

# 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  $\{ \mathbf{x}_i, y_i \}_{i=1}^n$  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  $\mathbf{x}$ .



## Time series data

Have a sequence  $\{ \mathbf{x}_t, y_t \}_{t=1}^T$  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.



## 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
```

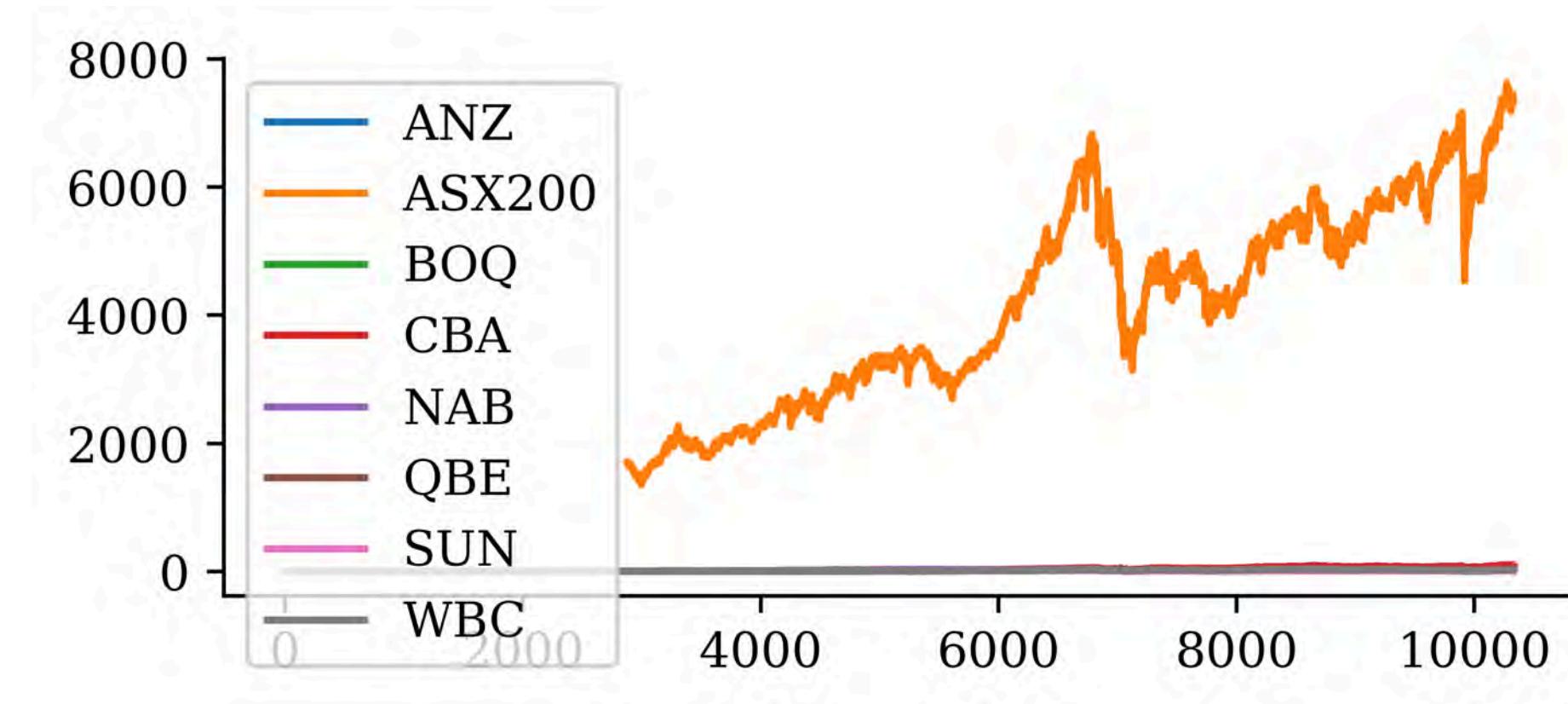
	Date	ANZ	ASX200	BOQ	CBA	NAB	QBE	SUN	WBC
0	1981-01-02	1.588896	NaN	NaN	NaN	1.791642	NaN	NaN	2.199454
1	1981-01-05	1.548452	NaN	NaN	NaN	1.791642	NaN	NaN	2.163397
2	1981-01-06	1.600452	NaN	NaN	NaN	1.791642	NaN	NaN	2.199454
...	...	...	...	...	...	...	...	...	...
10327	2021-10-28	28.600000	7430.4	8.97	106.86	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 asx200 = stocks.pop("ASX200")
```

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

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



# 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

```

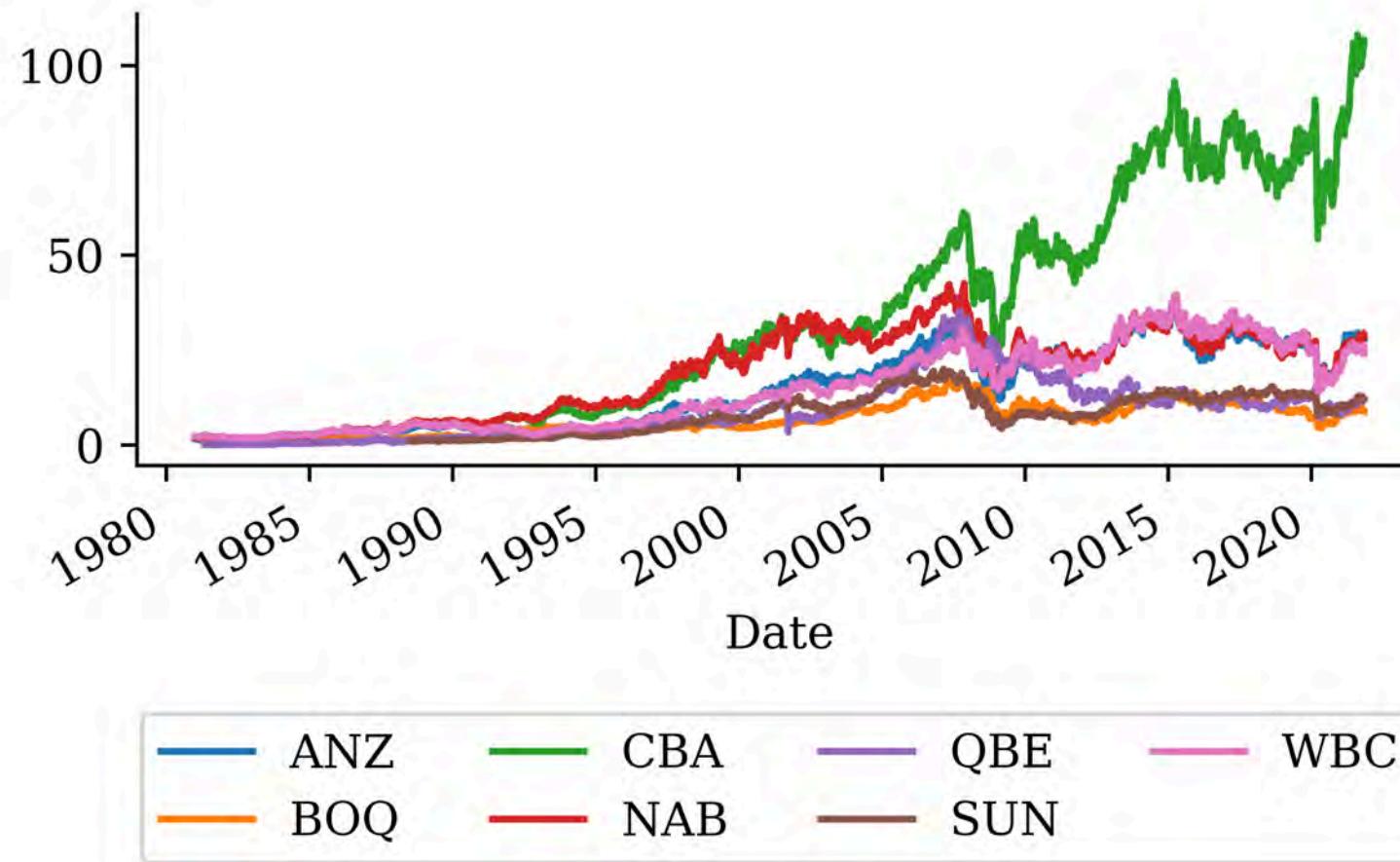
	<b>ANZ</b>	<b>BOQ</b>	<b>CBA</b>	<b>NAB</b>	<b>QBE</b>	<b>SUN</b>	<b>WBC</b>
<b>Date</b>							
1981-01-02	1.588896	NaN	NaN	1.791642	NaN	NaN	2.199454
1981-01-05	1.548452	NaN	NaN	1.791642	NaN	NaN	2.163397
1981-01-06	1.600452	NaN	NaN	1.791642	NaN	NaN	2.199454
...	...	...	...	...	...	...	...
2021-10-28	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"]
```

	ANZ	BOQ	CBA	NAB	QBE	SUN	WBC
Date							
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



# 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



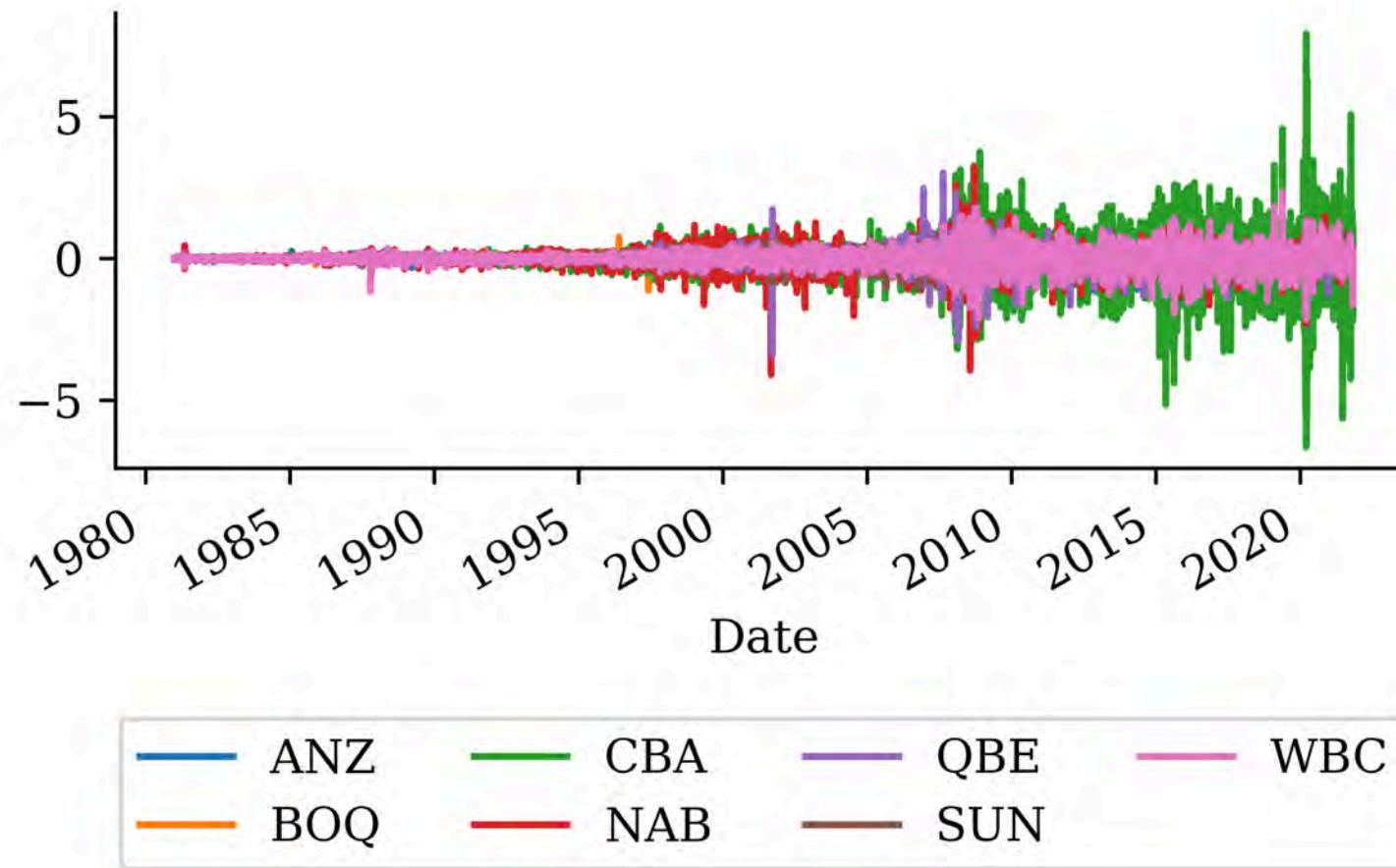
275 rows × 1 columns



UNSW  
SYDNEY

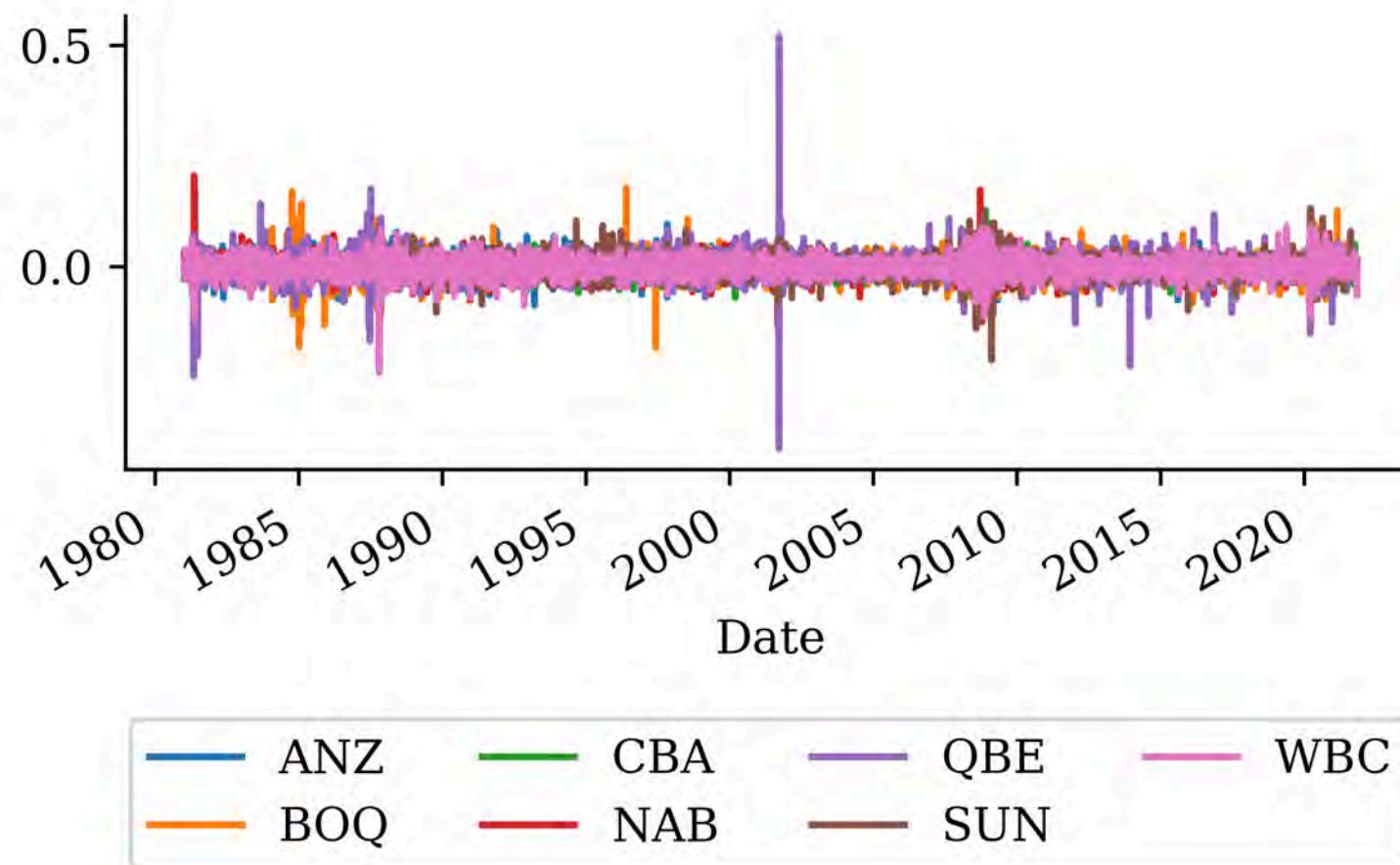
# Can look at the first differences

```
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);
```



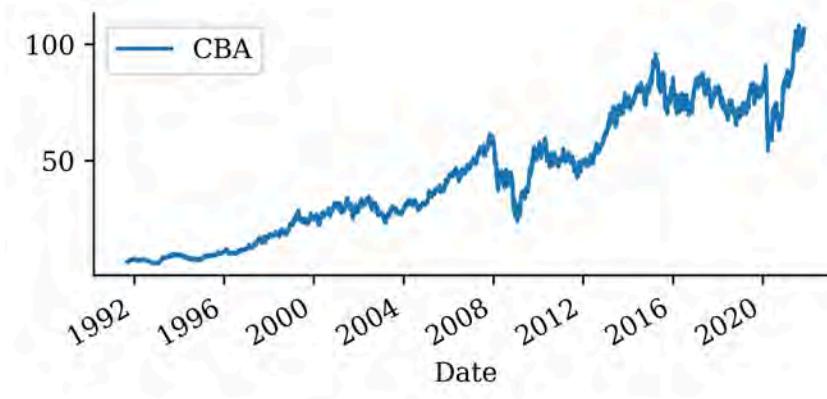
# Focus on one stock

```
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

```

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

```

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

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

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

CBA 0  
dtype: int64

```

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

```

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



# Lecture Outline

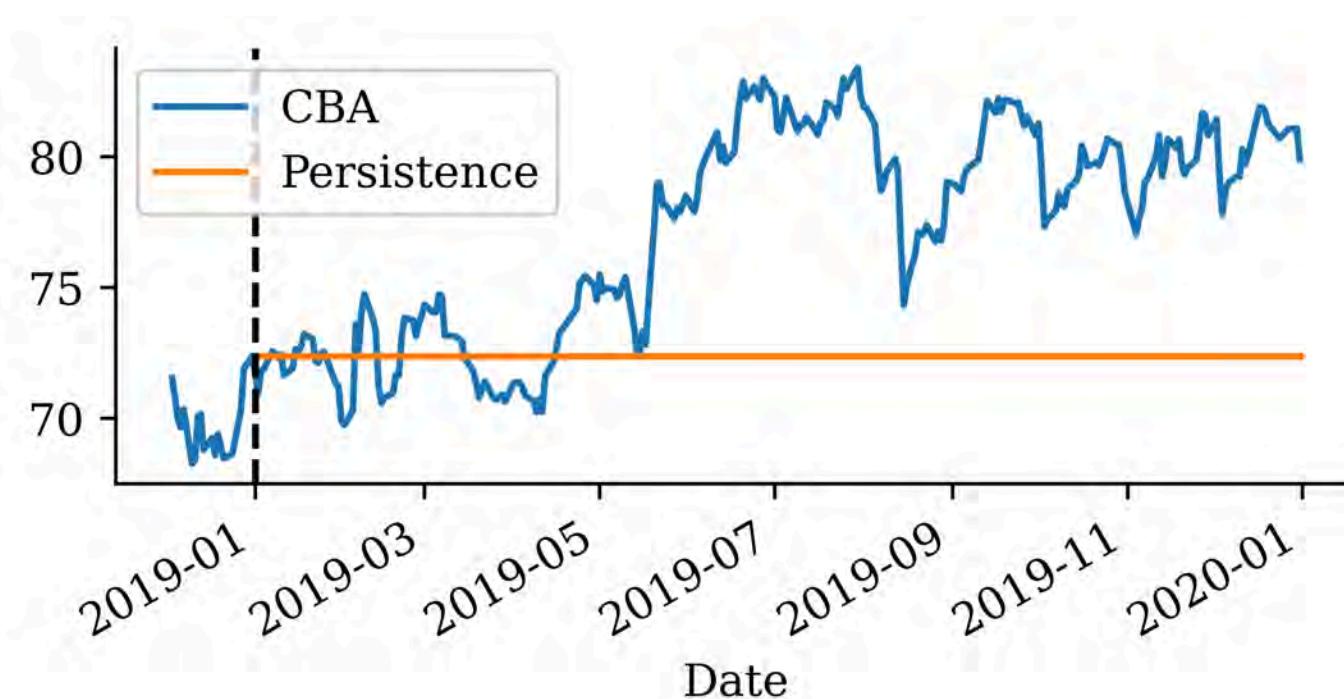
- Time Series
- **Baseline forecasts**
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- Stock prediction with recurrent networks
- Internals of the SimpleRNN
- Other recurrent network variants
- CoreLogic Hedonic Home Value Index
- Predicting Sydney House Prices
- Predicting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



# Persistence forecast

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

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



# Trend

We can extrapolate from recent trend:

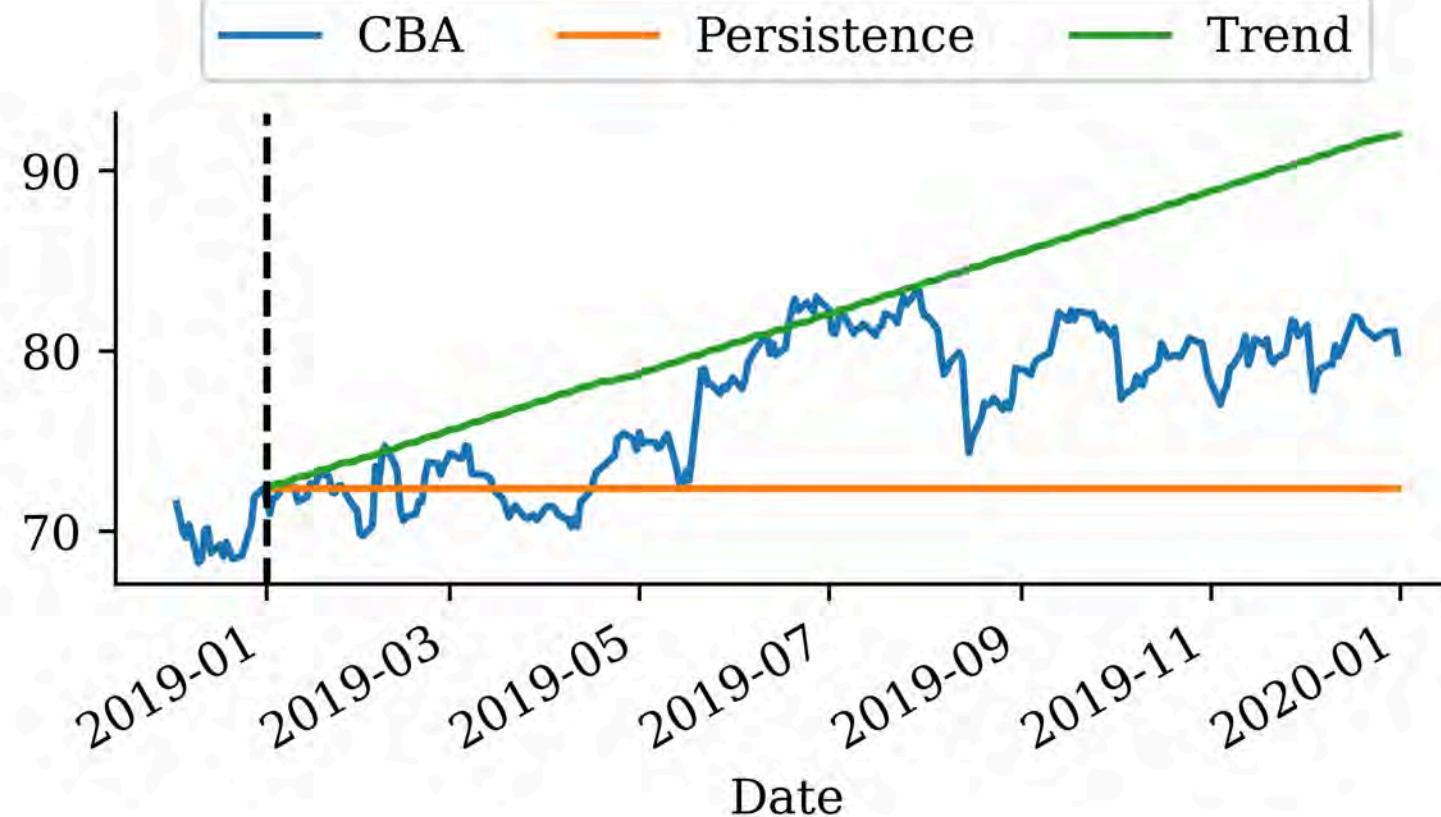
```
1 past_date = stock.loc["2018"].index[-30]
2 past = stock.loc[past_date, "CBA"]
3 latest_date = stock.loc["2018", "CBA"].index[-1]
4 latest = stock.loc[latest_date, "CBA"]
5
6 trend = (latest - past) / (latest_date - past_date).days
7 print(trend)
8
9 tdays_since_cutoff = np.arange(1, len(stock.loc["2019":]) + 1)
10 stock.loc["2019":, "Trend"] = latest + trend * tdays_since_cutoff
```

0.0775555555555545



# 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", "Persistence"])
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

```

	Today	Yesterday
Date		
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

```



# Lagged time series

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

	T-40	T-39	T-38	T-37	T-36	T-35	T-34	T-33	T-32	T-31	...	T-0
Date												
1991-09-12	NaN	...	NaN									
1991-09-13	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



# Split into training and testing

```
1 # Split the data in time
2 X_train = df_lags.loc[:"2018"]
3 X_val = df_lags.loc["2019"]
4 X_test = df_lags.loc["2020":]
5
6 # Remove any with NAs and split into X and y
7 X_train = X_train.dropna()
8 X_val = X_val.dropna()
9 X_test = X_test.dropna()
10
11 y_train = X_train.pop("T")
12 y_val = X_val.pop("T")
13 y_test = X_test.pop("T")
```

```
1 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,))
```



# Inspect the split data

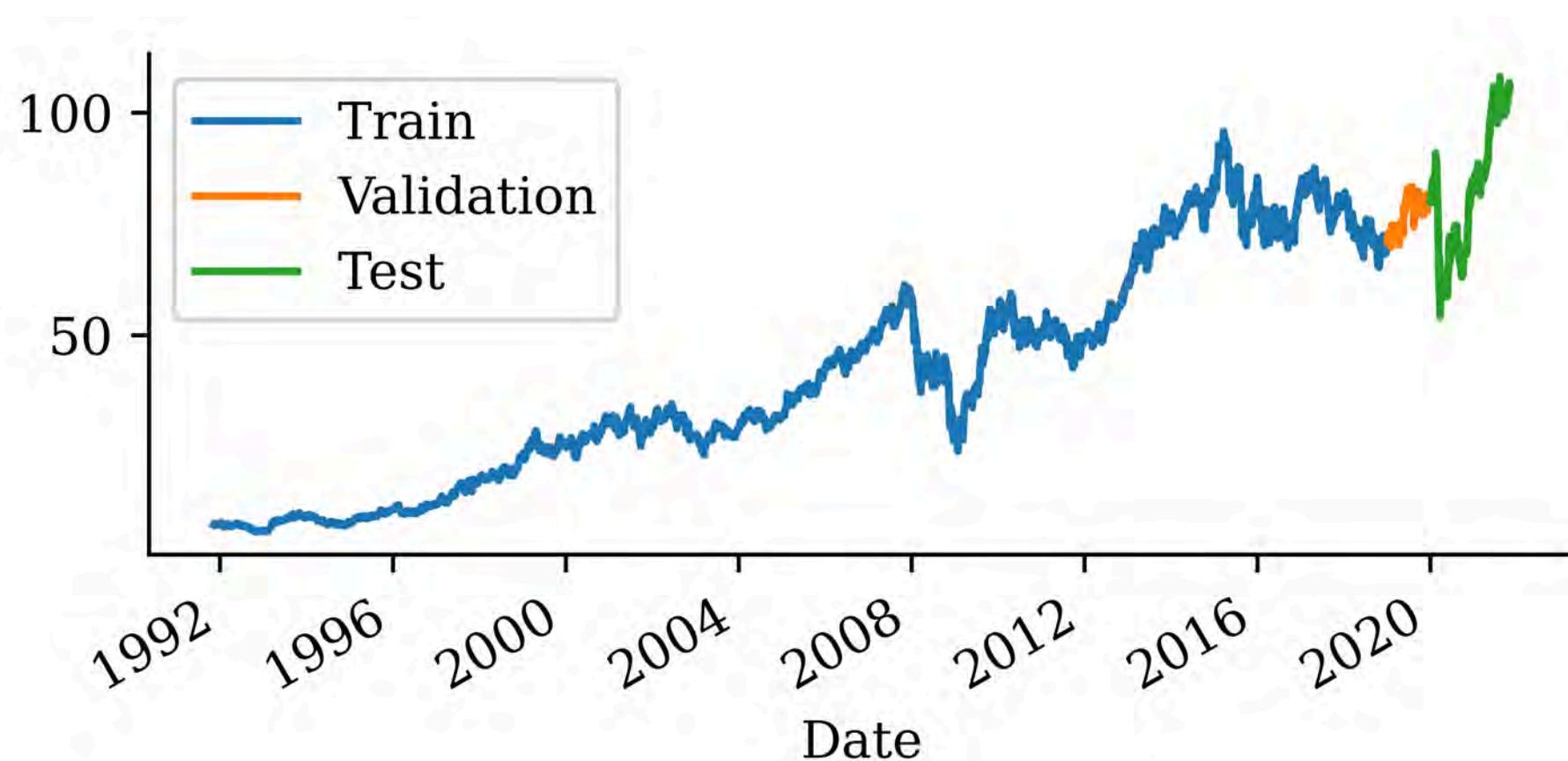
1 X\_train

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

6872 rows × 40 columns

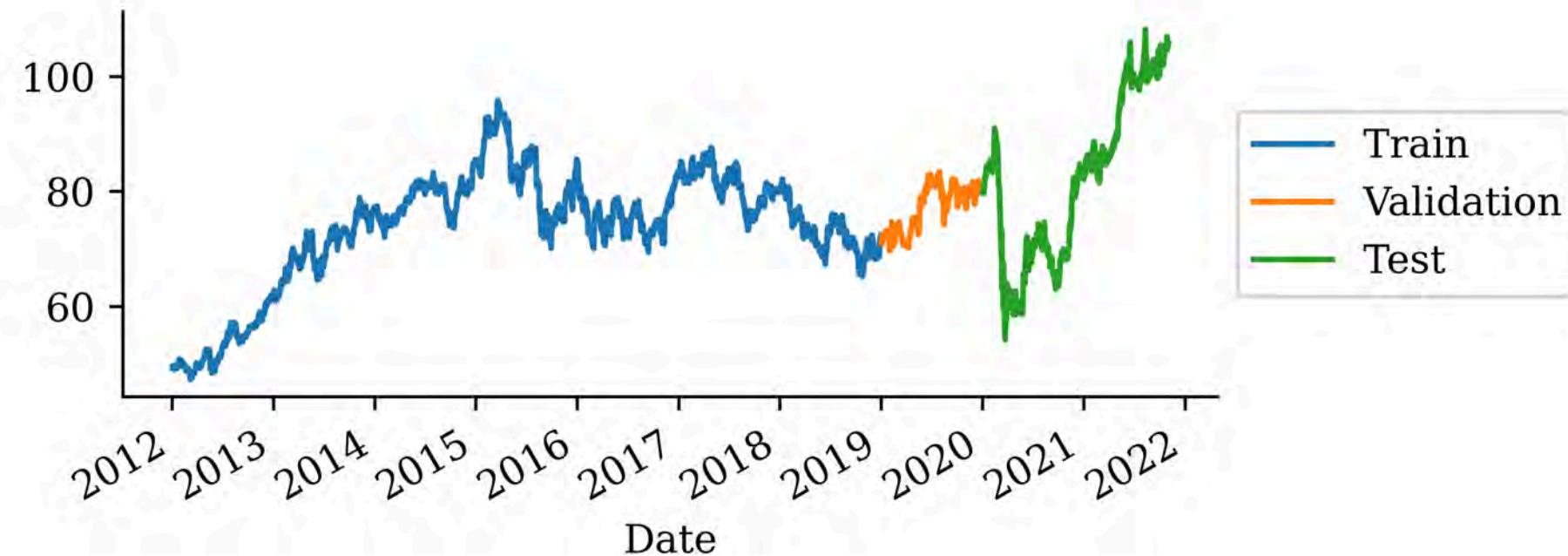


# Plot the split



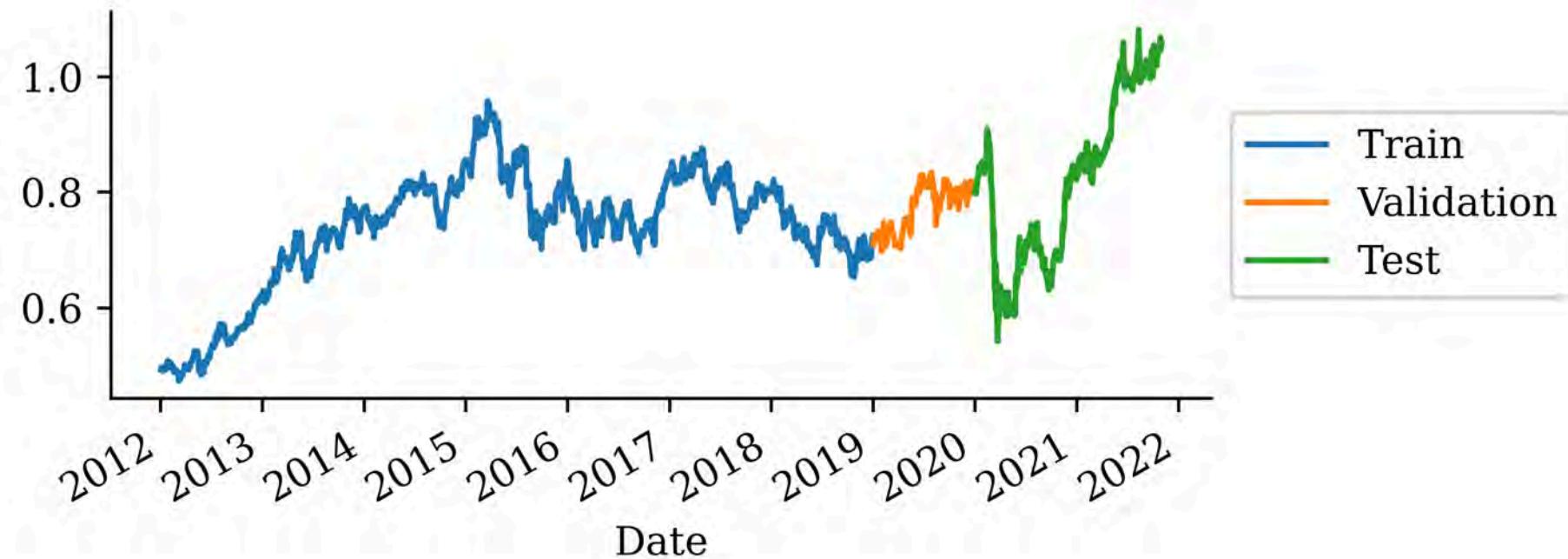
# Train on more recent data

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



# Rescale by eyeballing it

```
1 X_train = X_train / 100  
2 X_val = X_val / 100  
3 X_test = X_test / 100  
4 y_train = y_train / 100  
5 y_val = y_val / 100  
6 y_test = y_test / 100
```

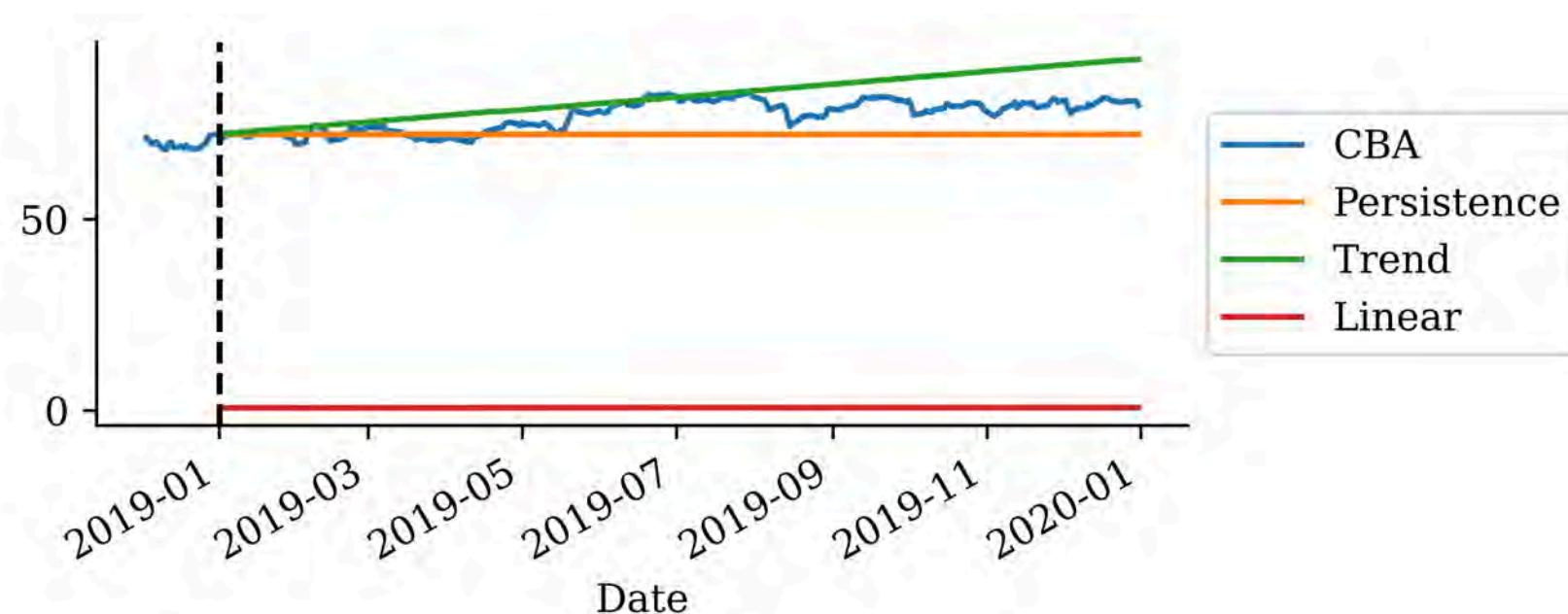


# Fit a linear model

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

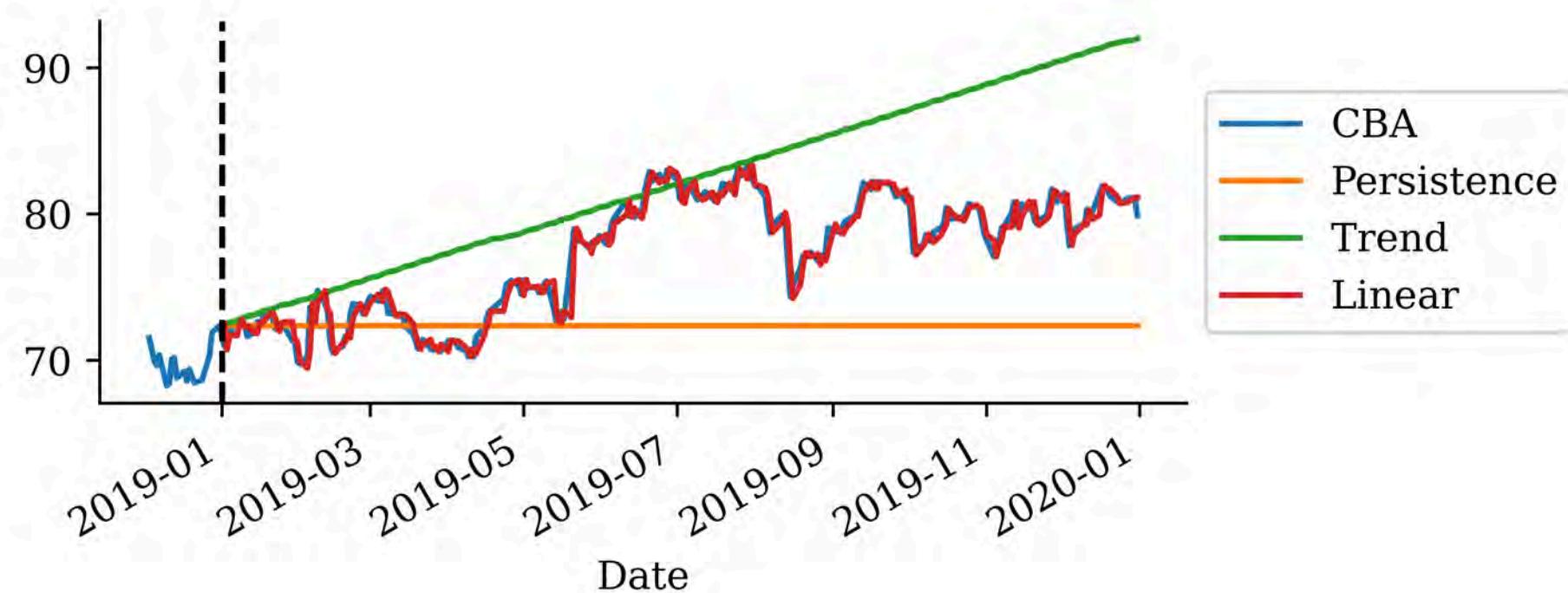
Make a forecast for the validation data:

```
1 y_pred = lr.predict(X_val)  
2 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

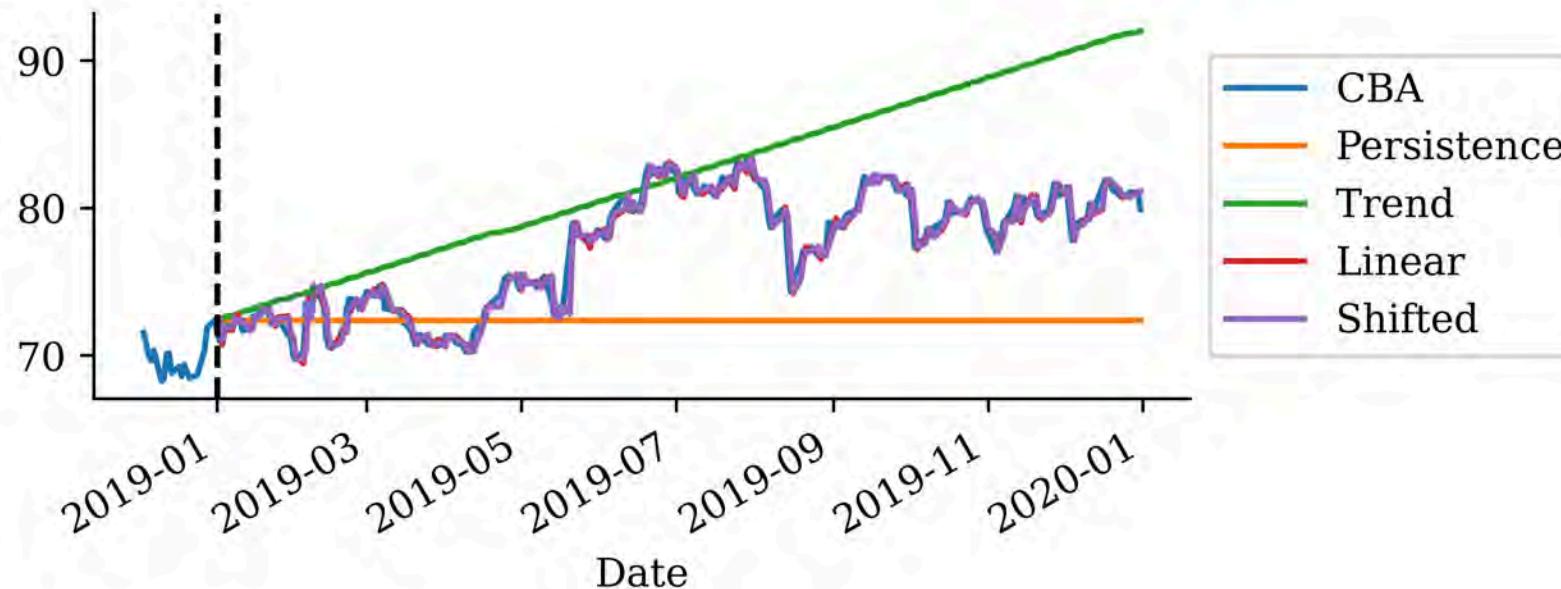


# Comparing apples to apples

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.

$$\hat{y}_t = \beta_0 + \beta_1 y_{t-1} + \beta_2 y_{t-2} + \dots + \beta_n y_{t-n}$$

$$\hat{y}_{t+1} = \beta_0 + \beta_1 \hat{y}_t + \beta_2 y_{t-1} + \dots + \beta_n y_{t-n+1}$$

$$\hat{y}_{t+2} = \beta_0 + \beta_1 \hat{y}_{t+1} + \beta_2 \hat{y}_t + \dots + \beta_n y_{t-n+2}$$

⋮

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



# Autoregressive forecasting function

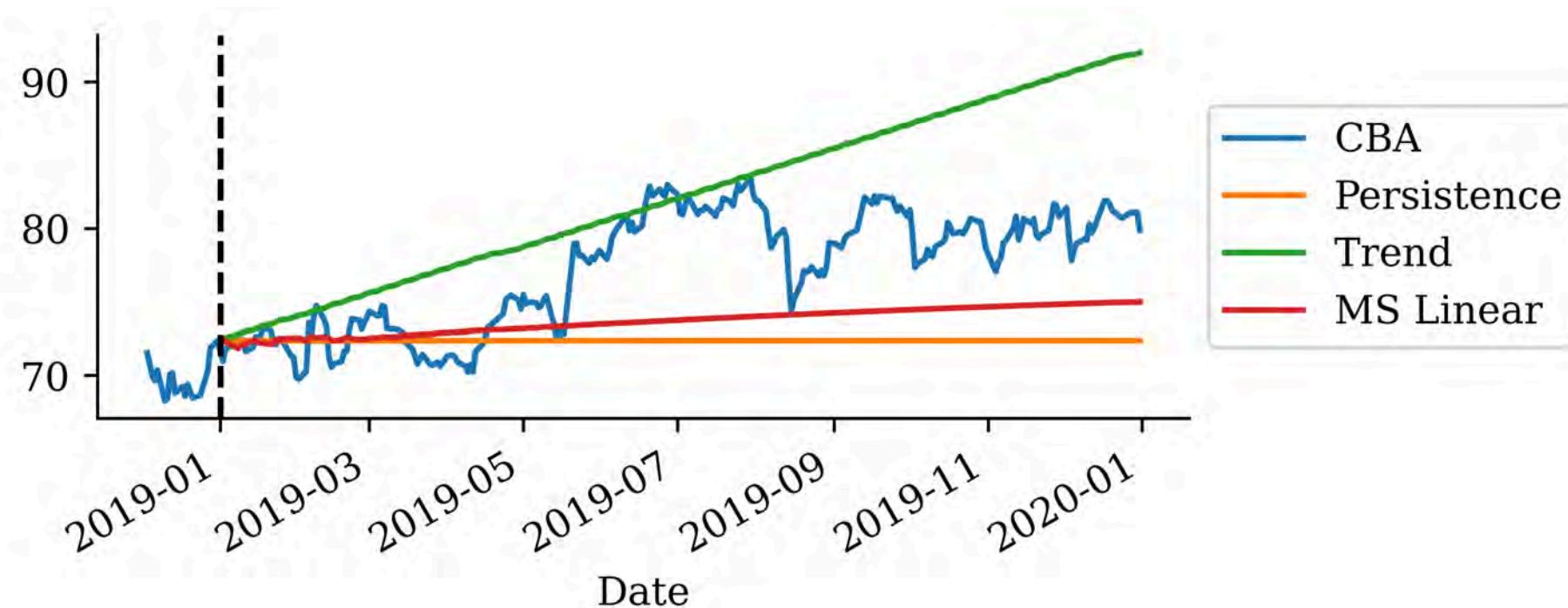
```
1 def autoregressive_forecast(model, X_val, suppress=False):
2     """
3     Generate a multi-step forecast using the given model.
4     """
5     multi_step = pd.Series(index=X_val.index, name="Multi Step")
6
7     # Initialize the input data for forecasting
8     input_data = X_val.iloc[0].values.reshape(1, -1)
9
10    for i in range(len(multi_step)):
11        # Ensure input_data has the correct feature names
12        input_df = pd.DataFrame(input_data, columns=X_val.columns)
13        if suppress:
14            next_value = model.predict(input_df, verbose=0)
15        else:
16            next_value = model.predict(input_df)
17
18        multi_step.iloc[i] = next_value
19
20        # Append that prediction to the input for the next forecast
21        if i + 1 < len(multi_step):
22            input_data = np.append(input_data[:, 1:], next_value).reshape(1, -1)
23
24    return multi_step
```



# Look at the autoregressive linear forecasts

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

```
1 stock.loc["2018-12":"2019"].drop(["Linear", "Shifted"], axis=1).plot()
2 plt.axvline("2019", color="black", linestyle="--")
3 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)
```

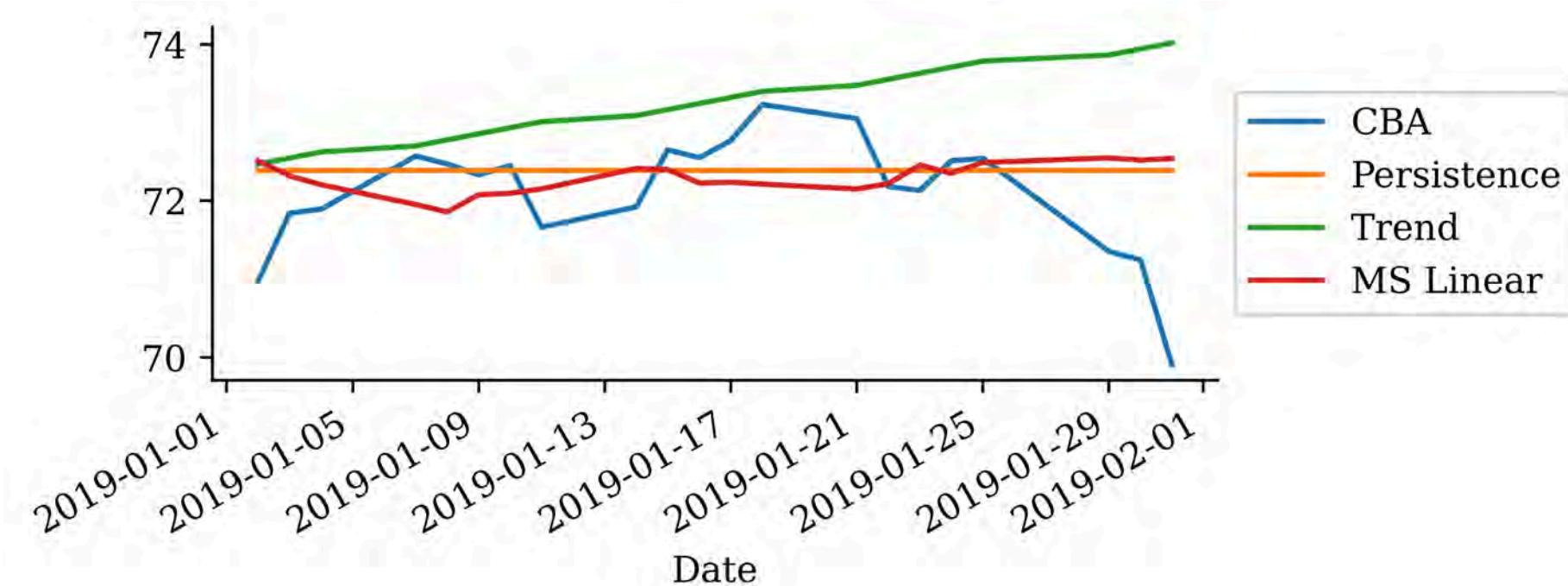
Multi-step-ahead forecasts:

```
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



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



# Simple feedforward neural network

```

1 model = Sequential([
2     Dense(64, activation="leaky_relu"),
3     Dense(1, "softplus")])
4
5 model.compile(optimizer="adam", loss="mean_squared_error")

```

```

1 if Path("aus_fin_fnn_model.h5").exists():
2     model = keras.models.load_model("aus_fin_fnn_model.h5")
3 else:
4     es = EarlyStopping(patience=15, restore_best_weights=True)
5     model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=500,
6                 callbacks=[es], verbose=0)
7     model.save("aus_fin_fnn_model.h5")
8
9 model.summary()

```

Model: "sequential"

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

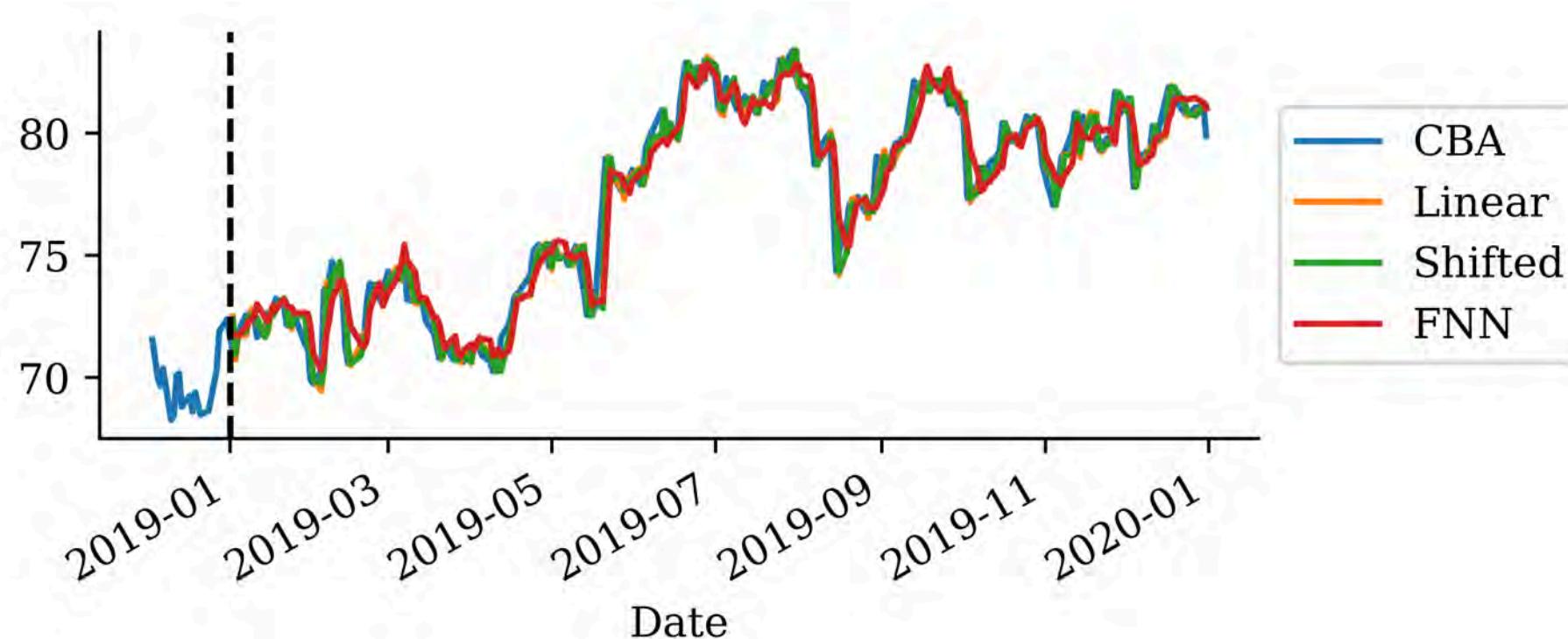
Total params: 2,691 (10.51 KB)  
Trainable params: 2,689 (10.50 KB)  
Non-trainable params: 0 (0.00 B)  
Optimizer params: 2 (8.00 B)



# Forecast and plot

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

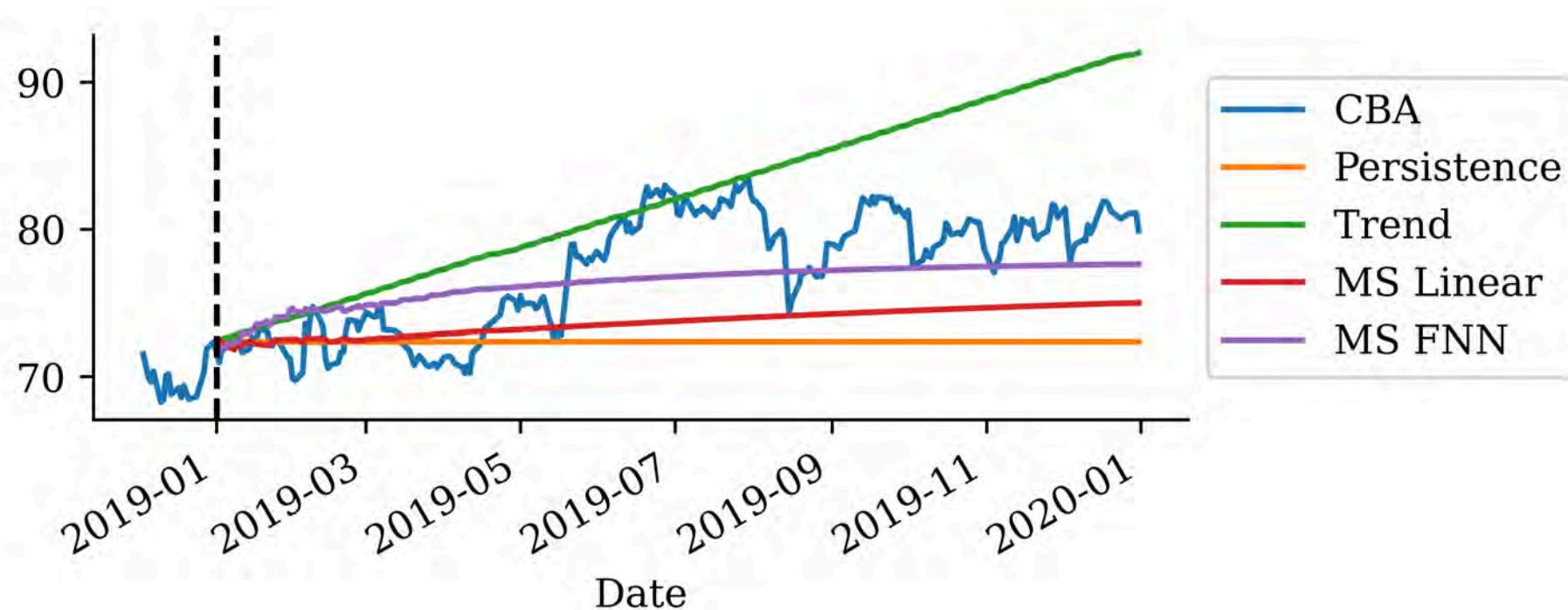
1 stock.loc["2018-12":"2019"].drop(["Persistence", "Trend", "MS Linear"], axis=1).plot()
2 plt.axvline("2019", color="black", linestyle="--")
3 plt.legend(loc="center left", bbox_to_anchor=(1, 0.5));
```



# Autoregressive forecasts

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

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



# Metrics

One-step-ahead forecasts:

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

(0.6329105517812197, 0.6367221343873524, 1.0445115378023873)

Multi-step-ahead forecasts:

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

(39.54629367588932, 37.87104674064297, 23.847003791127374, 10.150573162371526)



# 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

$$u_n = \psi(n, u_{n-1}) \quad \text{for } n > 0.$$

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



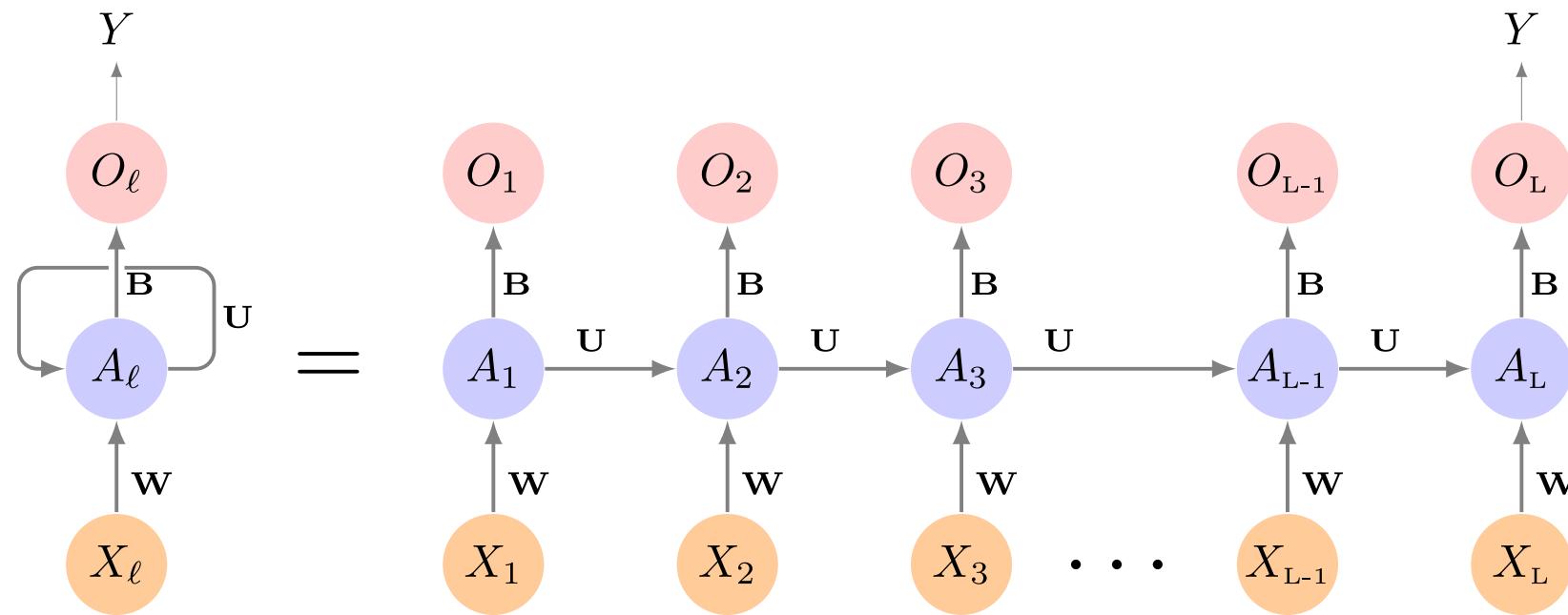
Source: Wikipedia, Recurrence relation.



UNSW  
SYDNEY

# 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

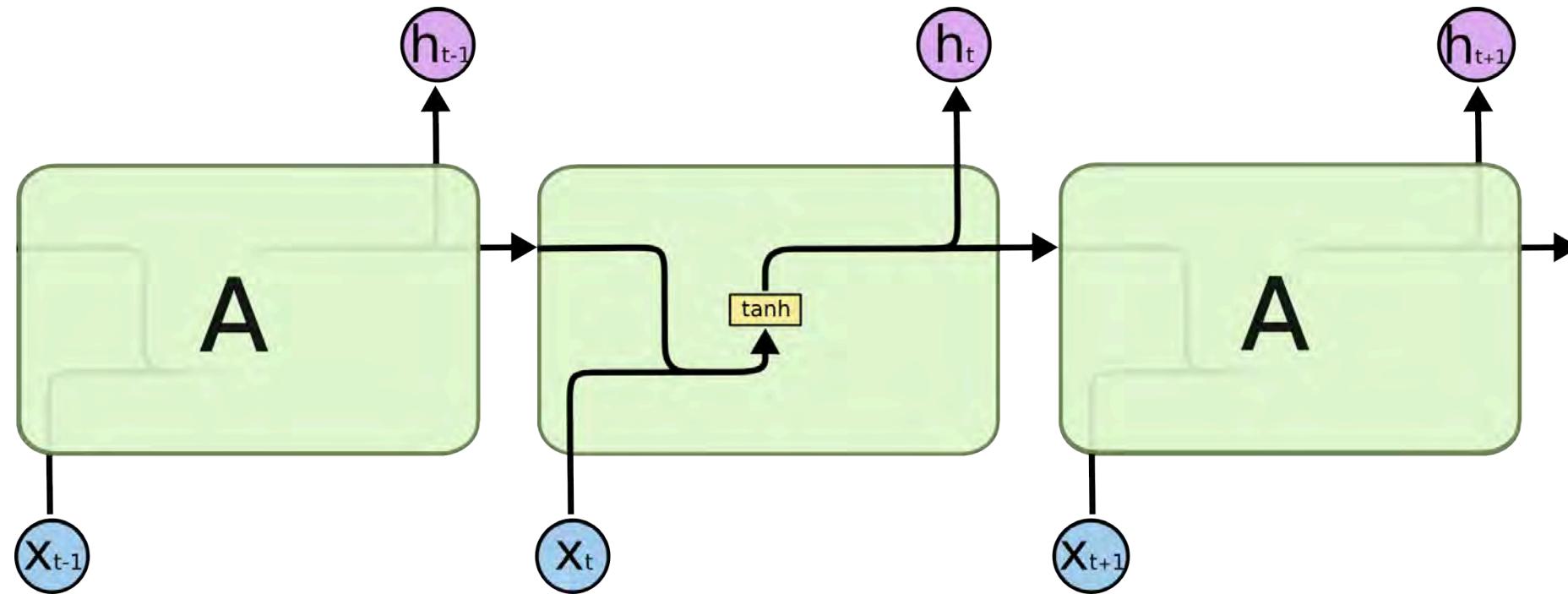
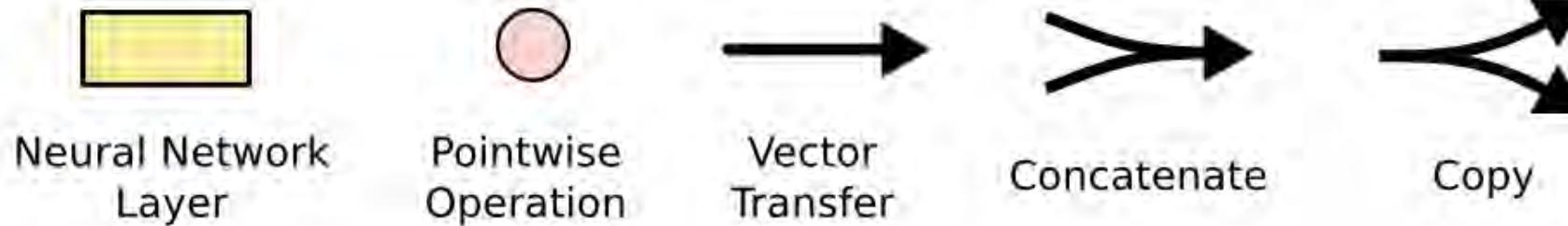
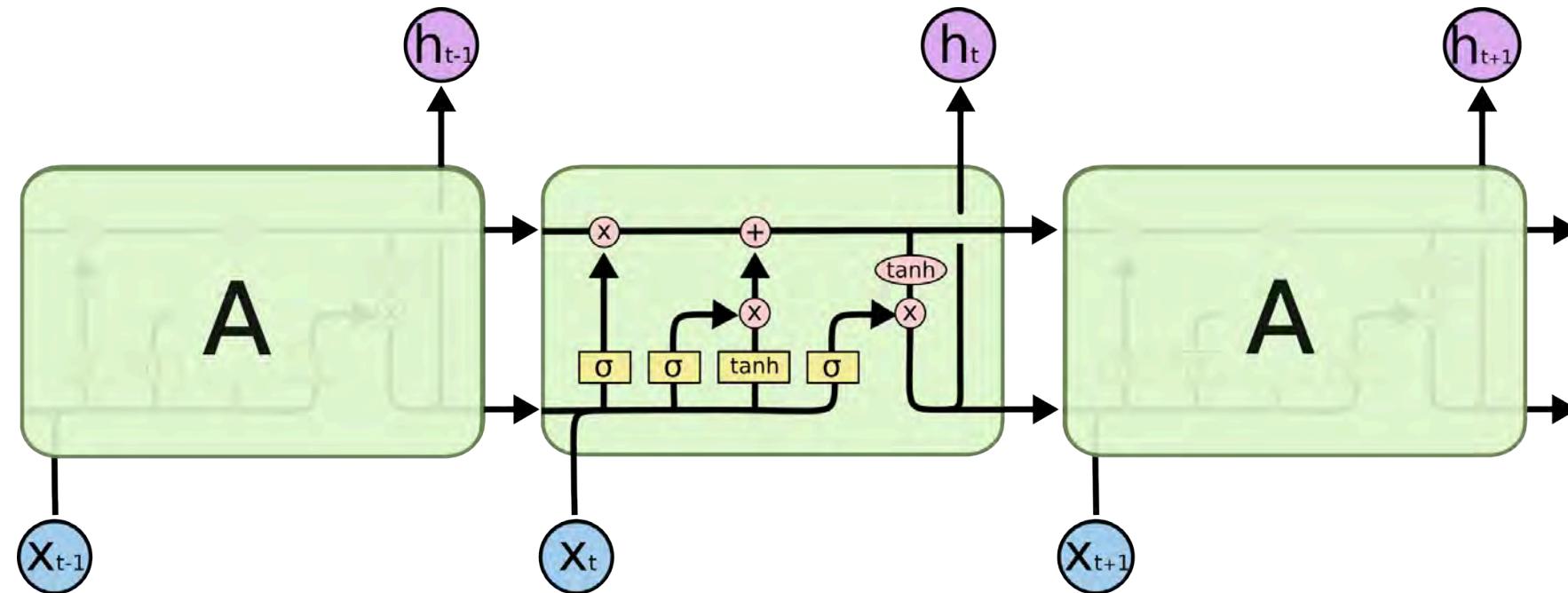


Diagram of a SimpleRNN cell.

All the outputs before the final one are often discarded.



# LSTM internals



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



# GRU internals

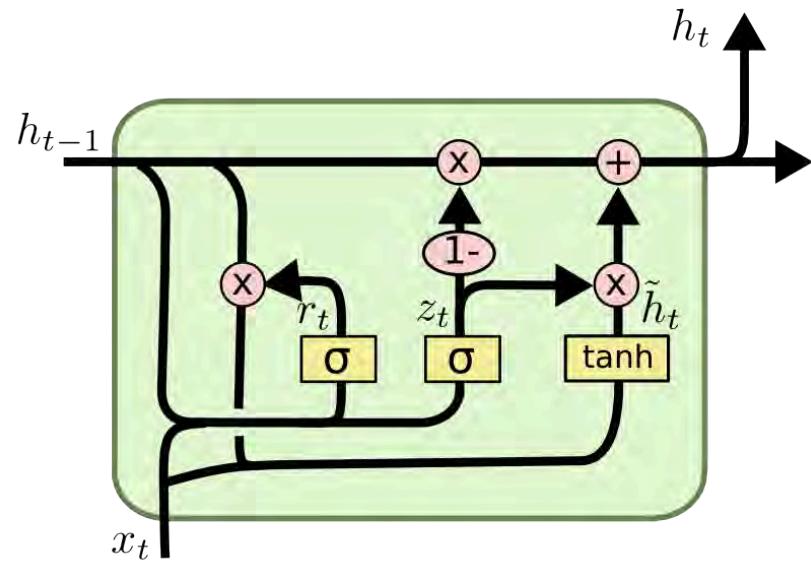


Diagram of a GRU cell.

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

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

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

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

# 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

```

1 from keras.layers import SimpleRNN, Reshape
2 model = Sequential([
3     Reshape((-1, 1)),
4     SimpleRNN(64, activation="tanh"),
5     Dense(1, "softplus")])
6 model.compile(optimizer="adam", loss="mean_squared_error")

```

```

1 es = EarlyStopping(patience=15, restore_best_weights=True)
2 model.fit(X_train, y_train, validation_data=(X_val, y_val),
3            epochs=500, callbacks=[es], verbose=0)
4 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 (16.76 KB)

Trainable params: 4,289 (16.75 KB)

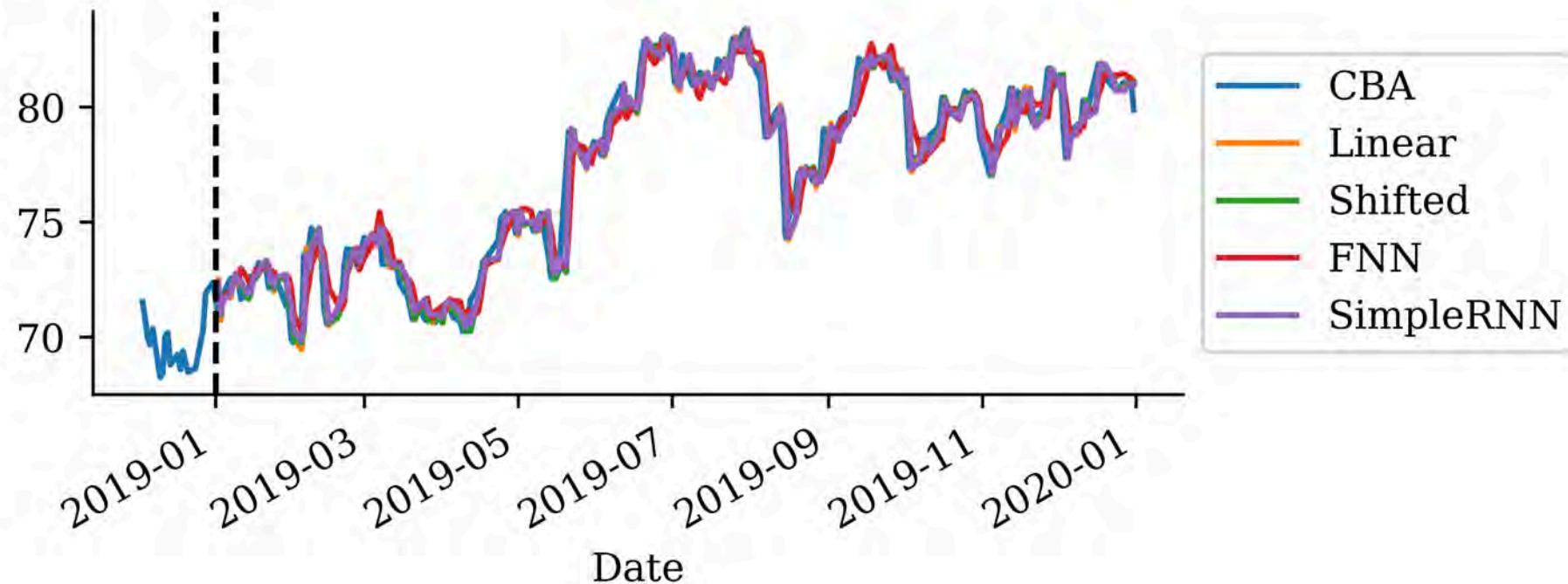
Non-trainable params: 0 (0.00 B)

Optimizer params: 2 (8.00 B)



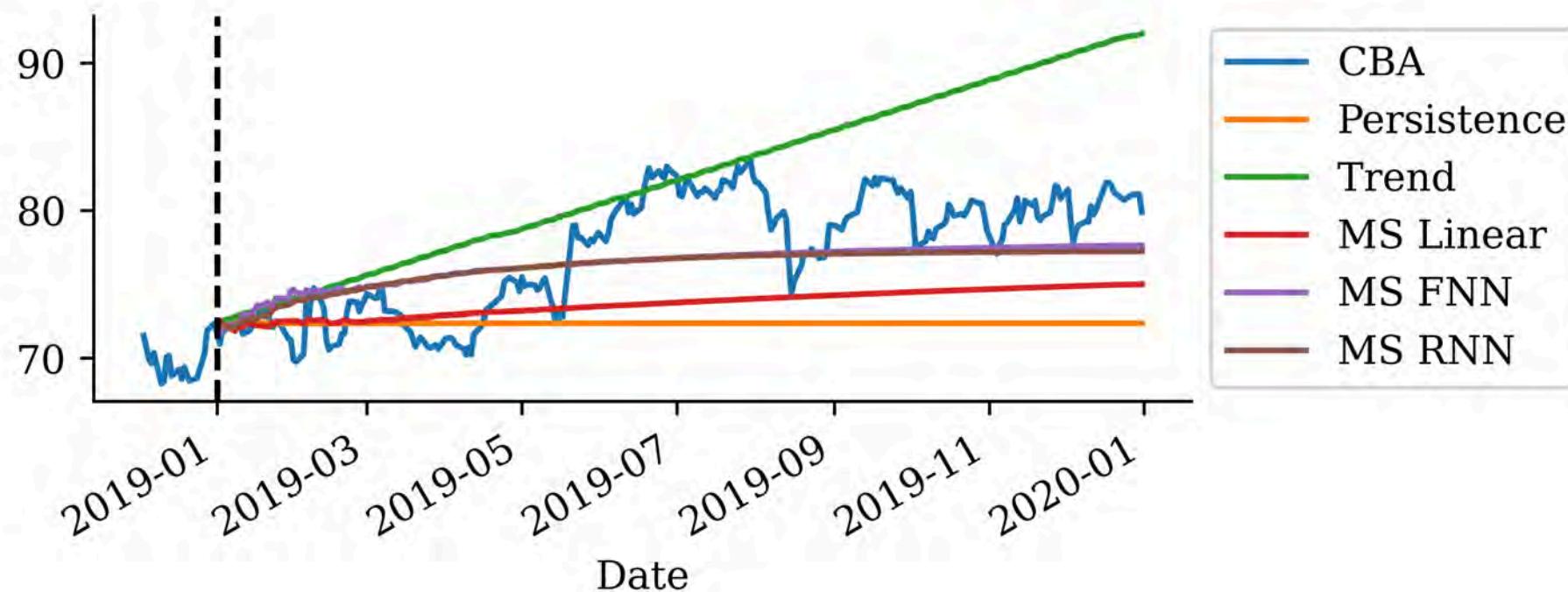
# Forecast and plot

```
1 y_pred = model.predict(X_val.to_numpy(), verbose=0)
2 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:

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

(0.6329105517812197,  
 0.6367221343873524,  
 1.0445115378023873,  
 0.6444506647025611)

Multi-step-ahead forecasts:

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

(39.54629367588932,  
 37.87104674064297,  
 23.847003791127374,  
 10.150573162371526,  
 10.58367263283111)



# GRU

```

1 from keras.layers import GRU
2
3 model = Sequential([Reshape((-1, 1)),
4                     GRU(16, activation="tanh"),
5                     Dense(1, "softplus")])
6 model.compile(optimizer="adam", loss="mean_squared_error")

```

```

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

```

Model: "sequential\_2"

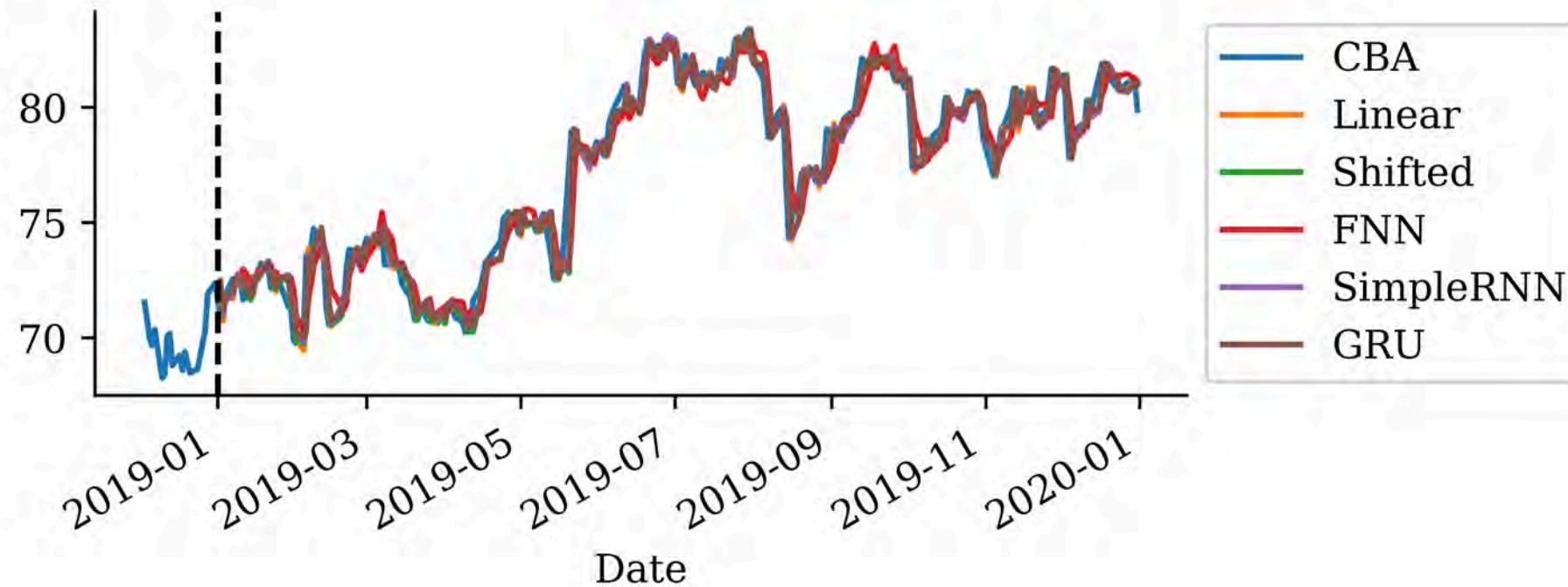
Layer (type)	Output Shape	Param #
reshape_1 (Reshape)	(32, 40, 1)	0
gru (GRU)	(32, 16)	912
dense_3 (Dense)	(32, 1)	17

Total params: 931 (3.64 KB)  
 Trainable params: 929 (3.63 KB)  
 Non-trainable params: 0 (0.00 B)  
 Optimizer params: 2 (8.00 B)



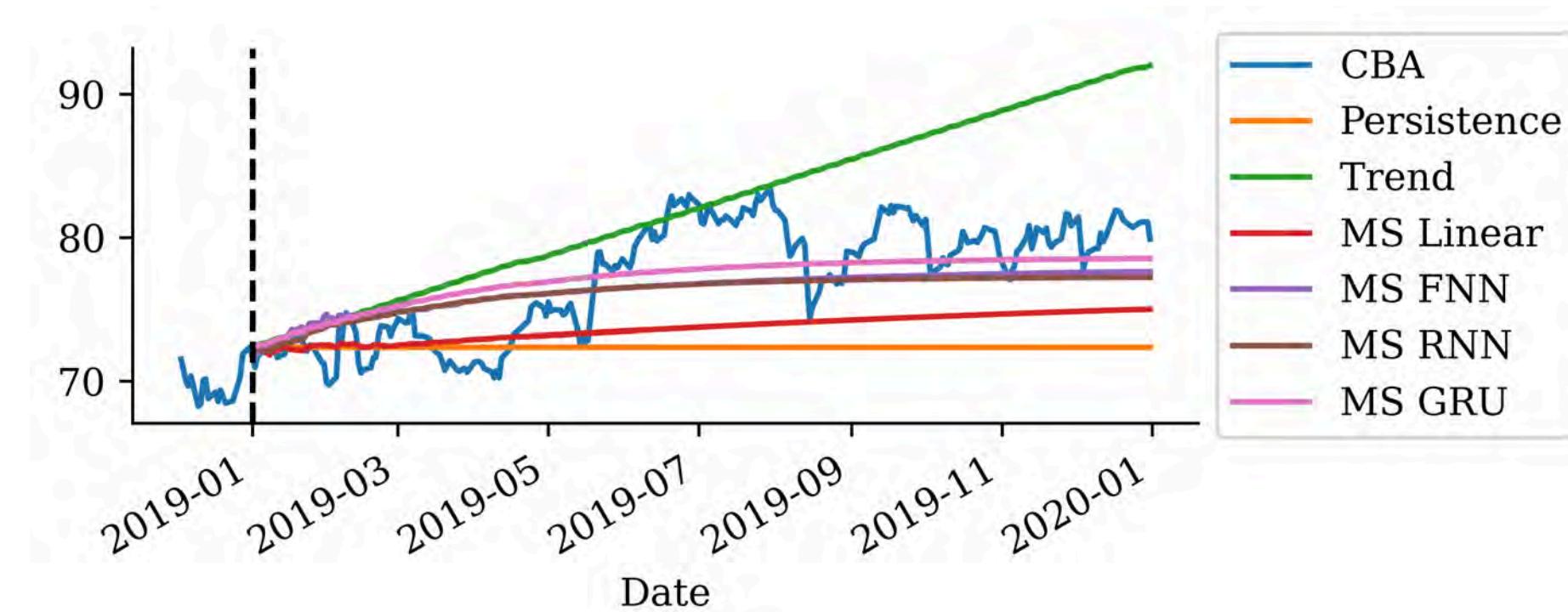
# Forecast and plot

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



# Multi-step forecasts

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



# Metrics

One-step-ahead forecasts:

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

(0.6329105517812197,  
 0.6367221343873524,  
 1.0445115378023873,  
 0.6444506647025611,  
 0.6390276531968386)

Multi-step-ahead forecasts:

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

(39.54629367588932,  
 37.87104674064297,  
 23.847003791127374,  
 10.150573162371526,  
 10.58367263283111,  
 8.111302768865865)



# 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  $x_1, x_2, \dots, x_n$ .

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

If instead we had a batch of  $b$  time series'

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

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

# SimpleRNN

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

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

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

Here,

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



# SimpleRNN (in batches)

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

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

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

Here,

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



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



# Simple Keras demo

```

1 num_obs = 4
2 num_time_steps = 3
3 num_time_series = 2
4
5 X = (
6     np.arange(num_obs * num_time_steps * num_time_series)
7     .astype(np.float32)
8     .reshape([num_obs, num_time_steps, num_time_series])
9 )
10
11 output_size = 1
12 y = np.array([0, 0, 1, 1])

```

1 X[:2]

```
array([[[ 0.,  1.],
       [ 2.,  3.],
       [ 4.,  5.]],
      [[ 6.,  7.],
       [ 8.,  9.],
       [10., 11.]]], dtype=float32)
```

1 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.

```
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)
```

```
[8.059103012084961, 0.5]
```

The predicted probabilities on the training set are:

```
1 model.predict(X, verbose=0)  
  
array([[2.19e-04],  
       [2.79e-09],  
       [3.52e-14],  
       [4.45e-19]], dtype=float32)
```



# SimpleRNN weights

```
1 model.get_weights()
```

```
[array([[-1.31],  
       [-0.57]], dtype=float32),  
 array([[-1.03]], dtype=float32),  
 array([-0.32], dtype=float32)]
```

```
1 def sigmoid(x):  
2     return 1 / (1 + np.exp(-x))  
3  
4  
5 W_x, W_y, b = model.get_weights()  
6  
7 Y = np.zeros((num_obs, output_size), dtype=np.float32)  
8 for t in range(num_time_steps):  
9     X_t = X[:, t, :]  
10    z = X_t @ W_x + Y @ W_y + b  
11    Y = sigmoid(z)  
12  
13 Y
```

```
array([[2.19e-04],  
       [2.79e-09],  
       [3.52e-14],  
       [4.45e-19]], dtype=float32)
```

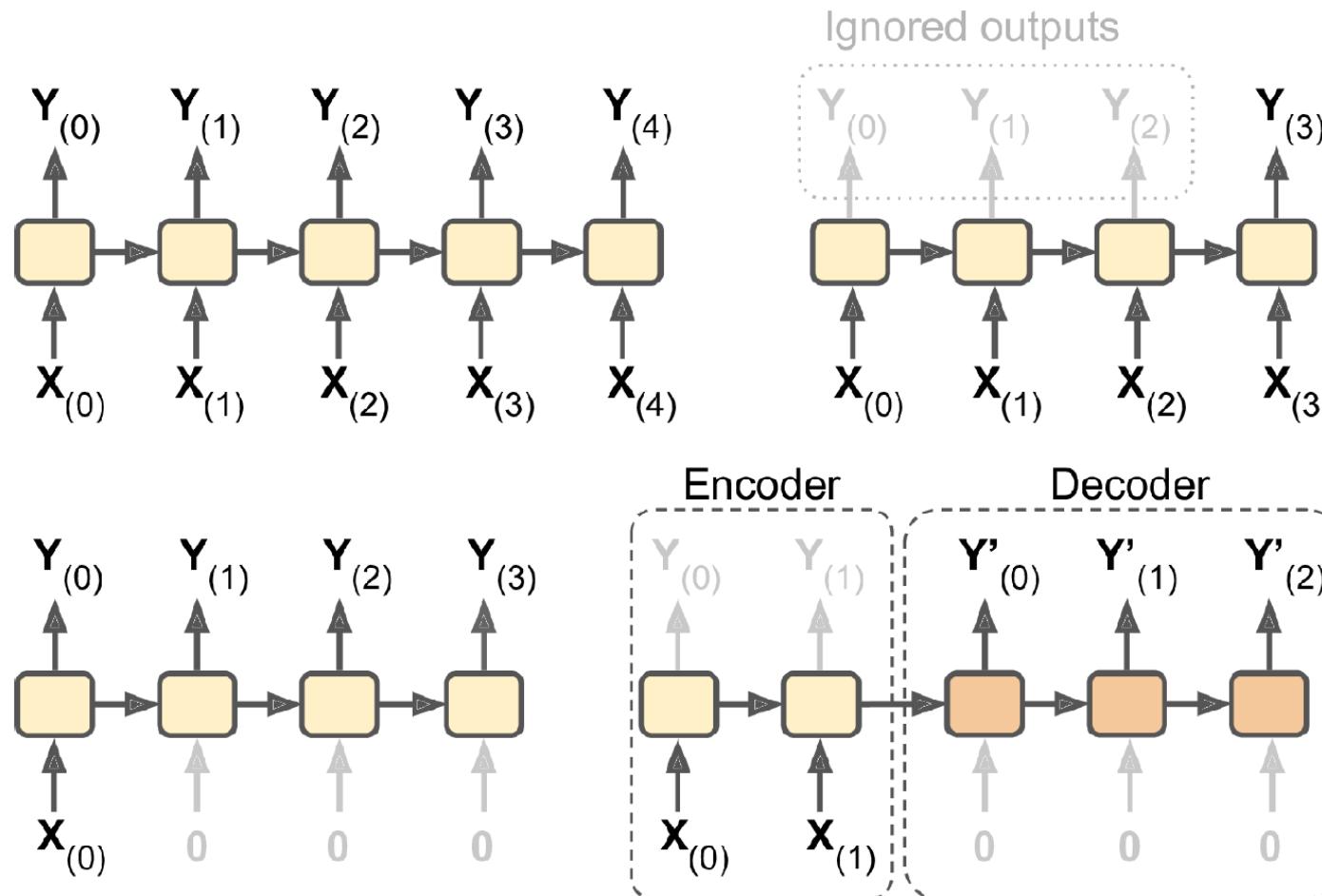


# Lecture Outline

- 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



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.

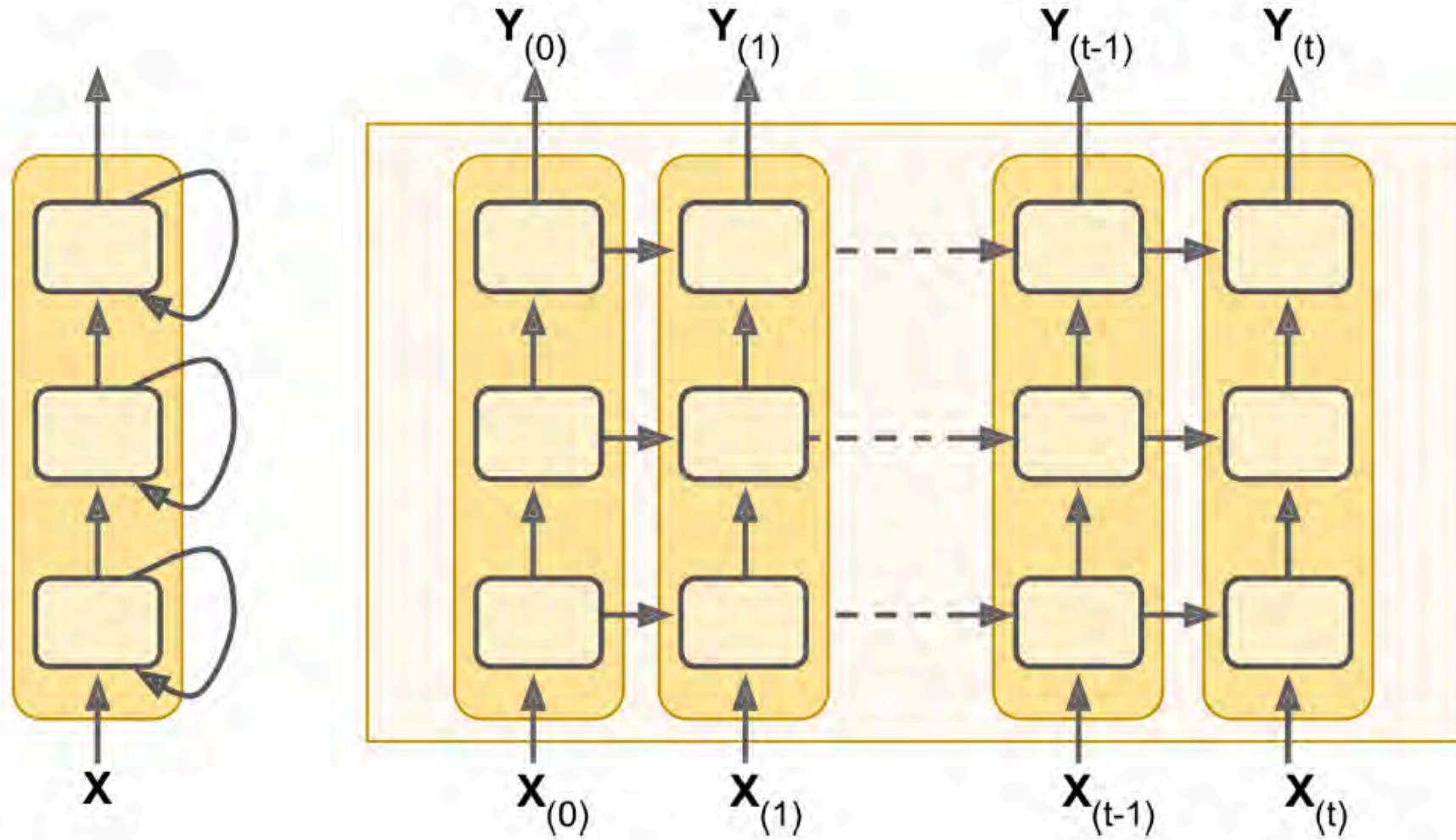


# Input and output sequences

- 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 sentence 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.



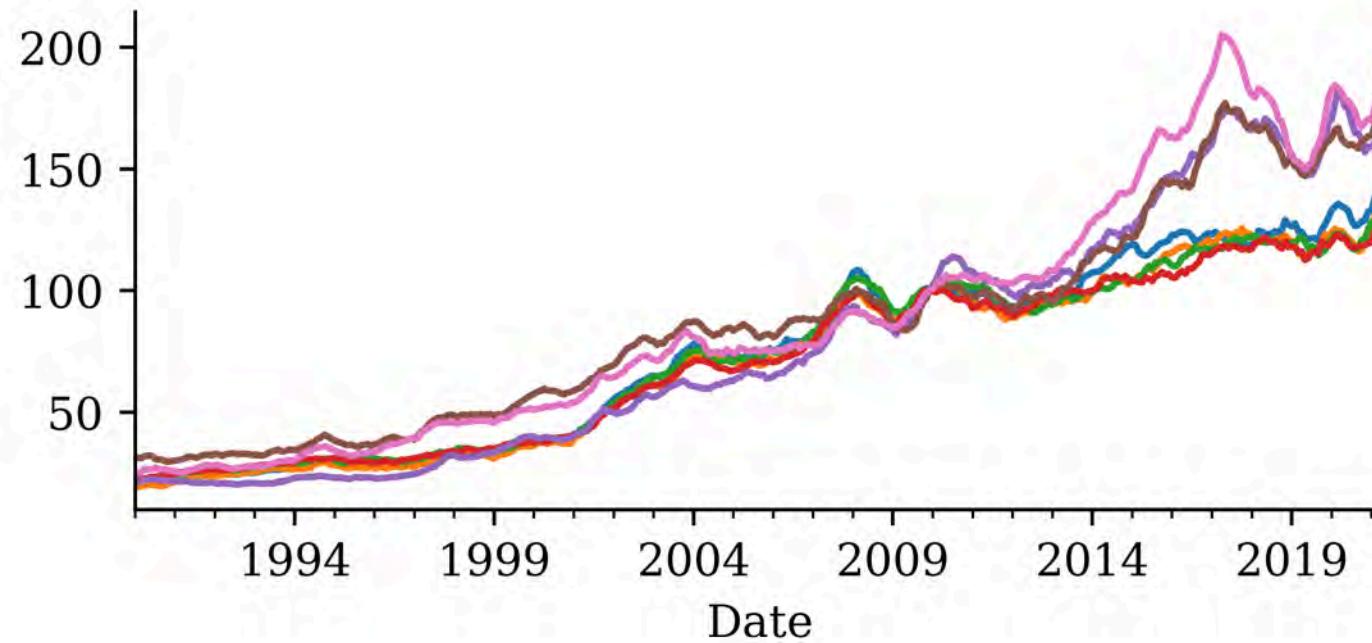
Source: Aurélien Géron (2019), *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition, Chapter 15.

# 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

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

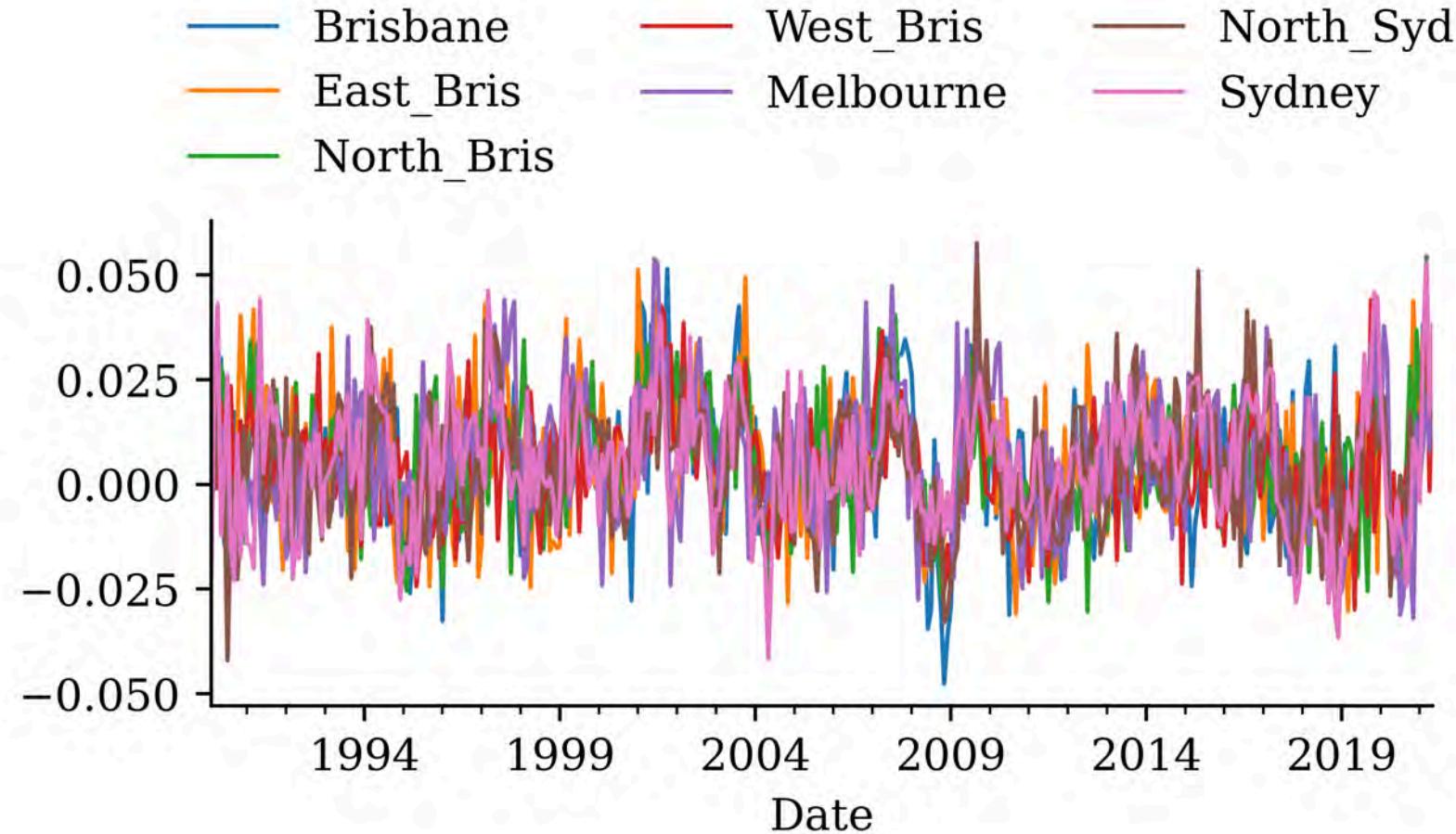
	Brisbane	East_Bris	North_Bris	West_Bris	Melbourne	North_Syd	Sydney
Date							
1990-02-28	0.03	-0.01	0.01	0.01	0.00	-0.00	-0.02
1990-03-31	0.01	0.03	0.01	0.01	0.02	-0.00	0.03
1990-04-30	0.02	0.02	0.01	-0.00	0.01	0.03	0.04
...	...	...	...	...	...	...	...
2021-03-31	0.04	0.04	0.03	0.04	0.02	0.05	0.05
2021-04-30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

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



# The size of the changes

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

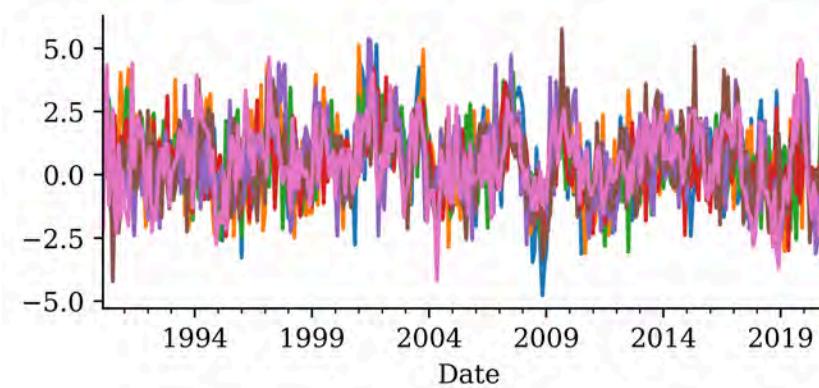
```
Brisbane      0.005496
East_Bris     0.005416
North_Bris    0.005024
West_Bris     0.004842
Melbourne     0.005677
North_Syd     0.004819
Sydney        0.005526
dtype: float64
```

```
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
```

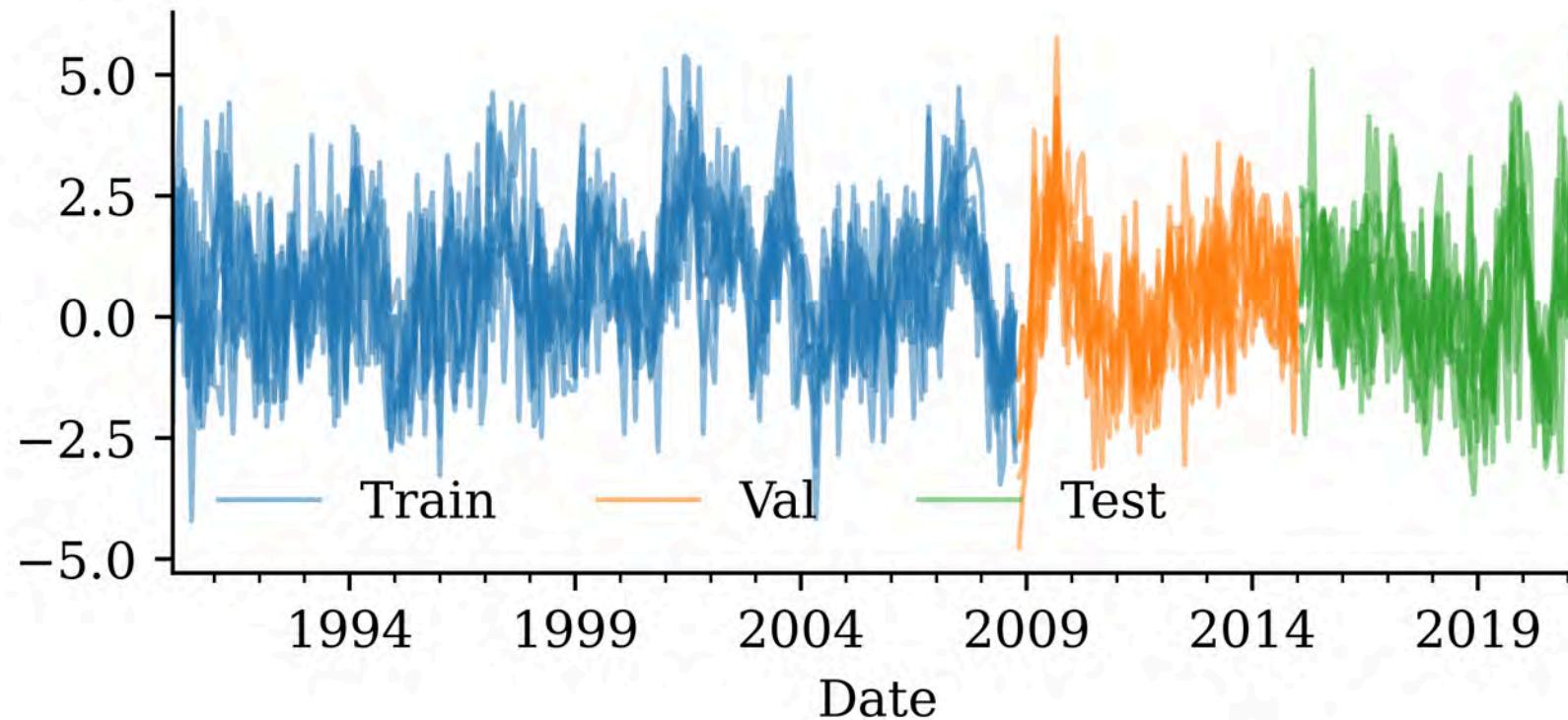
```
1 changes.plot(legend=False);
```



# Split *without* shuffling

```
1 num_train = int(0.6 * len(changes))
2 num_val = int(0.2 * len(changes))
3 num_test = len(changes) - num_train - num_val
4 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.

```
1 from keras.utils import timeseries_dataset_from_array
2
3 integers = range(10)
4 dummy_dataset = timeseries_dataset_from_array(
5     data=integers[:-3],
6     targets=integers[3:],
7     sequence_length=3,
8     batch_size=2,
9 )
10
11 for inputs, targets in dummy_dataset:
12     for i in range(inputs.shape[0]):
13         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

```

1 # Num. of input time series.
2 num_ts = changes.shape[1]
3
4 # How many prev. months to use.
5 seq_length = 6
6
7 # Predict the next month ahead.
8 ahead = 1
9
10 # The index of the first target.
11 delay = seq_length + ahead - 1

```

```

1 val_ds = timeseries_dataset_from_array(
2     changes[:-delay],
3     targets=target_suburb[delay:],
4     sequence_length=seq_length,
5     start_index=num_train,
6     end_index=num_train + num_val,
7 )

```

```

1 # Which suburb to predict.
2 target_suburb = changes["Sydney"]
3
4 train_ds = timeseries_dataset_from_array(
5     changes[:-delay],
6     targets=target_suburb[delay:],
7     sequence_length=seq_length,
8     end_index=num_train,
9 )

```

```

1 test_ds = timeseries_dataset_from_array(
2     changes[:-delay],
3     targets=target_suburb[delay:],
4     sequence_length=seq_length,
5     start_index=num_train + num_val,
6 )

```



# 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)))
```



# A dense network

```
1 from keras.layers import Input, Flatten
2 random.seed(1)
3 model_dense = Sequential([
4     Input((seq_length, num_ts)),
5     Flatten(),
6     Dense(50, activation="leaky_relu"),
7     Dense(20, activation="leaky_relu"),
8     Dense(1, activation="linear")
9 ])
10 model_dense.compile(loss="mse", optimizer="adam")
11 print(f"This model has {model_dense.count_params()} parameters.")
12
13 es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)
14 %time hist = model_dense.fit(X_train, y_train, epochs=1_000,
15     validation_data=(X_val, y_val), callbacks=[es], verbose=0);
```

This model has 3191 parameters.

Epoch 52: early stopping

Restoring model weights from the end of the best epoch: 2.

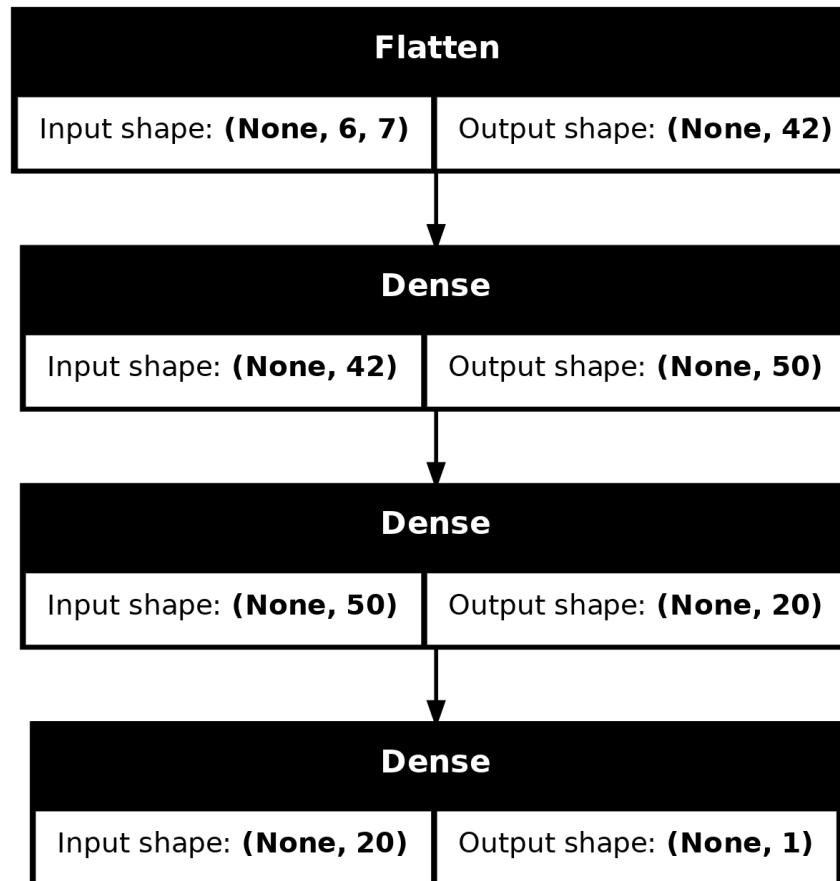
CPU times: user 901 ms, sys: 10.9 ms, total: 912 ms

Wall time: 941 ms



# Plot the model

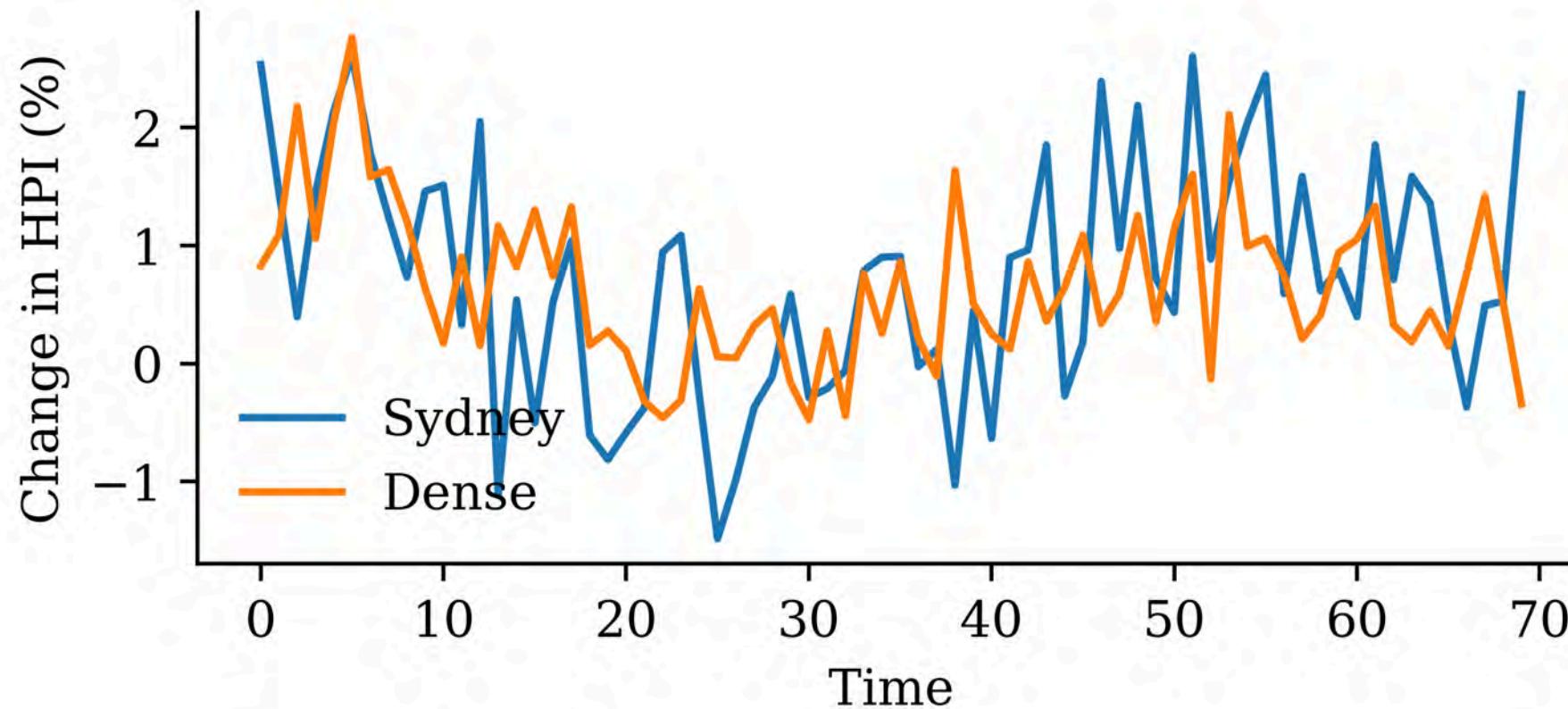
```
1 from keras.utils import plot_model  
2  
3 plot_model(model_dense, show_shapes=True)
```



# Assess the fits

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

```
1.043065071105957
```



# A SimpleRNN layer

```
1 random.seed(1)
2
3 model_simple = Sequential([
4     Input((seq_length, num_ts)),
5     SimpleRNN(50),
6     Dense(1, 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 2951 parameters.

Epoch 54: early stopping

Restoring model weights from the end of the best epoch: 4.

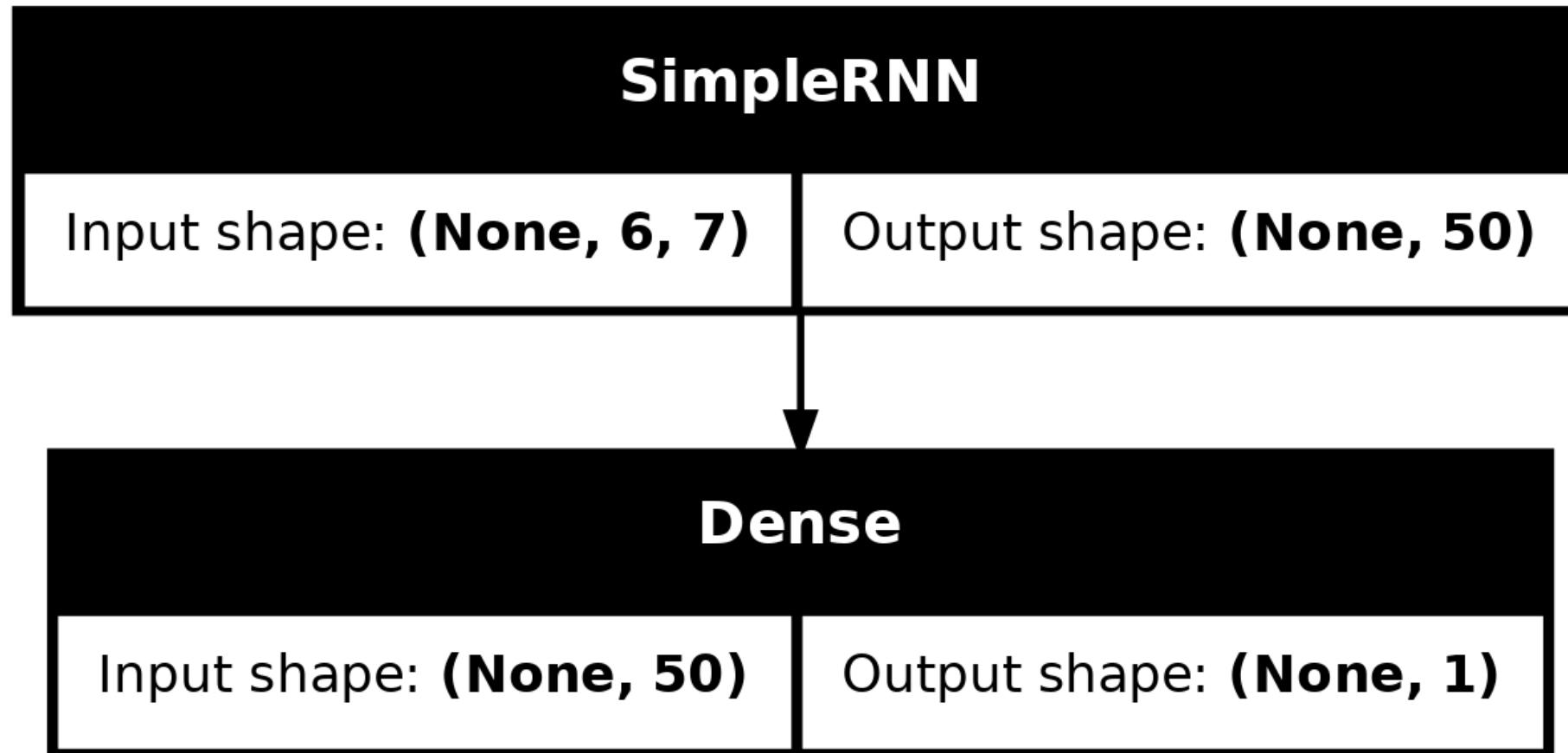
CPU times: user 1.76 s, sys: 1.94 ms, total: 1.77 s

Wall time: 1.8 s



# Plot the model

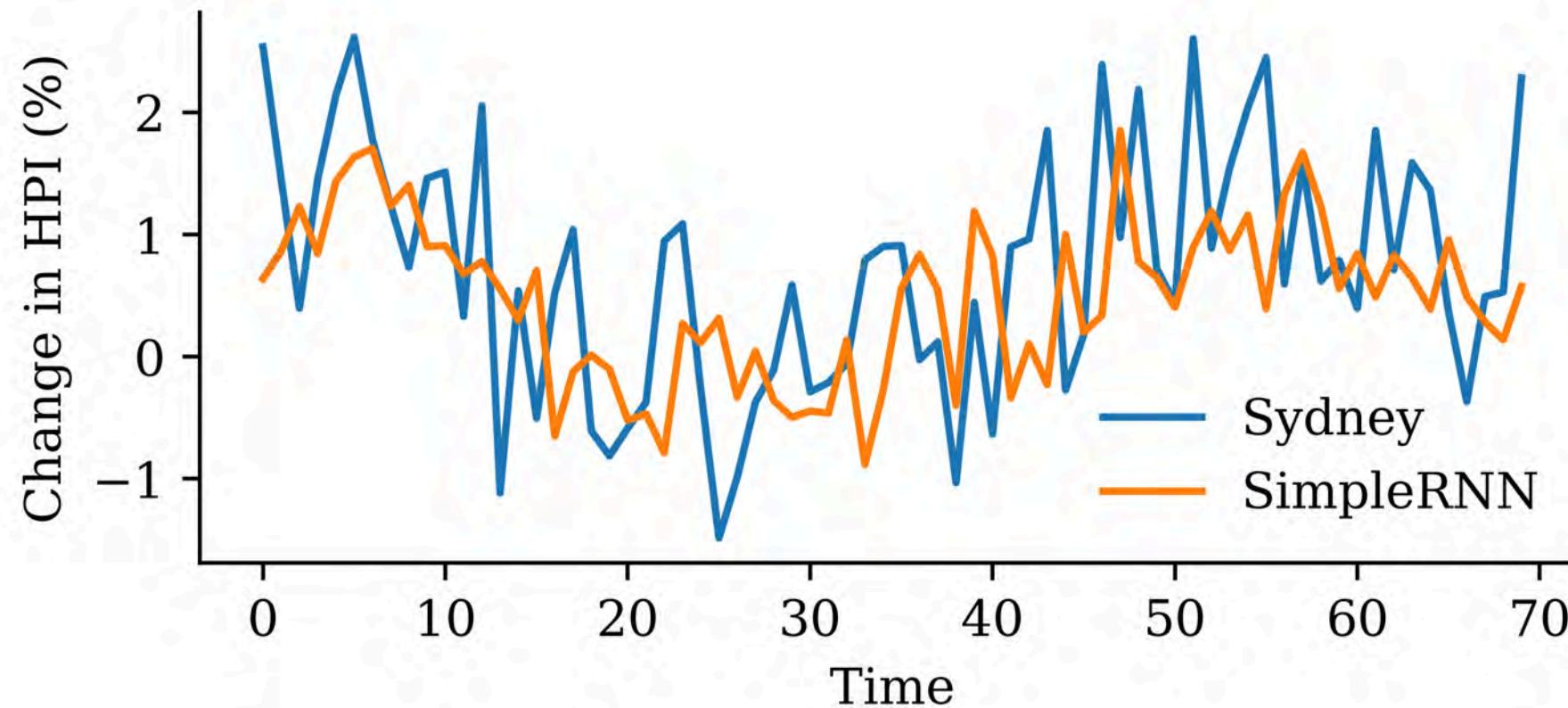
```
1 plot_model(model_simple, show_shapes=True)
```



# Assess the fits

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

```
0.9619883894920349
```



# A LSTM layer

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

Epoch 62: early stopping

Restoring model weights from the end of the best epoch: 12.

CPU times: user 2.56 s, sys: 30.8 ms, total: 2.6 s

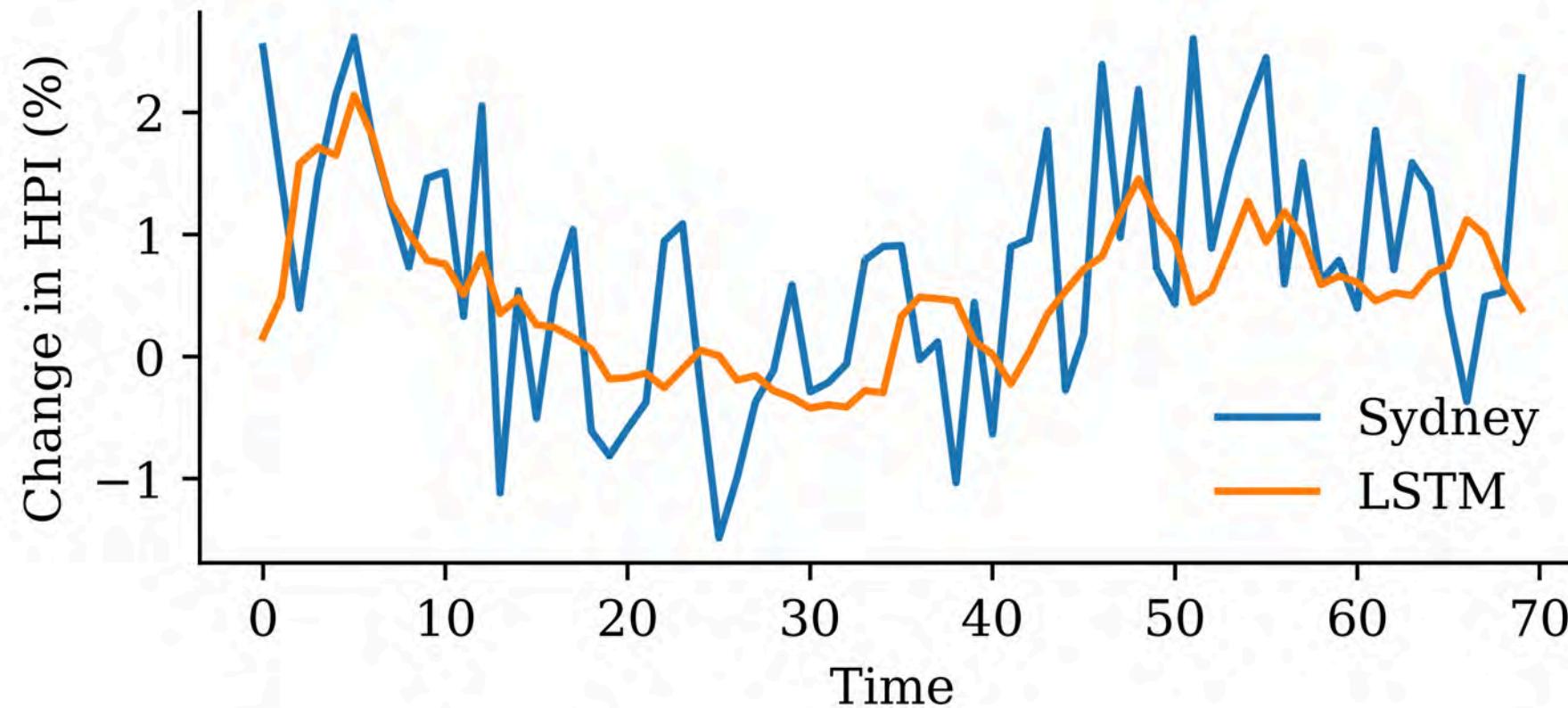
Wall time: 2.64 s



# Assess the fits

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

```
0.8037604093551636
```



# A GRU layer

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

Epoch 61: early stopping

Restoring model weights from the end of the best epoch: 11.

CPU times: user 2.95 s, sys: 25.1 ms, total: 2.97 s

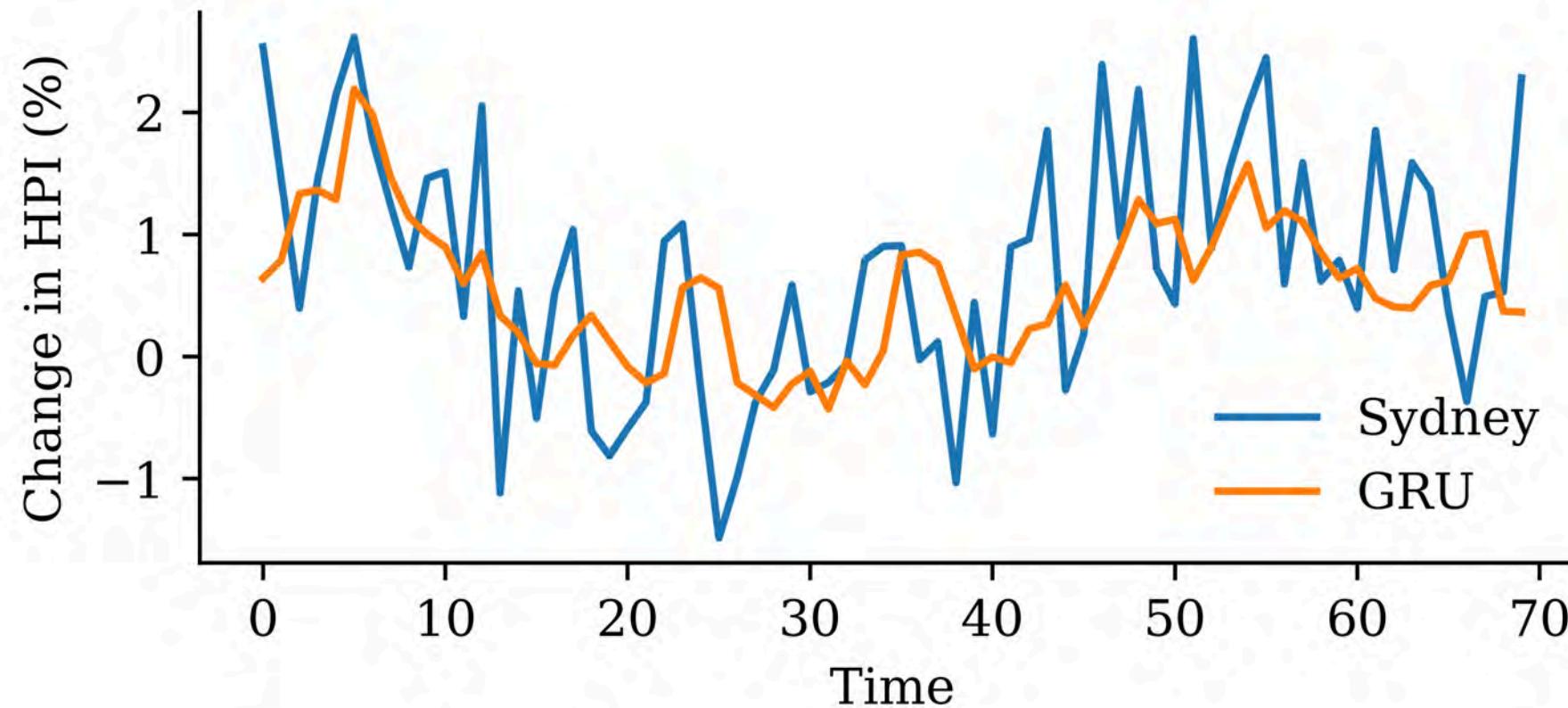
Wall time: 3.02 s



# Assess the fits

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

```
0.7643826007843018
```



# 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)
```

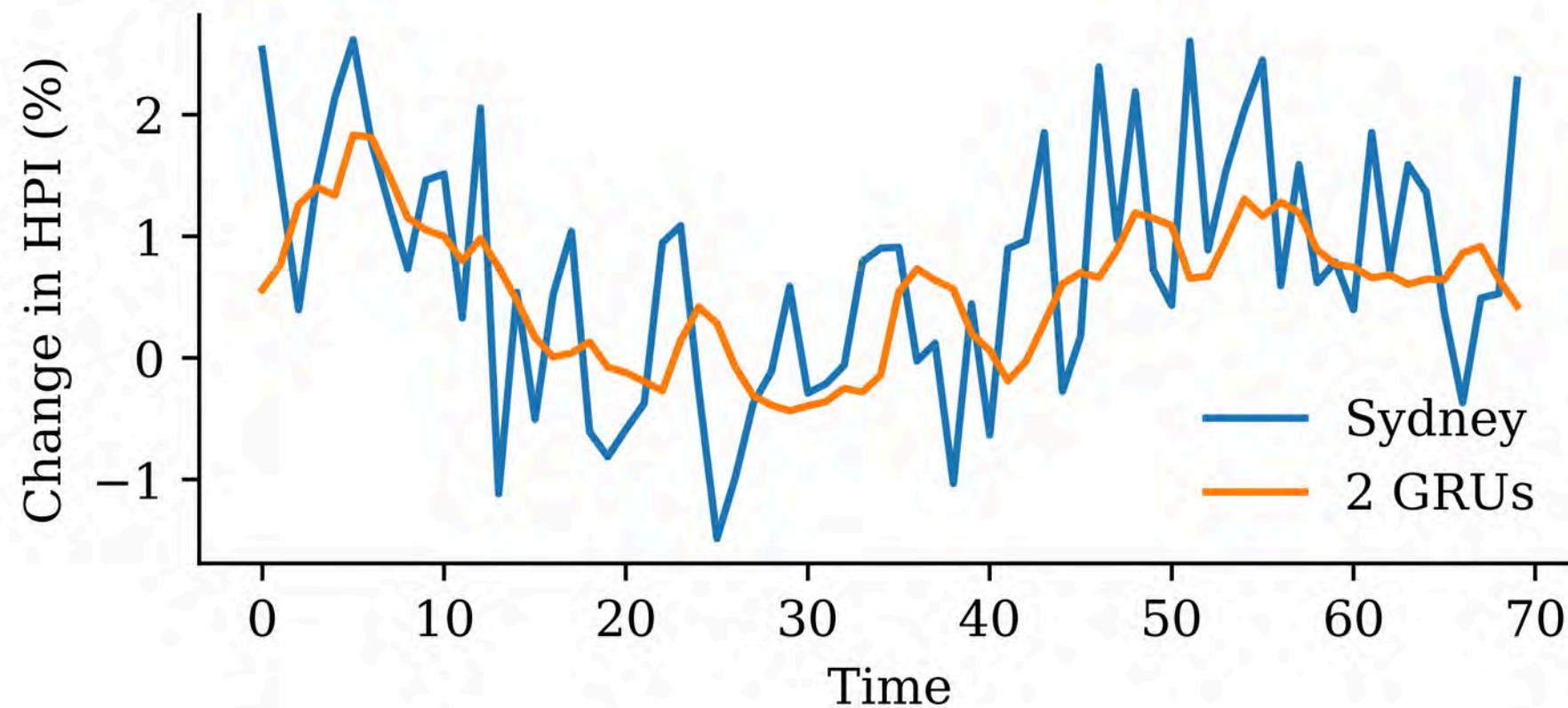
```
Epoch 55: early stopping
Restoring model weights from the end of the best epoch: 5.
CPU times: user 4.56 s, sys: 9.38 ms, total: 4.57 s
Wall time: 4.61 s
```



# Assess the fits

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

```
0.7825747728347778
```



# Compare the models

	Model	MSE
0	Dense	1.043065
1	SimpleRNN	0.961988
2	LSTM	0.803760
4	2 GRUs	0.782575
3	GRU	0.764383

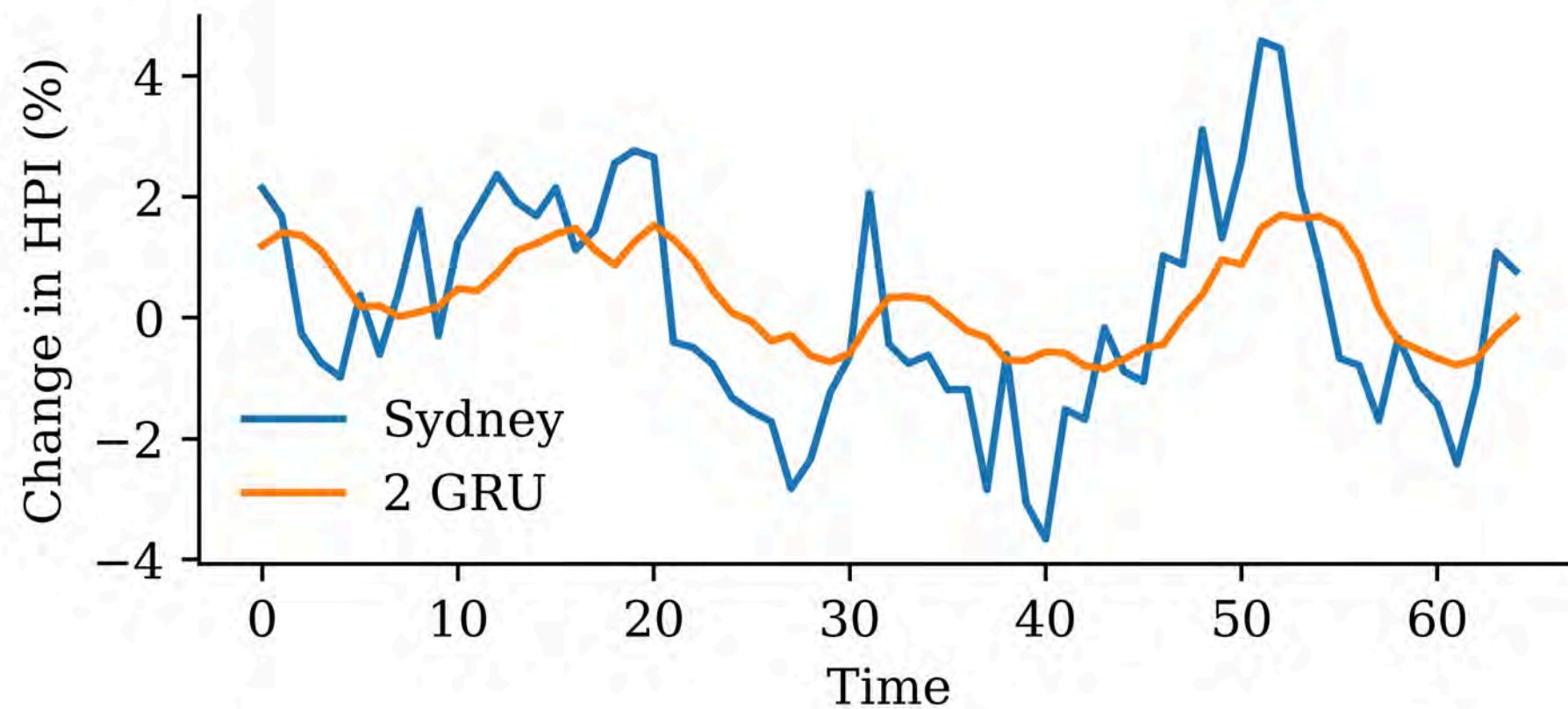
The network with two GRU layers is the best.

```
1 model_two_grus.evaluate(test_ds, verbose=0)
```

```
2.023635149002075
```



# 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.

```
1 val_ds = timeseries_dataset_from_array(  
2     changes[:-delay],  
3     targets=changes[delay:],  
4     sequence_length=seq_length,  
5     start_index=num_train,  
6     end_index=num_train + num_val,  
7 )
```

```
1 train_ds = timeseries_dataset_from_array(  
2     changes[:-delay],  
3     targets=changes[delay:],  
4     sequence_length=seq_length,  
5     end_index=num_train,  
6 )
```

```
1 test_ds = timeseries_dataset_from_array(  
2     changes[:-delay],  
3     targets=changes[delay:],  
4     sequence_length=seq_length,  
5     start_index=num_train + num_val,  
6 )
```



# 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

```
1 random.seed(1)
2 model_dense = Sequential([
3     Input((seq_length, num_ts)),
4     Flatten(),
5     Dense(50, activation="leaky_relu"),
6     Dense(20, activation="leaky_relu"),
7     Dense(num_ts, activation="linear")
8 ])
9 model_dense.compile(loss="mse", optimizer="adam")
10 print(f"This model has {model_dense.count_params()} parameters.")
11
12 es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)
13 %time hist = model_dense.fit(X_train, y_train, epochs=1_000, \
14     validation_data=(X_val, y_val), callbacks=[es], verbose=0);
```

This model has 3317 parameters.

Epoch 69: early stopping

Restoring model weights from the end of the best epoch: 19.

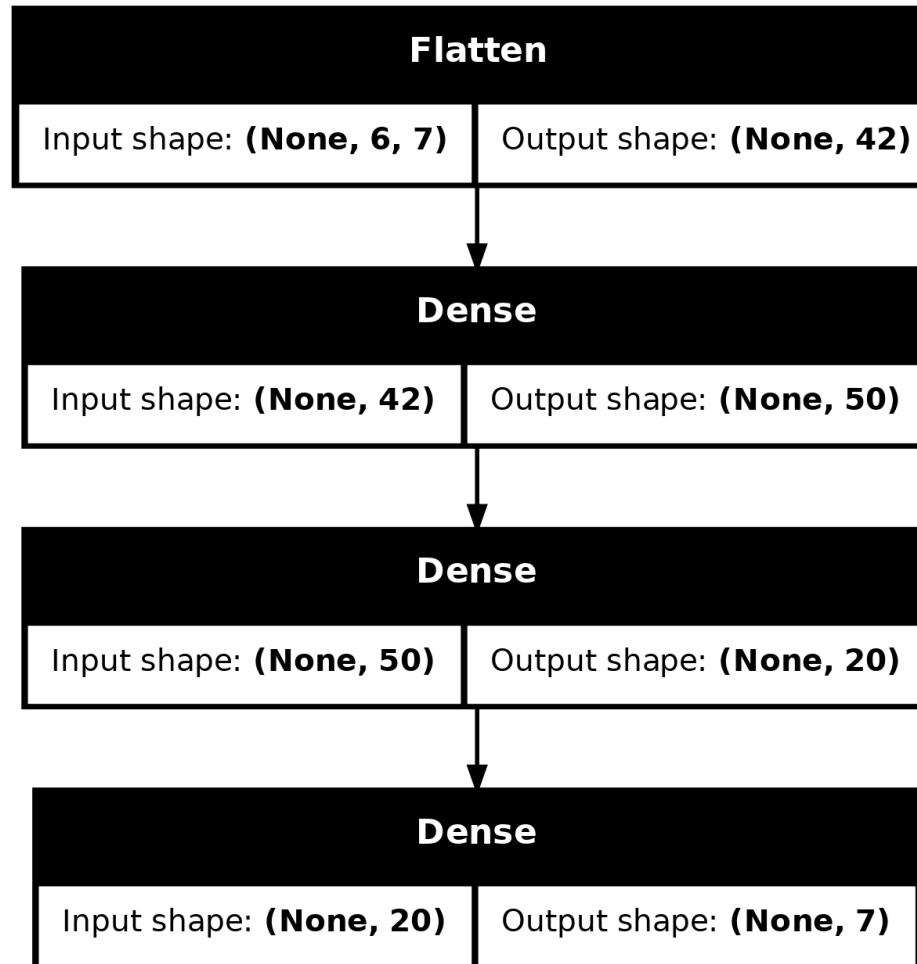
CPU times: user 1.24 s, sys: 6.45 ms, total: 1.25 s

Wall time: 1.28 s



# Plot the model

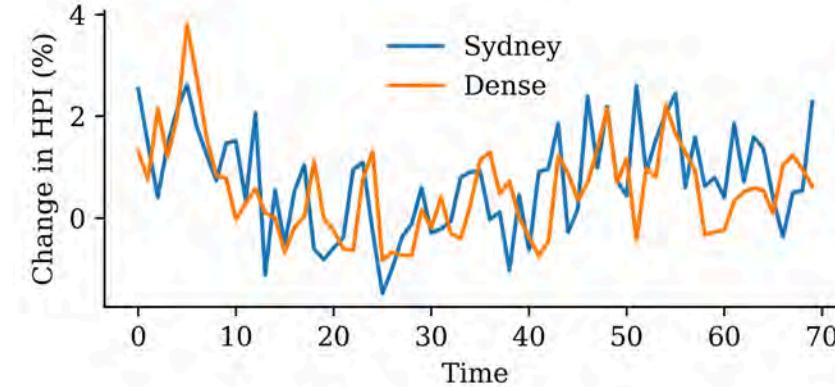
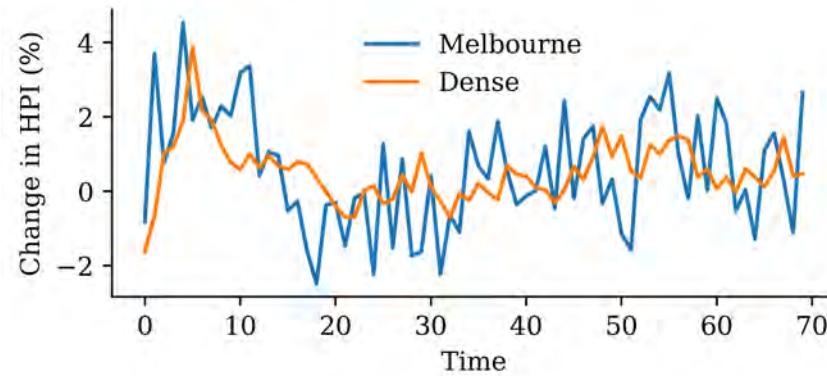
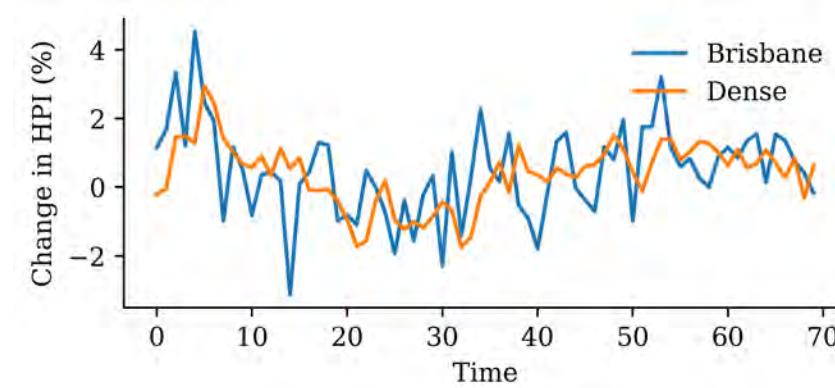
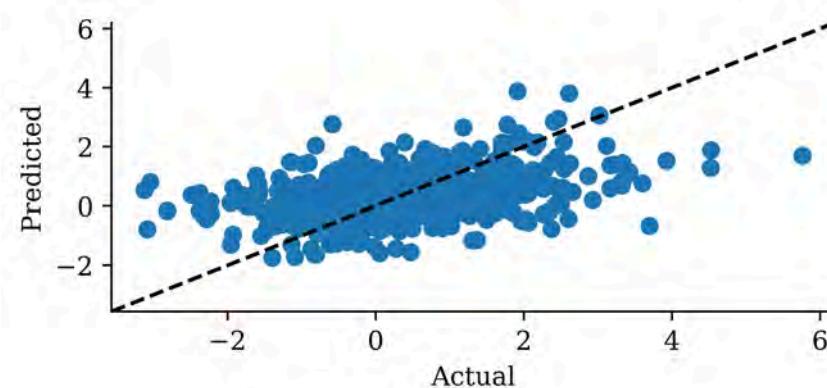
```
1 plot_model(model_dense, show_shapes=True)
```



# Assess the fits

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

1.5469738245010376



# 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 62: early stopping

Restoring model weights from the end of the best epoch: 12.

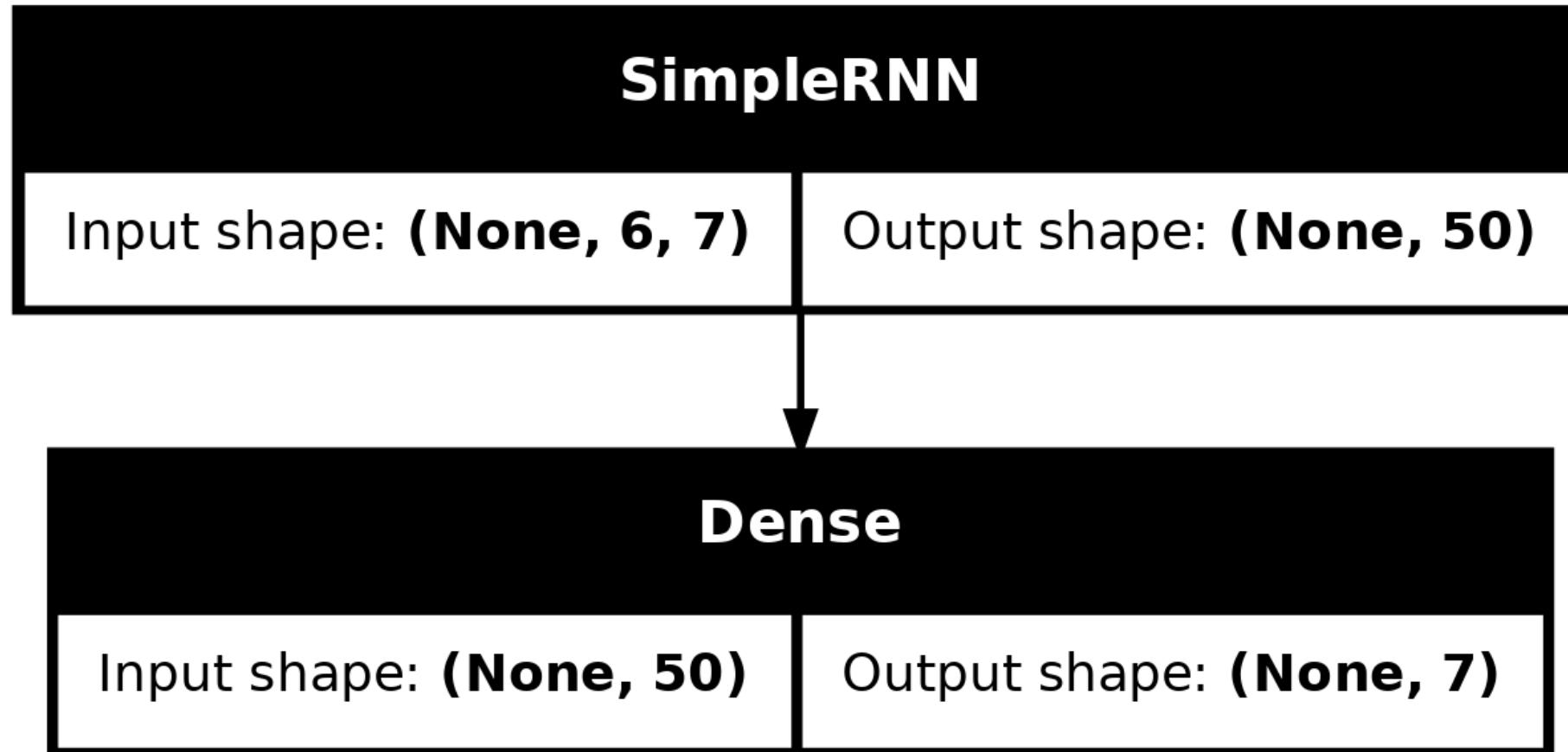
CPU times: user 2.01 s, sys: 33.4 ms, total: 2.05 s

Wall time: 2.09 s



# Plot the model

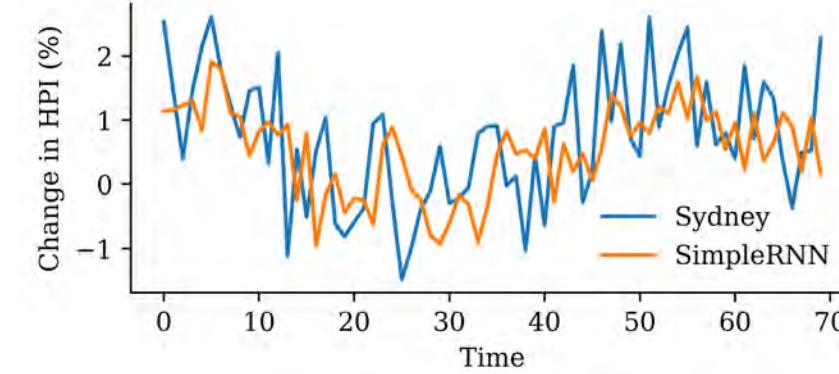
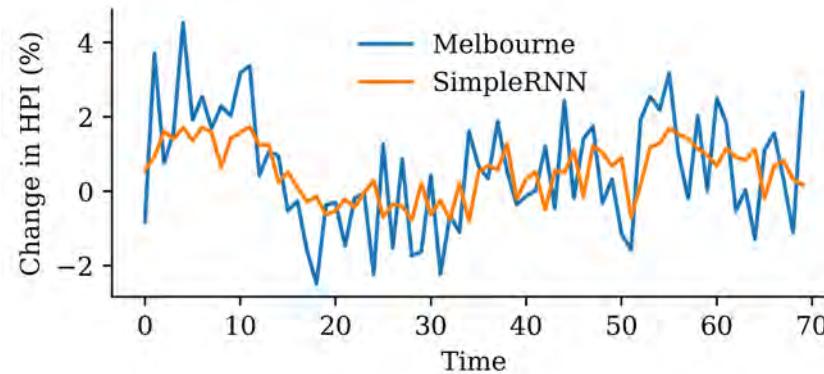
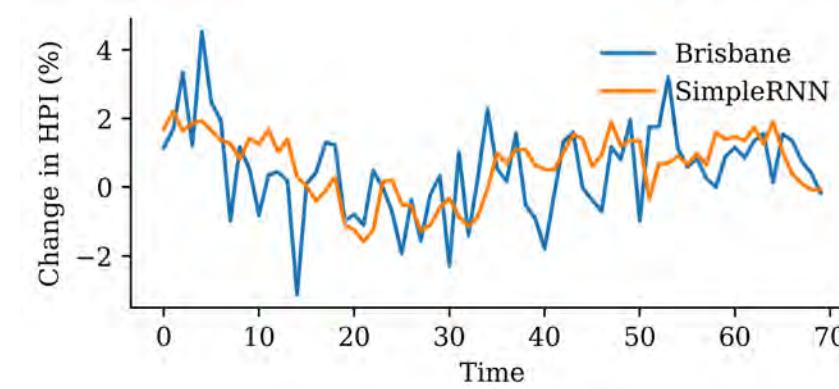
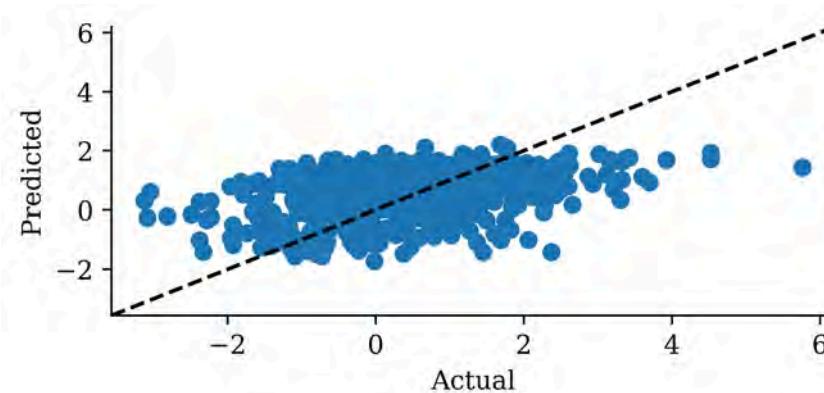
```
1 plot_model(model_simple, show_shapes=True)
```



# Assess the fits

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

1.473482370376587



# A LSTM layer

```
1 random.seed(1)
2
3 model_lstm = Sequential([
4     Input((seq_length, num_ts)),
5     LSTM(50),
6     Dense(num_ts, activation="linear")
7 ])
8
9 model_lstm.compile(loss="mse", optimizer="adam")
10
11 es = EarlyStopping(patience=50, restore_best_weights=True, verbose=1)
12
13 %time hist = model_lstm.fit(X_train, y_train, epochs=1_000, \
14     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 3.08 s, sys: 20 ms, total: 3.1 s

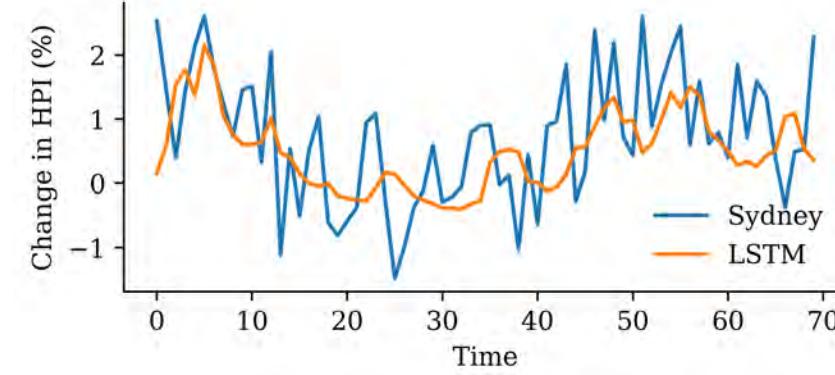
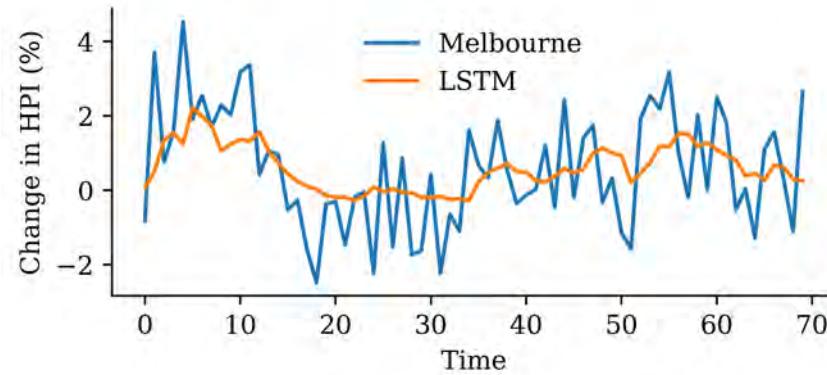
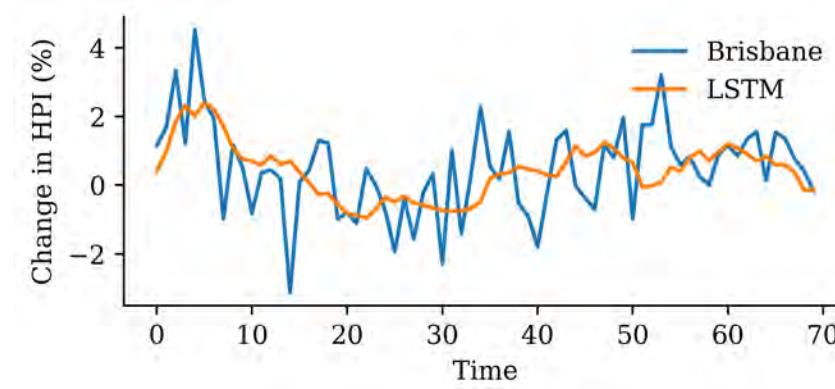
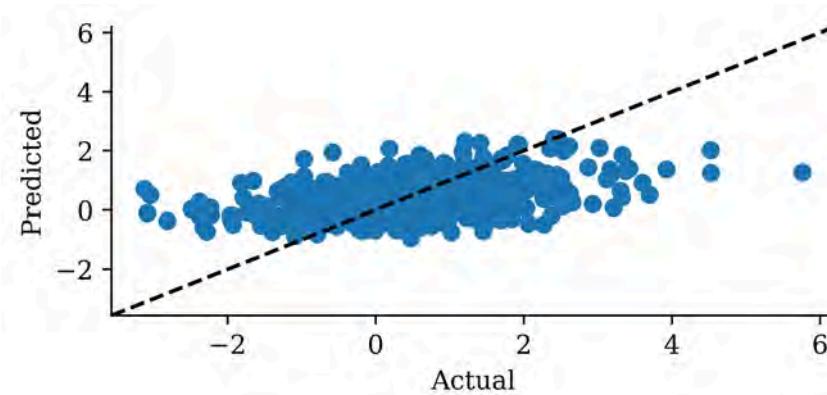
Wall time: 3.16 s



# Assess the fits

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

1.360884428024292



# A GRU layer

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

Epoch 77: early stopping

Restoring model weights from the end of the best epoch: 27.

CPU times: user 3.74 s, sys: 27.5 ms, total: 3.77 s

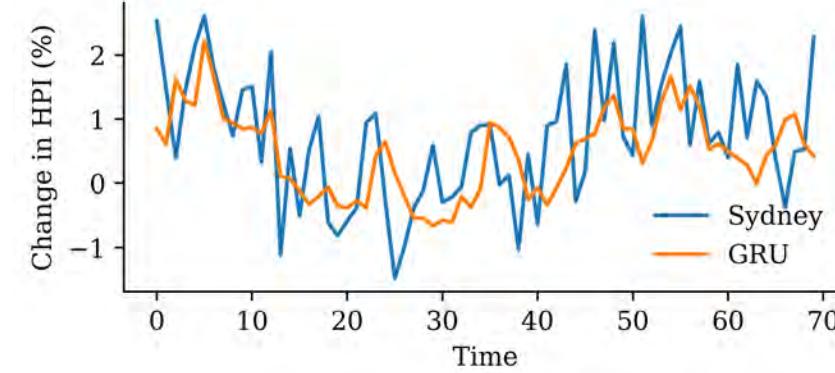
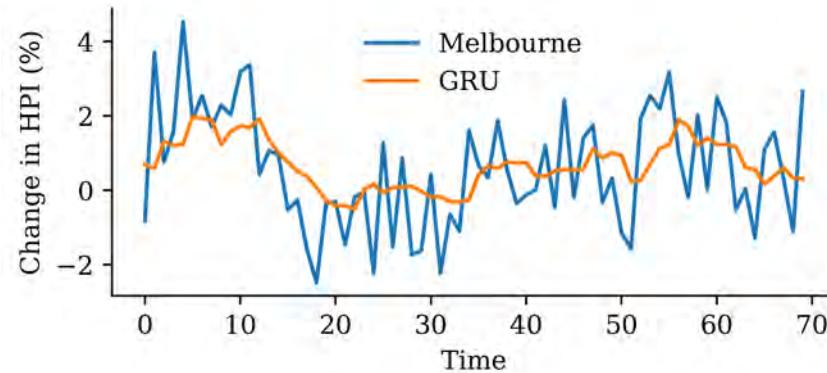
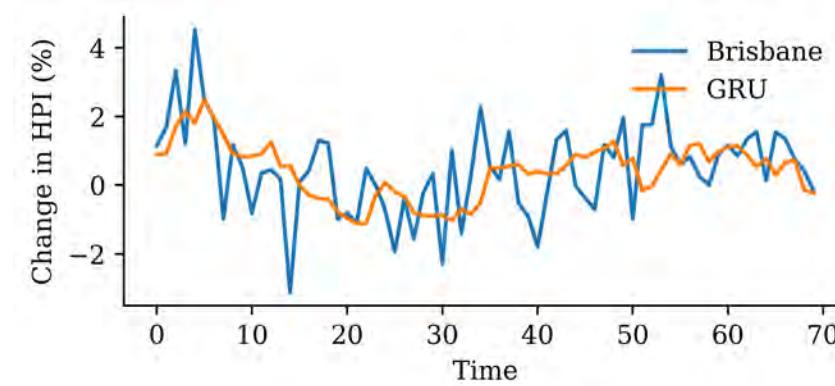
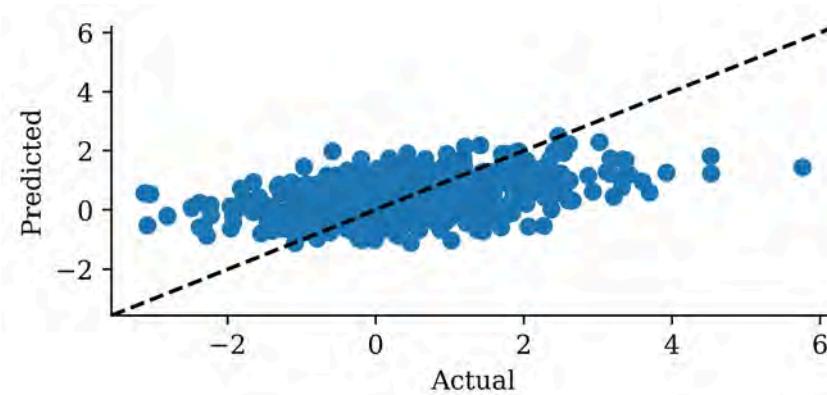
Wall time: 3.84 s



# Assess the fits

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

1.3418978452682495



# 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(num_ts, 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 65: early stopping

Restoring model weights from the end of the best epoch: 15.

CPU times: user 5.42 s, sys: 8 ms, total: 5.43 s

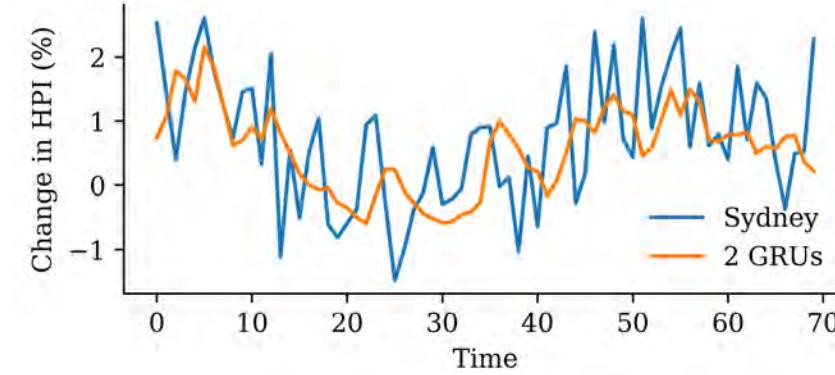
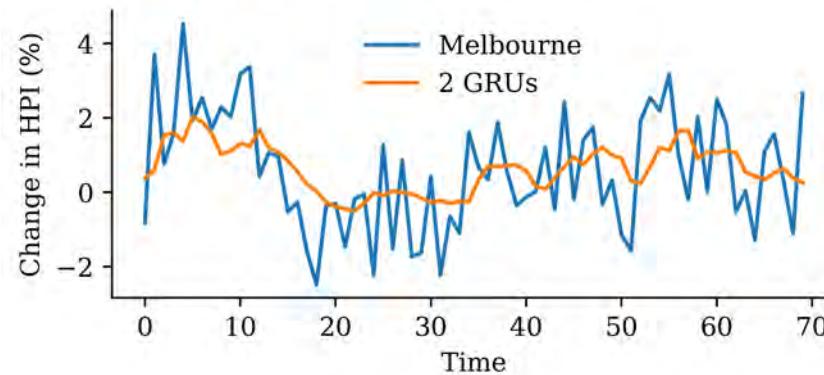
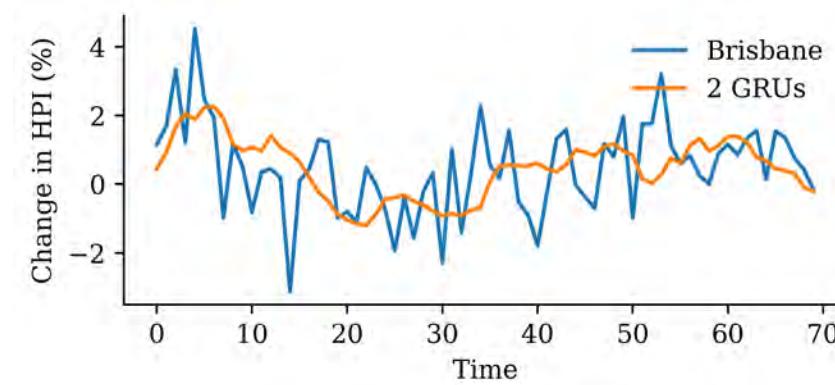
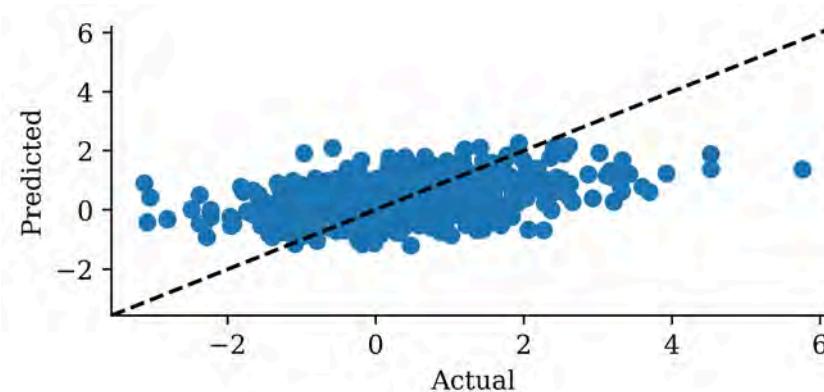
Wall time: 5.48 s



# Assess the fits

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

1.378563404083252



# Compare the models

	Model	MSE
0	Dense	1.546974
1	SimpleRNN	1.473482
4	2 GRUs	1.378563
2	LSTM	1.360884
3	GRU	1.341898

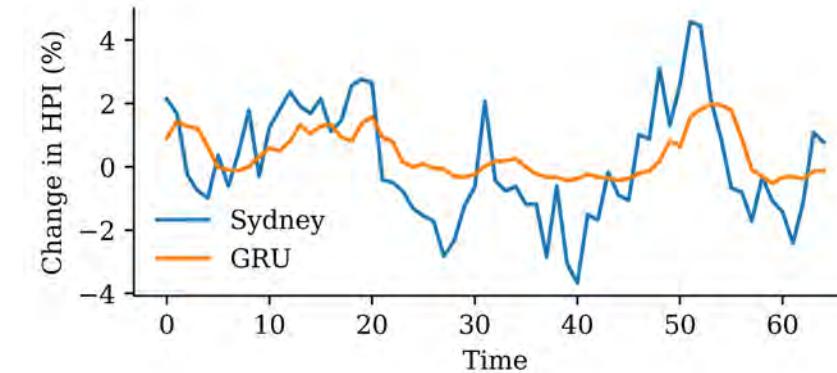
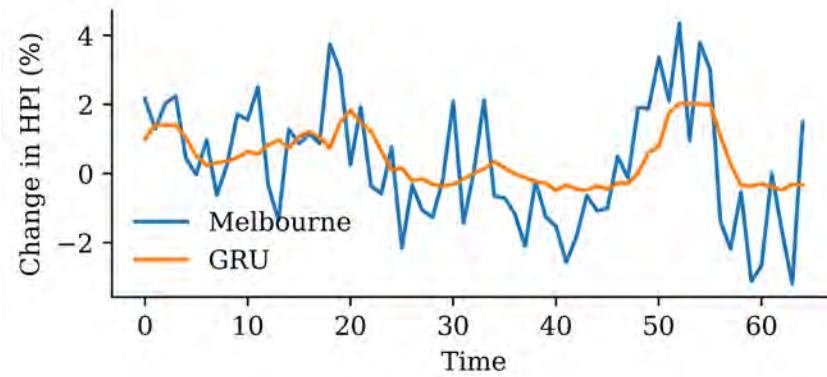
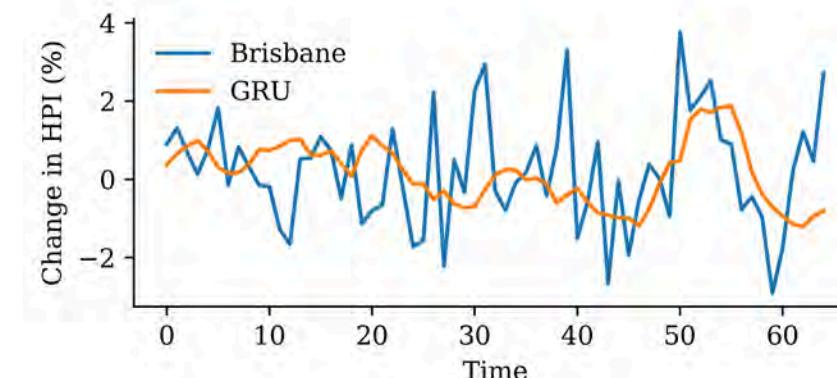
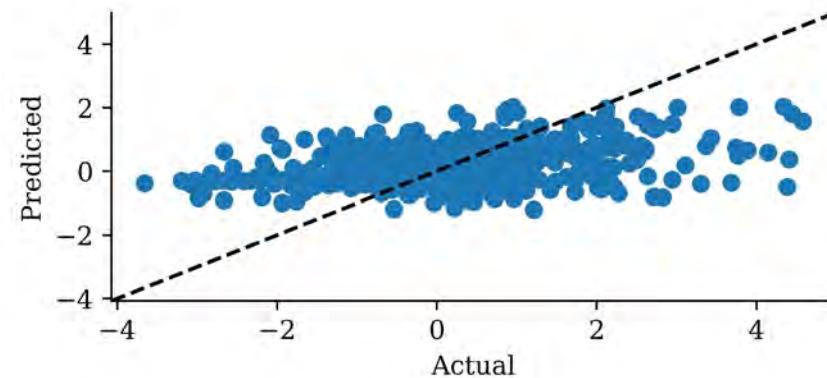
The network with an LSTM layer is the best.

```
1 model_lstm.evaluate(test_ds, verbose=0)
```

```
1.9254661798477173
```



# Test set



# 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 : 3.11.9

IPython version : 8.24.0

keras : 3.3.3

matplotlib: 3.9.0

numpy : 1.26.4

pandas : 2.2.2

seaborn : 0.13.2

scipy : 1.11.0

torch : 2.3.1

tensorflow: 2.16.1

tf\_keras : 2.16.0



# Glossary

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

