04	0.03	0.04	0.02	0.05	0.05
2021-04-30	0.03	0.01	0.01	-0.00	0.01	0.02	0.02
2021-05-31	0.03	0.03	0.03	0.03	0.03	0.02	0.04

376 rows × 7 columns

Percentage changes

```
changes.plot();
```

The size of the changes

```
1  changes.mean()
```

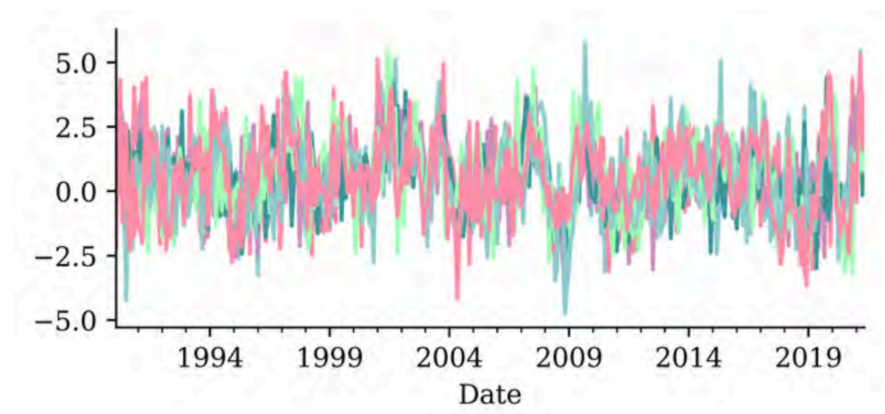
```
Brisbane       0.005496
East_Bris      0.005416
North_Bris     0.005024
West_Bris      0.004842
Melbourne      0.005677
North_Syd      0.004819
Sydney         0.005526
dtype: float64
```

```
1  changes *= 100
```

```
1  changes.mean()
```

```
Brisbane       0.549605
East_Bris      0.541562
North_Bris     0.502390
West_Bris      0.484204
Melbourne      0.567700
North_Syd      0.481863
Sydney         0.552641
dtype: float64
```

```
1  changes.plot(legend=False);
```



Split *without* shuffling

```python
num_train = int(0.6 * len(changes))
num_val = int(0.2 * len(changes))
num_test = len(changes) - num_train - num_val
print(f"# Train: {num_train}, # Val: {num_val}, # Test: {num_test}")
```

```
# Train: 225, # Val: 75, # Test: 76
```

Subsequences of a time series

We can use numpy to convert a time series into subsequences/chunks using a sliding window.

```
integers = np.arange(10)
seq_len = 3

X = np.stack([integers[i : i + seq_len] for i in range(len(integers) - seq_len)])
y = integers[seq_len:]

for i in range(len(X)):
    print(X[i].tolist(), int(y[i]))
```

```
[0, 1, 2] 3
[1, 2, 3] 4
[2, 3, 4] 5
[3, 4, 5] 6
[4, 5, 6] 7
[5, 6, 7] 8
[6, 7, 8] 9
```

Source: Code snippet in Chapter 10 of Chollet.

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- **Predicting Sydney House Prices**
- Predicting Multiple Time Series

Creating input arrays

```python
def make_sequences(data, targets, seq_len, start_index=0, end_index=None):
    data = np.asarray(data, dtype=np.float32)
    targets = np.asarray(targets, dtype=np.float32)
    if end_index is None:
        end_index = len(data)
    n = end_index - start_index - seq_len + 1
    X = np.stack([data[start_index + i : start_index + i + seq_len] for i in range(n)])
    y = targets[start_index : start_index + n]
    return X, y
```

```python
# Num. of input time series.
num_ts = changes.shape[1]

# How many prev. months to use.
seq_length = 6

# Predict the next month ahead.
ahead = 1

# The index of the first target.
delay = seq_length + ahead - 1
```

```python
# Which suburb to predict.
target_suburb = changes["Sydney"]

X_train, y_train = make_sequences(
    changes[:-delay], target_suburb[delay
    seq_length, end_index=num_train,
)
```

```python
X_val, y_val = make_sequences(
    changes[:-delay], target_suburb[delay
    seq_length, start_index=num_train,
    end_index=num_train + num_val,
```

```python
X_test, y_test = make_sequences(
    changes[:-delay], target_suburb[delay
    seq_length, start_index=num_train + r
)
```

Training set shape

```
1 X_train.shape
```

```
(220, 6, 7)
```

```
1 y_train.shape
```

```
(220,)
```

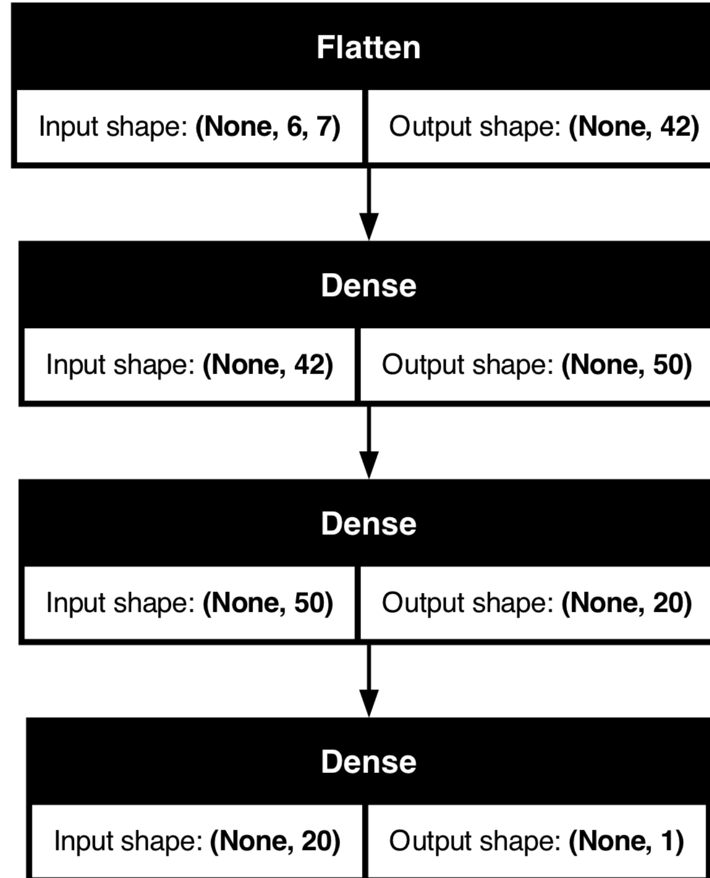
A dense network

```
from keras.layers import Input, Flatten
random.seed(1)
model_dense = Sequential([
    Input((seq_length, num_ts)),
    Flatten(),
    Dense(50, activation="leaky_relu"),
    Dense(20, activation="leaky_relu"),
    Dense(1, activation="linear")
])
model_dense.compile(loss="mse", optimizer="adam")
print(f"This model has {model_dense.count_params()} parameters.")

es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)
%time hist = model_dense.fit(X_train, y_train, epochs=1_000, \
  validation_data=(X_val, y_val), callbacks=[es], verbose=0);
```

```
This model has 3191 parameters.
Epoch 76: early stopping
Restoring model weights from the end of the best epoch: 26.
CPU times: user 1.63 s, sys: 73.2 ms, total: 1.7 s
Wall time: 1.66 s
```

Plot the model

```python
from keras.utils import plot_model

plot_model(model_dense, show_shapes=True)
```



Assess the fits

```python
model_dense.evaluate(X_val, y_val, verbose=0)
```

```
1.3529319763183594
```

Change in HPI (%)
2
0
−2
0 10 20 30 40 50 60 70
Time
Sydney
Dense

A `SimpleRNN` layer

```
random.seed(1)

model_simple = Sequential([
    Input((seq_length, num_ts)),
    SimpleRNN(50),
    Dense(1, activation="linear")
])
model_simple.compile(loss="mse", optimizer="adam")
print(f"This model has {model_simple.count_params()} parameters.")

es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)
%time hist = model_simple.fit(X_train, y_train, epochs=1_000, \
  validation_data=(X_val, y_val), callbacks=[es], verbose=0);
```

```
This model has 2951 parameters.
Epoch 53: early stopping
Restoring model weights from the end of the best epoch: 3.
CPU times: user 2.34 s, sys: 71.4 ms, total: 2.41 s
Wall time: 2.38 s
```

Plot the model

```
1  plot_model(model_simple, show_shapes=True)
```

SimpleRNN

Input shape: **(None, 6, 7)** | Output shape: **(None, 50)**

Dense

Input shape: **(None, 50)** | Output shape: **(None, 1)**

Assess the fits

```
1  model_simple.evaluate(X_val, y_val, verbose=0)
```

```
0.8649981021881104
```

A LSTM layer

```
from keras.layers import LSTM

random.seed(1)

model_lstm = Sequential([
    Input((seq_length, num_ts)),
    LSTM(50),
    Dense(1, activation="linear")
])

model_lstm.compile(loss="mse", optimizer="adam")

es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)

%time hist = model_lstm.fit(X_train, y_train, epochs=1_000, \
  validation_data=(X_val, y_val), callbacks=[es], verbose=0);
```

```
Epoch 58: early stopping
Restoring model weights from the end of the best epoch: 8.
CPU times: user 1.93 s, sys: 468 ms, total: 2.39 s
Wall time: 2.09 s
```

Assess the fits

```
1 model_lstm.evaluate(X_val, y_val, verbose=0)
```

0.8229678273200989

A GRU layer

```
from keras.layers import GRU

random.seed(1)

model_gru = Sequential([
    Input((seq_length, num_ts)),
    GRU(50),
    Dense(1, activation="linear")
])

model_gru.compile(loss="mse", optimizer="adam")

es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)

%time hist = model_gru.fit(X_train, y_train, epochs=1_000, \
  validation_data=(X_val, y_val), callbacks=[es], verbose=0)
```

```
Epoch 59: early stopping
Restoring model weights from the end of the best epoch: 9.
CPU times: user 3.39 s, sys: 72 ms, total: 3.47 s
Wall time: 3.43 s
```

Assess the fits

```
1  model_gru.evaluate(X_val, y_val, verbose=0)
```

0.813169002532959

Two GRU layers

```
1  random.seed(1)
2
3  model_two_grus = Sequential([
4      Input((seq_length, num_ts)),
5      GRU(50, return_sequences=True),
6      GRU(50),
7      Dense(1, activation="linear")
8  ])
9
10 model_two_grus.compile(loss="mse", optimizer="adam")
11
12 es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)
13
14 %time hist = model_two_grus.fit(X_train, y_train, epochs=1_000, \
15   validation_data=(X_val, y_val), callbacks=[es], verbose=0)
```

```
Epoch 57: early stopping
Restoring model weights from the end of the best epoch: 7.
CPU times: user 5.7 s, sys: 97.4 ms, total: 5.8 s
Wall time: 5.74 s
```

Assess the fits

```
1 model_two_grus.evaluate(X_val, y_val, verbose=0)
```

```
0.7832801342010498
```

Change in HPI (%)
2
1
0
−1
0 10 20 30 40 50 60 70
Time
Sydney
2 GRUs

Compare the models

	Model	MSE
0	Dense	1.352932
1	SimpleRNN	0.864998
2	LSTM	0.822968
3	GRU	0.813169
4	2 GRUs	0.783280

The network with two GRU layers is the best.

```
model_two_grus.evaluate(X_test, y_test, verbose=0)
```

```
1.9257299900054932
```

Test set

Change in HPI (%)

4
2
0
−2
−4

Sydney
2 GRU

0 10 20 30 40 50 60

Time

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- **Predicting Multiple Time Series**

Creating input arrays

Change the `targets` argument to include all the suburbs.

```
1  X_train, y_train = make_sequences(
2      changes[:-delay], changes[delay:],
3      seq_length, end_index=num_train,
4  )
```

```
1  X_val, y_val = make_sequences(
2      changes[:-delay], changes[delay:],
3      seq_length, start_index=num_train,
4      end_index=num_train + num_val,
5  )
```

```
1  X_test, y_test = make_sequences(
2      changes[:-delay], changes[delay:],
3      seq_length, start_index=num_train + r
4  )
```

Training set shape

The shape of our training set is now:

```
1  X_train.shape
```

```
(220, 6, 7)
```

```
1  y_train.shape
```

```
(220, 7)
```

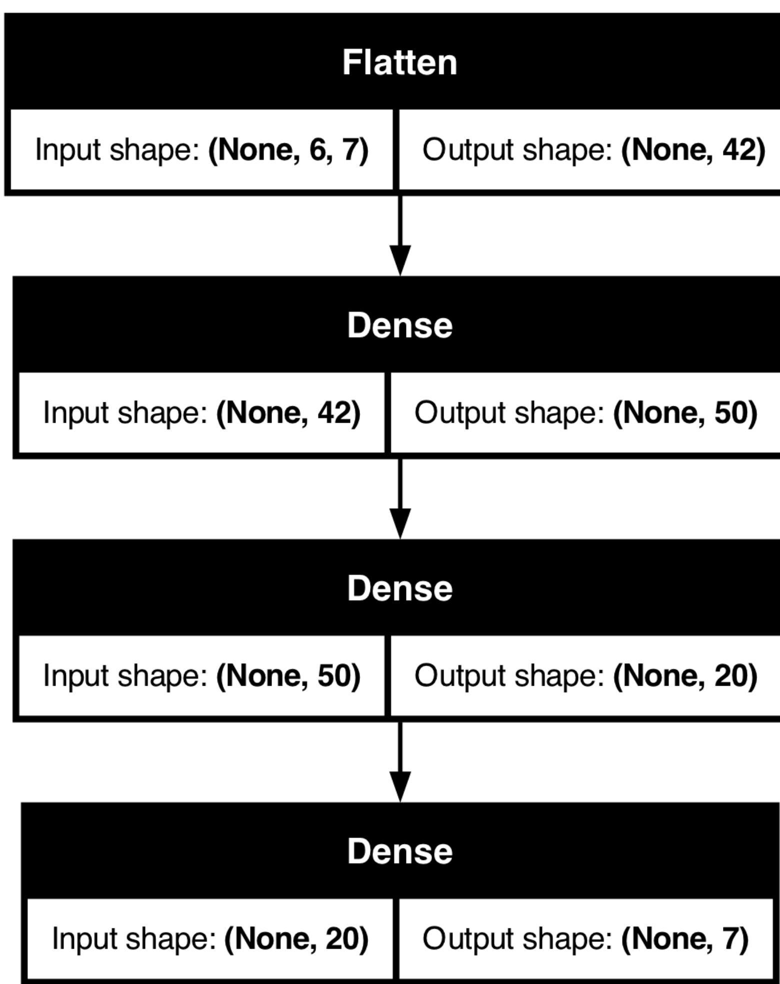
A dense network

```
random.seed(1)
model_dense = Sequential([
    Input((seq_length, num_ts)),
    Flatten(),
    Dense(50, activation="leaky_relu"),
    Dense(20, activation="leaky_relu"),
    Dense(num_ts, activation="linear")
])
model_dense.compile(loss="mse", optimizer="adam")
print(f"This model has {model_dense.count_params()} parameters.")

es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)
%time hist = model_dense.fit(X_train, y_train, epochs=1_000, \
  validation_data=(X_val, y_val), callbacks=[es], verbose=0);
```

```
This model has 3317 parameters.
Epoch 70: early stopping
Restoring model weights from the end of the best epoch: 20.
CPU times: user 1.42 s, sys: 66.5 ms, total: 1.49 s
Wall time: 1.44 s
```

Plot the model

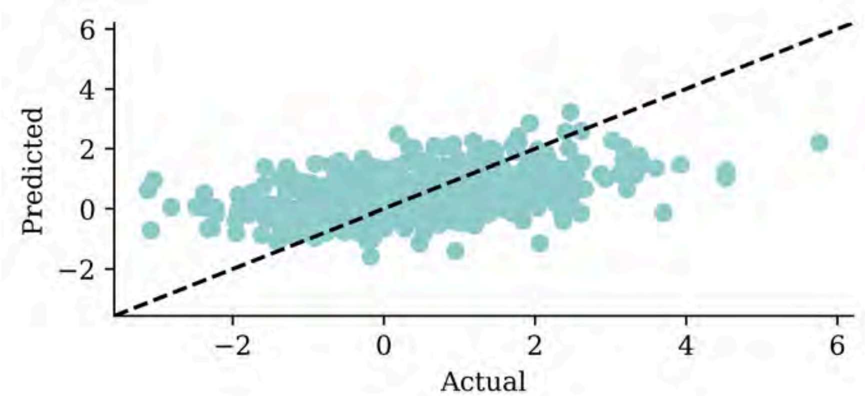
```
1 plot_model(model_dense, show_shapes=True)
```

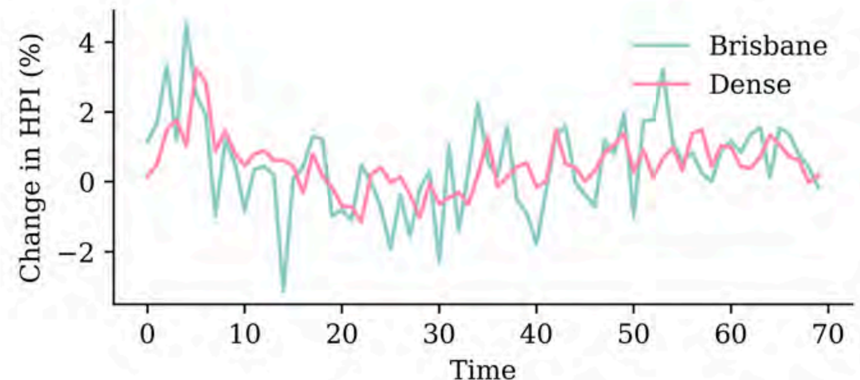


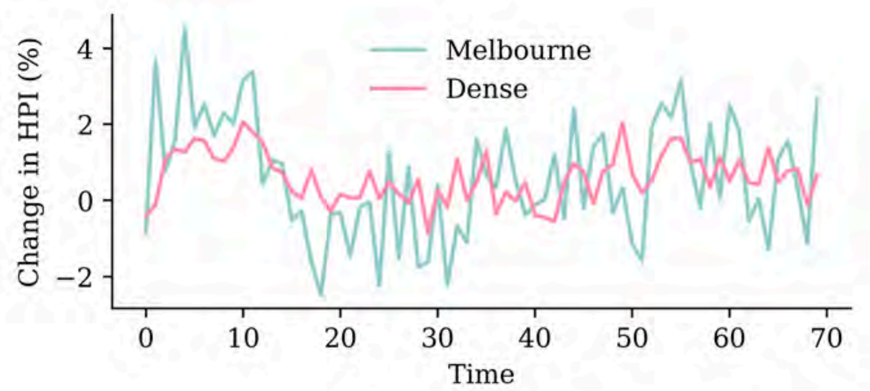
Assess the fits

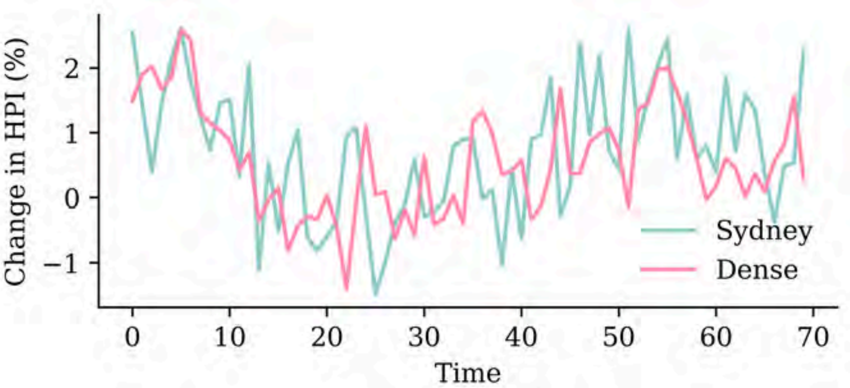
```
1 model_dense.evaluate(X_val, y_val, verbose=0)
```

```
1.4135159254074097
```









A SimpleRNN layer

```
1  random.seed(1)
2
3  model_simple = Sequential([
4      Input((seq_length, num_ts)),
5      SimpleRNN(50),
6      Dense(num_ts, activation="linear")
7  ])
8  model_simple.compile(loss="mse", optimizer="adam")
9  print(f"This model has {model_simple.count_params()} parameters.")
10
11 es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)
12 %time hist = model_simple.fit(X_train, y_train, epochs=1_000, \
13   validation_data=(X_val, y_val), callbacks=[es], verbose=0);
```

```
This model has 3257 parameters.
Epoch 69: early stopping
Restoring model weights from the end of the best epoch: 19.
CPU times: user 3.04 s, sys: 86.7 ms, total: 3.13 s
Wall time: 3.09 s
```

Plot the model

```python
plot_model(model_simple, show_shapes=True)
```

SimpleRNN	
Input shape: **(None, 6, 7)**	Output shape: **(None, 50)**

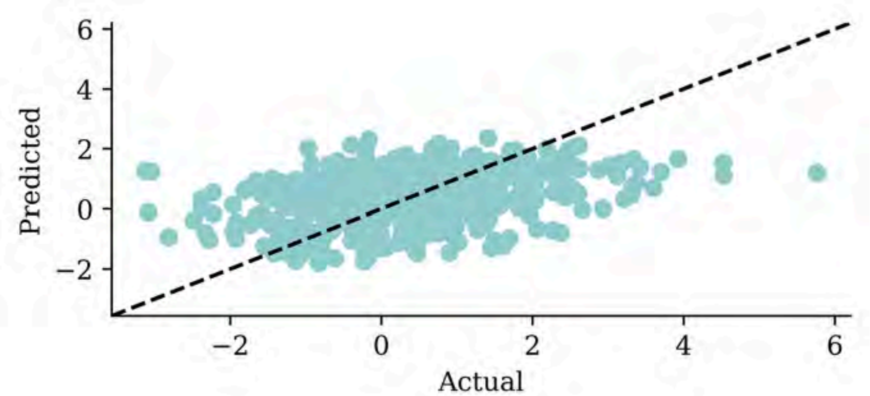
↓

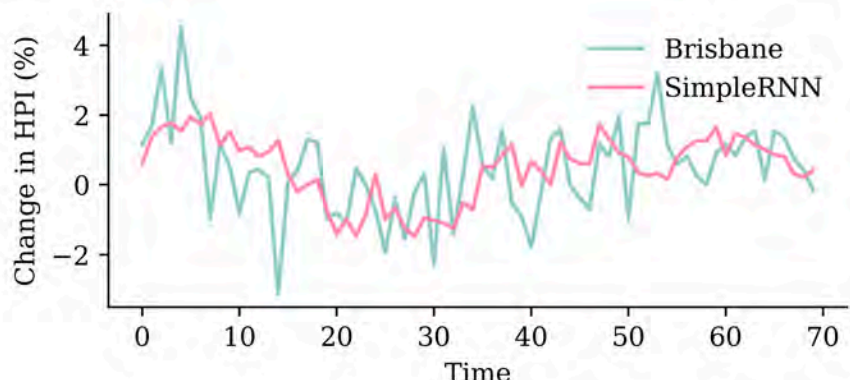
Dense	
Input shape: **(None, 50)**	Output shape: **(None, 7)**

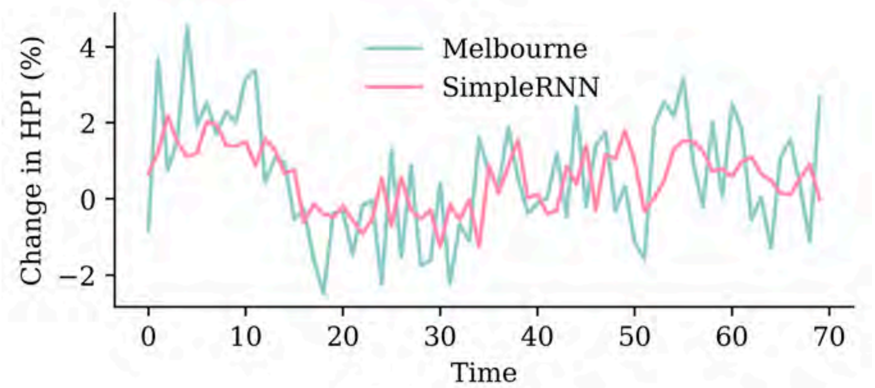
Assess the fits

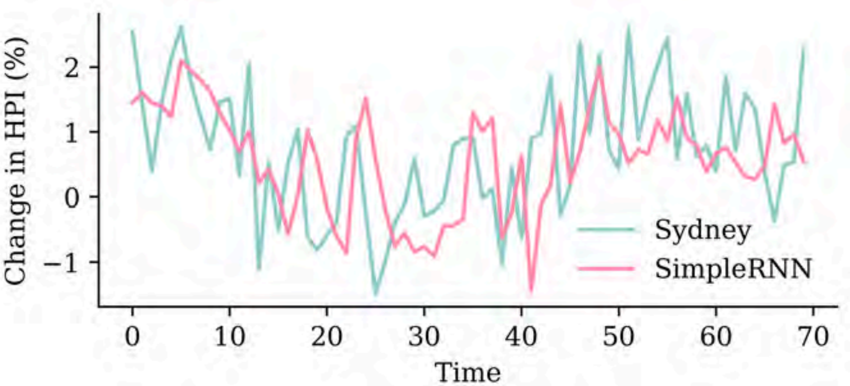
```python
model_simple.evaluate(X_val, y_val, verbose=0)
```

```
1.6102873086929321
```









A LSTM layer

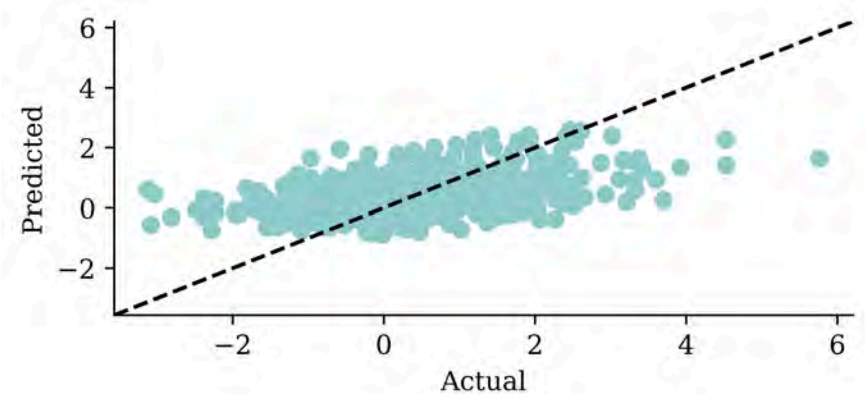
```
random.seed(1)

model_lstm = Sequential([
    Input((seq_length, num_ts)),
    LSTM(50),
    Dense(num_ts, activation="linear")
])

model_lstm.compile(loss="mse", optimizer="adam")

es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)

%time hist = model_lstm.fit(X_train, y_train, epochs=1_000, \
  validation_data=(X_val, y_val), callbacks=[es], verbose=0);
```

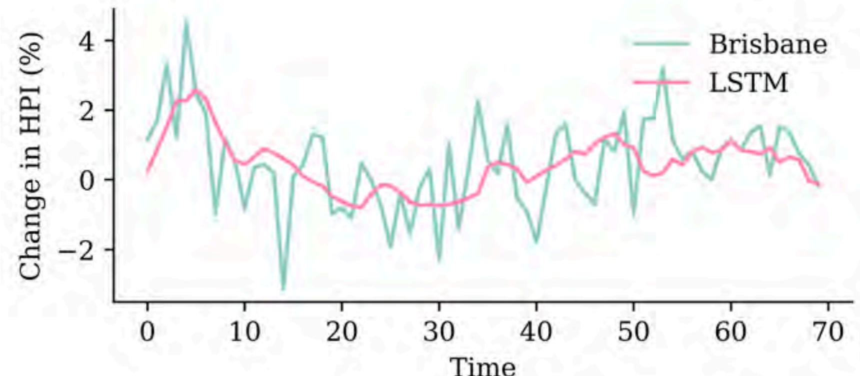
```
Epoch 76: early stopping
Restoring model weights from the end of the best epoch: 26.
CPU times: user 2.5 s, sys: 596 ms, total: 3.1 s
Wall time: 2.71 s
```

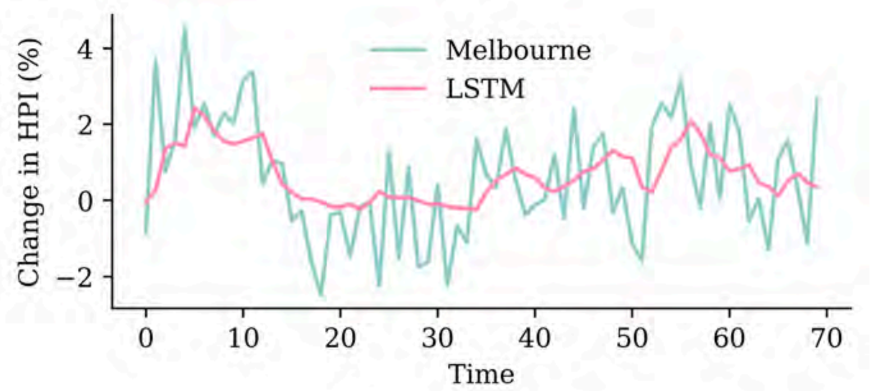
Assess the fits

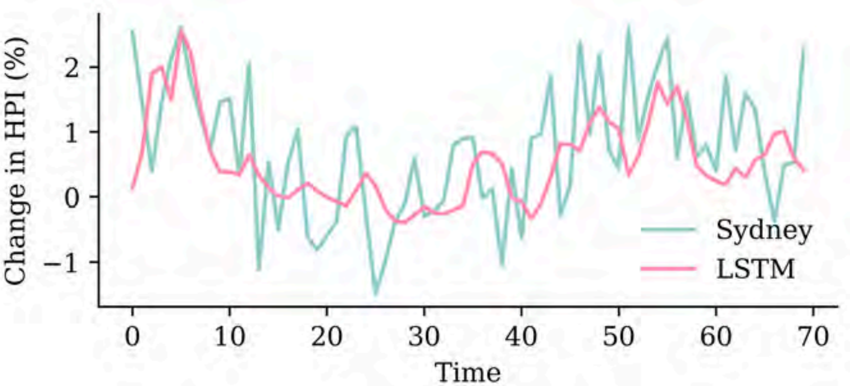
```
1 model_lstm.evaluate(X_val, y_val, verbose=0)
```

1.3229057788848877









A GRU layer

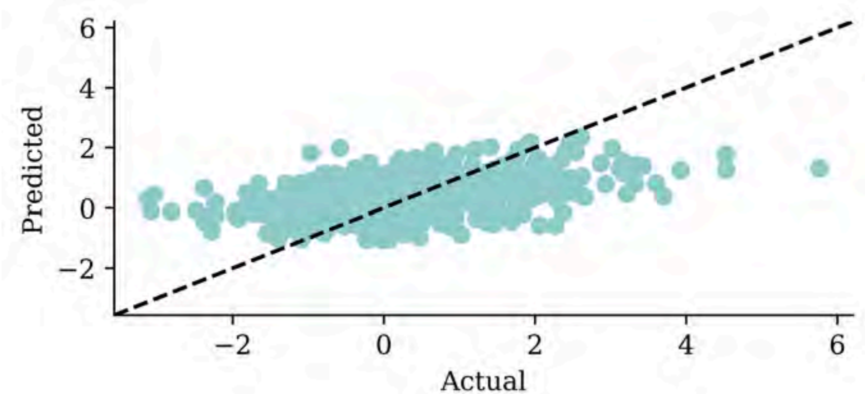
```
random.seed(1)

model_gru = Sequential([
    Input((seq_length, num_ts)),
    GRU(50),
    Dense(num_ts, activation="linear")
])

model_gru.compile(loss="mse", optimizer="adam")

es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)

%time hist = model_gru.fit(X_train, y_train, epochs=1_000, \
  validation_data=(X_val, y_val), callbacks=[es], verbose=0)
```

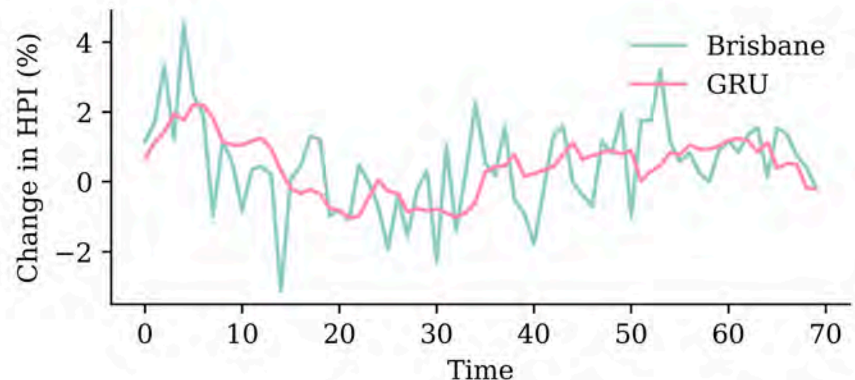
```
Epoch 69: early stopping
Restoring model weights from the end of the best epoch: 19.
CPU times: user 3.92 s, sys: 83.3 ms, total: 4 s
Wall time: 3.96 s
```

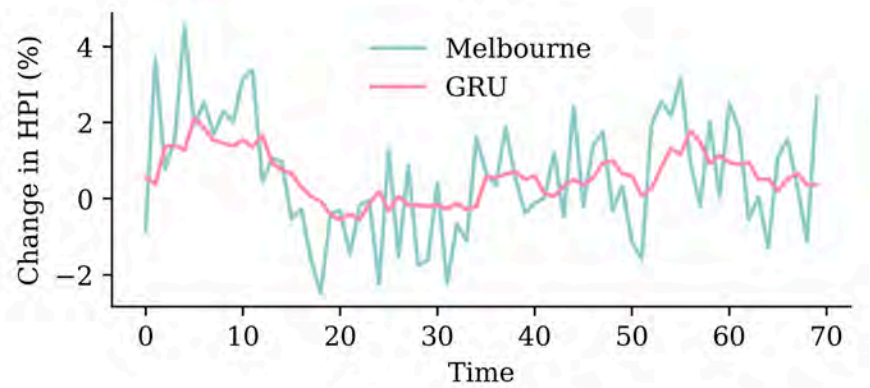
Assess the fits

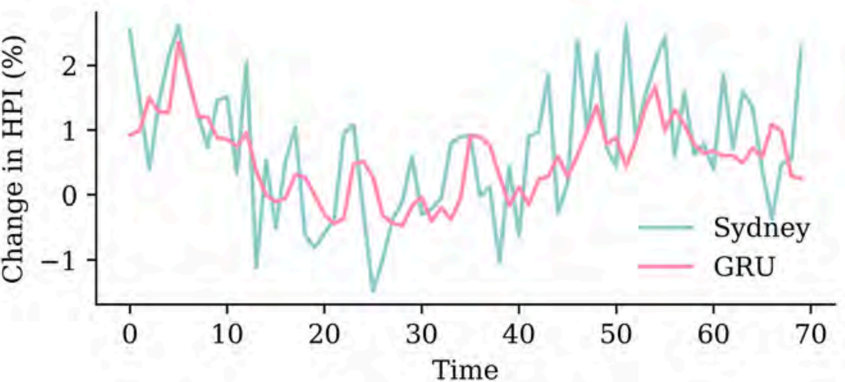
```python
model_gru.evaluate(X_val, y_val, verbose=0)
```

```
1.340381383895874
```









Two GRU layers

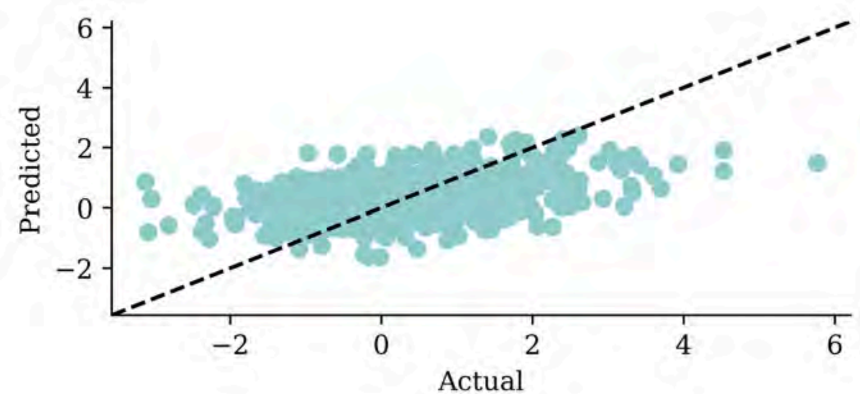
```python
random.seed(1)

model_two_grus = Sequential([
    Input((seq_length, num_ts)),
    GRU(50, return_sequences=True),
    GRU(50),
    Dense(num_ts, activation="linear")
])

model_two_grus.compile(loss="mse", optimizer="adam")

es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)

%time hist = model_two_grus.fit(X_train, y_train, epochs=1_000, \
  validation_data=(X_val, y_val), callbacks=[es], verbose=0)
```

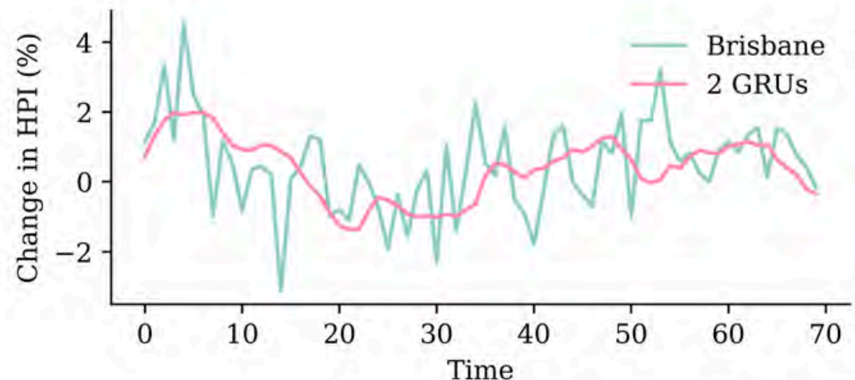
```
Epoch 74: early stopping
Restoring model weights from the end of the best epoch: 24.
CPU times: user 7.37 s, sys: 125 ms, total: 7.49 s
Wall time: 7.42 s
```

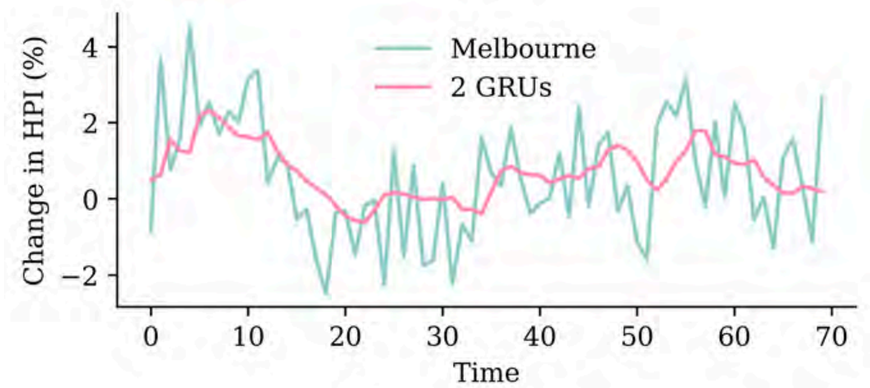
Assess the fits

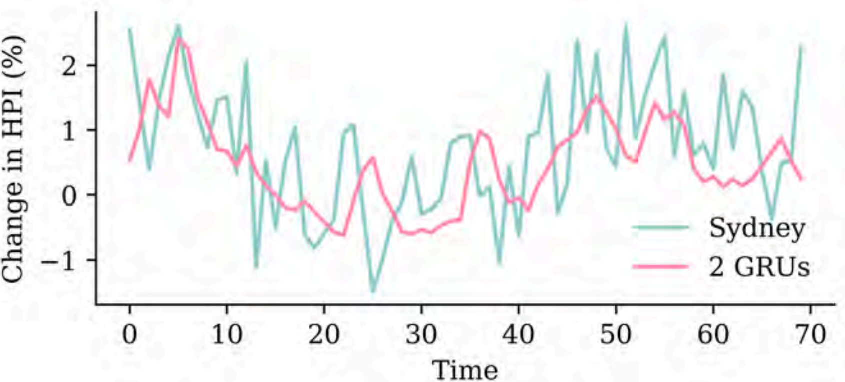
```
1 model_two_grus.evaluate(X_val, y_val, verbose=0)
```

1.3471126556396484









Compare the models

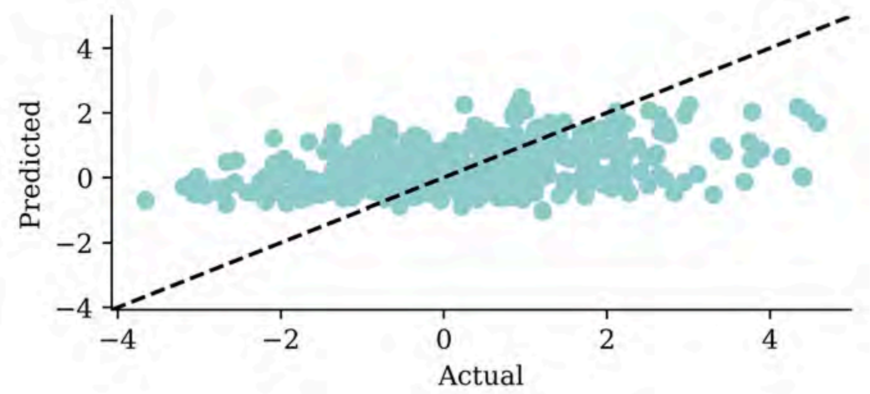
	Model	MSE
1	SimpleRNN	1.610287
0	Dense	1.413516
4	2 GRUs	1.347113
3	GRU	1.340381
2	LSTM	1.322906

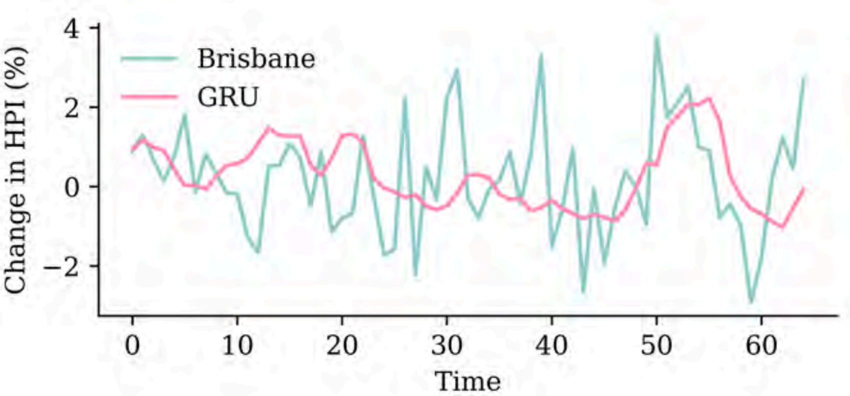
The network with an LSTM layer is the best.

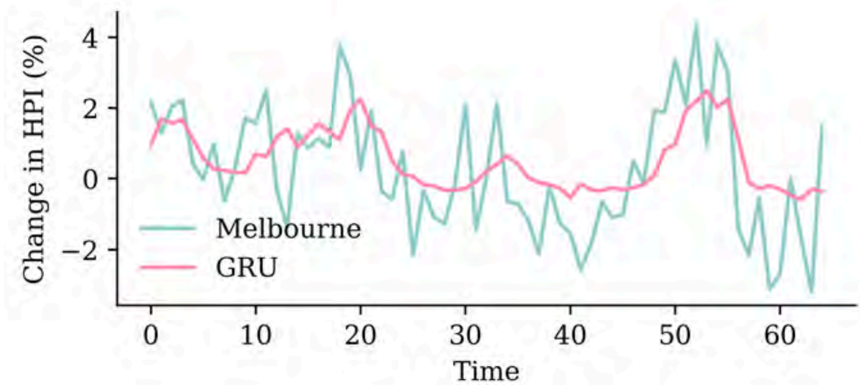
```
1  model_lstm.evaluate(X_test, y_test, verbose=0)
```

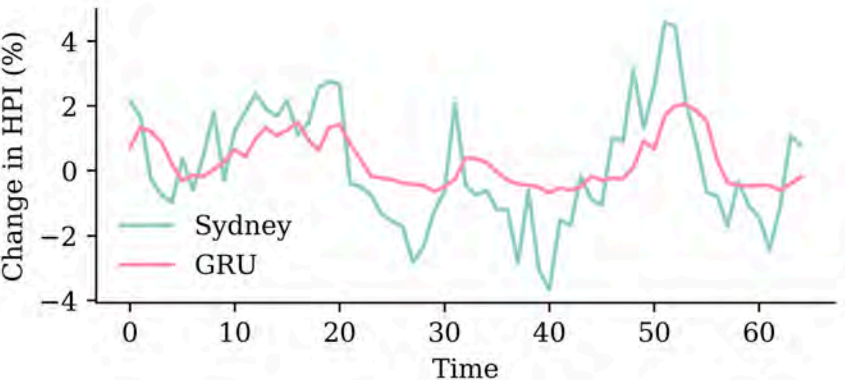
```
1.8732062578201294
```

Test set









Package Versions

```python
from watermark import watermark
print(watermark(python=True, packages="keras,matplotlib,numpy,pandas,seaborn,scipy,torch"
```

```
Python implementation: CPython
Python version       : 3.14.3
IPython version      : 9.13.0

keras     : 3.14.1
matplotlib: 3.10.9
numpy     : 2.4.4
pandas    : 3.0.2
seaborn   : 0.13.2
scipy     : 1.17.1
torch     : 2.11.0
```

Glossary

- autoregressive forecasting
- forecasting
- GRU
- LSTM
- one-step/multi-step ahead forecasting
- persistence forecast
- recurrent neural networks
- SimpleRNN