Time Series & Recurrent Neural Networks

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Patrick Laub

UNSW SYDNEY

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.

UNSW SYDNEY

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.

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

Australian financial stocks

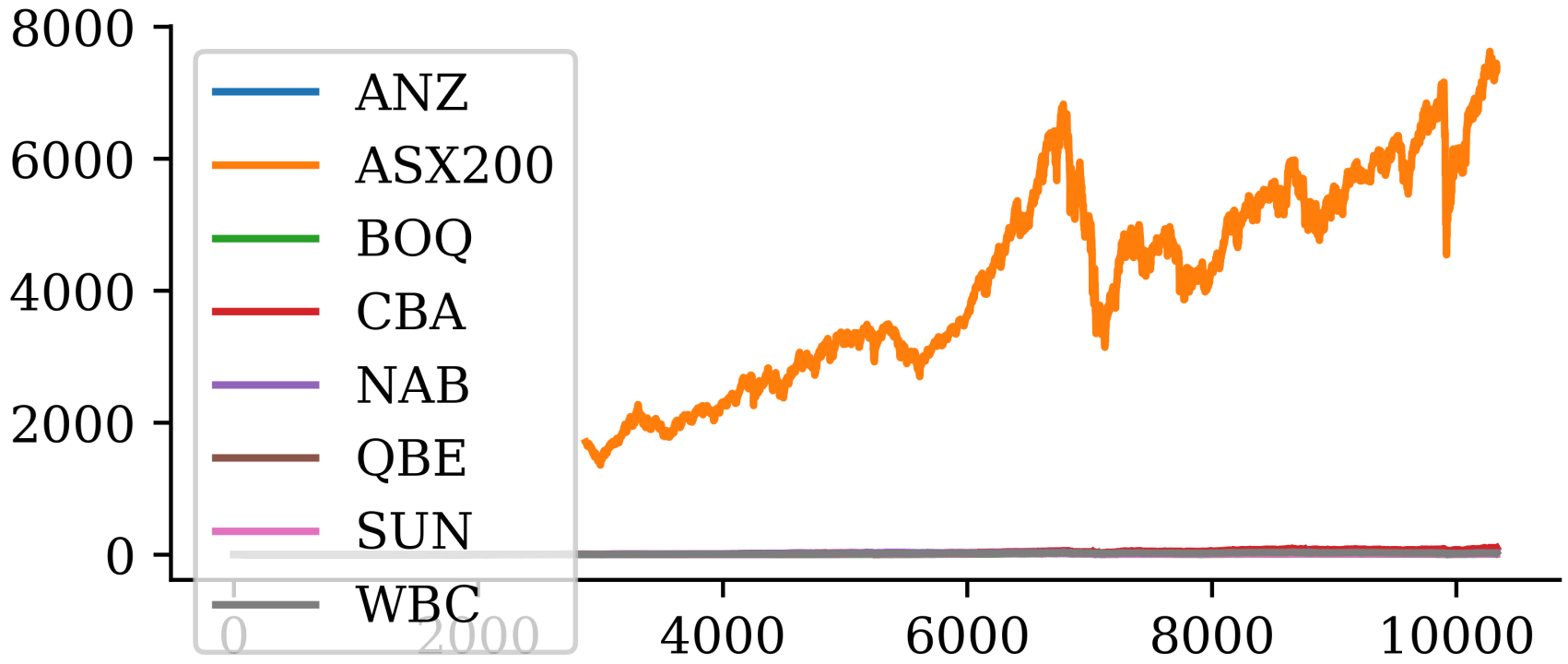
```
1 stocks = pd.read_csv("aus_fin_stocks.csv")
2 stocks
```

	Date	ANZ	ASX200	BOQ	CBA	NAB	QBE	SUN	WBC
0	1981-01-02	1.588896	NaN	NaN	NaN	1.791642	NaN	NaN	2.199454
1	1981-01-05	1.548452	NaN	NaN	NaN	1.791642	NaN	NaN	2.163397
2	1981-01-06	1.600452	NaN	NaN	NaN	1.791642	NaN	NaN	2.199454
...	...	...	...	...	...	...	...	...	...
10327	2021-10-28	28.600000	7430.4	8.97	106.86	29.450000	12.10	12.02	26.230000
10328	2021-10-29	28.140000	7323.7	8.80	104.68	28.710000	11.83	11.72	25.670000
10329	2021-11-01	27.900000	7357.4	8.79	105.71	28.565000	12.03	11.83	24.050000

10330 rows × 9 columns

Plot

```
1 stocks.plot()
```



Data types and NA values

```
1 stocks.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10330 entries, 0 to 10329
Data columns (total 9 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Date    10330 non-null  object 
 1   ANZ     10319 non-null  float64
 2   ASX200  7452 non-null   float64
 3   BOQ     8970 non-null   float64
 4   CBA     7624 non-null   float64
 5   NAB     10316 non-null  float64
 6   QBE     9441 non-null   float64
 7   SUN     8424 non-null   float64
 8   WBC     10323 non-null  float64
dtypes: float64(8), object(1)
memory usage: 726.5+ KB
```

```
1 for col in stocks.columns:
2     print(f"{col}: {stocks[col].isna().s
```

```
Date: 0
ANZ: 11
ASX200: 2878
BOQ: 1360
CBA: 2706
NAB: 14
QBE: 889
SUN: 1906
WBC: 7
```

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

UNSW SYDNEY

Set the index to the date

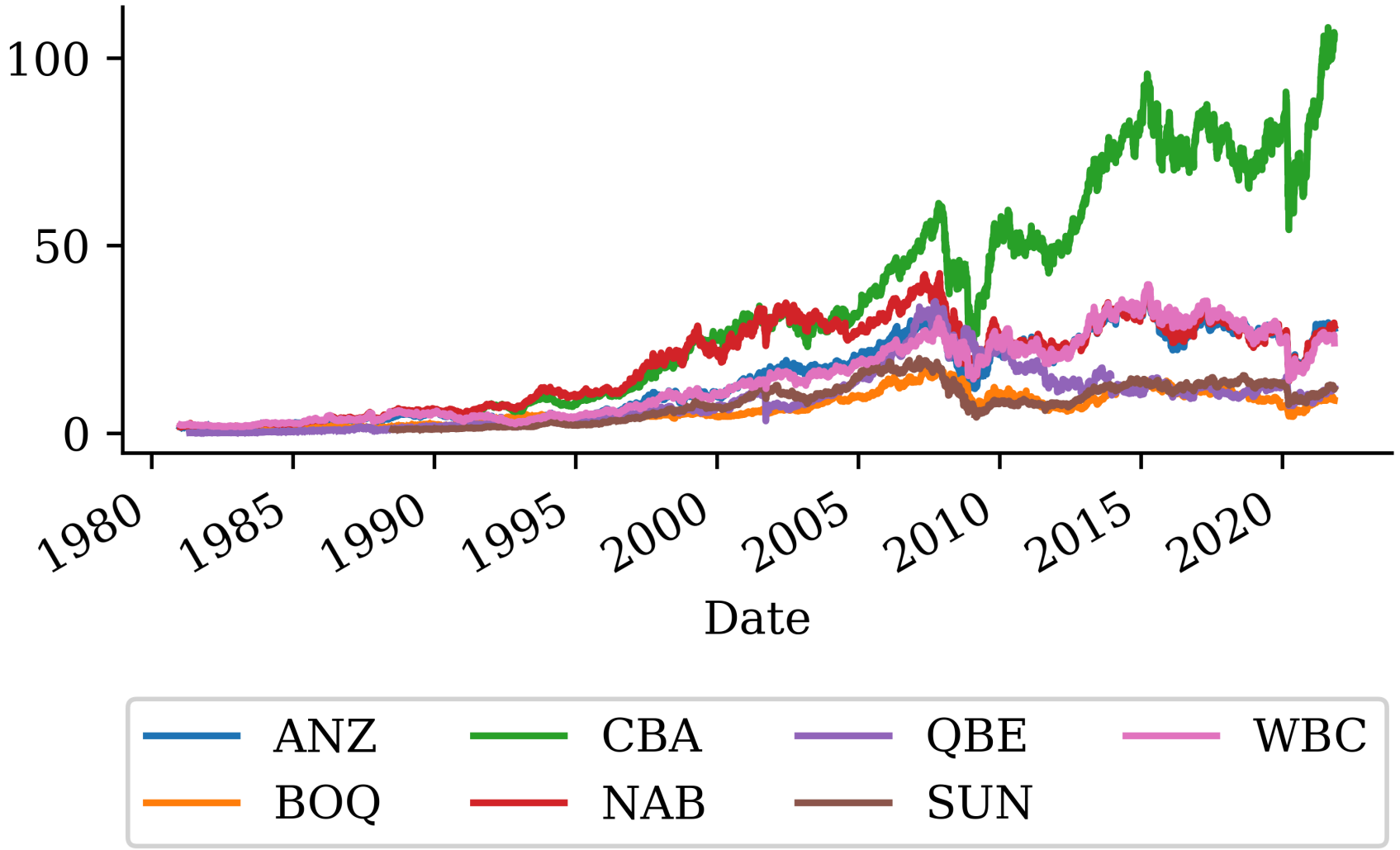
```
stocks["Date"] = pd.to_datetime(stocks["Date"])
stocks = stocks.set_index("Date") # or `stocks.set_index("Date", inplace=True)`
stocks
```

Date	ANZ	BOQ	CBA	NAB	QBE	SUN	WBC
1981-01-02	1.588896	NaN	NaN	1.791642	NaN	NaN	2.199454
1981-01-05	1.548452	NaN	NaN	1.791642	NaN	NaN	2.163397
1981-01-06	1.600452	NaN	NaN	1.791642	NaN	NaN	2.199454
...	...	...	...	...	...	...	...
2021-10-28	28.600000	8.97	106.86	29.450000	12.10	12.02	26.230000
2021-10-29	28.140000	8.80	104.68	28.710000	11.83	11.72	25.670000
2021-11-01	27.900000	8.79	105.71	28.565000	12.03	11.83	24.050000

10330 rows × 7 columns

Plot II

```
1 stocks.plot()
2 plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.5), ncol=4);
```



Can index using dates I

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

Date	ANZ	BOQ	CBA	NAB	QBE	SUN	WBC
2010-01-04	22.89	10.772147	54.573702	26.046571	25.21	8.142453	25.012620
2010-01-05	23.00	10.910369	55.399220	26.379283	25.34	8.264684	25.220235
2010-01-06	22.66	10.855080	55.677708	25.865956	24.95	8.086039	25.101598
2010-01-07	22.12	10.523346	55.140624	25.656823	24.50	8.198867	24.765460
2010-01-08	22.25	10.781361	55.856736	25.571269	24.77	8.245879	24.864324

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

UNSW SYDNEY

Can index using dates II

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

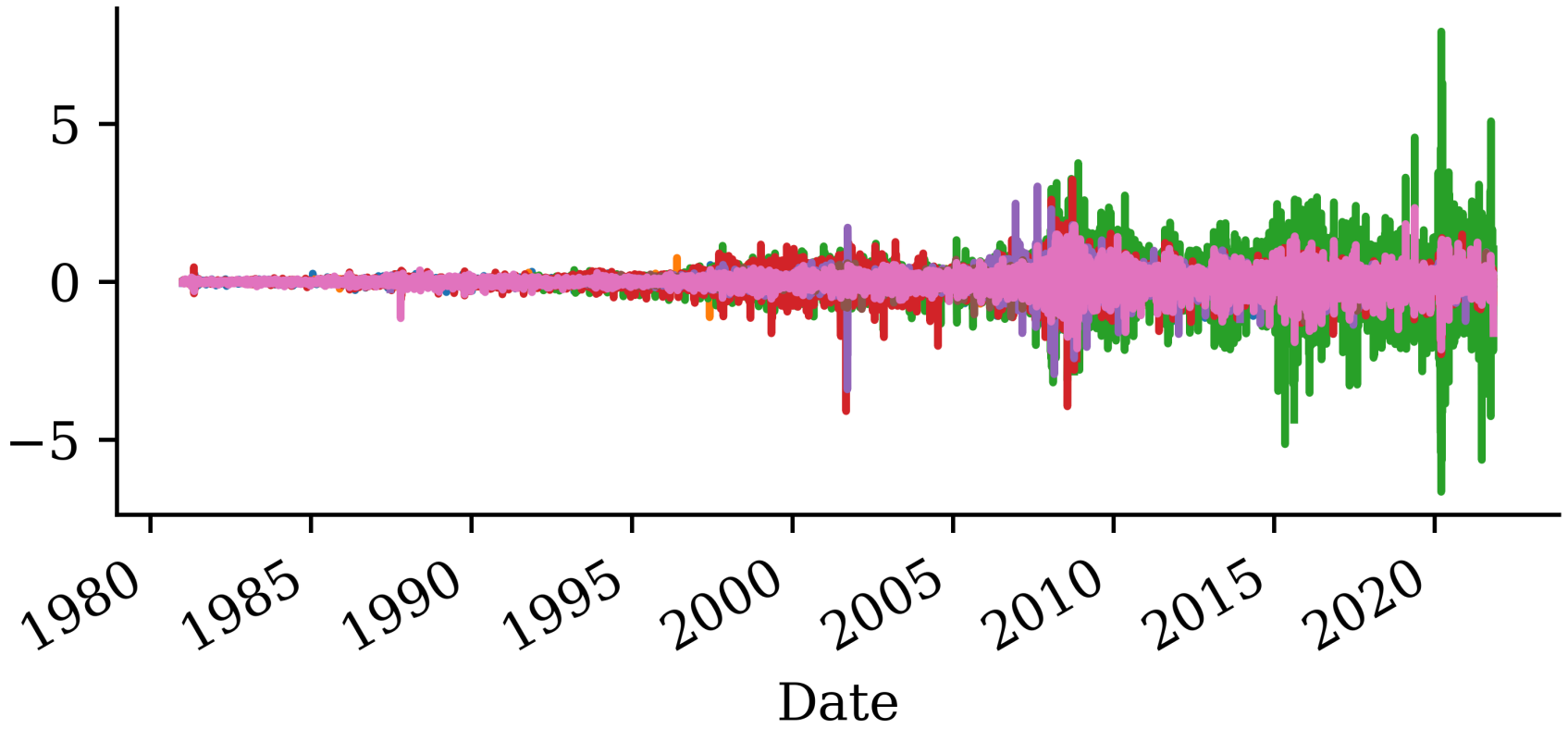
```
1 stocks.loc["2019-12":"2020", ["CBA"]]
```

	CBA
Date	
2019-12-02	81.43
2019-12-03	79.34
2019-12-04	77.81
...	...
2020-12-29	84.01
2020-12-30	83.59
2020-12-31	82.11

275 rows × 1 columns

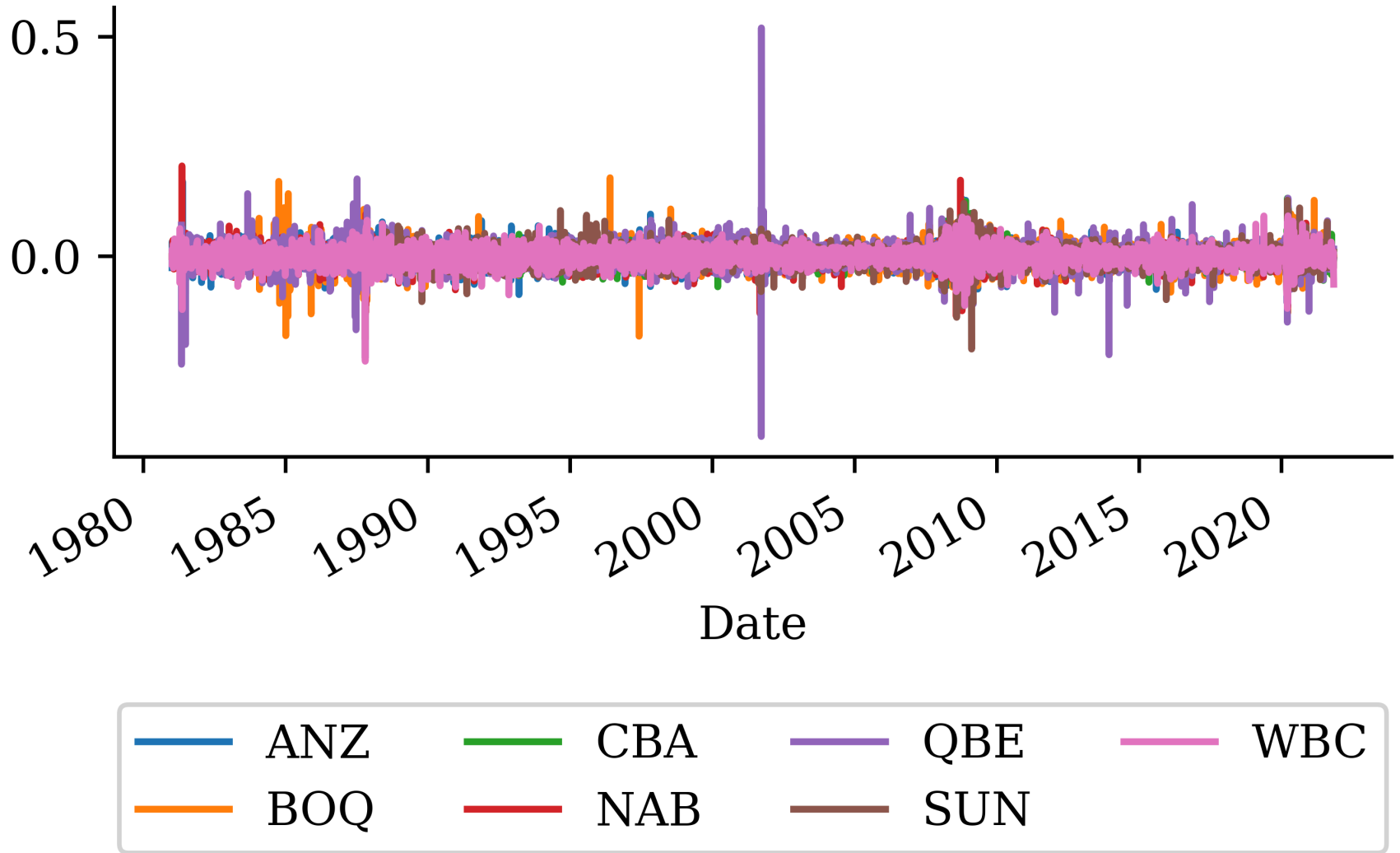
Can look at the first differences

```
1 stocks.diff().plot()
2 plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.5), ncol=4);
```



Can look at the percentage changes

```
1 stocks.pct_change().plot()
2 plt.legend(loc="upper center", bbox_to_anchor=(0.5, -0.5), ncol=4);
```



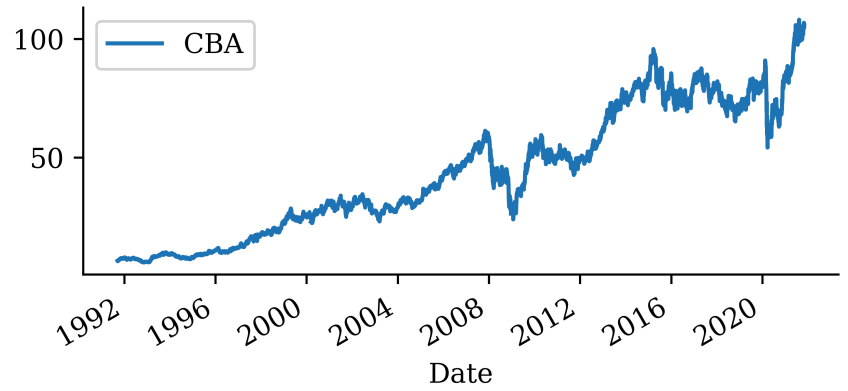
Focus on one stock

```
1 stock = stocks[["CBA"]]
2 stock
```

	CBA
Date	
1981-01-02	NaN
1981-01-05	NaN
1981-01-06	NaN
...	...
2021-10-28	106.86
2021-10-29	104.68
2021-11-01	105.71

10330 rows × 1 columns

```
1 stock.plot()
```



Find first non-missing value

```
1 first_day = stock.dropna().index[0]
2 first_day
```

```
Timestamp('1991-09-12 00:00:00')
```

```
1 stock = stock.loc[first_day:]
```

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

```
CBA    8
dtype: int64
```

Fill in the missing values

```
missing_day = stock[stock["CBA"].isna()].index[0]
prev_day = missing_day - pd.Timedelta(days=1)
after = missing_day + pd.Timedelta(days=3)
```

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

Date	CBA
2000-03-07	24.56662
2000-03-08	NaN
2000-03-09	NaN
2000-03-10	22.87580

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

Date	CBA
2000-03-07	24.56662
2000-03-08	24.56662
2000-03-09	24.56662
2000-03-10	22.87580

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

```
CBA    0
dtype: int64
```

UNSW SYDNEY

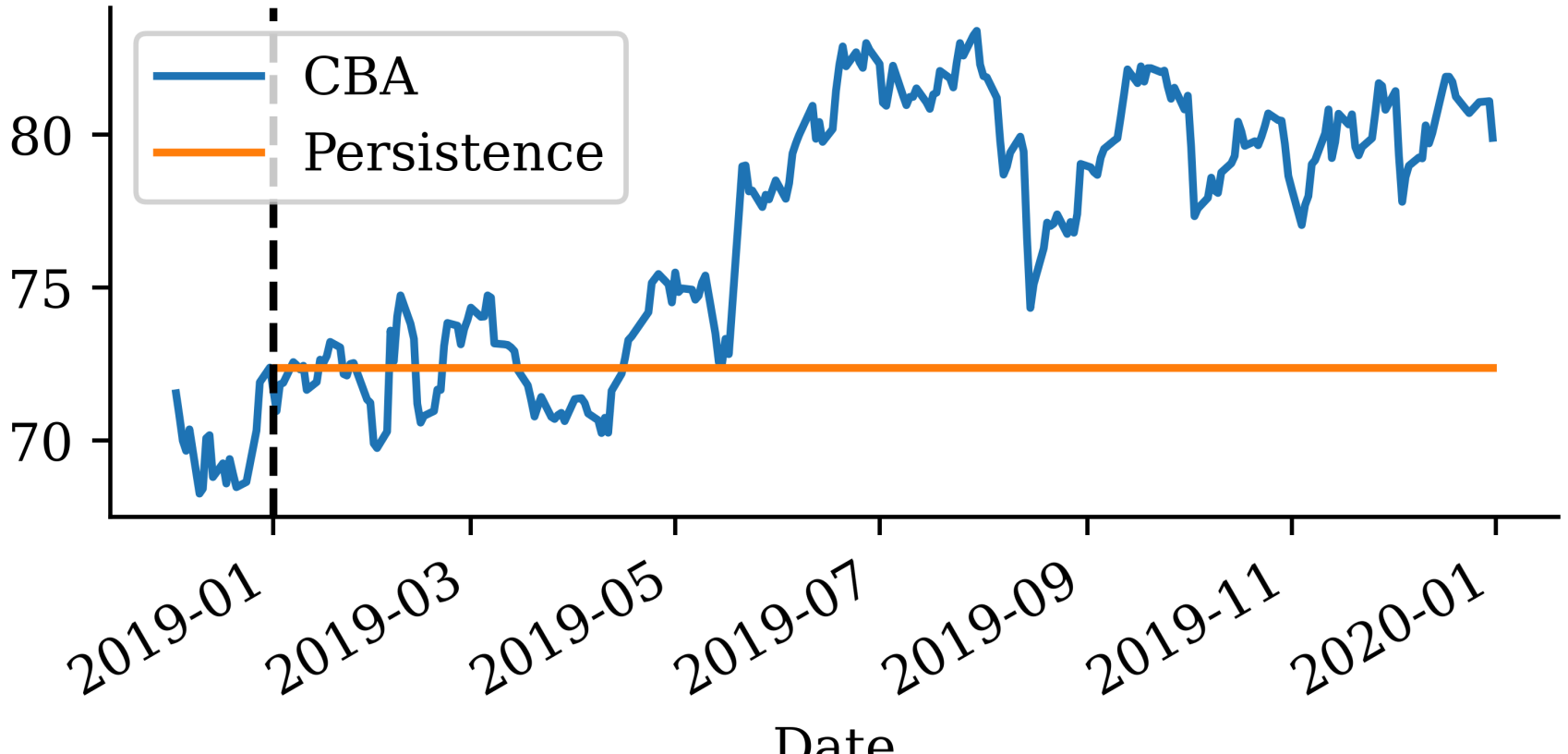
Lecture Outline

- 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

Persistence forecast

The simplest model is to predict the next value to be the same as the current value.

```
stock.loc["2019":, "Persistence"] = stock.loc["2018"].iloc[-1].values[0]
stock.loc["2018-12":"2019"].plot()
plt.axvline("2019", color="black", linestyle="--")
```



Trend

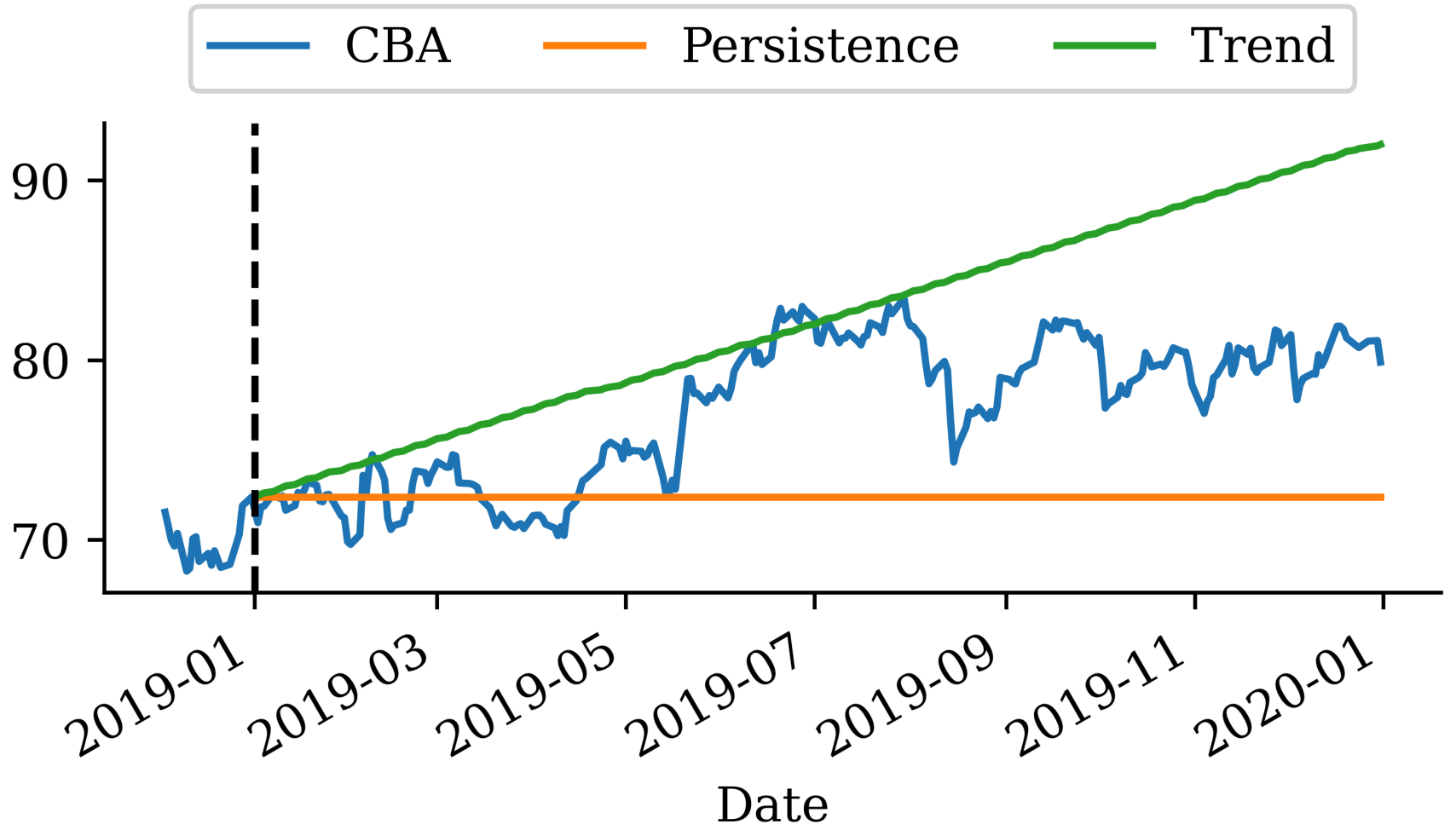
We can extrapolate from recent trend:

```
past_date = stock.loc["2018"].index[-30]
past = stock.loc[past_date, "CBA"]
latest_date = stock.loc["2018", "CBA"].index[-1]
latest = stock.loc[latest_date, "CBA"]

trend = (latest - past) / (latest_date - past_date).days
print(trend)

tdays_since_cutoff = np.arange(1, len(stock.loc["2019":]) + 1)
stock.loc["2019":, "Trend"] = latest + trend * tdays_since_cutoff
```

```
0.07755555555555545
```

Trend forecasts

```
1 stock.loc["2018-12":"2019"].plot()
2 plt.axvline("2019", color="black", linestyle="--")
3 plt.legend(ncol=3, loc="upper center", bbox_to_anchor=(0.5, 1.3))
```



Which is better?

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

```
1 persistence_mse = mean_squared_error(stock.loc["2019", "CBA"], stock.loc["2019", "Persis
2 trend_mse = mean_squared_error(stock.loc["2019", "CBA"], stock.loc["2019", "Trend"])
3 persistence_mse, trend_mse
```

```
(39.54629367588932, 37.87104674064297)
```

Use the history

```
1 cba_shifted = stock["CBA"].head().shift(1)
2 both = pd.concat([stock["CBA"].head(), cba_shifted], axis=1, keys=["Today", "Yesterday"])
3 both
```

Date	Today	Yesterday
1991-09-12	6.425116	NaN
1991-09-13	6.365440	6.425116
1991-09-16	6.305764	6.365440
1991-09-17	6.285872	6.305764
1991-09-18	6.325656	6.285872

```
1 def lagged_timeseries(df, target, window=30):
2     lagged = pd.DataFrame()
3     for i in range(window, 0, -1):
4         lagged[f"T-{i}"] = df[target].shift(i)
5     lagged["T"] = df[target].values
6     return lagged
```

UNSW SYDNEY

Lagged time series

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

	T-40	T-39	T-38	T-37	T-36	T-35	T-34	T-33	T-32	T-31	...	T-9
Date												
1991-09-12	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
1991-09-13	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...
2021-10-29	101.84	102.16	102.14	102.92	100.55	101.09	101.30	101.58	101.41	102.85	...	103.94
2021-11-01	102.16	102.14	102.92	100.55	101.09	101.30	101.58	101.41	102.85	102.88	...	103.89

7632 rows × 41 columns

UNSW SYDNEY

Split into training and testing

```
# Split the data in time
X_train = df_lags.loc[:"2018"]
X_val = df_lags.loc["2019"]
X_test = df_lags.loc["2020":]

# 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")
```

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

```
((6872, 40), (6872,), (253, 40), (253,), (467, 40), (467,))
```

UNSW SYDNEY

Inspect the split data

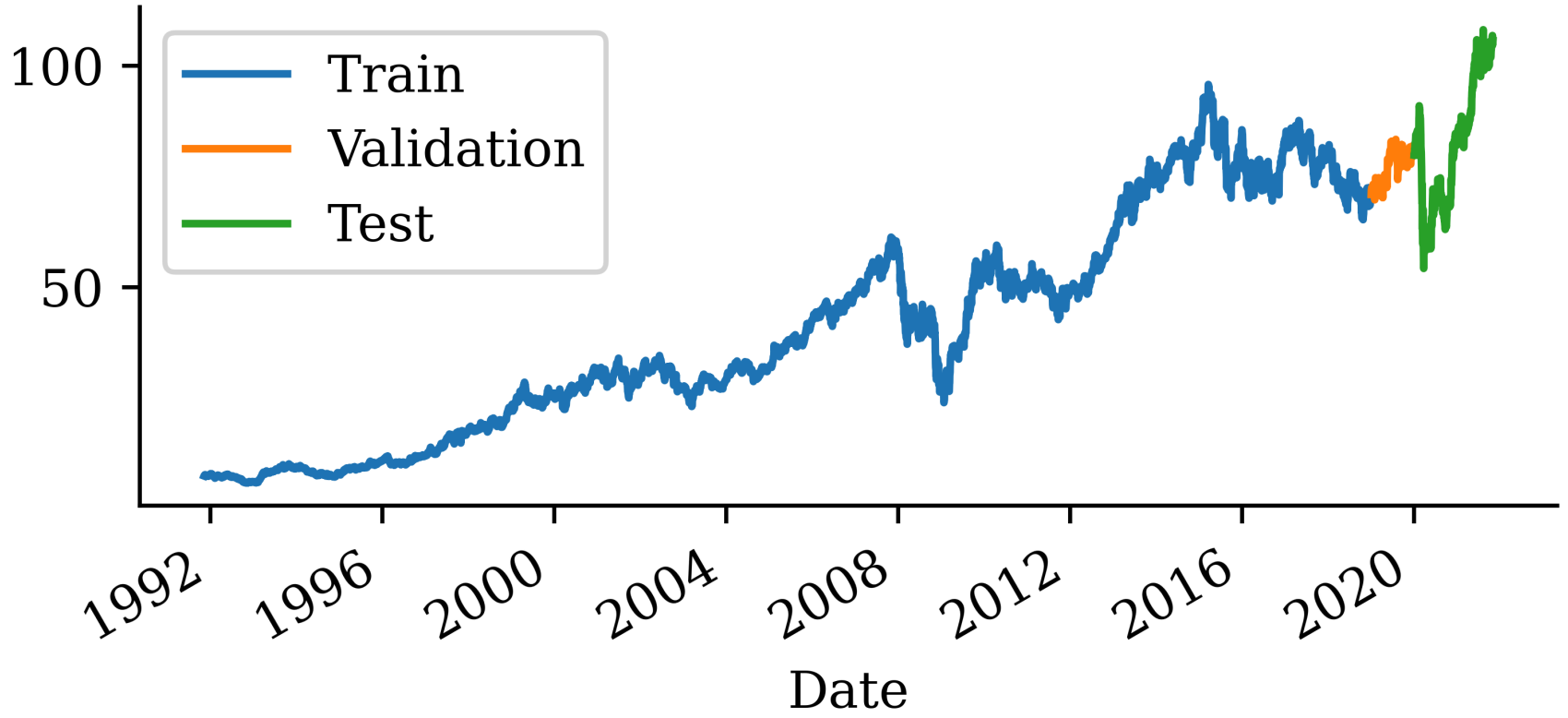
```
1 X_train
```

Date	T-40	T-39	T-38	T-37	T-36	T-35	T-34	
1991-11-07	6.425116	6.365440	6.305764	6.285872	6.325656	6.385332	6.445008	6.4
1991-11-08	6.365440	6.305764	6.285872	6.325656	6.385332	6.445008	6.445008	6.5
1991-11-11	6.305764	6.285872	6.325656	6.385332	6.445008	6.445008	6.504684	6.5
...	...	...	...	...	...	...	...	...
2018-12-27	68.160000	69.230000	68.940000	68.350000	67.980000	68.950000	69.350000	70.
2018-12-28	69.230000	68.940000	68.350000	67.980000	68.950000	69.350000	70.620000	70.
2018-12-31	68.940000	68.350000	67.980000	68.950000	69.350000	70.620000	70.950000	71.

6872 rows × 40 columns

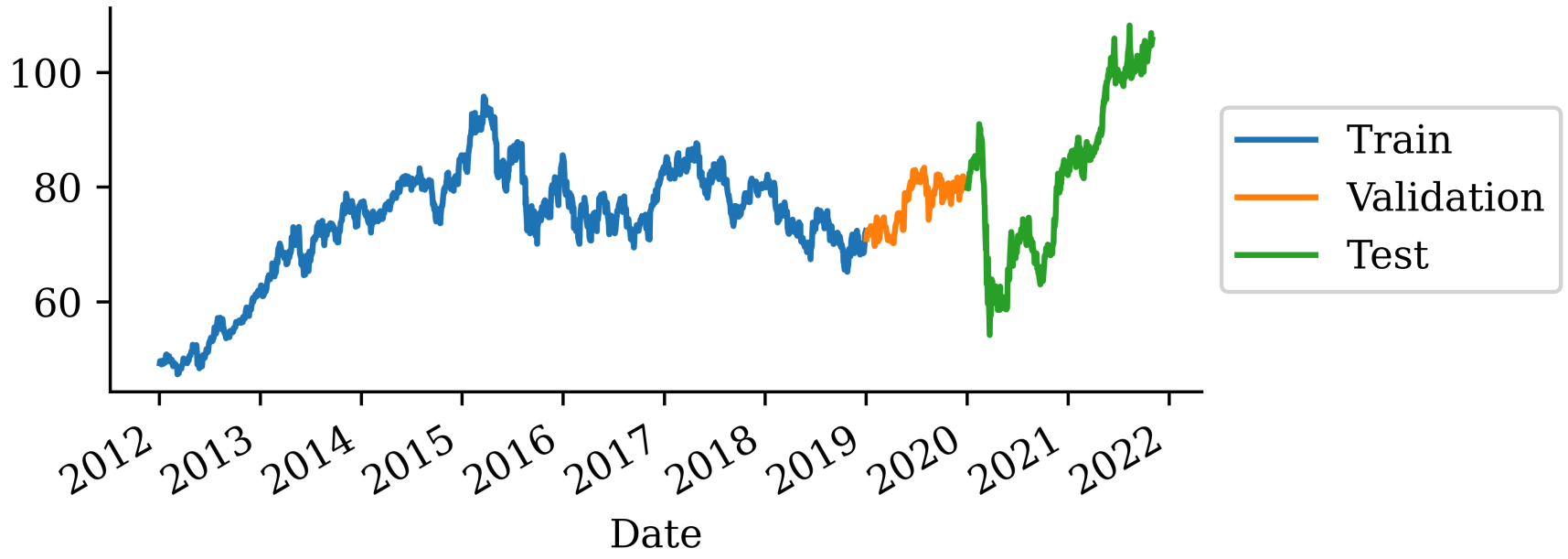
UNSW SYDNEY

Plot the split



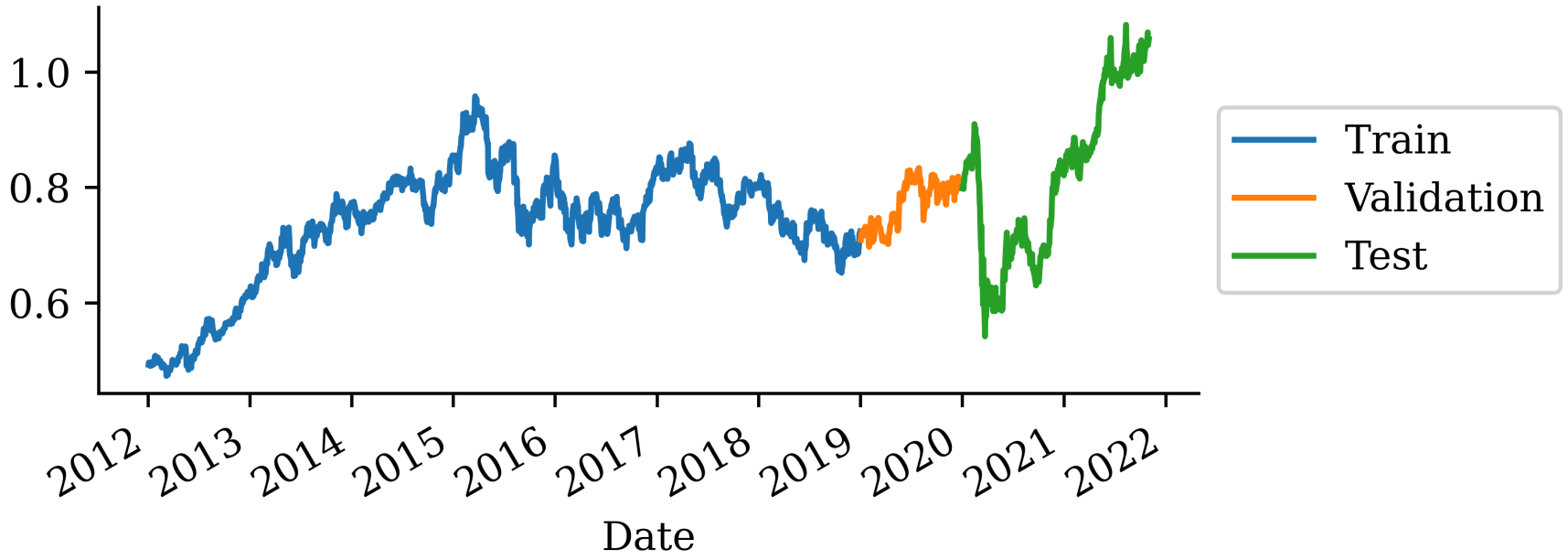
Train on more recent data

```
1 X_train = X_train.loc["2012":]
2 y_train = y_train.loc["2012":]
```



Rescale by eyeballing it

```
X_train = X_train / 100
X_val = X_val / 100
X_test = X_test / 100
y_train = y_train / 100
y_val = y_val / 100
y_test = y_test / 100
```

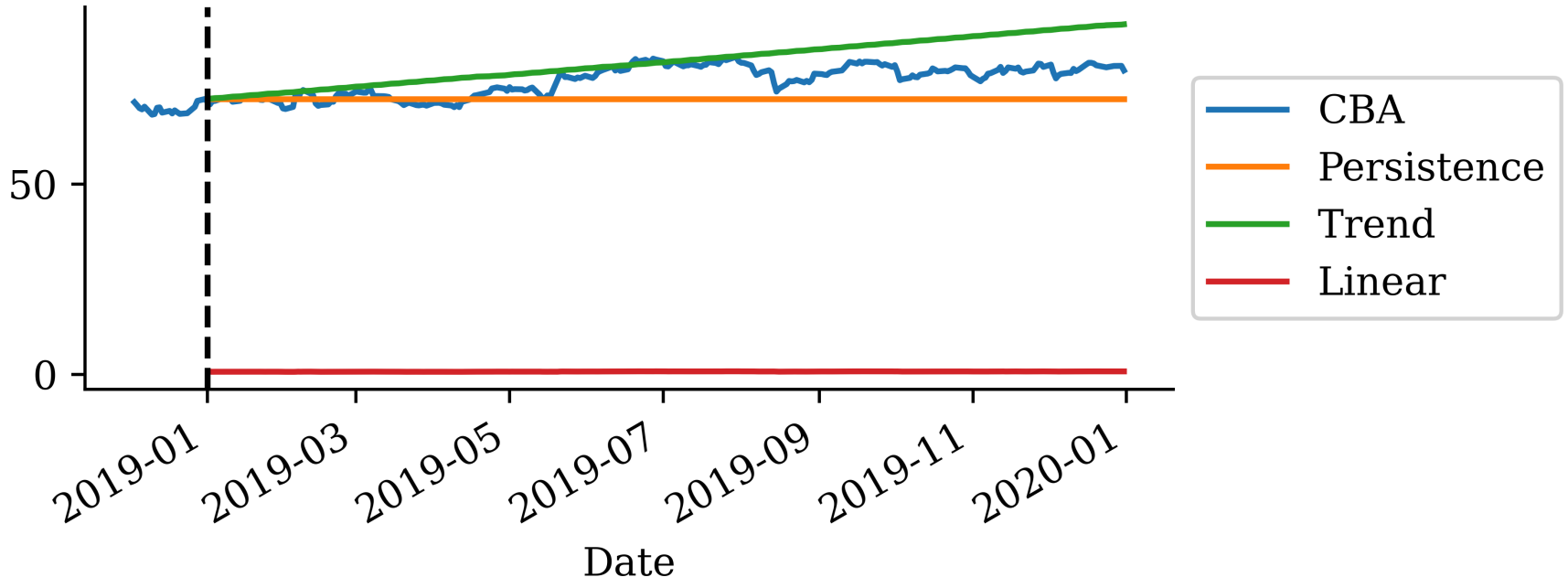


Fit a linear model

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

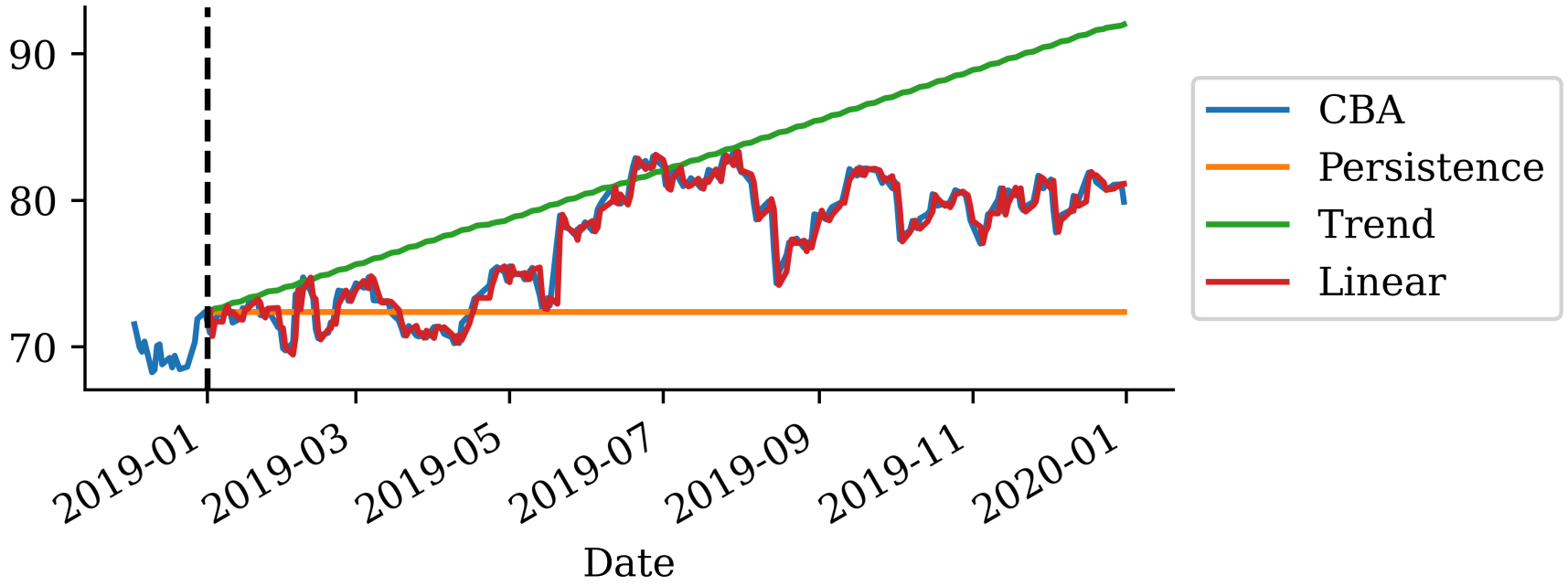
Make a forecast for the validation data:

```
y_pred = lr.predict(X_val)
stock.loc[X_val.index, "Linear"] = y_pred
```



Inverse-transform the forecasts

```
1 stock.loc[X_val.index, "Linear"] = 100 * y_pred
```



Careful with the metrics

```
1 mean_squared_error(y_val, y_pred)
```

```
6.329105517812197e-05
```

```
1 mean_squared_error(100 * y_val, 100 * y_pred)
```

```
0.6329105517812198
```

```
1 100**2 * mean_squared_error(y_val, y_pred)
```

```
0.6329105517812197
```

```
1 linear_mse = 100**2 * mean_squared_error(y_val, y_pred)
2 persistence_mse, trend_mse, linear_mse
```

```
(39.54629367588932, 37.87104674064297, 0.6329105517812197)
```

Lecture Outline

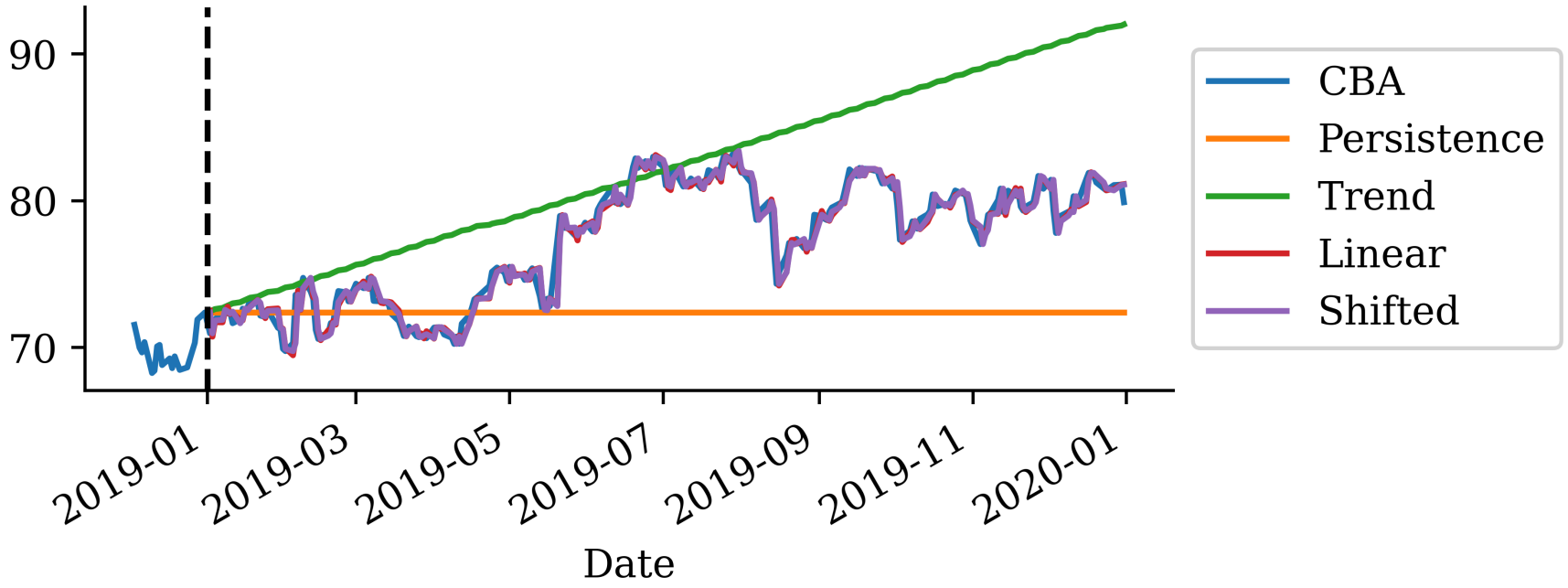
- Time Series
- Baseline forecasts
- **Multi-step forecasts**
- Neural network forecasts
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- Predicting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

```
1  stock.loc["2019":, "Shifted"] = stock["CBA"].shift(1).loc["2019":]
```




```
1  shifted_mse = mean_squared_error(stock.loc["2019", "CBA"], stock.loc["2019", "Shifted"])
2  persistence_mse, trend_mse, linear_mse, shifted_mse
```

```
(39.54629367588932, 37.87104674064297, 0.6329105517812197, 0.6367221343873524)
```

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.

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

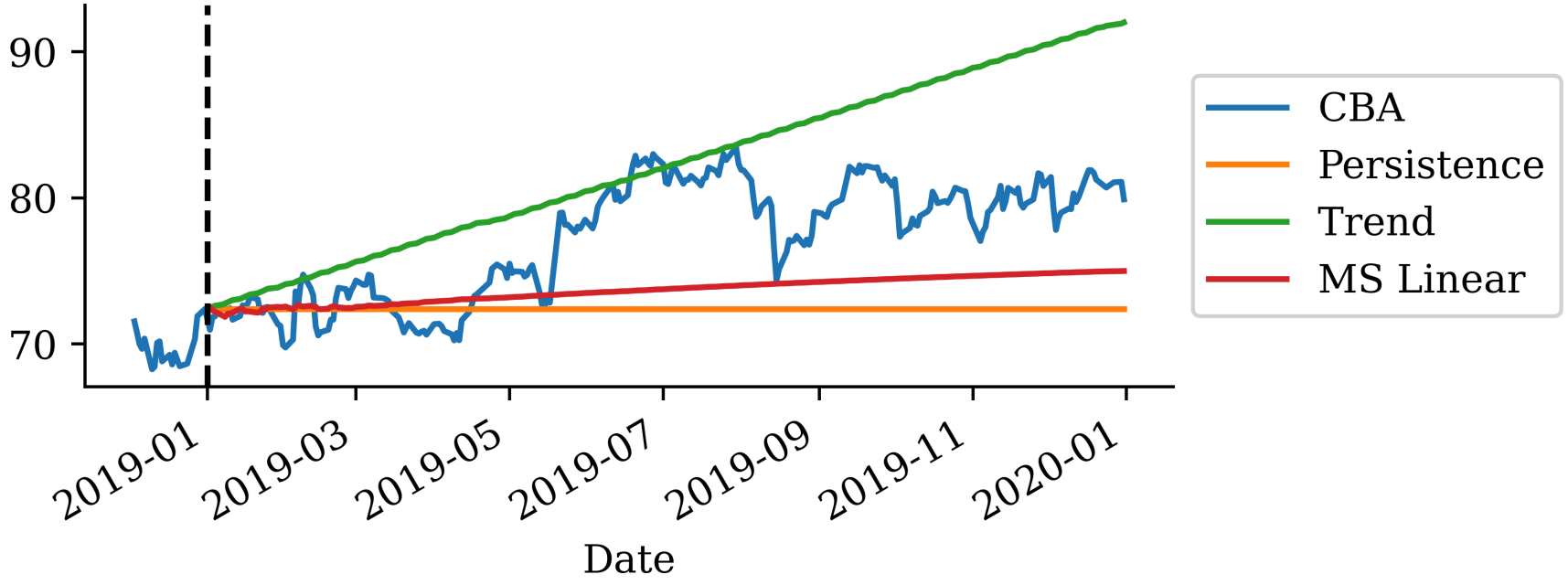
Autoregressive forecasting function

```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

        # 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

```
lr_forecast = autoregressive_forecast(lr, X_val)
stock.loc[lr_forecast.index, "MS Linear"] = 100 * lr_forecast
```

```
stock.loc["2018-12":"2019"].drop(["Linear", "Shifted"], axis=1).plot()
plt.axvline("2019", color="black", linestyle="--")
plt.legend(loc="center left", bbox_to_anchor=(1, 0.5));
```



Metrics

One-step-ahead forecasts:

```
1 linear_mse, shifted_mse
```

```
(0.6329105517812197, 0.6367221343873524)
```

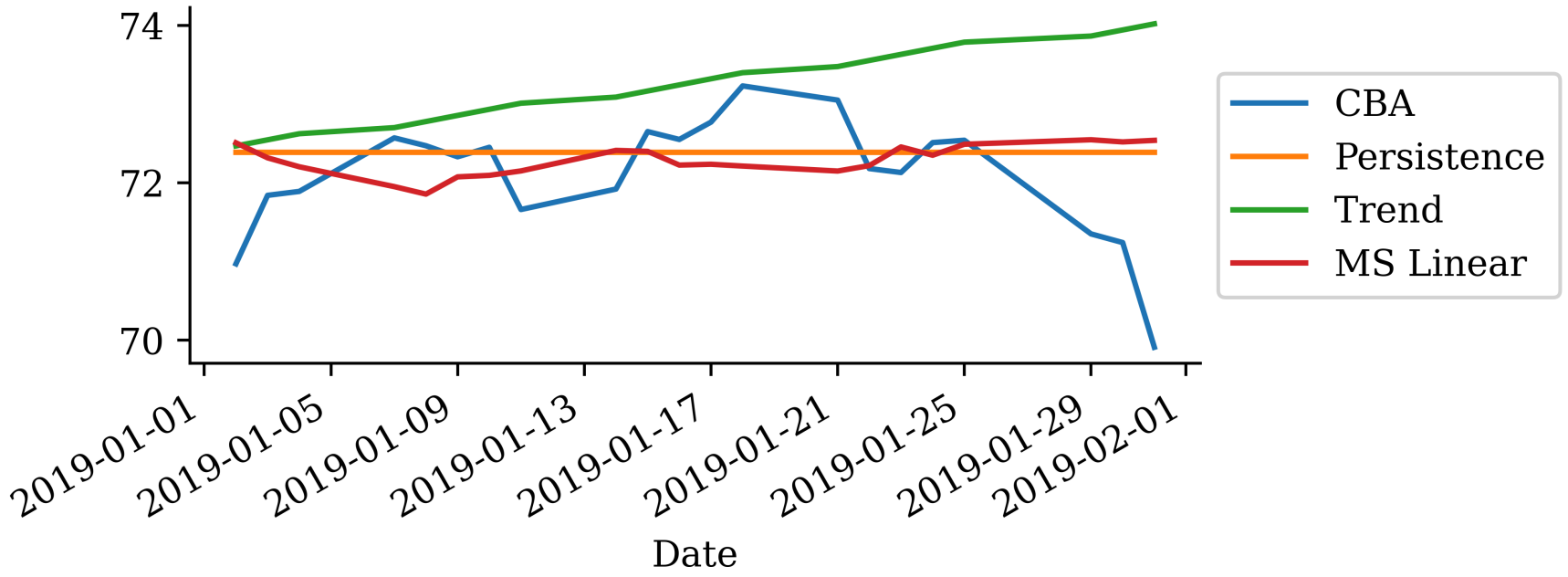
Multi-step-ahead forecasts:

```
1 multi_step_linear_mse = 100**2 * mean_squared_error(y_val, lr_forecast)
2 persistence_mse, trend_mse, multi_step_linear_mse
```

```
(39.54629367588932, 37.87104674064297, 23.847003791127374)
```

Prefer only short windows

```
1 stock.loc["2019":"2019-1"].drop(["Linear", "Shifted"], axis=1).plot();
2 plt.legend(loc="center left", bbox_to_anchor=(1, 0.5));
```



> "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

```
model = Sequential([
        Dense(64, activation="leaky_relu"),
        Dense(1, "softplus")])

model.compile(optimizer="adam", loss="mean_squared_error")
```

```
if Path("aus_fin_fnn_model.h5").exists():
    model = keras.models.load_model("aus_fin_fnn_model.h5")
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("aus_fin_fnn_model.h5")

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(32, 64)	2,624
dense_1 (Dense)	(32, 1)	65

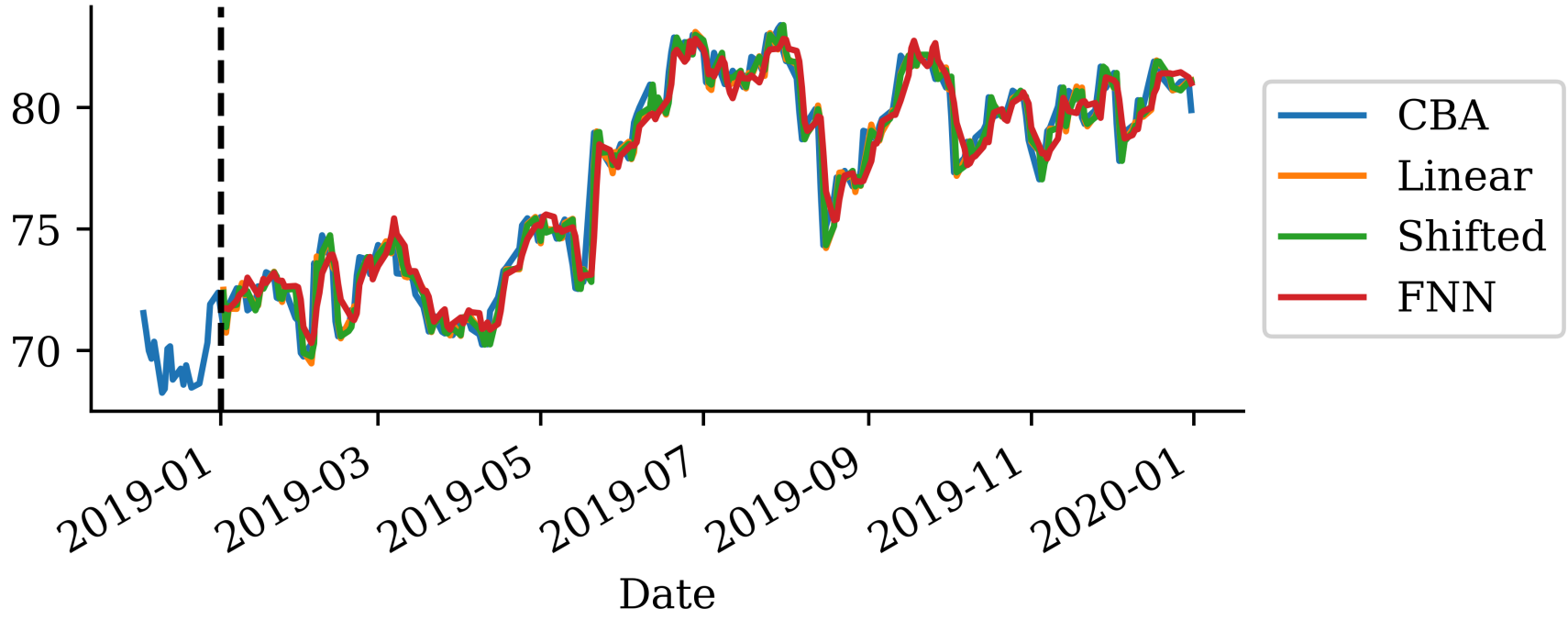
Total params: 2,691 (10.52 KB)
Trainable params: 2,689 (10.50 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 2 (12.00 B)

UNSW SYDNEY

Forecast and plot

```
y_pred = model.predict(X_val, verbose=0)
stock.loc[X_val.index, "FNN"] = 100 * y_pred
```

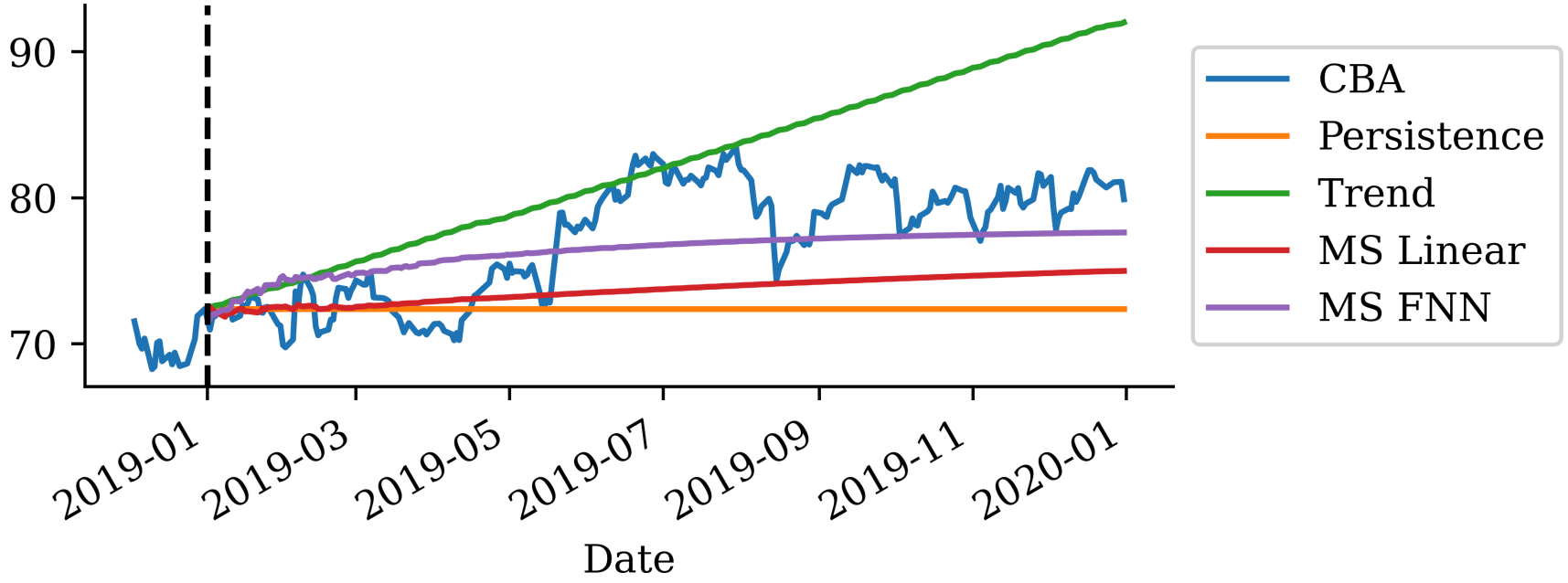
```
stock.loc["2018-12":"2019"].drop(["Persistence", "Trend", "MS Linear"], axis=1).plot()
plt.axvline("2019", color="black", linestyle="--")
plt.legend(loc="center left", bbox_to_anchor=(1, 0.5));
```



Autoregressive forecasts

```
nn_forecast = autoregressive_forecast(model, X_val, True)
stock.loc[nn_forecast.index, "MS FNN"] = 100 * nn_forecast
```

```
stock.loc["2018-12":"2019"].drop(["Linear", "Shifted", "FNN"], axis=1).plot()
plt.axvline("2019", color="black", linestyle="--")
plt.legend(loc="center left", bbox_to_anchor=(1, 0.5));
```



Metrics

One-step-ahead forecasts:

```
nn_mse = 100**2 * mean_squared_error(y_val, y_pred)
linear_mse, shifted_mse, nn_mse
```

```
(0.6329105517812197, 0.6367221343873524, 1.044511310373669)
```

Multi-step-ahead forecasts:

```
multi_step_fnn_mse = 100**2 * mean_squared_error(y_val, nn_forecast)
persistence_mse, trend_mse, multi_step_linear_mse, multi_step_fnn_mse
```

```
(39.54629367588932, 37.87104674064297, 23.847003791127374, 10.150589118262795)
```

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

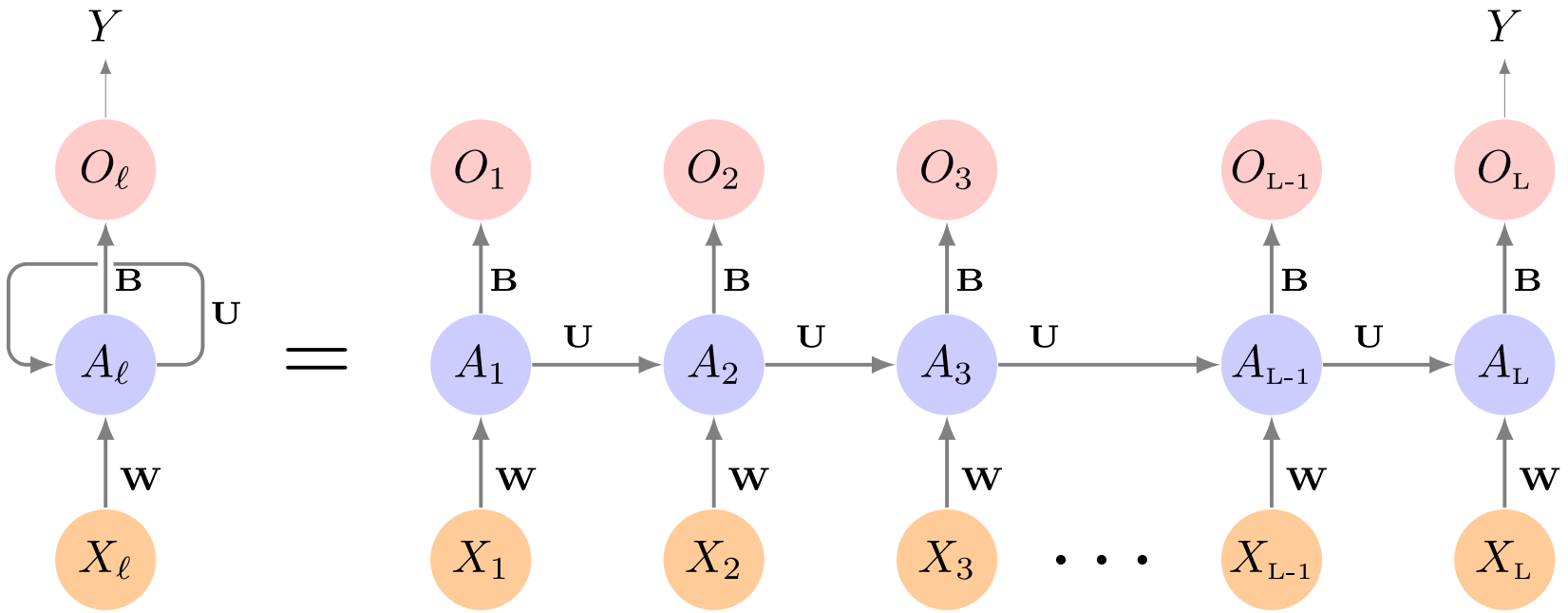
> A recurrence relation is an equation that expresses each element of a sequence as a function of the preceding ones. More precisely, in the case where only the immediately preceding element is involved, a recurrence relation has the form
>
> $$u_n = \psi(n, u_{n-1}) \quad \text{for} \quad n > 0.$$

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: James et al (2022), An Introduction to Statistical Learning, 2nd edition, Figure 10.12.

A SimpleRNN cell

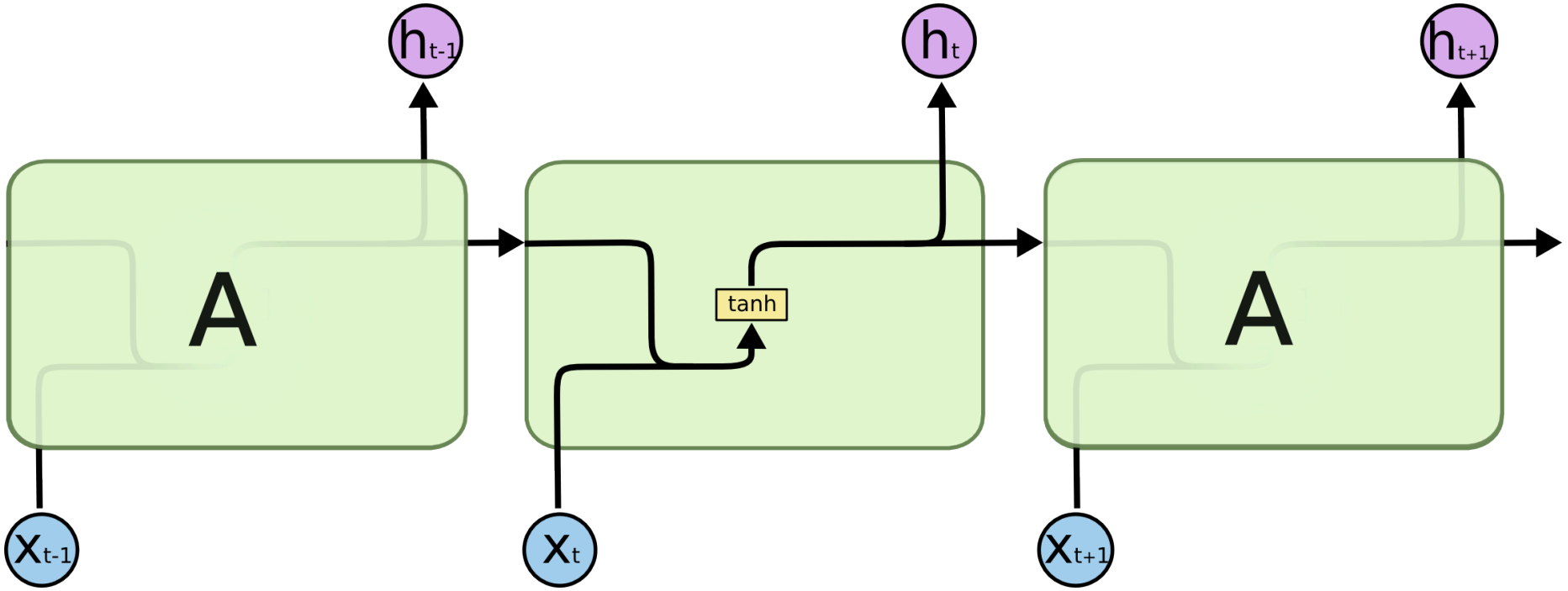
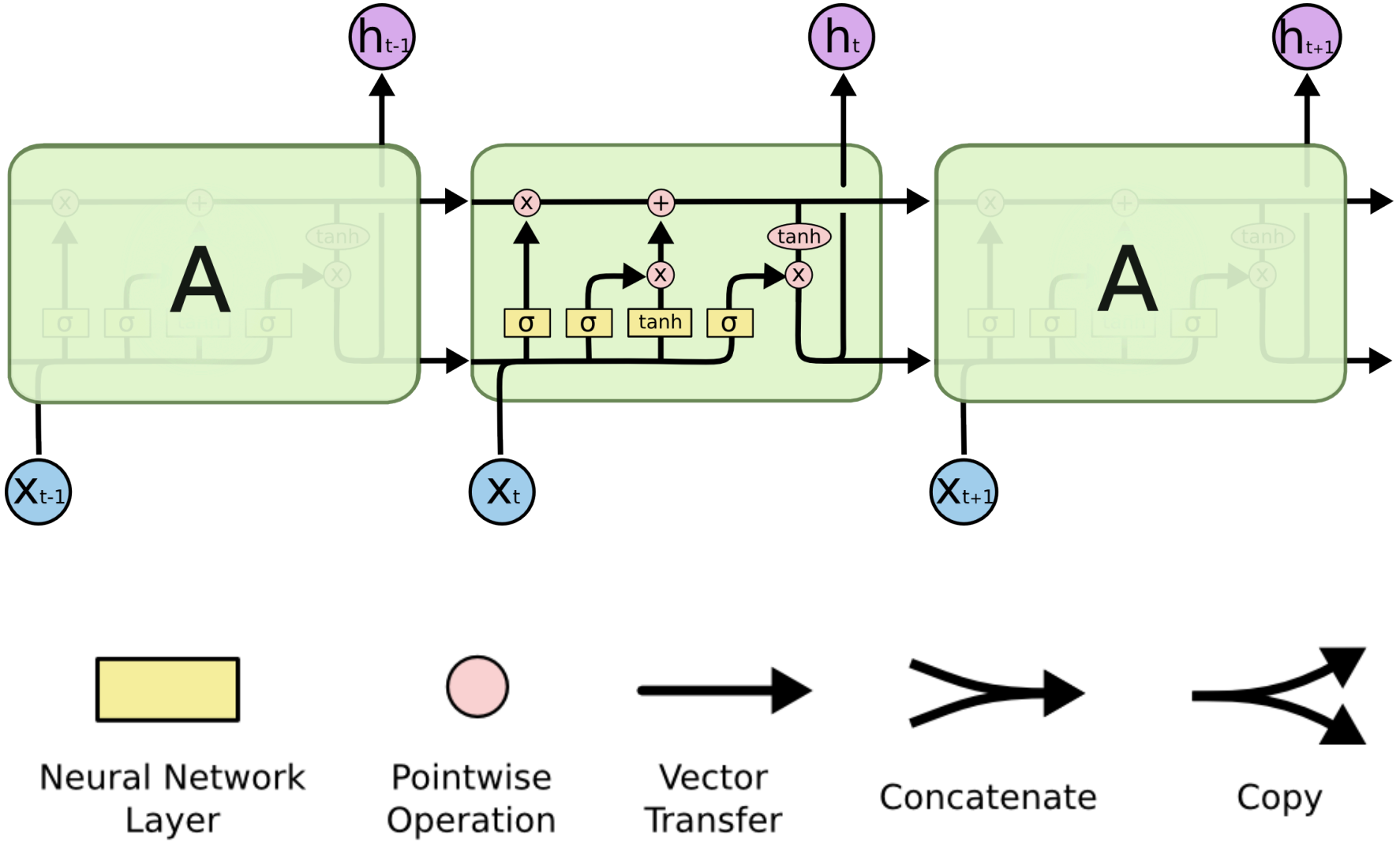


Diagram of a SimpleRNN cell.

All the outputs before the final one are often discarded.

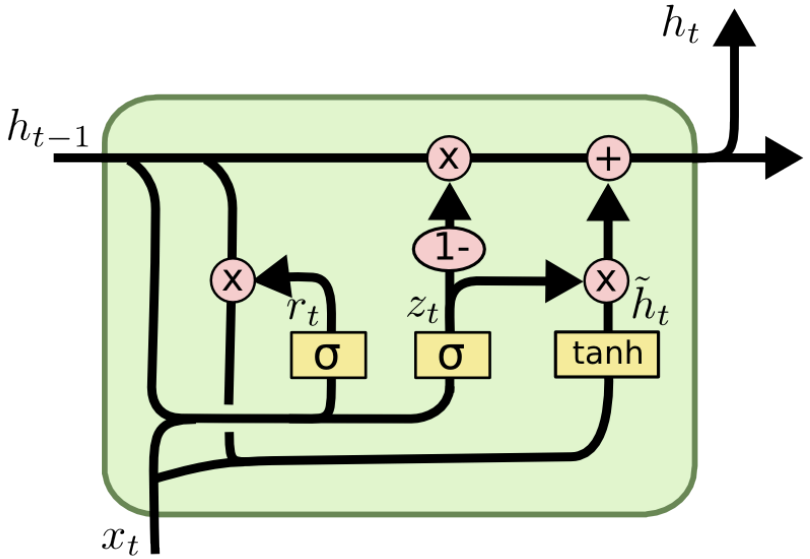
Source: Christopher Olah (2015), Understanding LSTM Networks, Colah's Blog.

LSTM internals



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

GRU internals



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

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([
        Reshape((-1, 1)),
        SimpleRNN(64, activation="tanh"),
        Dense(1, "softplus")])
model.compile(optimizer="adam", loss="mean_squared_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"

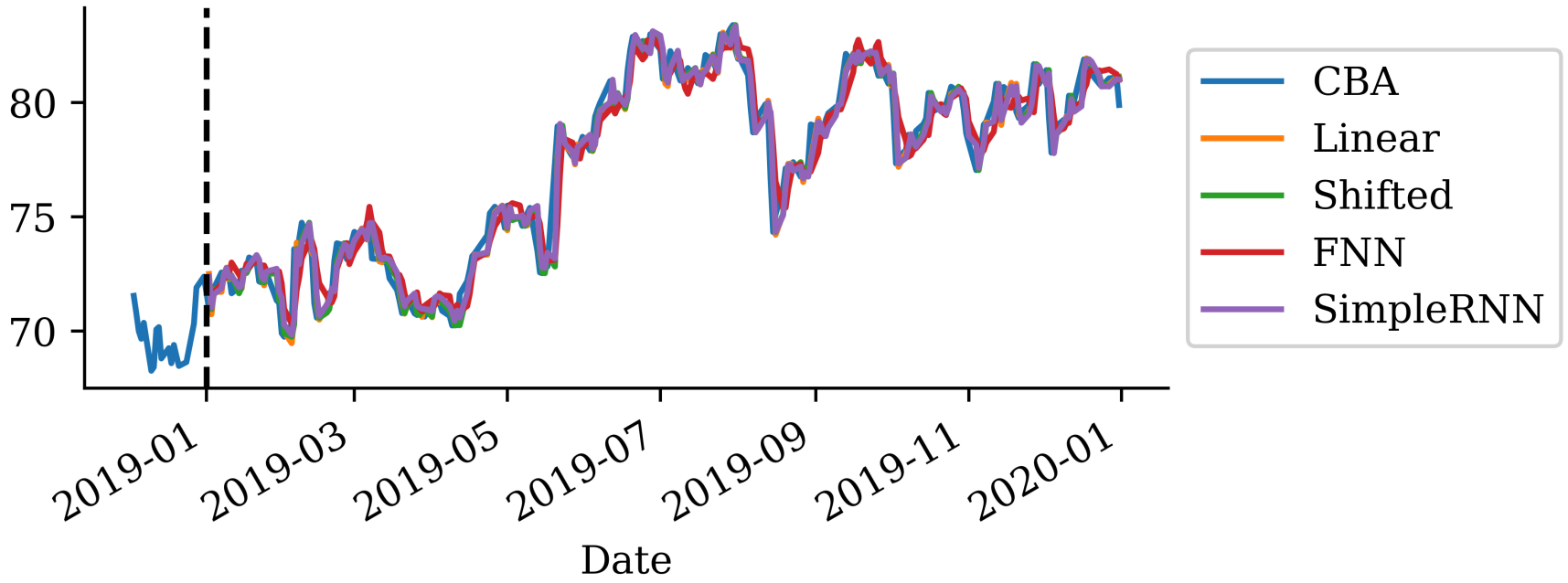
Layer (type)	Output Shape	Param #
reshape (Reshape)	(32, 40, 1)	0
simple_rnn (SimpleRNN)	(32, 64)	4,224
dense_2 (Dense)	(32, 1)	65

Total params: 4,291 (16.77 KB)
Trainable params: 4,289 (16.75 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 2 (12.00 B)

UNSW SYDNEY

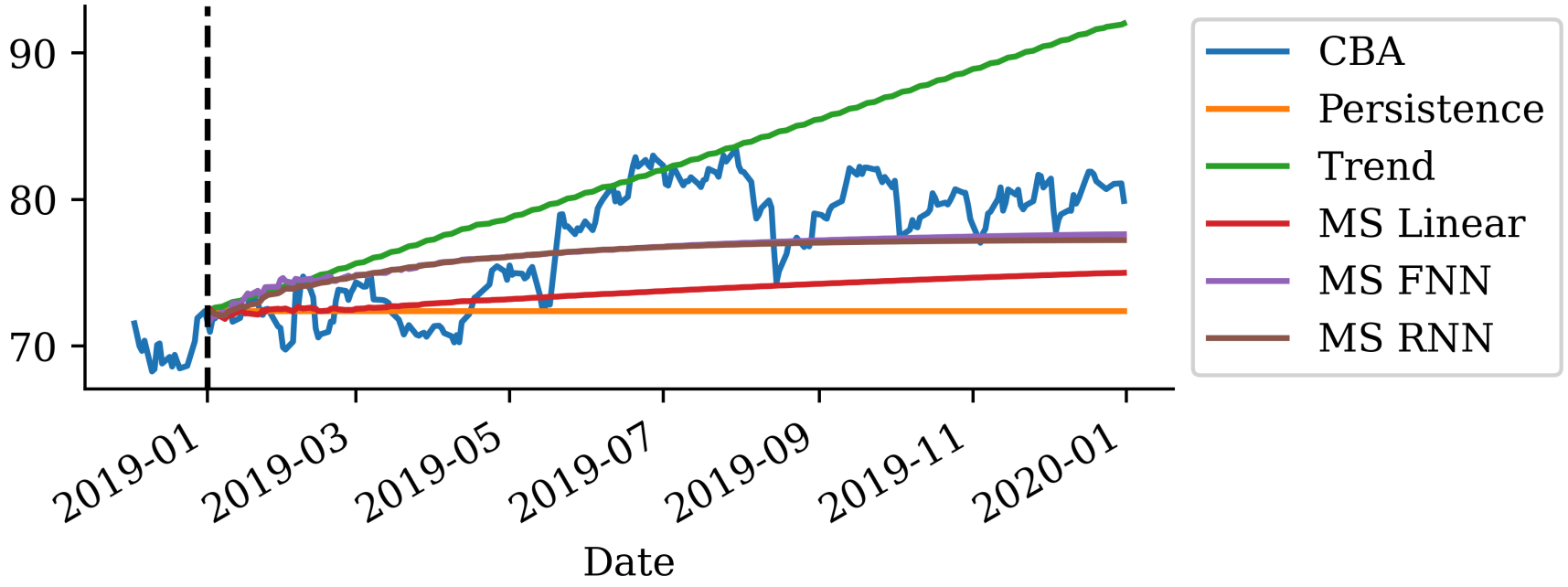
Forecast and plot

```
y_pred = model.predict(X_val.to_numpy(), verbose=0)
stock.loc[X_val.index, "SimpleRNN"] = 100 * y_pred
```



Multi-step forecasts

```
1 rnn_forecast = autoregressive_forecast(model, X_val, True)
2 stock.loc[rnn_forecast.index, "MS RNN"] = 100 * rnn_forecast
```



Metrics

One-step-ahead forecasts:

```
rnn_mse = 100**2 * mean_squared_error(y_val, y_pred)
linear_mse, shifted_mse, nn_mse, rnn_mse
```

```
(0.6329105517812197, 0.6367221343873524, 1.044511310373669, 0.6444509354197702)
```

Multi-step-ahead forecasts:

```
multi_step_rnn_mse = 100**2 * mean_squared_error(y_val, rnn_forecast)
persistence_mse, trend_mse, multi_step_linear_mse, multi_step_fnn_mse, multi_step_rnn_ms
```

```
(39.54629367588932,
 37.87104674064297,
 23.847003791127374,
 10.150589118262795,
 10.583806822953129)
```

UNSW SYDNEY

GRU

```
from keras.layers import GRU

model = Sequential([Reshape((-1, 1)),
        GRU(16, activation="tanh"),
        Dense(1, "softplus")])
model.compile(optimizer="adam", loss="mean_squared_error")
```

```
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"

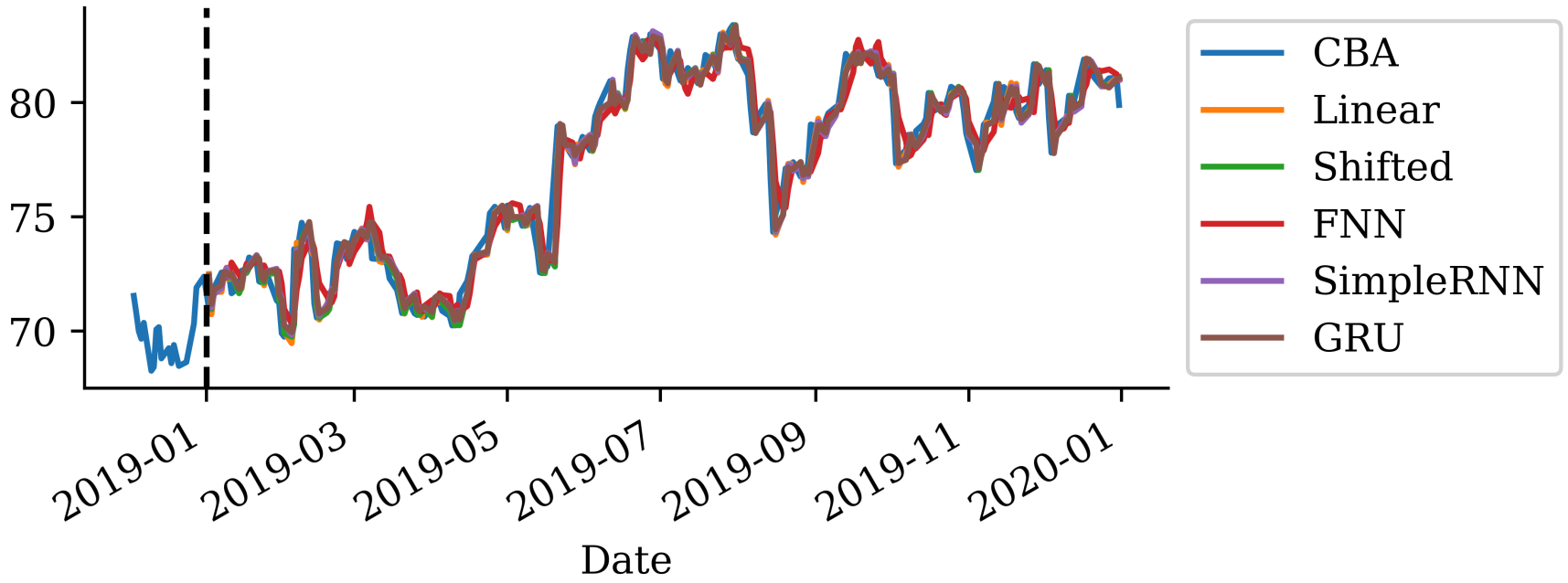
Layer (type)	Output Shape	Param #
reshape_1 (Reshape)	(32, 40, 1)	0
gru (GRU)	(32, 16)	912
dense_3 (Dense)	(32, 1)	17

Total params: 931 (3.64 KB)
Trainable params: 929 (3.63 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 2 (12.00 B)

UNSW SYDNEY

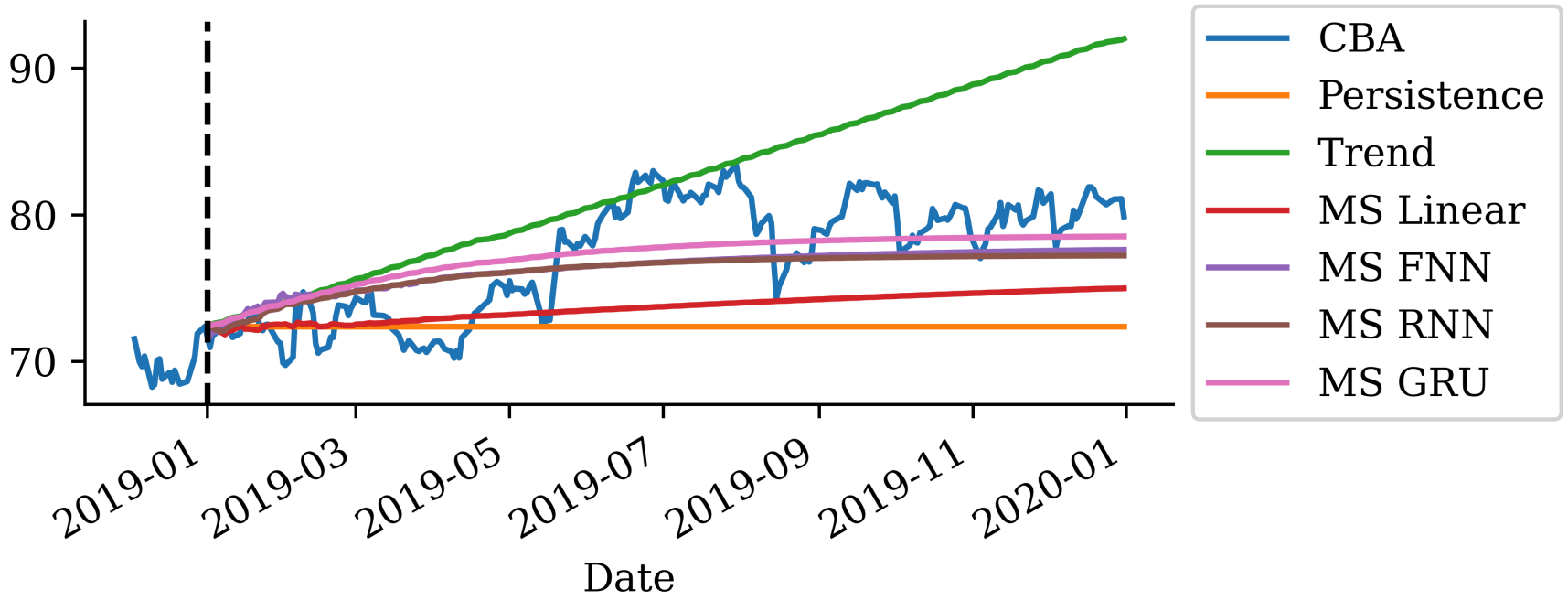
Forecast and plot

```
y_pred = model.predict(X_val, verbose=0)
stock.loc[X_val.index, "GRU"] = 100 * y_pred
```



Multi-step forecasts

```
gru_forecast = autoregressive_forecast(model, X_val, True)
stock.loc[gru_forecast.index, "MS GRU"] = 100 * gru_forecast
```



Metrics

One-step-ahead forecasts:

```
gru_mse = 100**2 * mean_squared_error(y_val, y_pred)
linear_mse, shifted_mse, nn_mse, rnn_mse, gru_mse
```

```
(0.6329105517812197,
 0.6367221343873524,
 1.044511310373669,
 0.6444509354197702,
 0.6390277225953965)
```

Multi-step-ahead forecasts:

```
multi_step_gru_mse = 100**2 * mean_squared_error(y_val, gru_forecast)
persistence_mse, trend_mse, multi_step_linear_mse, multi_step_fnn_mse, multi_step_rnn_ms
```

```
(39.54629367588932,
 37.87104674064297,
 23.847003791127374,
 10.150589118262795,
 10.583806822953129,
 8.111263804249768)
```

UNSW SYDNEY

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 $\boldsymbol{y}_0 = \mathbf{0}$, 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

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

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, :]`.

UNSW SYDNEY

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)
```

UNSW SYDNEY

Keras' SimpleRNN

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

```
1 from keras.layers import SimpleRNN
2
3 random.seed(1234)
4 model = Sequential([SimpleRNN(output_size, activation="sigmoid")])
5 model.compile(loss="binary_crossentropy", metrics=["accuracy"])
6
7 hist = model.fit(X, y, epochs=500, verbose=False)
8 model.evaluate(X, y, verbose=False)
```

```
[3.148719310760498, 0.5]
```

The predicted probabilities on the training set are:

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

```
array([[0.97],
       [1.  ],
       [1.  ],
       [1.  ]], dtype=float32)
```

SimpleRNN weights

```
model.get_weights()
```

```
[array([[0.68],
        [0.2 ]], dtype=float32),
 array([[0.49]], dtype=float32),
 array([-0.51], 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
```

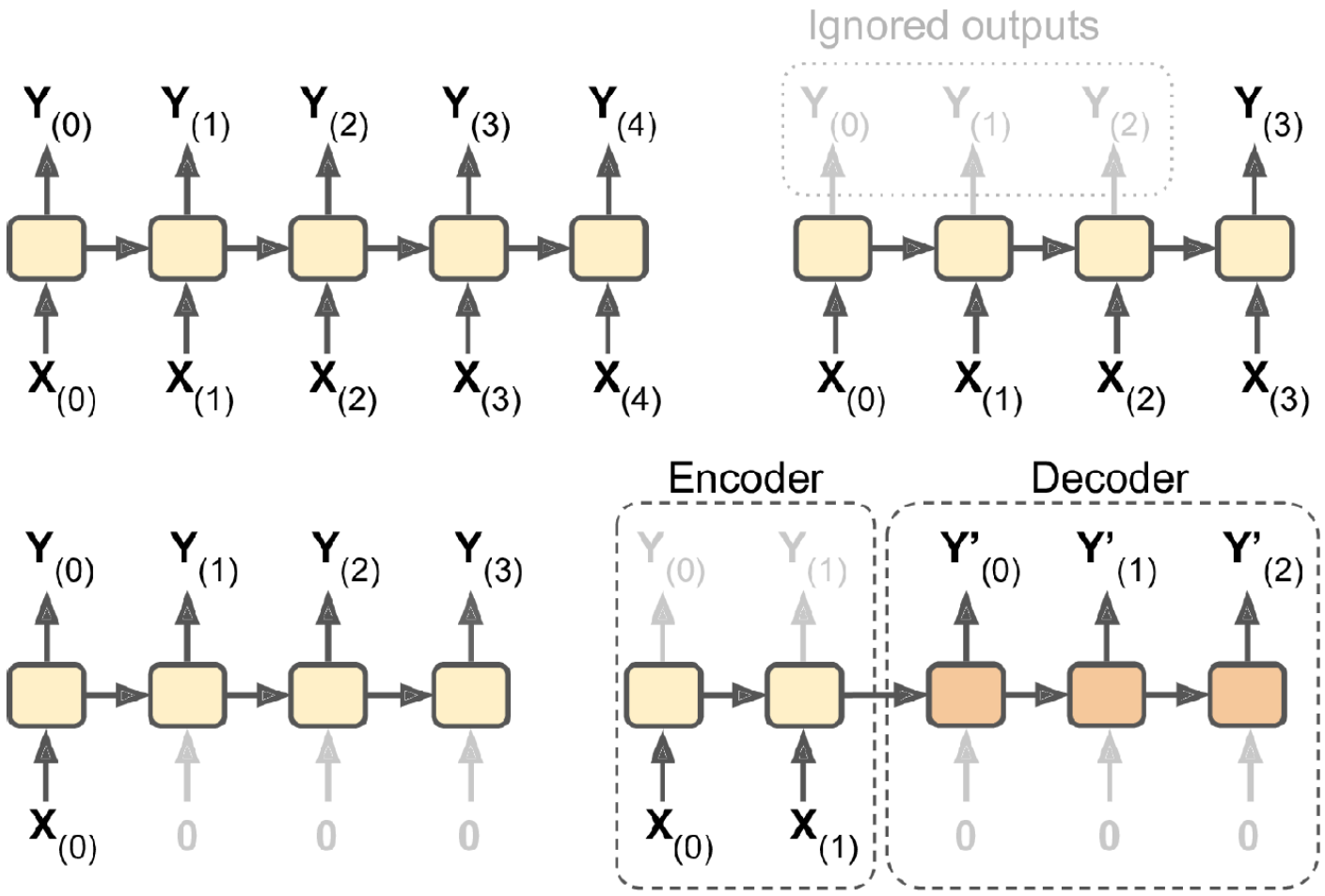
```
array([[0.97],
       [1.  ],
       [1.  ],
       [1.  ]], dtype=float32)
```

UNSW SYDNEY

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



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

Source: Aurélien Géron (2019), *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition, Chapter 15.

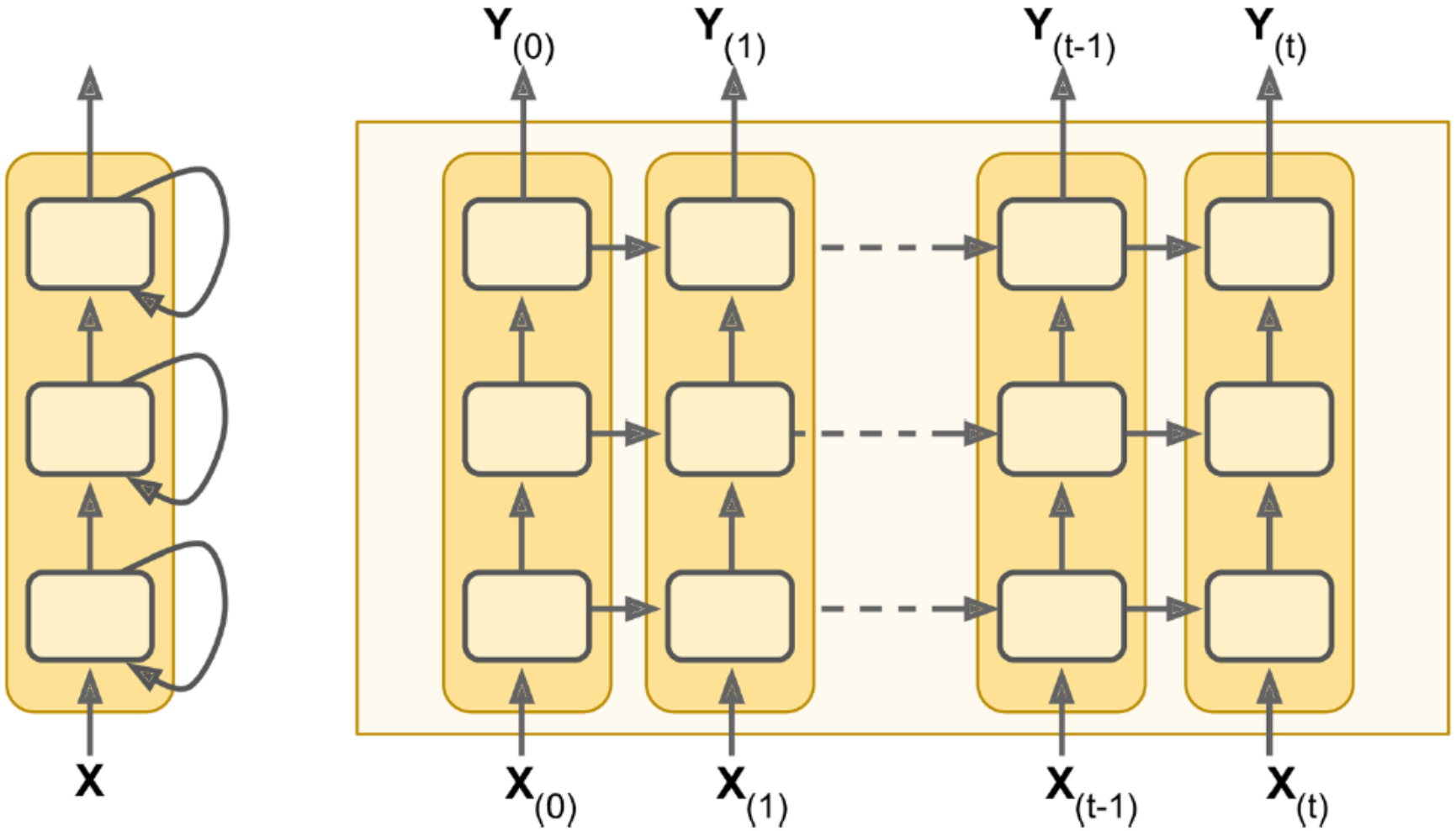
Input and output sequences

- 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.



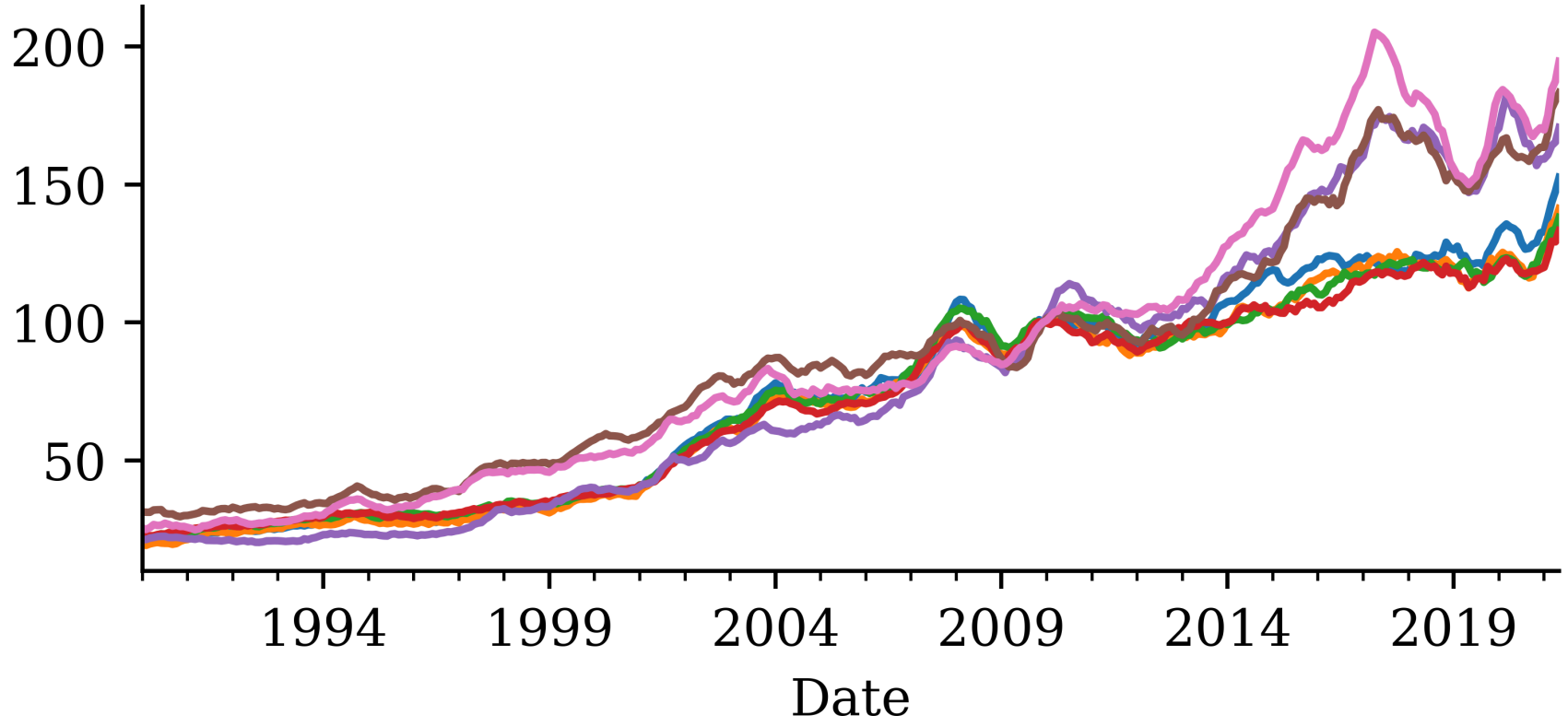
Deep RNN unrolled through time.

Source: Aurélien Géron (2019), *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition, Chapter 15.

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



Note

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

Percentage changes

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

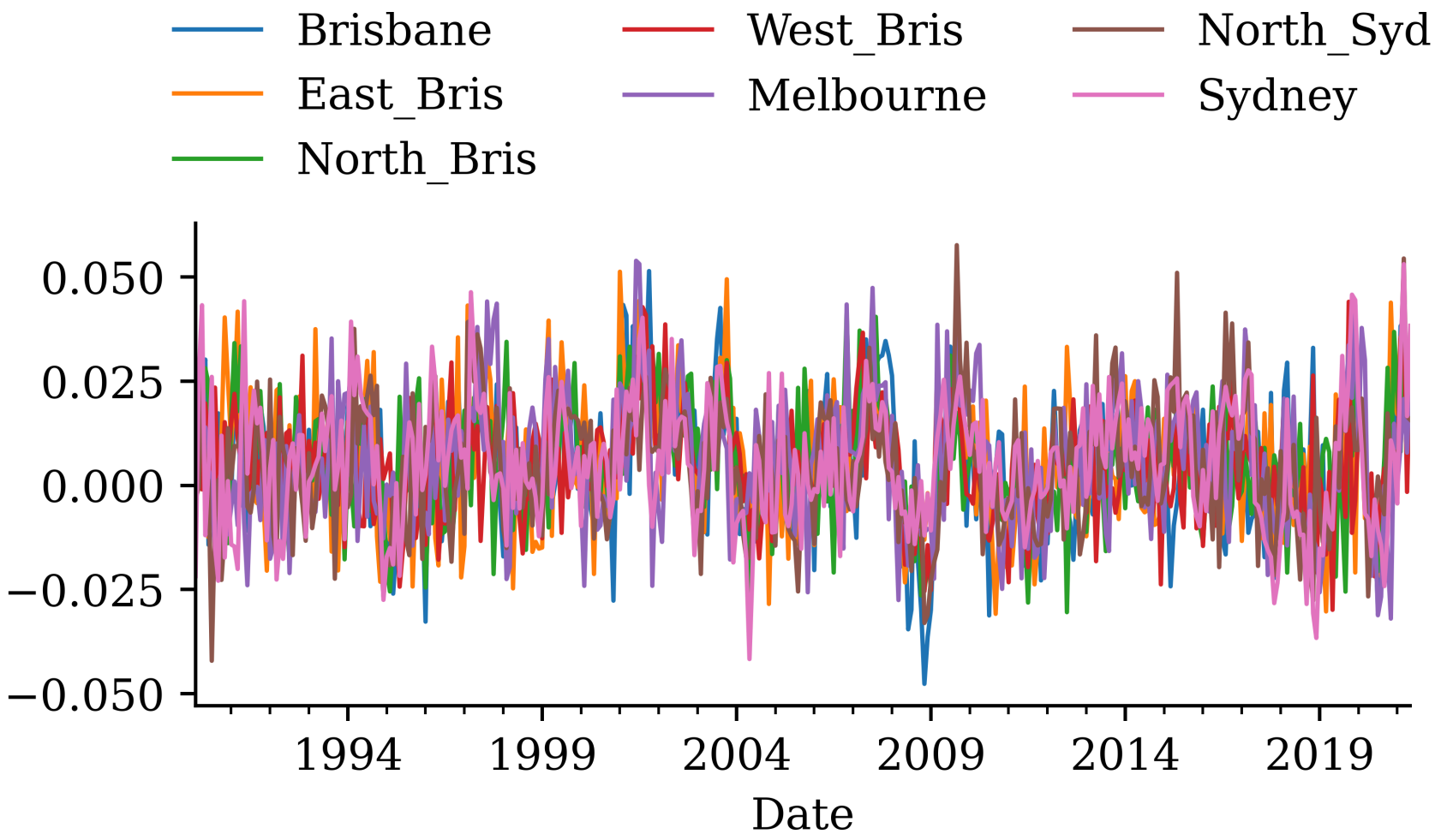
Date	Brisbane	East_Bris	North_Bris	West_Bris	Melbourne	North_Syd	Sydney
1990-02-28	0.03	-0.01	0.01	0.01	0.00	-0.00	-0.02
1990-03-31	0.01	0.03	0.01	0.01	0.02	-0.00	0.03
1990-04-30	0.02	0.02	0.01	-0.00	0.01	0.03	0.04
...	...	...	...	...	...	...	...
2021-03-31	0.04	0.04	0.03	0.04	0.02	0.05	0.05
2021-04-30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

UNSW SYDNEY

Percentage changes

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



UNSW SYDNEY

The size of the changes

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

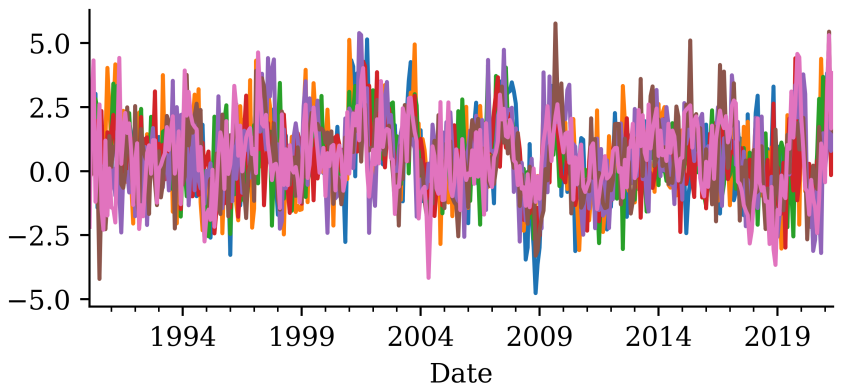
```
Brisbane      0.005496
East_Bris     0.005416
North_Bris    0.005024
West_Bris     0.004842
Melbourne     0.005677
North_Syd     0.004819
Sydney        0.005526
dtype: float64
```

```
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
```

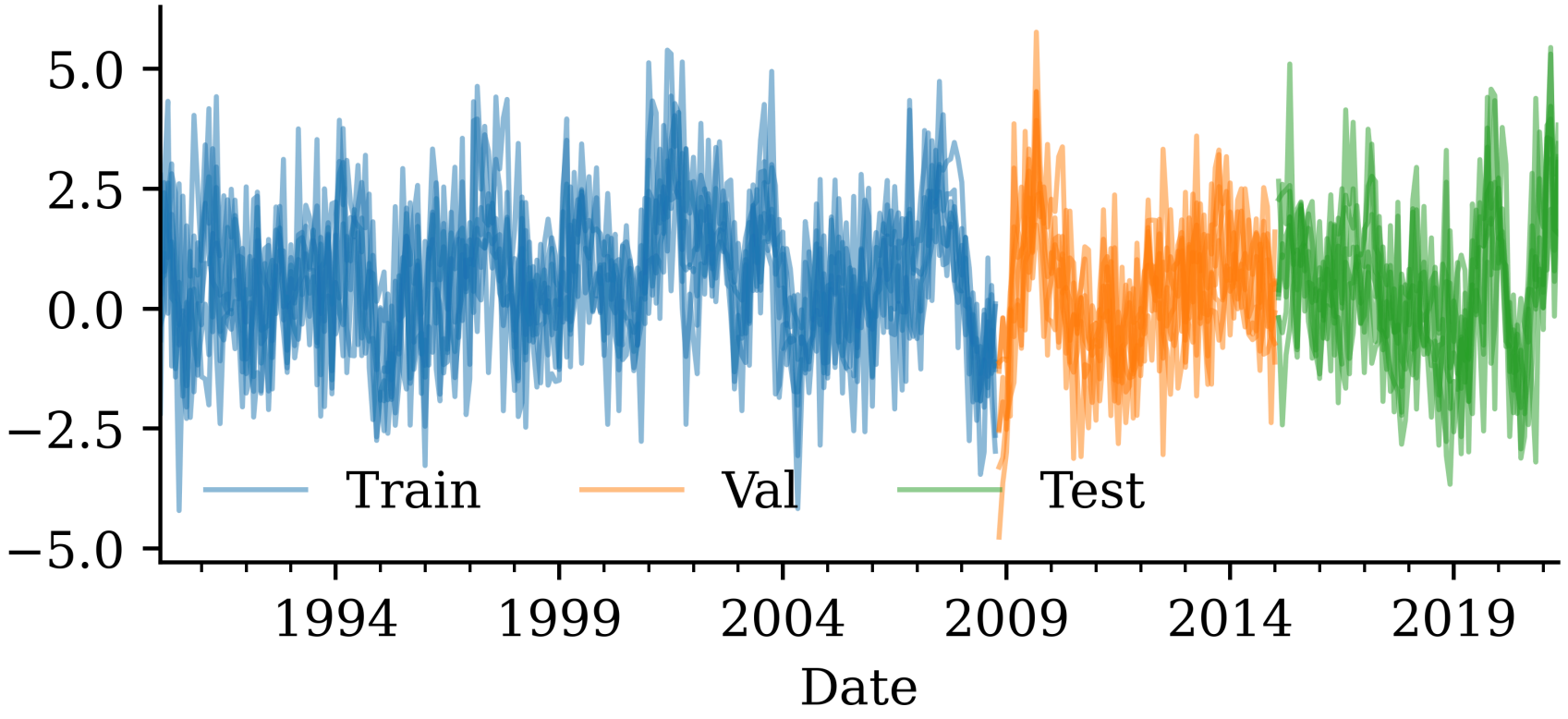
```
1 changes.plot(legend=False);
```



Split *without* shuffling

```
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

Keras has a built-in method for converting a time series into subsequences/chunks.

```python
from keras.utils import timeseries_dataset_from_array

integers = range(10)
dummy_dataset = timeseries_dataset_from_array(
    data=integers[:-3],
    targets=integers[3:],
    sequence_length=3,
    batch_size=2,
)

for inputs, targets in dummy_dataset:
    for i in range(inputs.shape[0]):
        print([int(x) for x in inputs[i]], int(targets[i]))
```

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

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 dataset objects

```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"]

train_ds = timeseries_dataset_from_array
    changes[:-delay],
    targets=target_suburb[delay:],
    sequence_length=seq_length,
    end_index=num_train,
)
```

```python
val_ds = timeseries_dataset_from_array(
    changes[:-delay],
    targets=target_suburb[delay:],
    sequence_length=seq_length,
    start_index=num_train,
    end_index=num_train + num_val,
)
```

```python
test_ds = timeseries_dataset_from_array(
    changes[:-delay],
    targets=target_suburb[delay:],
    sequence_length=seq_length,
    start_index=num_train + num_val,
)
```

UNSW SYDNEY

Converting `Dataset` to numpy

The `Dataset` object can be handed to Keras directly, but if we really need a numpy array, we can run:

```
1 X_train = np.concatenate(list(train_ds.map(lambda x, y: x)))
2 y_train = np.concatenate(list(train_ds.map(lambda x, y: y)))
```

The shape of our training set is now:

```
1 X_train.shape
```

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

```
1 y_train.shape
```

```
(220,)
```

Converting the rest to numpy arrays:

```
1 X_val = np.concatenate(list(val_ds.map(lambda x, y: x)))
2 y_val = np.concatenate(list(val_ds.map(lambda x, y: y)))
3 X_test = np.concatenate(list(test_ds.map(lambda x, y: x)))
4 y_test = np.concatenate(list(test_ds.map(lambda x, y: y)))
```

UNSW SYDNEY

A dense network

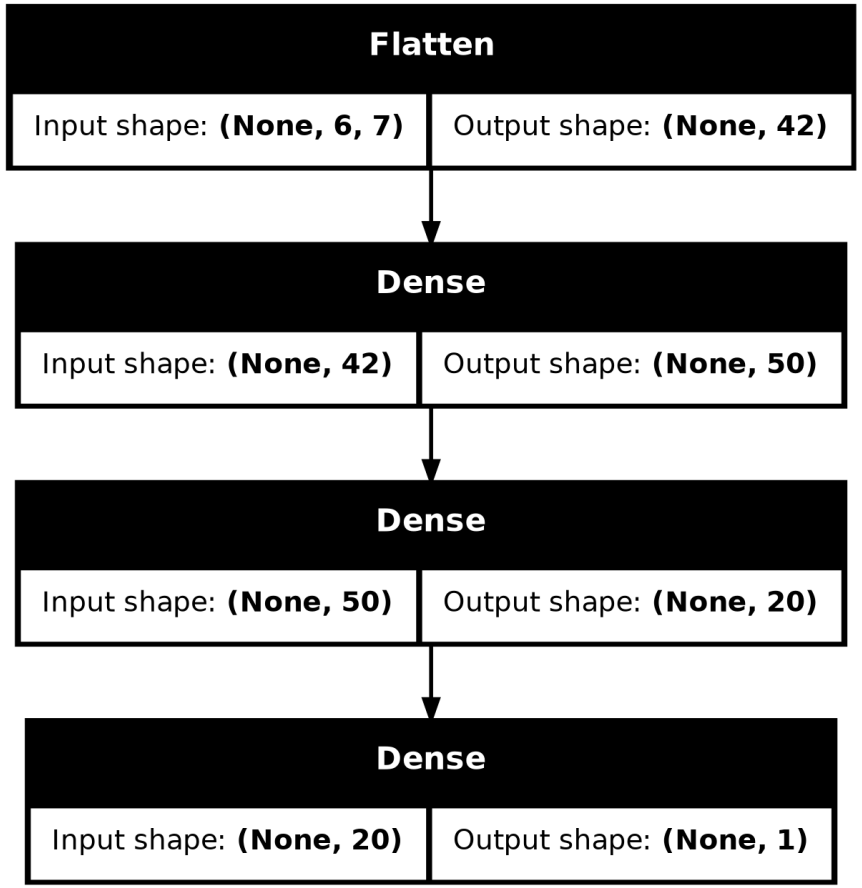
```
from keras.layers import Input, Flatten
random.seed(1)
model_dense = Sequential([
    Input((seq_length, num_ts)),
    Flatten(),
    Dense(50, activation="leaky_relu"),
    Dense(20, activation="leaky_relu"),
    Dense(1, activation="linear")
])
model_dense.compile(loss="mse", optimizer="adam")
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 57: early stopping
Restoring model weights from the end of the best epoch: 7.
CPU times: user 3.81 s, sys: 476 ms, total: 4.28 s
Wall time: 3.82 s
```

UNSW SYDNEY

Plot the model

```
1 from keras.utils import plot_model
2
3 plot_model(model_dense, show_shapes=True)
```

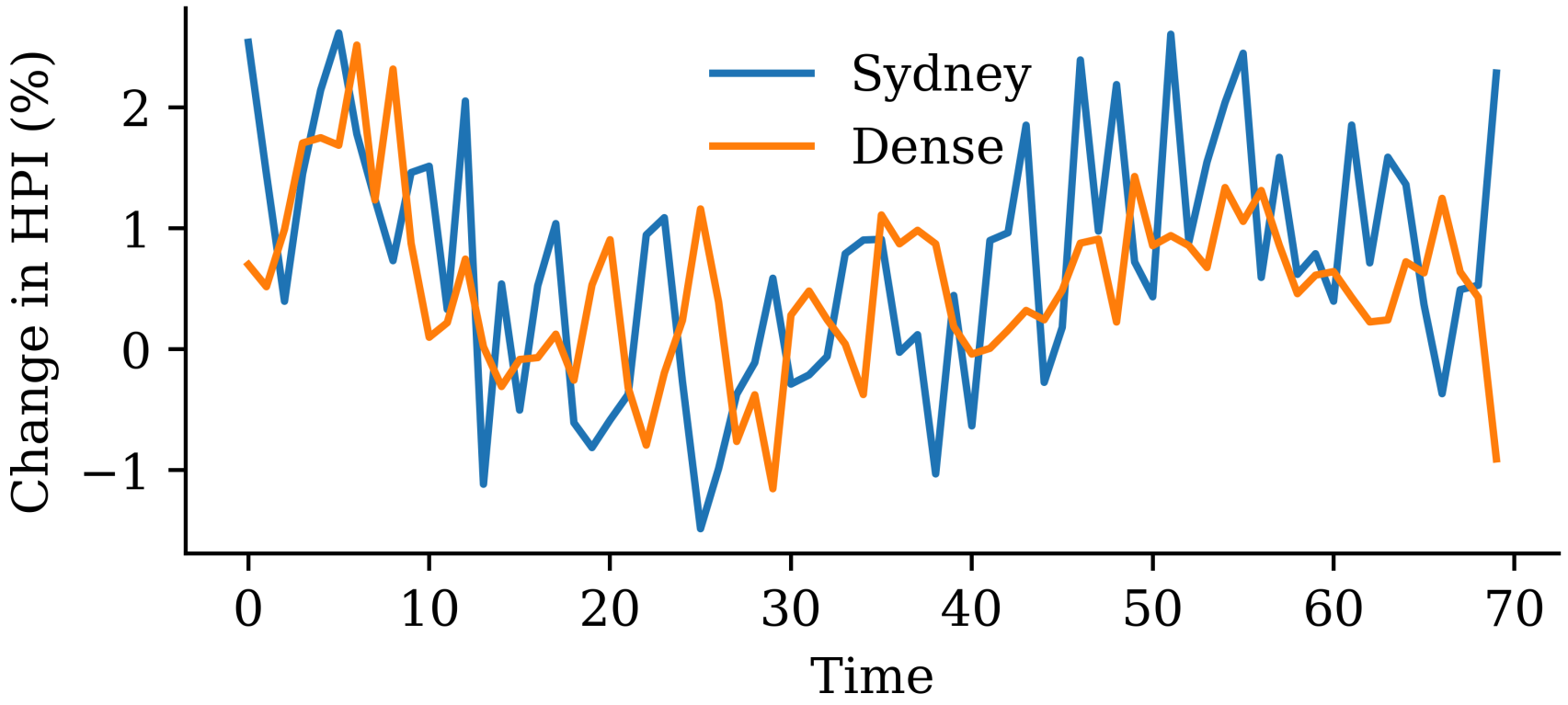


UNSW SYDNEY

Assess the fits

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

```
1.1644608974456787
```



A `SimpleRNN` layer

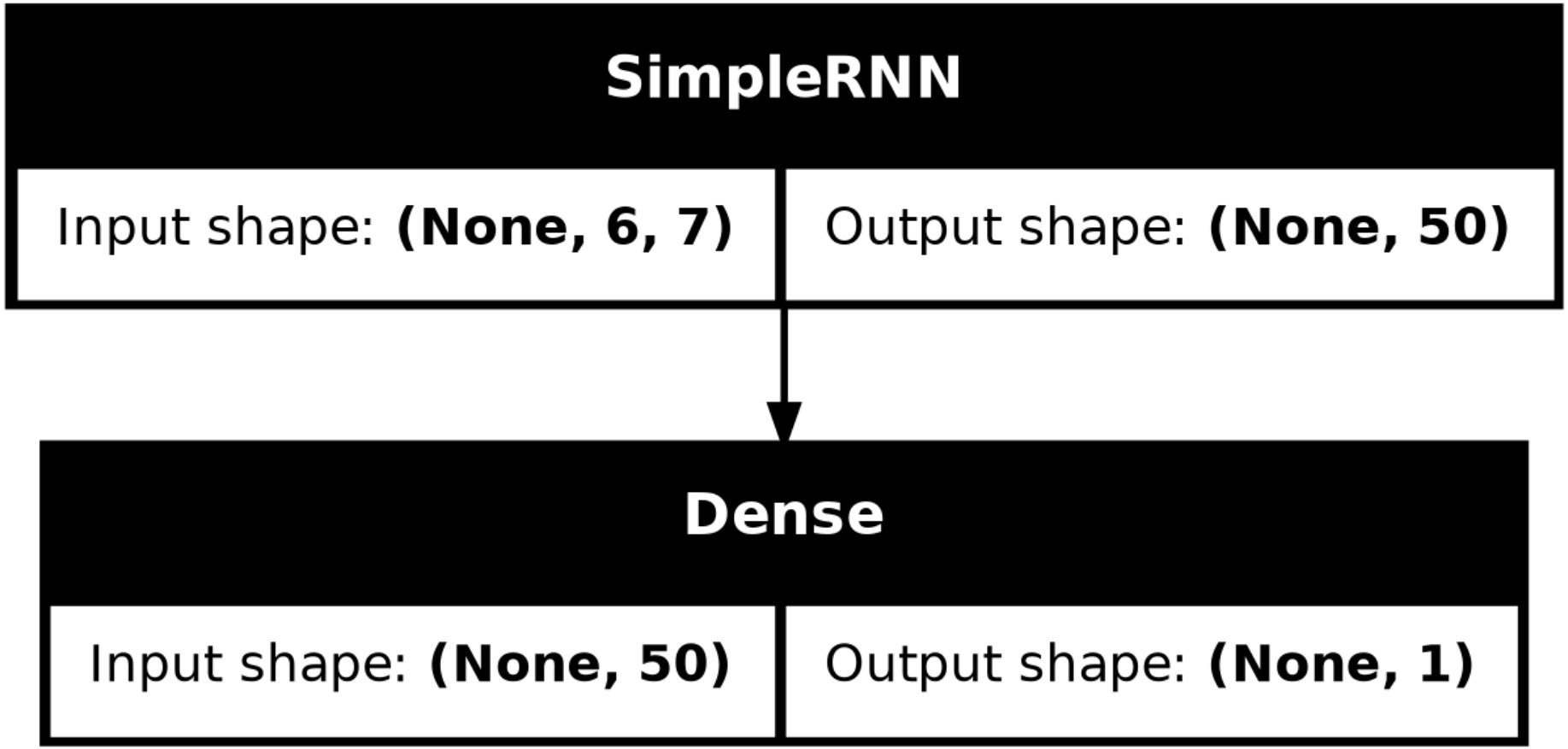
```
random.seed(1)

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 62: early stopping
Restoring model weights from the end of the best epoch: 12.
CPU times: user 4.6 s, sys: 552 ms, total: 5.15 s
Wall time: 4.71 s
```

UNSW SYDNEY

Plot the model

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



Assess the fits

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

```
1.2508097887039185
```

```
WARNING:tensorflow:5 out of the last 259 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at
0×779849a75620> triggered tf.function retracing. Tracing is expensive and the excessive
number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2)
passing tensors with different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2), @tf.function has
reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
```

```
WARNING:tensorflow:6 out of the last 261 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at
0×779849a75620> triggered tf.function retracing. Tracing is expensive and the excessive
number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2)
passing tensors with different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2), @tf.function has
reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
```

A LSTM layer

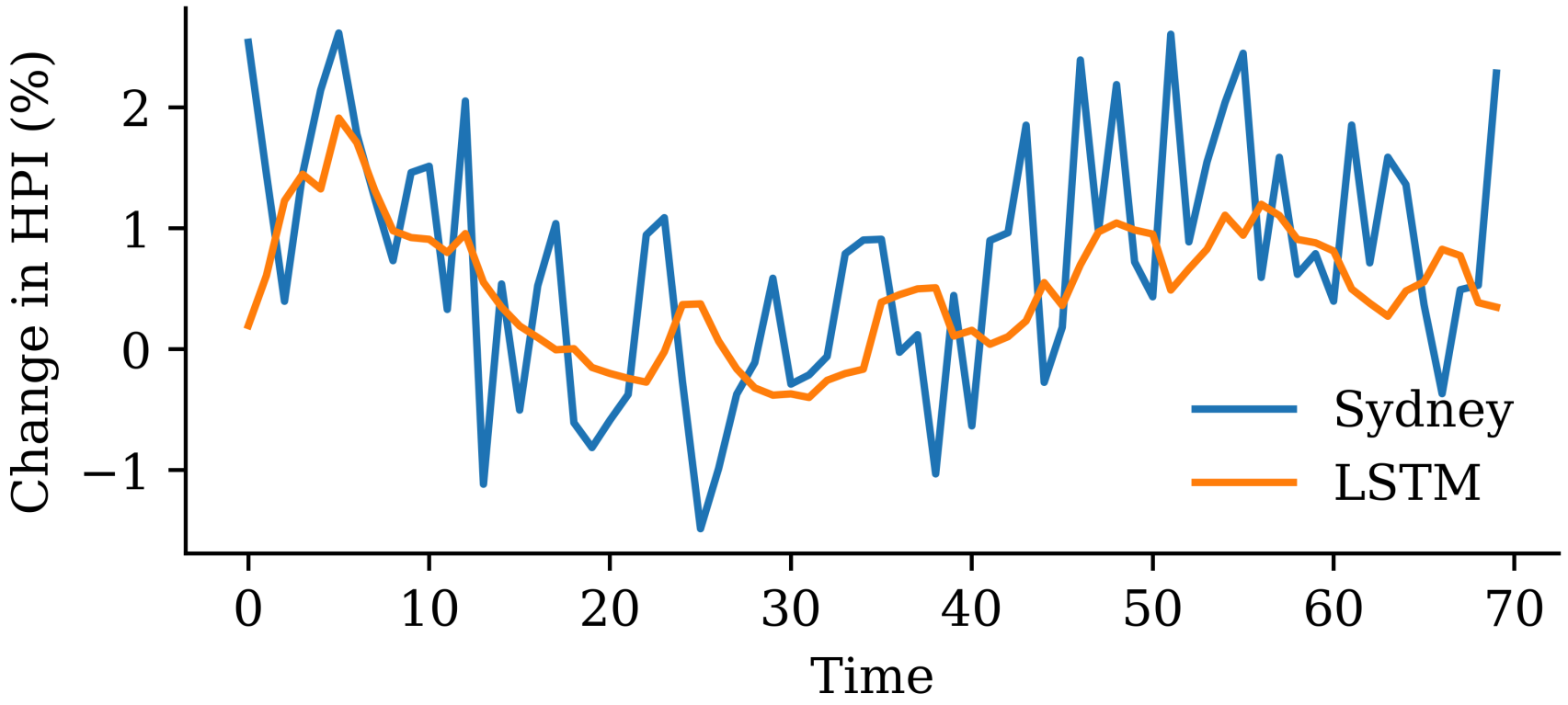
```
from keras.layers import LSTM

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 59: early stopping
Restoring model weights from the end of the best epoch: 9.
CPU times: user 4.92 s, sys: 586 ms, total: 5.51 s
Wall time: 4.6 s
```

Assess the fits

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

```
0.8353261947631836
```



A GRU layer

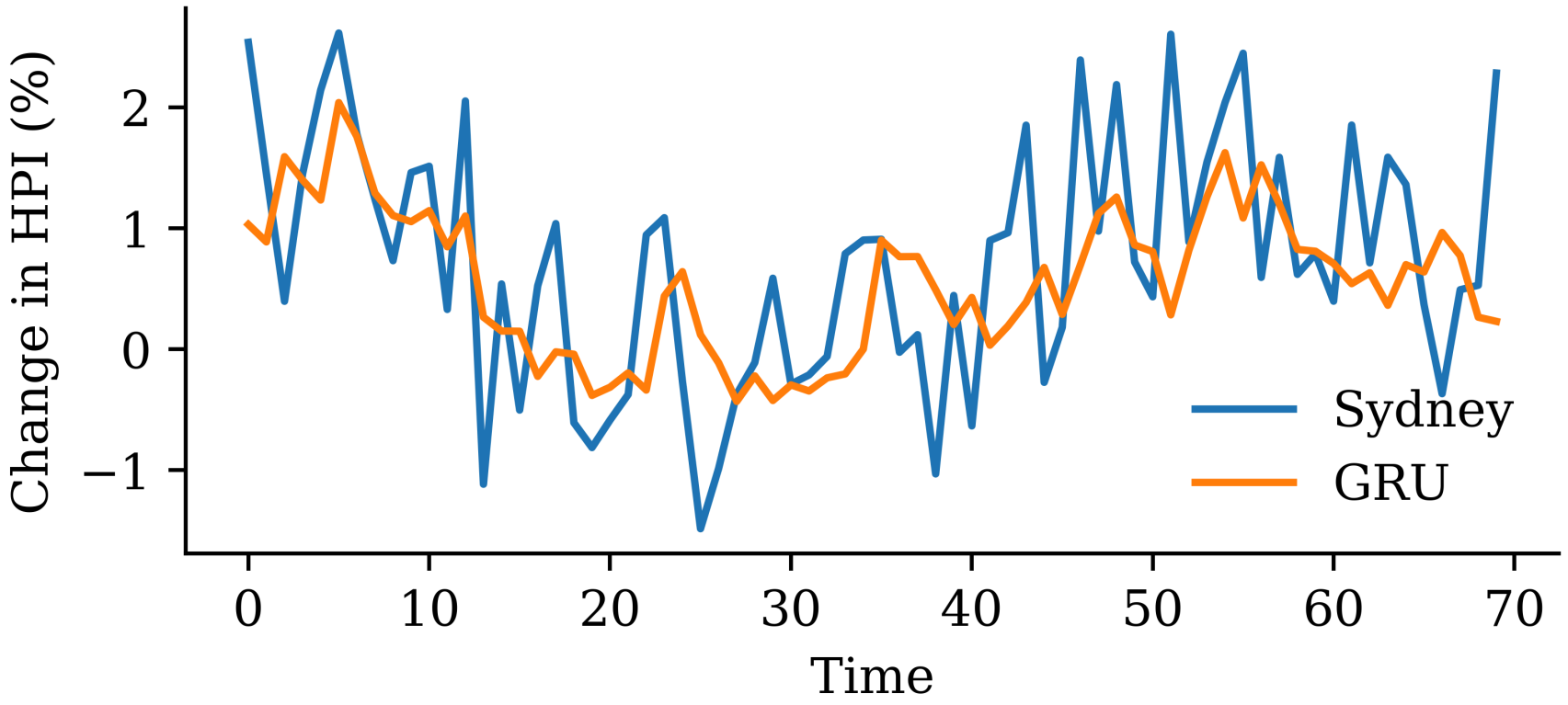
```
from keras.layers import GRU

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 57: early stopping
Restoring model weights from the end of the best epoch: 7.
CPU times: user 4.39 s, sys: 542 ms, total: 4.93 s
Wall time: 4.1 s
```

Assess the fits

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

```
0.7435099482536316
```



Two GRU layers

```
random.seed(1)

model_two_grus = Sequential([
    Input((seq_length, num_ts)),
    GRU(50, return_sequences=True),
    GRU(50),
    Dense(1, 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)
```

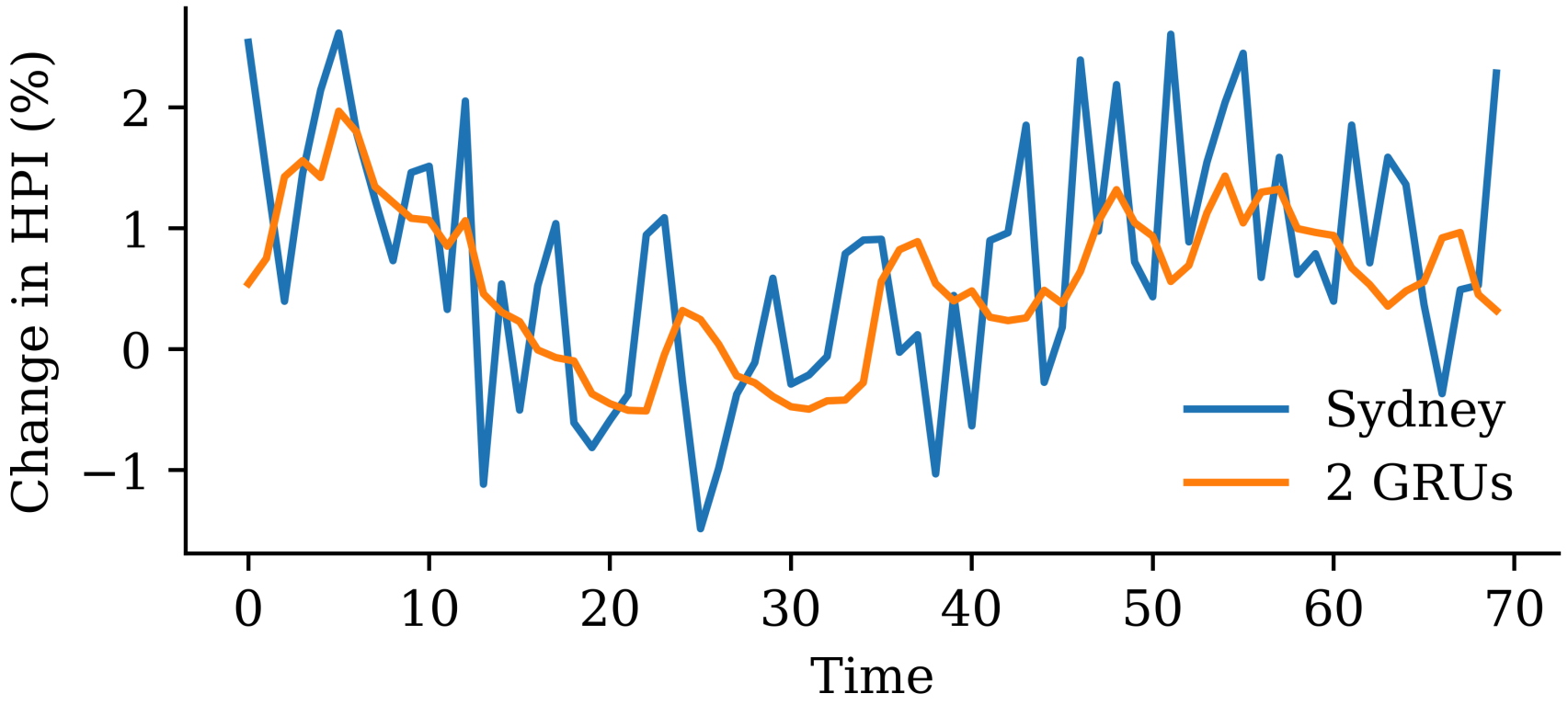
```
Epoch 56: early stopping
Restoring model weights from the end of the best epoch: 6.
CPU times: user 5.29 s, sys: 608 ms, total: 5.9 s
Wall time: 5 s
```

UNSW SYDNEY

Assess the fits

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

```
0.7989509105682373
```



Compare the models

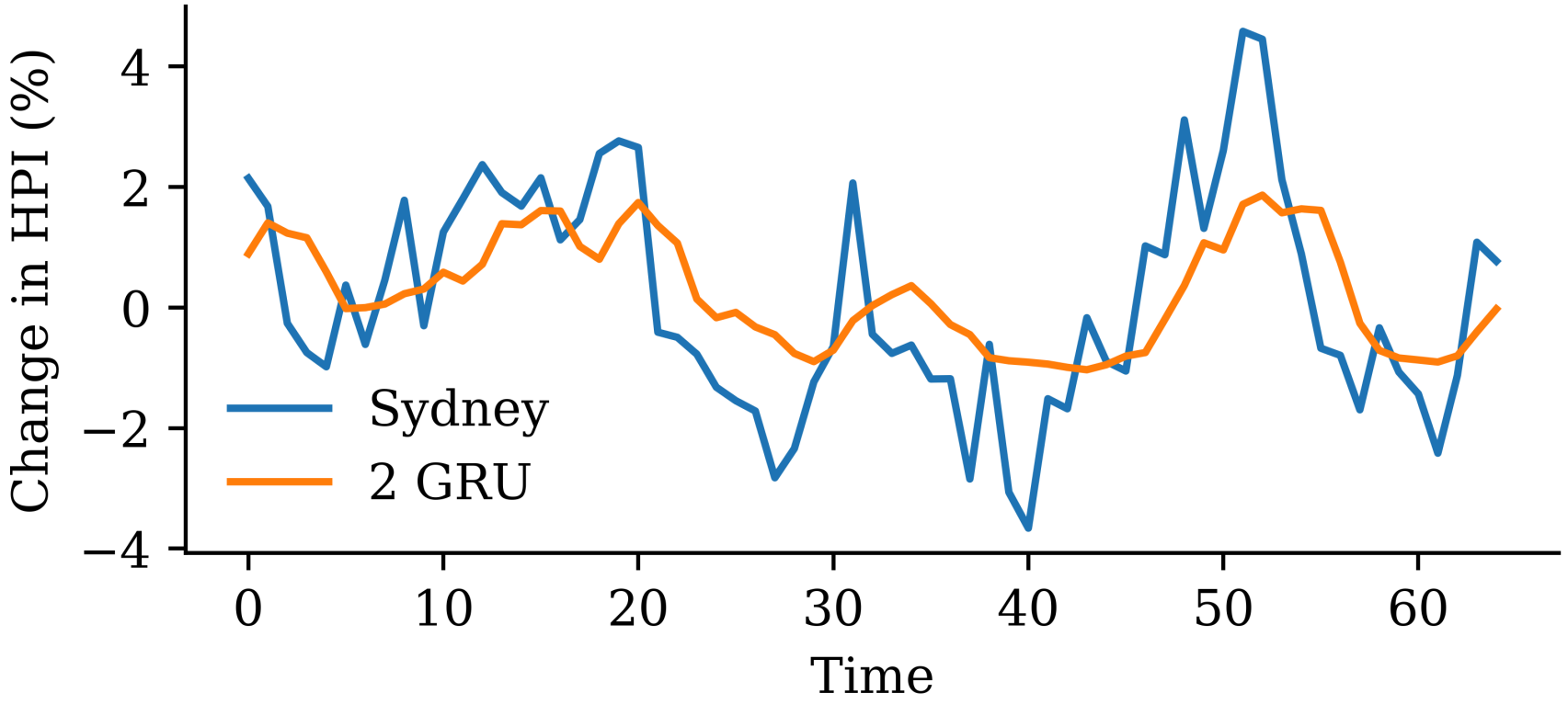
	Model	MSE
1	SimpleRNN	1.250810
0	Dense	1.164461
2	LSTM	0.835326
4	2 GRUs	0.798951
3	GRU	0.743510

The network with two GRU layers is the best.

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

```
1.855254888534546
```

Test set



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 dataset objects

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

```
train_ds = timeseries_dataset_from_array
    changes[:-delay],
    targets=changes[delay:],
    sequence_length=seq_length,
    end_index=num_train,
)
```

```
val_ds = timeseries_dataset_from_array(
    changes[:-delay],
    targets=changes[delay:],
    sequence_length=seq_length,
    start_index=num_train,
    end_index=num_train + num_val,
)
```

```
test_ds = timeseries_dataset_from_array(
    changes[:-delay],
    targets=changes[delay:],
    sequence_length=seq_length,
    start_index=num_train + num_val,
)
```

Converting `Dataset` to numpy

The shape of our training set is now:

```
1 X_train = np.concatenate(list(train_ds.map(lambda x, y: x)))
2 X_train.shape
```

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

```
1 y_train = np.concatenate(list(train_ds.map(lambda x, y: y)))
2 y_train.shape
```

```
(220, 7)
```

Converting the rest to numpy arrays:

```
1 X_val = np.concatenate(list(val_ds.map(lambda x, y: x)))
2 y_val = np.concatenate(list(val_ds.map(lambda x, y: y)))
3 X_test = np.concatenate(list(test_ds.map(lambda x, y: x)))
4 y_test = np.concatenate(list(test_ds.map(lambda x, y: y)))
```

A dense network

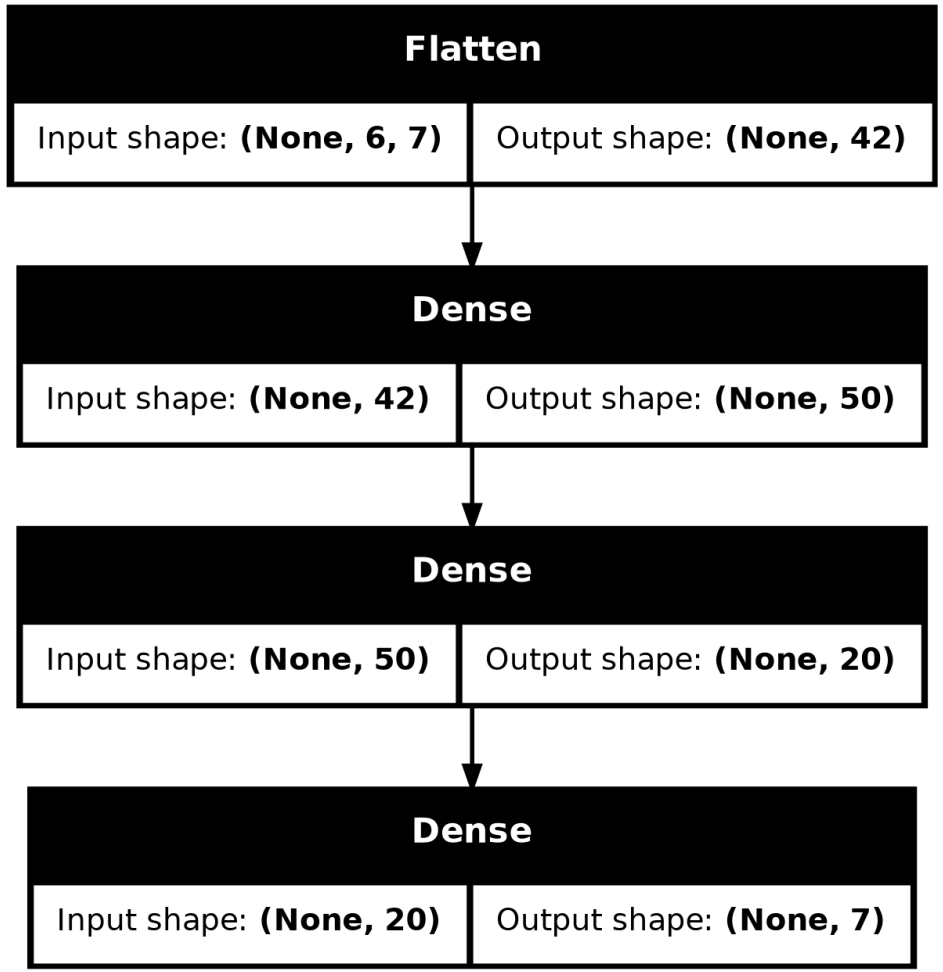
```
random.seed(1)
model_dense = Sequential([
    Input((seq_length, num_ts)),
    Flatten(),
    Dense(50, activation="leaky_relu"),
    Dense(20, activation="leaky_relu"),
    Dense(num_ts, activation="linear")
])
model_dense.compile(loss="mse", optimizer="adam")
print(f"This model has {model_dense.count_params()} 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 75: early stopping
Restoring model weights from the end of the best epoch: 25.
CPU times: user 5.17 s, sys: 599 ms, total: 5.77 s
Wall time: 5.09 s
```

UNSW SYDNEY

Plot the model

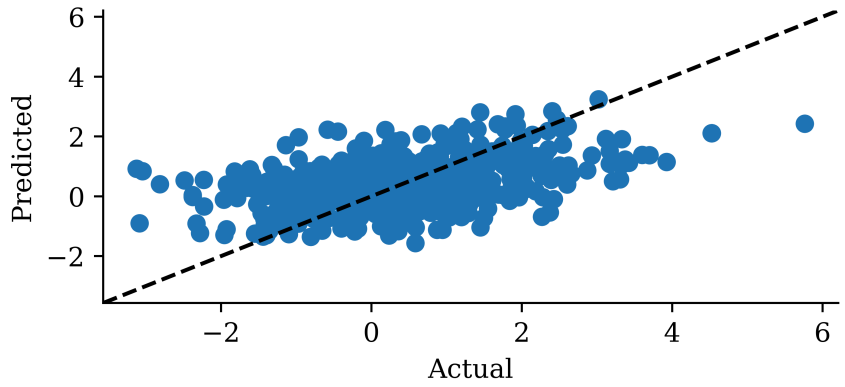
```
1 plot_model(model_dense, show_shapes=True)
```

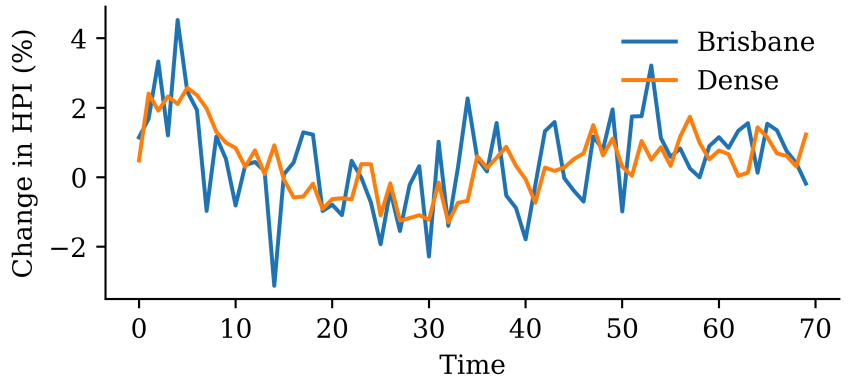


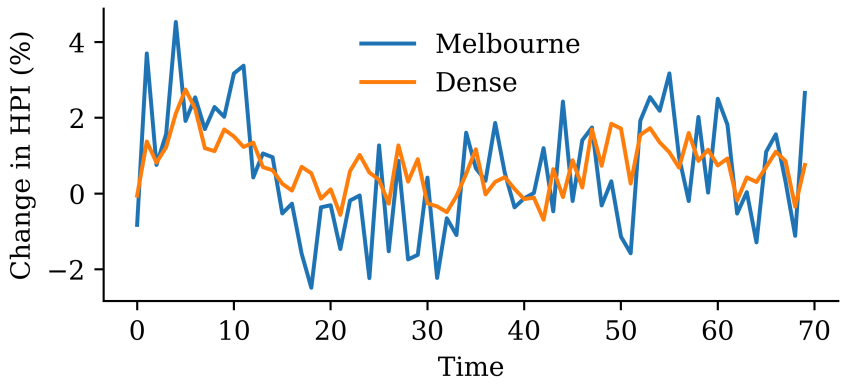
Assess the fits

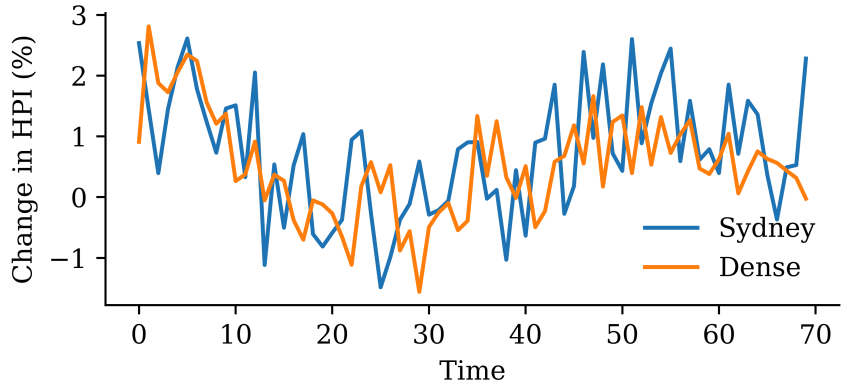
```
1 model_dense.evaluate(X_val, y_val, verbose=0)
```

```
1.4289127588272095
```









UNSW SYDNEY

A `SimpleRNN` layer

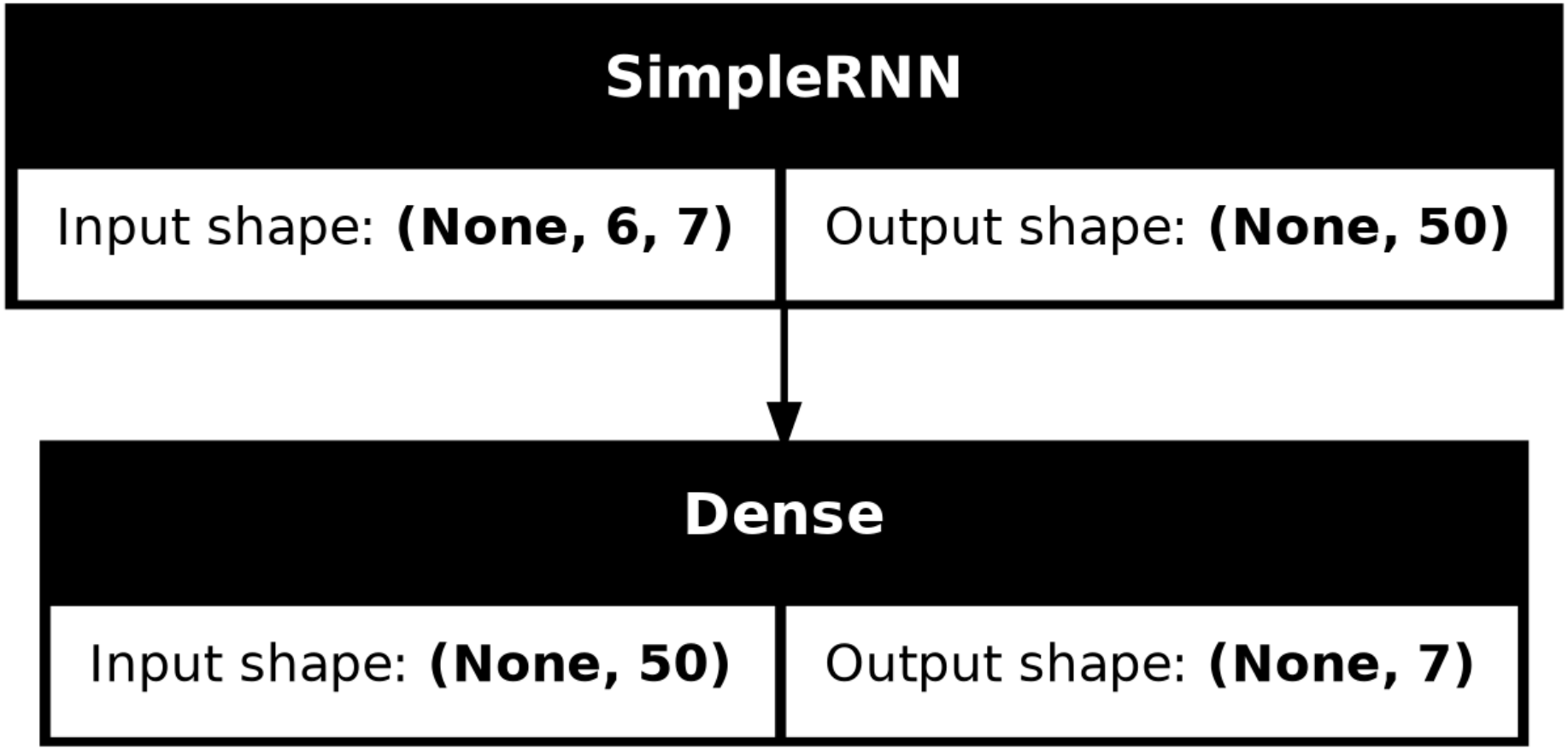
```
random.seed(1)

model_simple = Sequential([
    Input((seq_length, num_ts)),
    SimpleRNN(50),
    Dense(num_ts, 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 3257 parameters.
Epoch 70: early stopping
Restoring model weights from the end of the best epoch: 20.
CPU times: user 5.99 s, sys: 653 ms, total: 6.64 s
Wall time: 5.42 s
```

UNSW SYDNEY

Plot the model

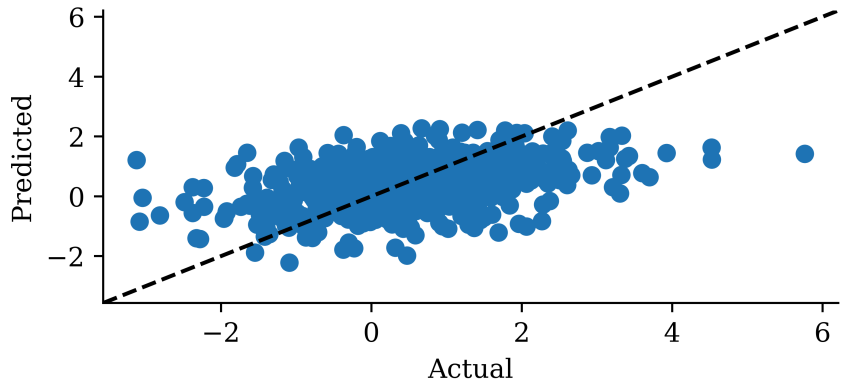
```
1 plot_model(model_simple, show_shapes=True)
```

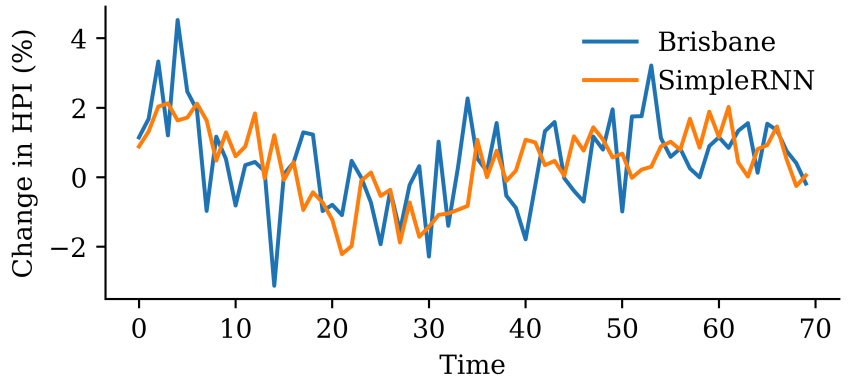


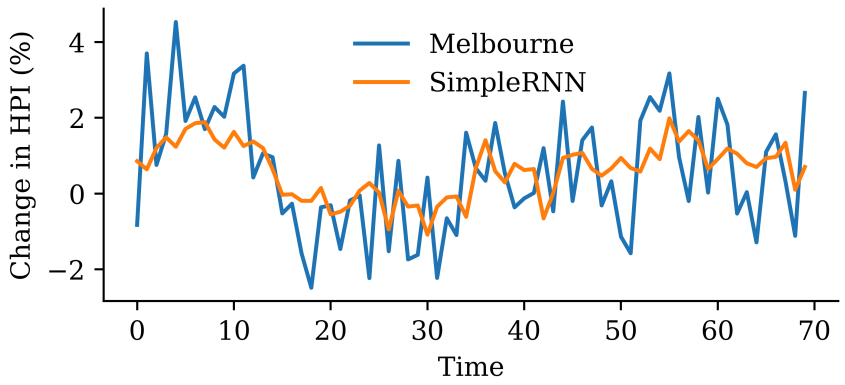
Assess the fits

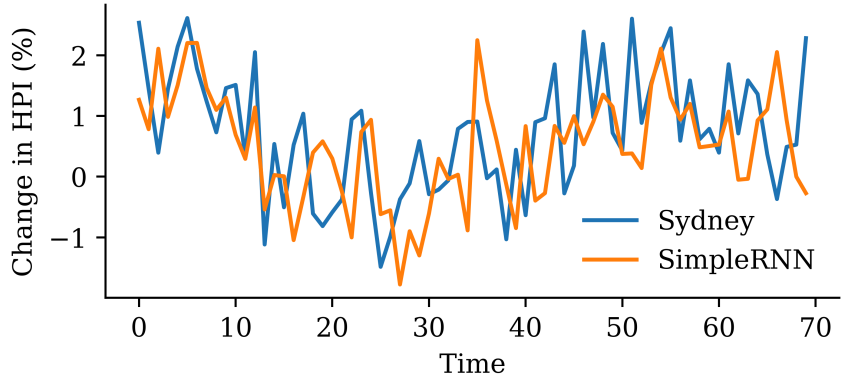
```
model_simple.evaluate(X_val, y_val, verbose=0)
```

```
1.4916425943374634
```









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);
```

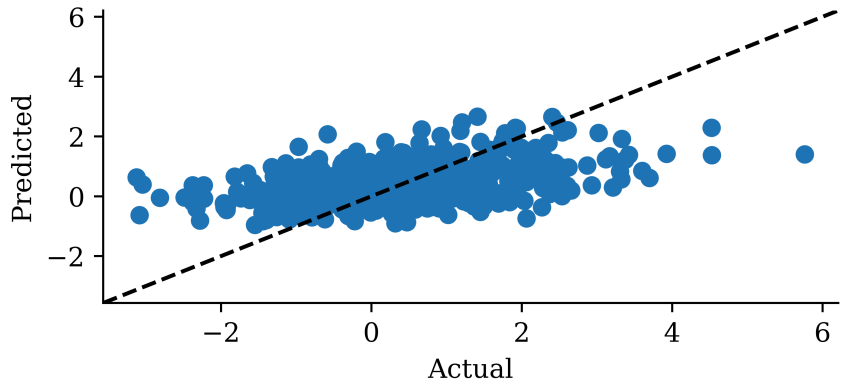
```
Epoch 74: early stopping
Restoring model weights from the end of the best epoch: 24.
CPU times: user 5.34 s, sys: 680 ms, total: 6.02 s
Wall time: 4.95 s
```

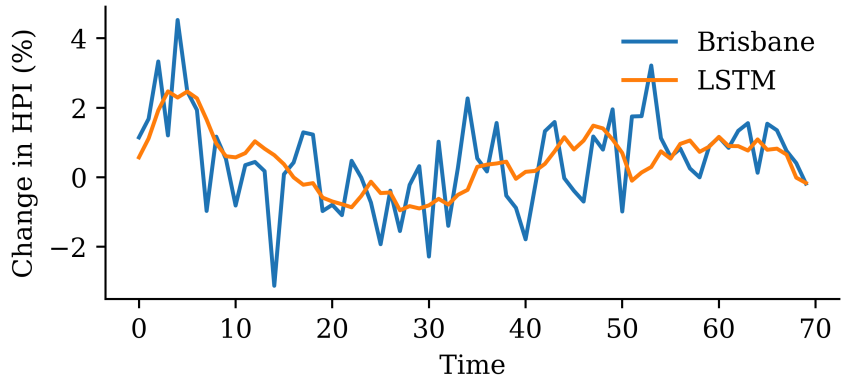
UNSW SYDNEY

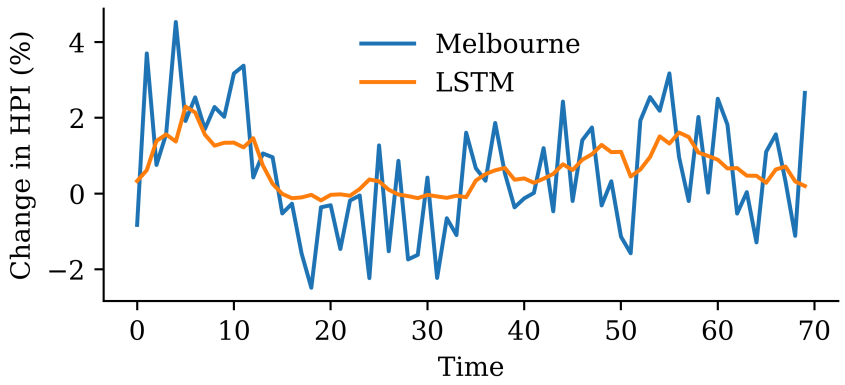
Assess the fits

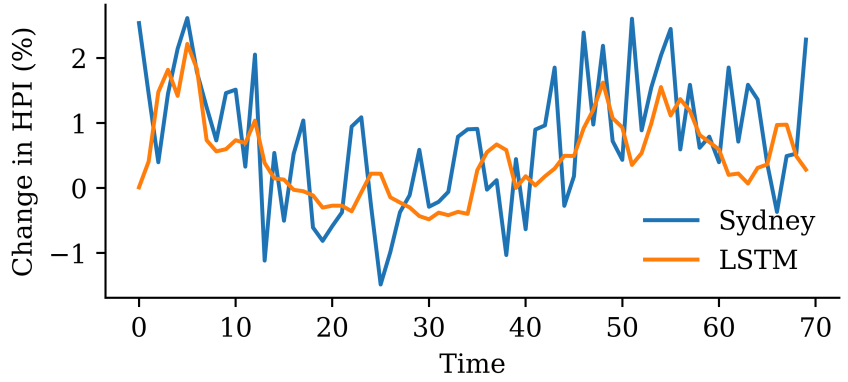
```
1 model_lstm.evaluate(X_val, y_val, verbose=0)
```

```
1.3311247825622559
```









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)
```

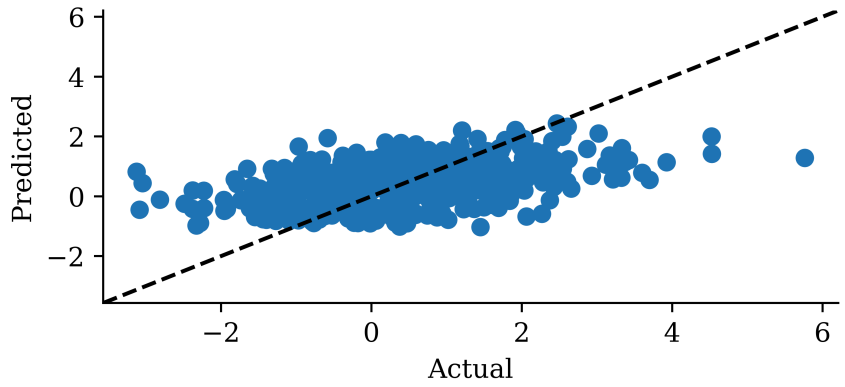
```
Epoch 70: early stopping
Restoring model weights from the end of the best epoch: 20.
CPU times: user 5.23 s, sys: 630 ms, total: 5.86 s
Wall time: 4.84 s
```

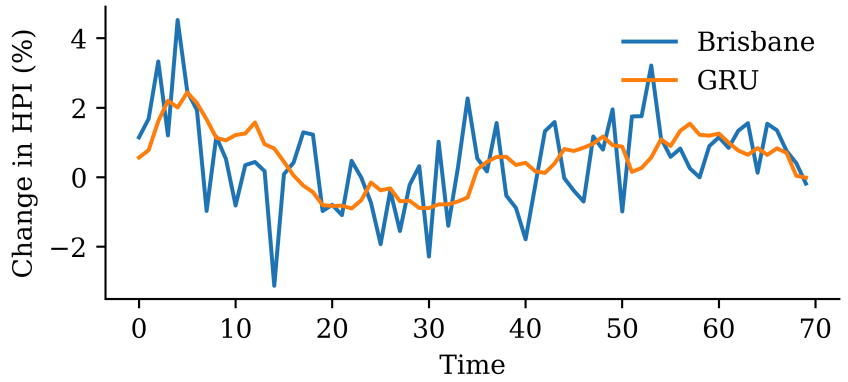
UNSW SYDNEY

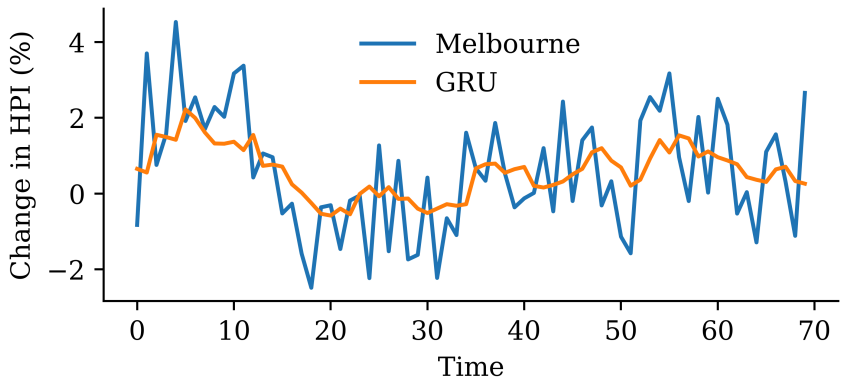
Assess the fits

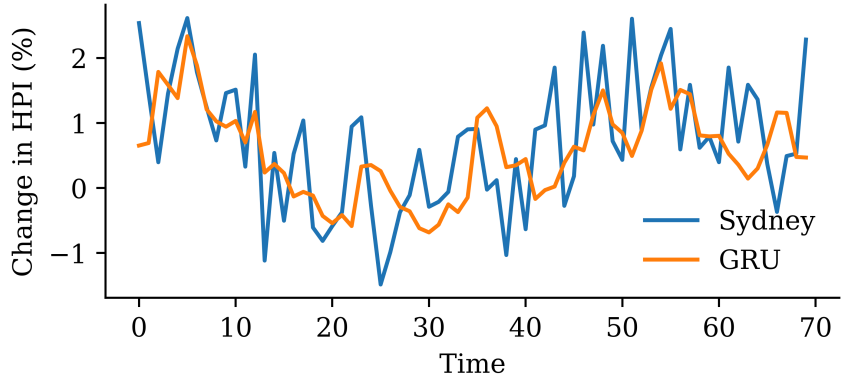
```
1 model_gru.evaluate(X_val, y_val, verbose=0)
```

```
1.344503402709961
```









UNSW SYDNEY

Two GRU layers

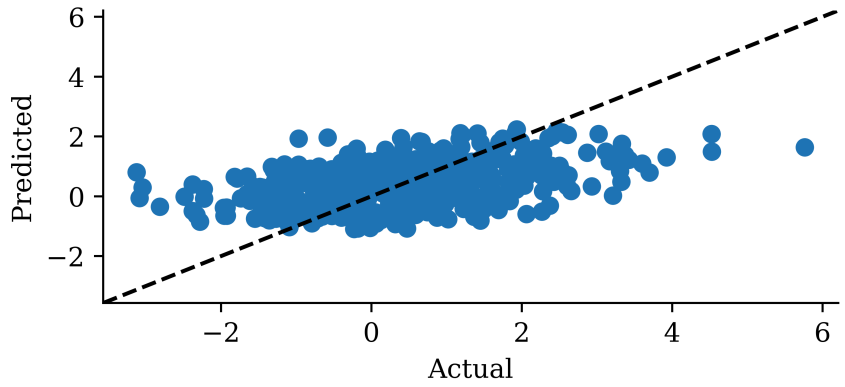
```
random.seed(1)

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)
```

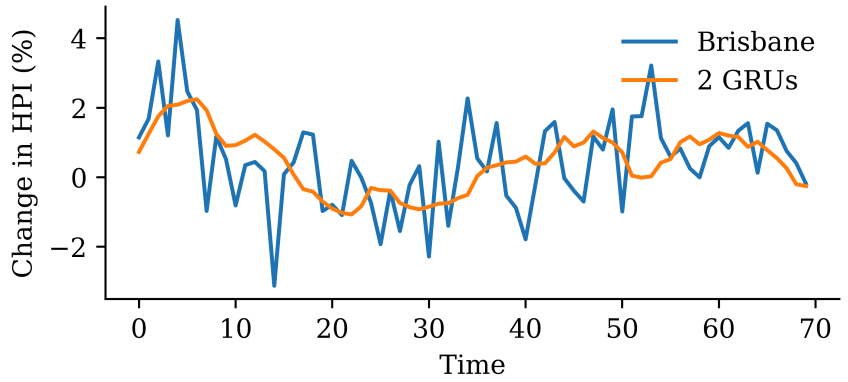
```
Epoch 67: early stopping
Restoring model weights from the end of the best epoch: 17.
CPU times: user 5.78 s, sys: 747 ms, total: 6.53 s
Wall time: 5.46 s
```

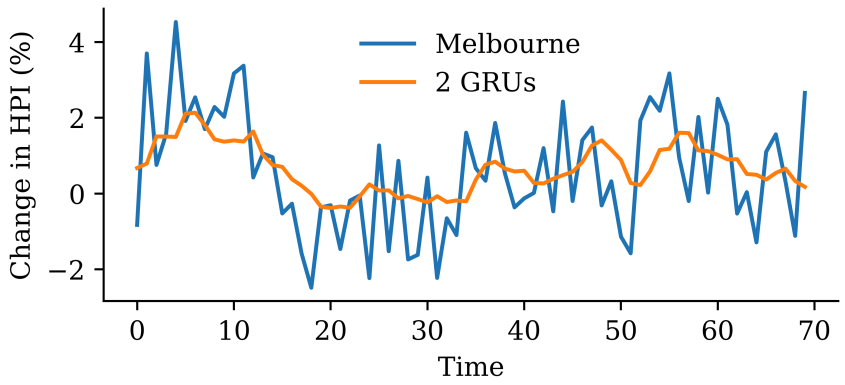
Assess the fits

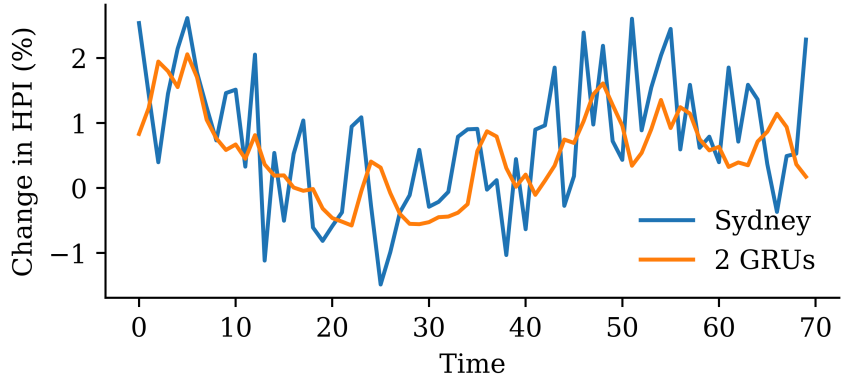
```
1 model_two_grus.evaluate(X_val, y_val, verbose=0)
```

```
1.3586511611938477
```









UNSW SYDNEY

Compare the models

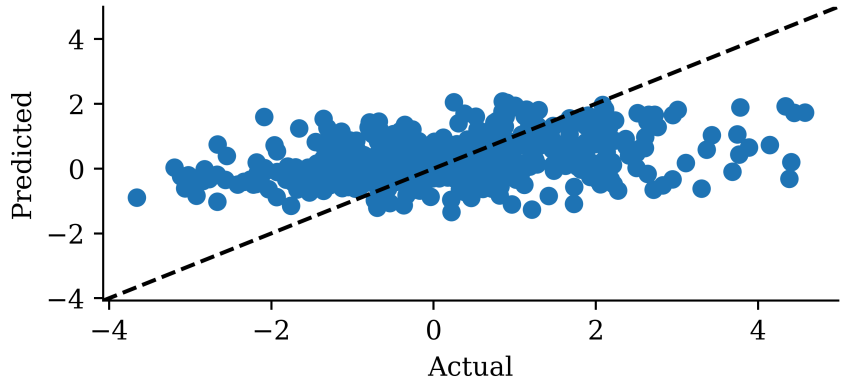
	Model	MSE
1	SimpleRNN	1.491643
0	Dense	1.428913
4	2 GRUs	1.358651
3	GRU	1.344503
2	LSTM	1.331125

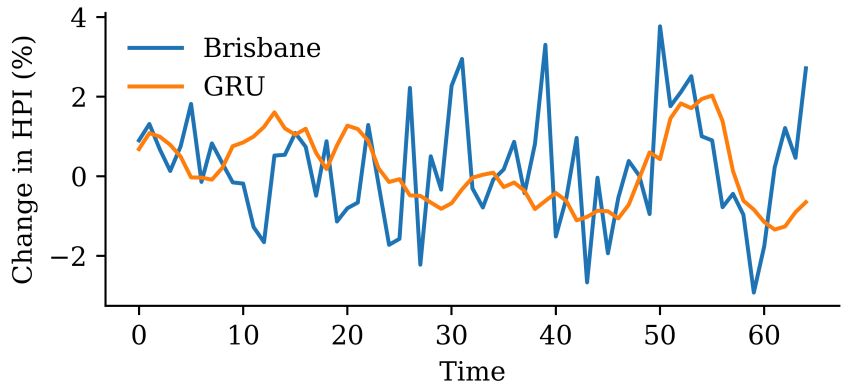
The network with an LSTM layer is the best.

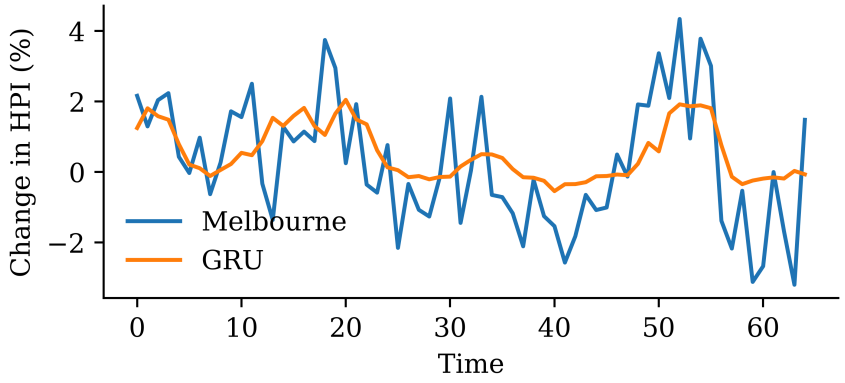
```
1 model_lstm.evaluate(X_test, y_test, verbose=0)
```

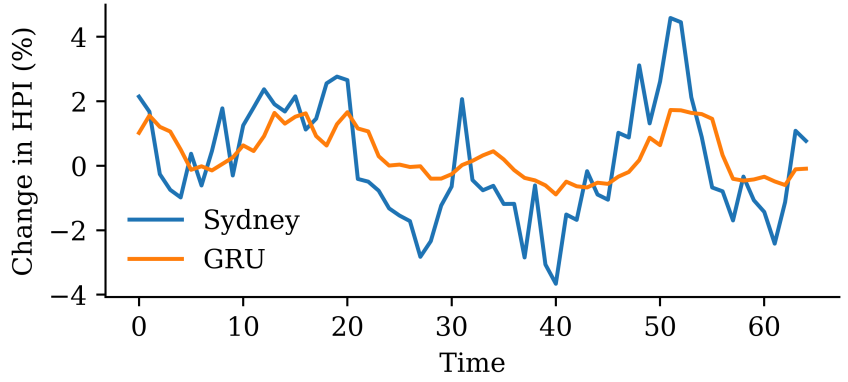
```
1.932026982307434
```

Test set









UNSW SYDNEY

Package Versions

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

```
Python implementation: CPython
Python version       : 3.11.11
IPython version      : 8.32.0

keras     : 3.8.0
matplotlib: 3.10.0
numpy     : 1.26.4
pandas    : 2.2.3
seaborn   : 0.13.2
scipy     : 1.13.1
torch     : 2.5.1+cu124
tensorflow: 2.18.0
tf_keras  : 2.18.0
```

UNSW SYDNEY

Glossary

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