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
- RNN Demo
- SimpleRNN maths
- More complex RNNs

Tabular data vs time series data

Tabular data

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

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

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

Time series data

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

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

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

Attributes of time series data

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

Note

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

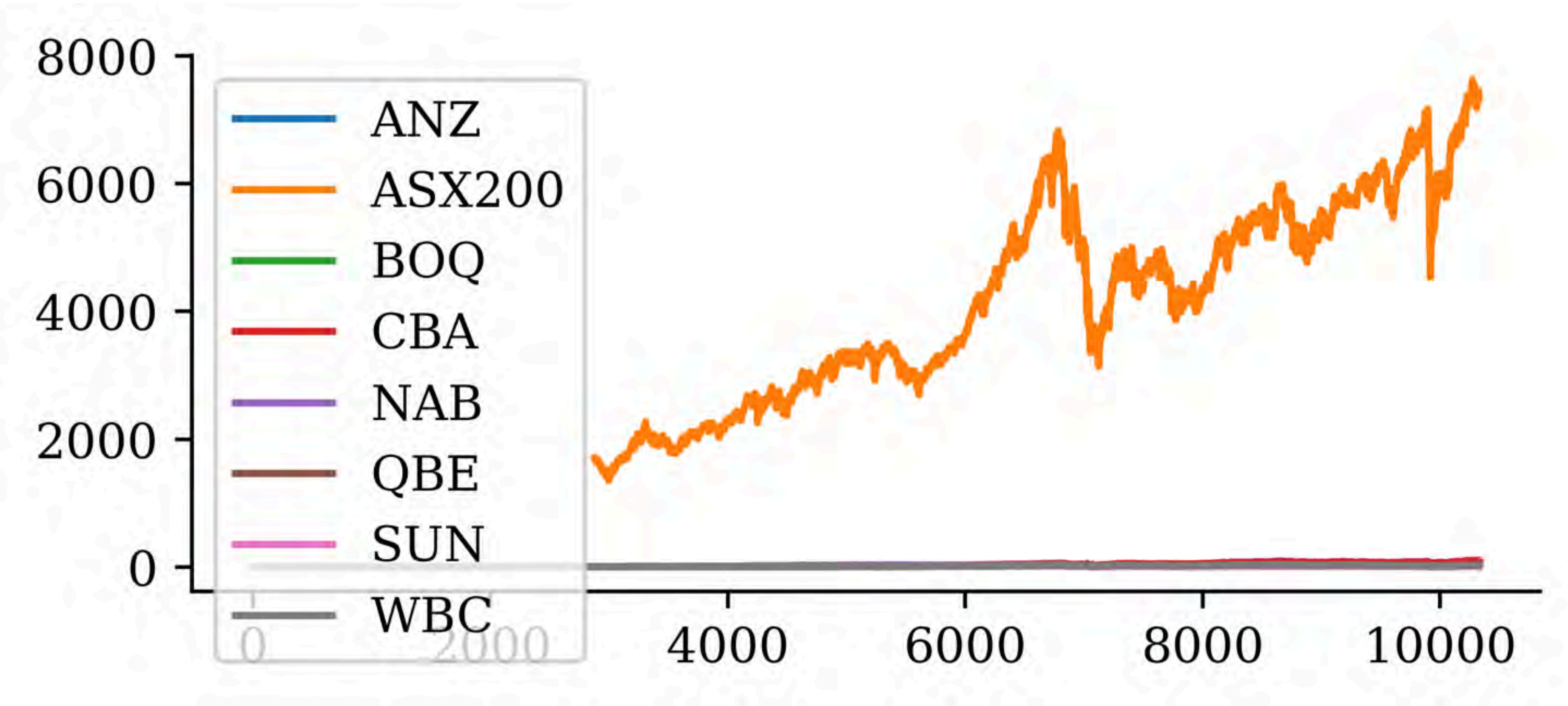
Australian financial stocks

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

	Date	ANZ	ASX200	BOQ	CBA	NAB	Q
0	1981-01-02	1.588896	NaN	NaN	NaN	1.791642	Na
1	1981-01-05	1.548452	NaN	NaN	NaN	1.791642	Na
2	1981-01-06	1.600452	NaN	NaN	NaN	1.791642	Na
…	…	…	…	…	…	…	…
10327	2021-10	28.600000	7430.4	8.97	106.86	29.450000	12

Plot

```
stocks.plot()
```



Data types and NA values

```python
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
```

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

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

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

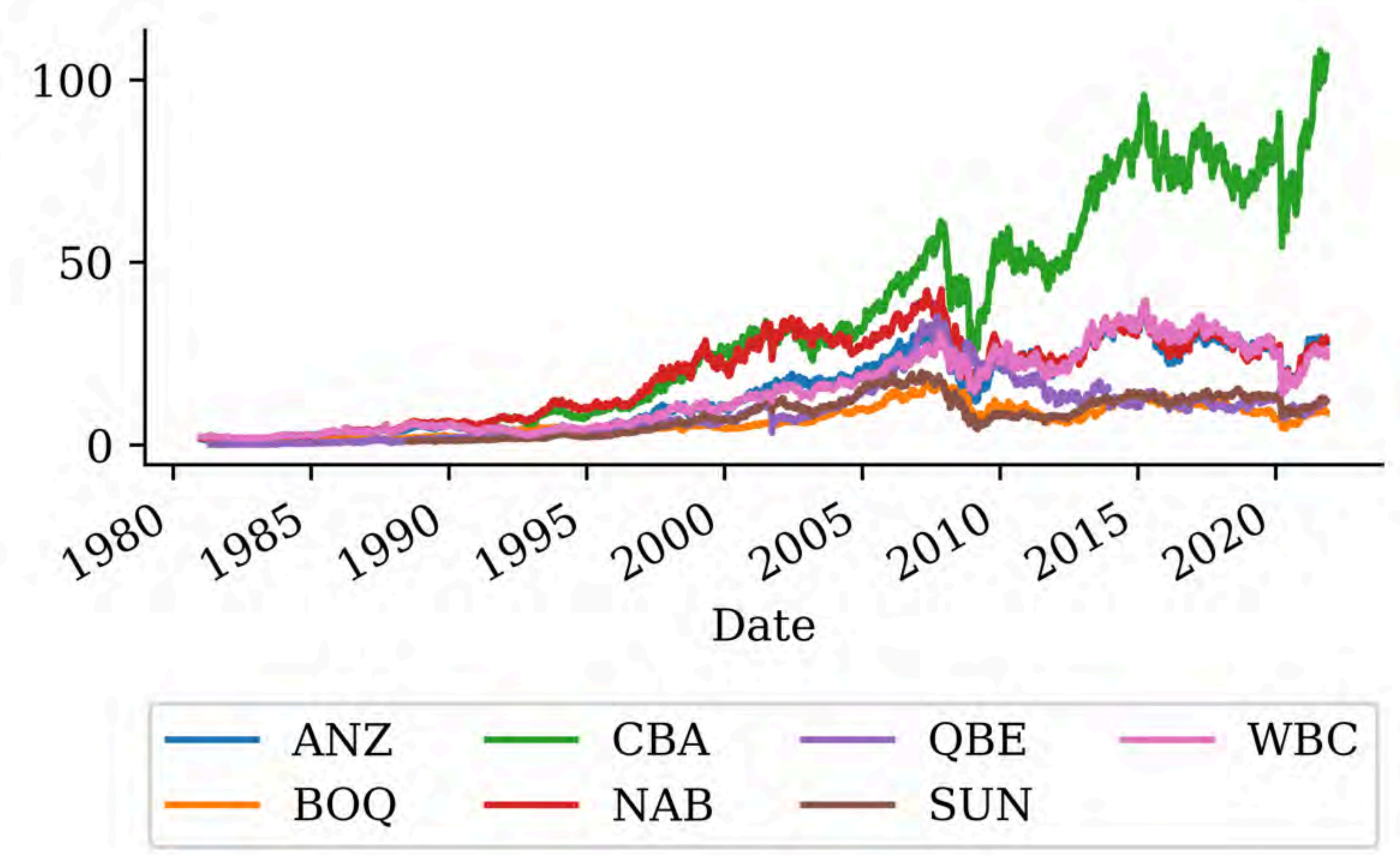
Set the index to the date

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

	ANZ	BOQ	CBA	NAB	QBE	SUN	
Date							
1981-01-02	1.588896	NaN	NaN	1.791642	NaN	NaN	2.1994
1981-01-05	1.548452	NaN	NaN	1.791642	NaN	NaN	2.1633
1981-01-06	1.600452	NaN	NaN	1.791642	NaN	NaN	2.1994
...	...	...	...	...	...	...	...

Plot II

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



Can index using dates I

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

	ANZ	BOQ	CBA	NAB	QBE	SUN
Date						
2010-01-04	22.89	10.772147	54.573702	26.046571	25.21	8.142453
2010-01-05	23.00	10.910369	55.399220	26.379283	25.34	8.264684
2010-01-06	22.66	10.855080	55.677708	25.865956	24.95	8.086039
2010-01-07	22.12	10.523346	55.140624	25.656823	24.50	8.198867

Can index using dates II

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

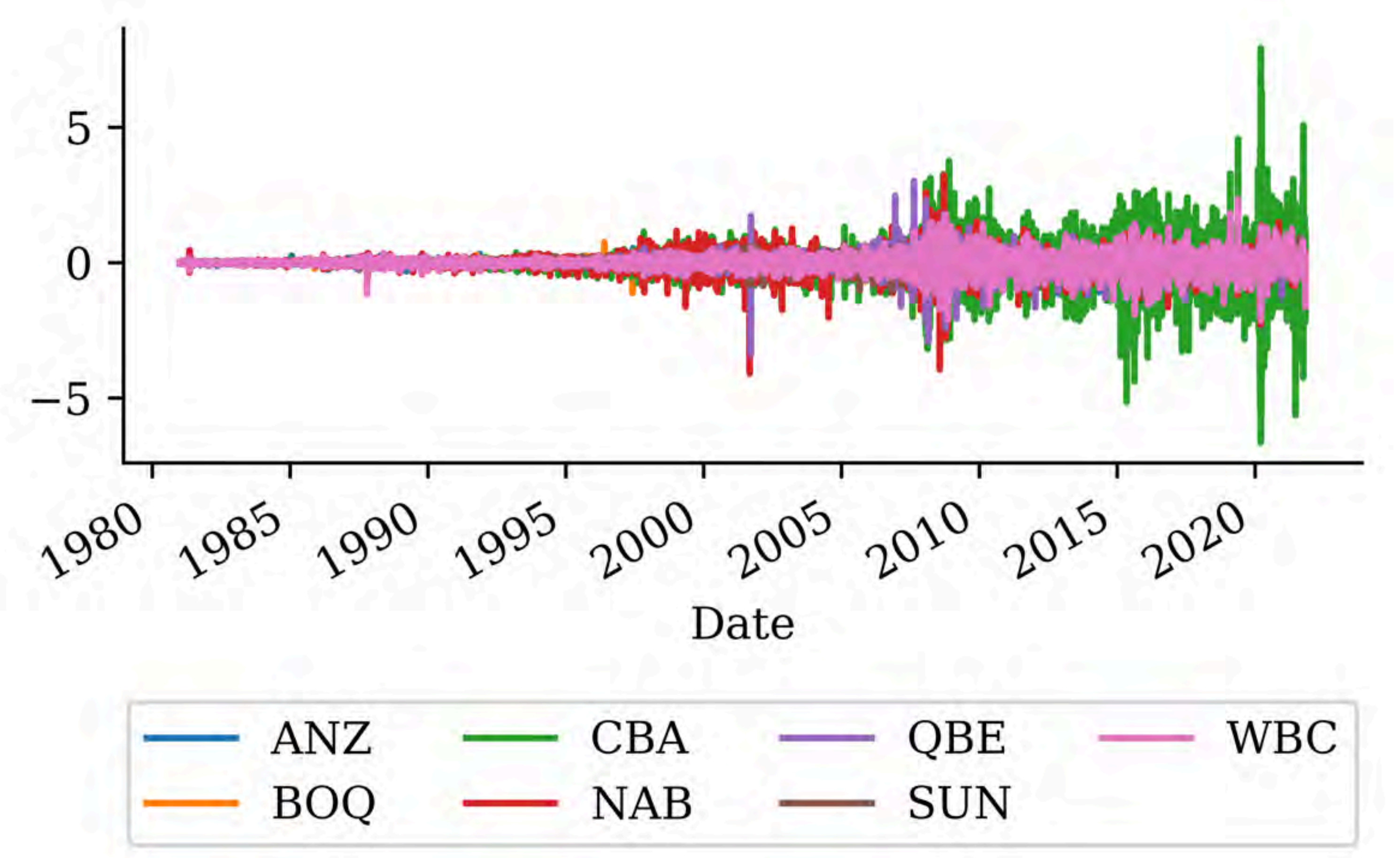
```
stocks.loc["2019-12":"2020", ["CBA"]]
```

	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

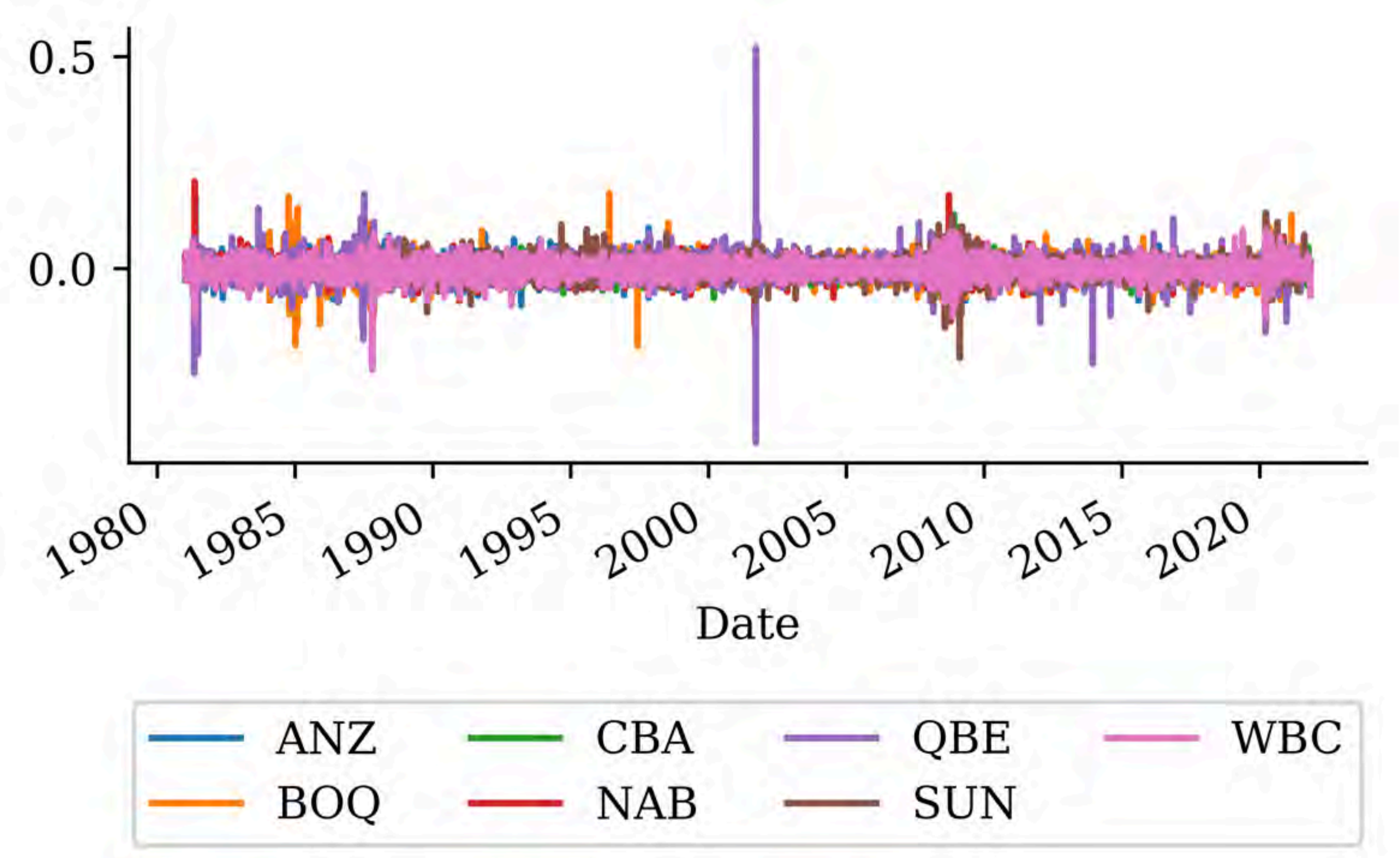
Can look at the first differences

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



Can look at the percentage changes

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



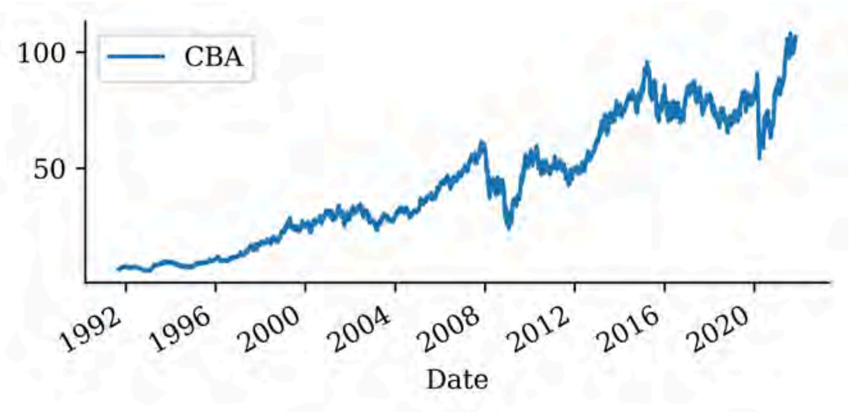
Focus on one stock

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

```
stock.plot()
```



Find first non-missing value

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

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

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

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

```
CBA      8
dtype: int64
```

Fill in the missing values

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

```python
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

```python
stock = stock.ffill()
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

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

```
CBA    0
dtype: int64
```

UNSW SYDNEY

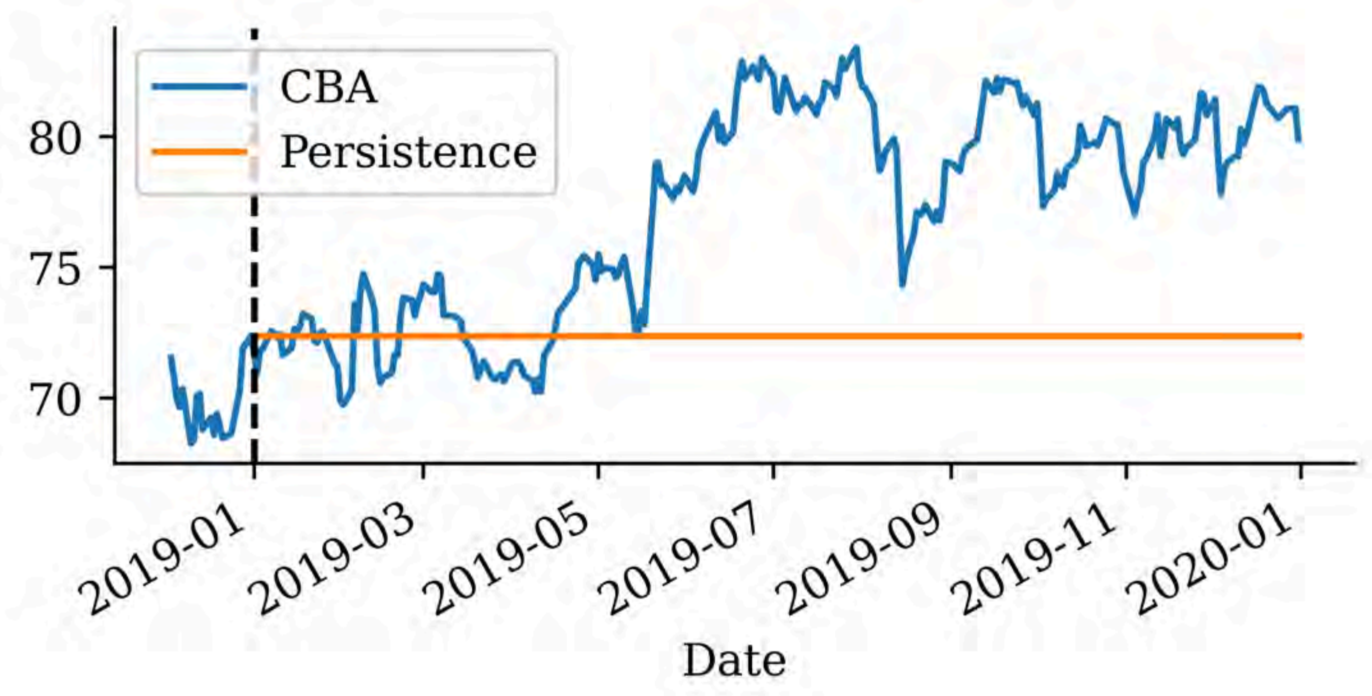
Lecture Outline

- Time Series
- **Baseline forecasts**
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- RNN Demo
- SimpleRNN maths
- More complex RNNs

Persistence forecast

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

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



Trend

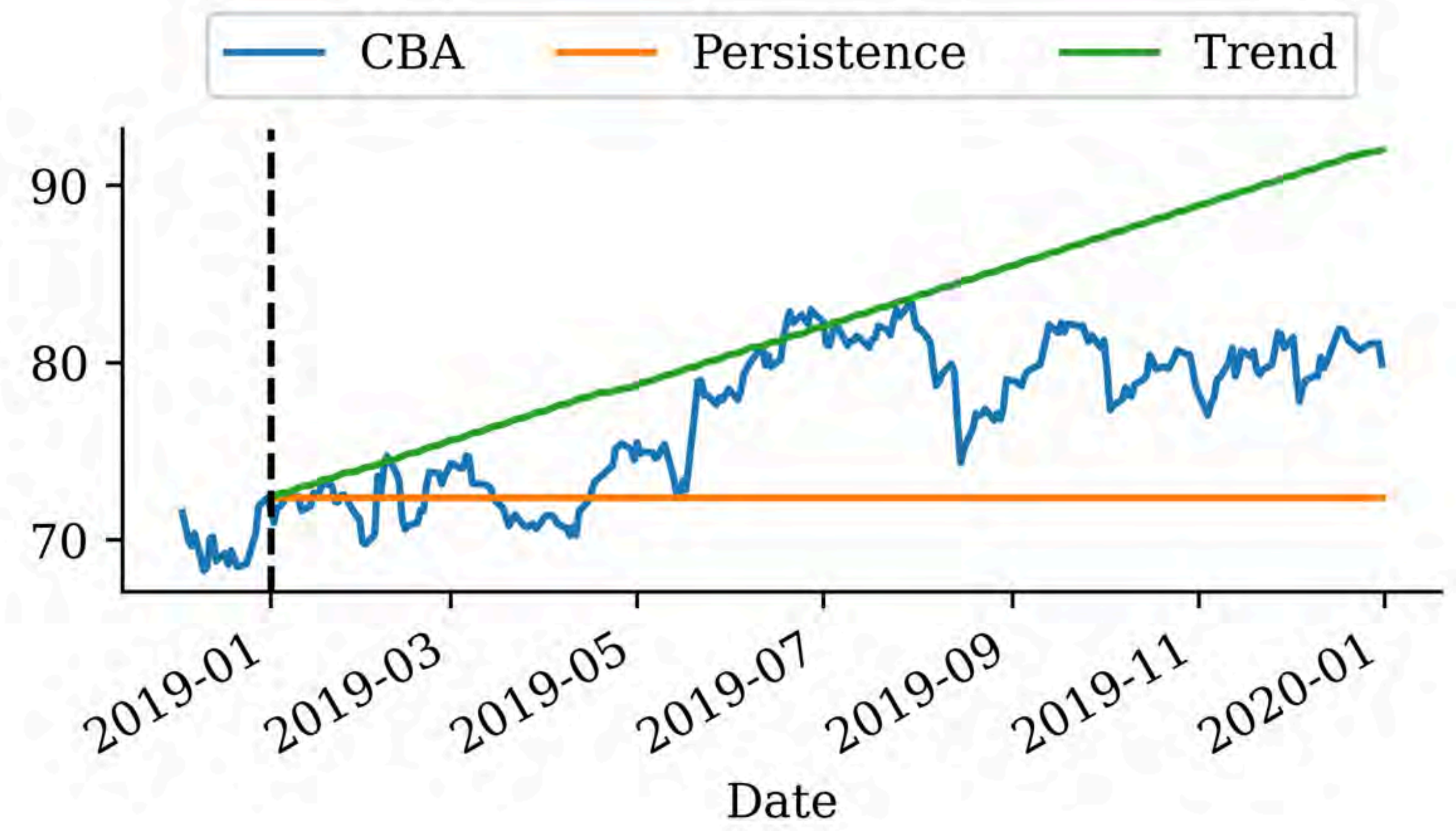
We can extrapolate from recent trend:

```python
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

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

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

```
(39.54629367588932, 37.87104674064297)
```

Use the history

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

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

Lagged time series

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

	T-40	T-39	T-38	T-37	T-36	T-35	T-34	T-
Date								
1991-09-12	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1991-09-13	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1991-09-16	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
…	…	…	…	…	…	…	…	…
2021-	101.37	101.84	102.16	102.14	102.92	100.55	101.09	101.

Split into training and testing

```python
# 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")
```

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

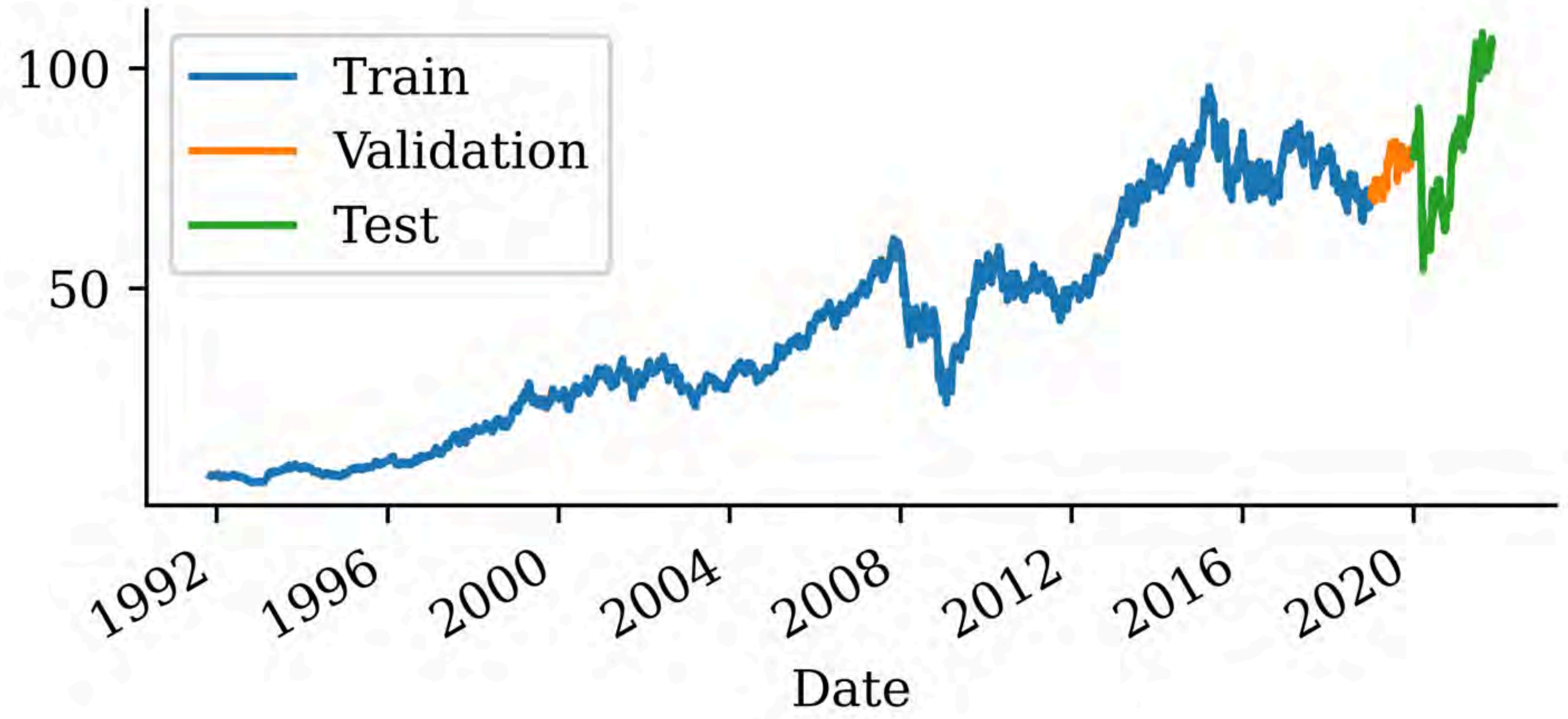
```
((6872, 40), (6872,), (253, 40), (253,), (467, 40), (467,))
```

Inspect the split data

```
X_train
```

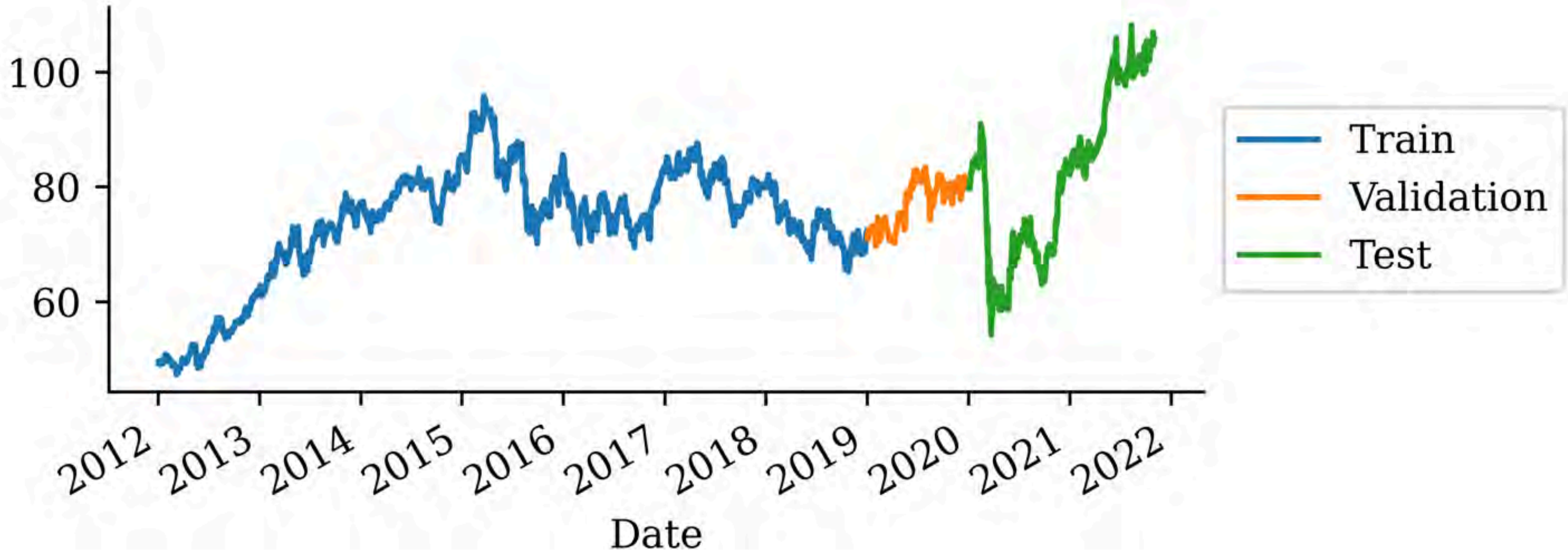
Date	T-40	T-39	T-38	T-37	T-36
1991-11-07	6.425116	6.365440	6.305764	6.285872	6.325656
1991-11-08	6.365440	6.305764	6.285872	6.325656	6.385332
1991-11-11	6.305764	6.285872	6.325656	6.385332	6.445008
…	…	…	…	…	…
2018-12-27	68.160000	69.230000	68.940000	68.350000	67.980000
2018-	69.230000	68.940000	68.350000	67.980000	68.950000

Plot the split



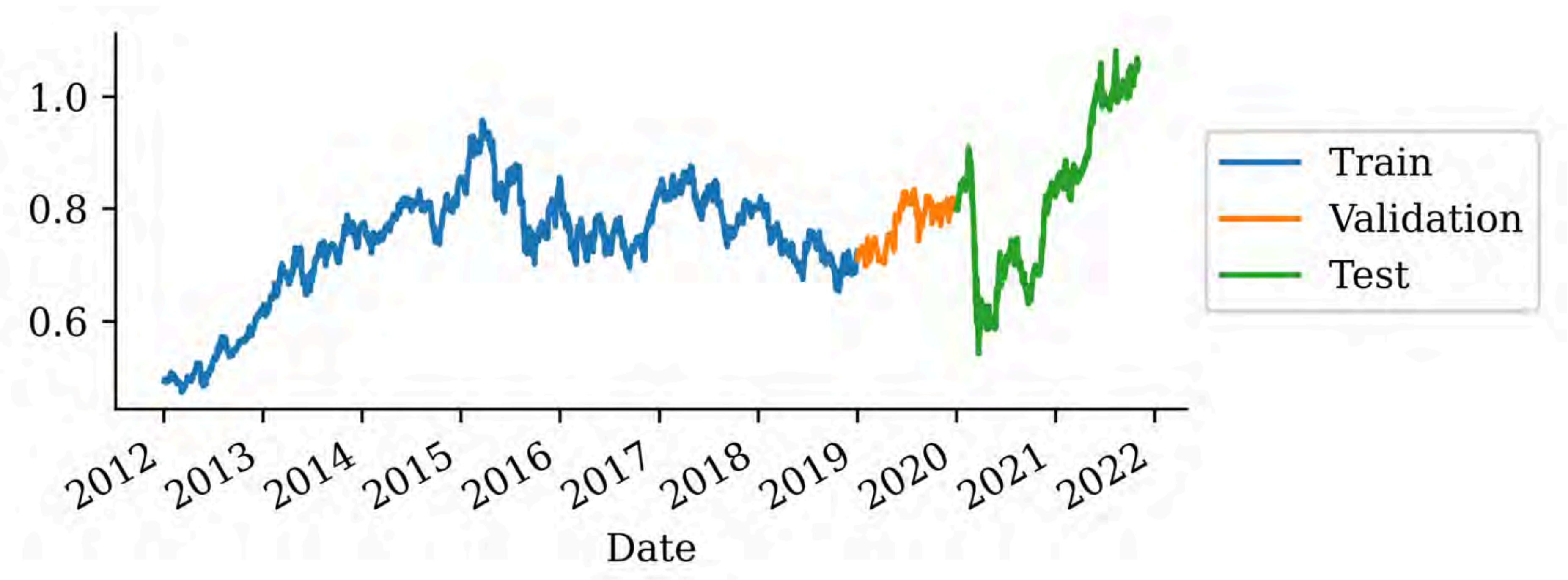
Train on more recent data

```
X_train = X_train.loc["2012":]
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
```

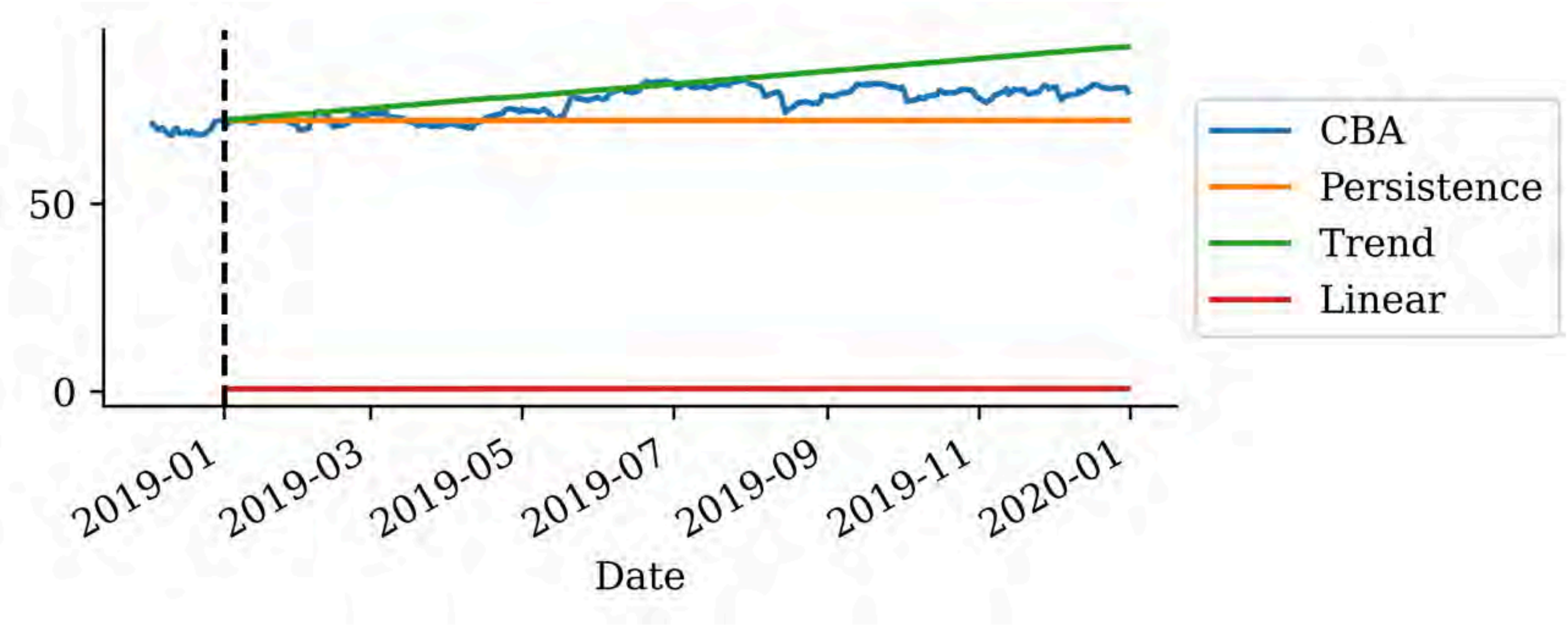


Fit a linear model

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

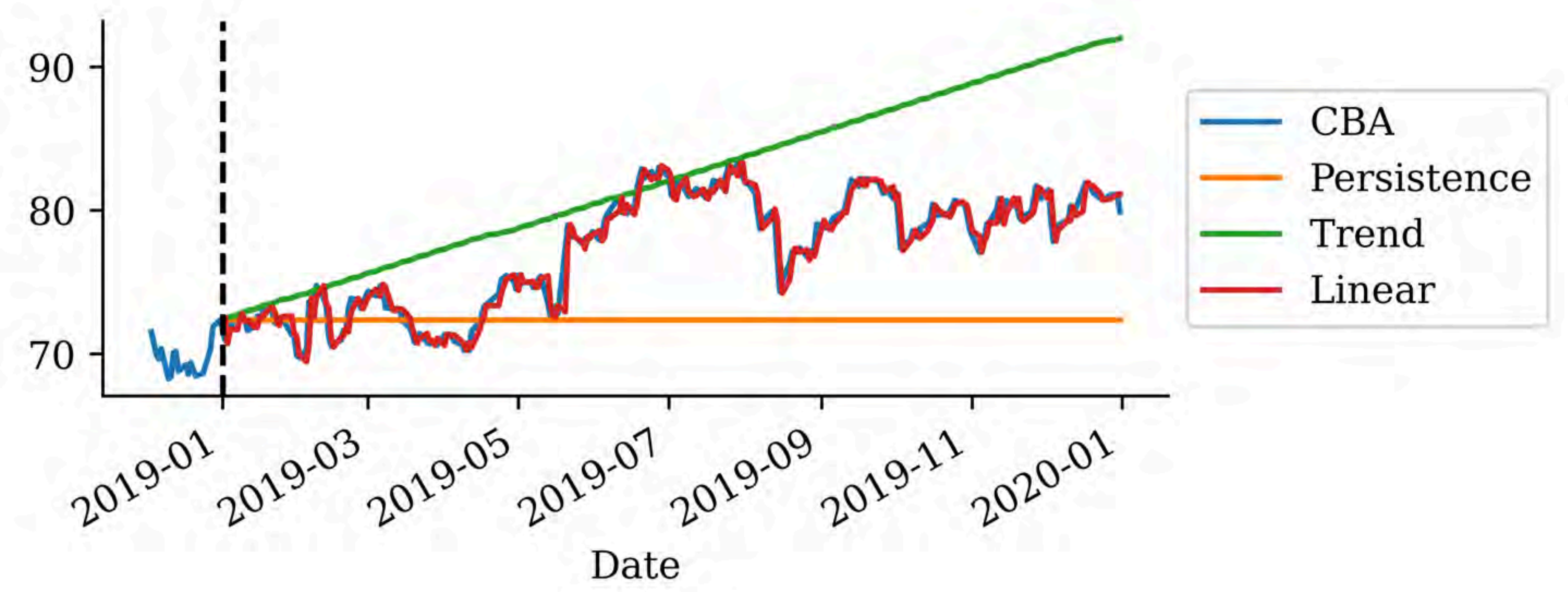
Make a forecast for the validation data:

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



Inverse-transform the forecasts

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



Careful with the metrics

```python
mean_squared_error(y_val, y_pred)
```

```
6.329105517812206e-05
```

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

```
0.6329105517812207
```

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

```
0.6329105517812206
```

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

```
(39.54629367588932, 37.87104674064297, 0.6329105517812206)
```

Lecture Outline

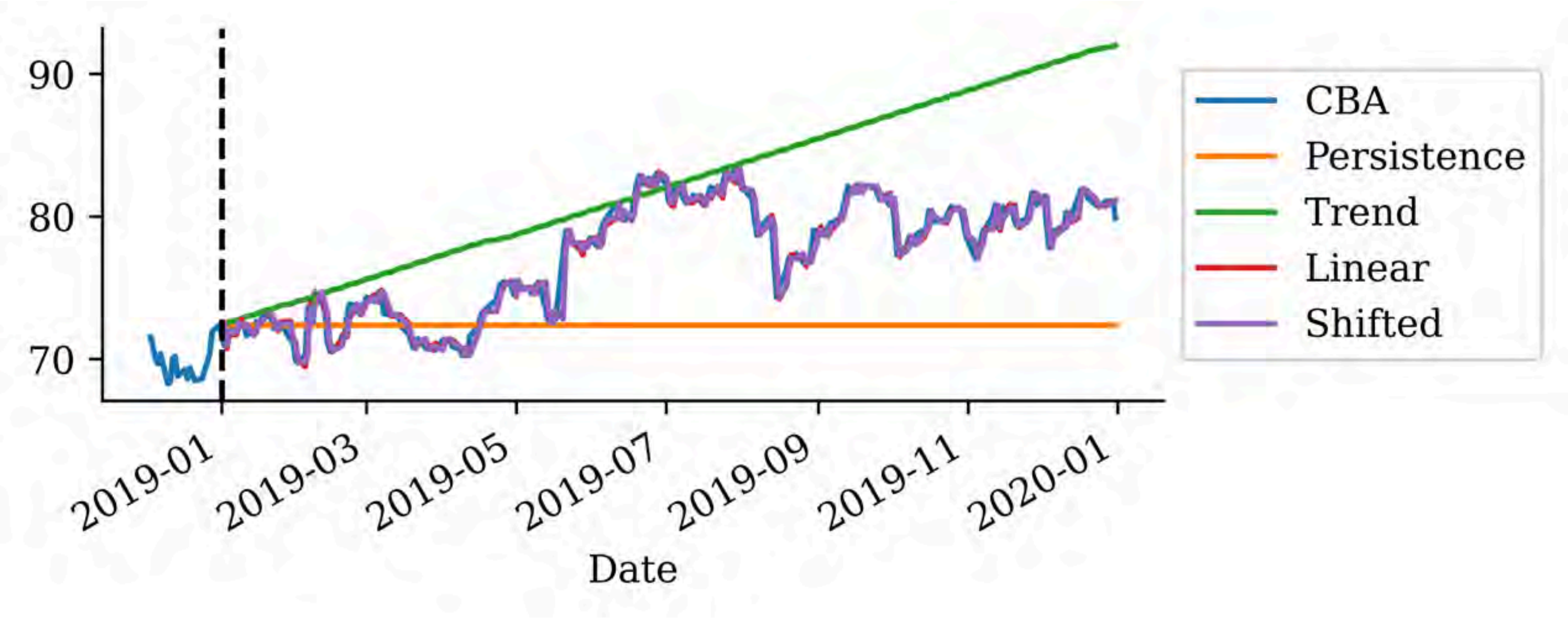
- Time Series
- Baseline forecasts
- **Multi-step forecasts**
- Neural network forecasts
- Recurrent Neural Networks
- RNN Demo
- SimpleRNN maths
- More complex RNNs

Comparing apples to apples

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

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

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




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

```
(39.54629367588932, 37.87104674064297, 0.6329105517812206, 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.

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

$$\vdots$$

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

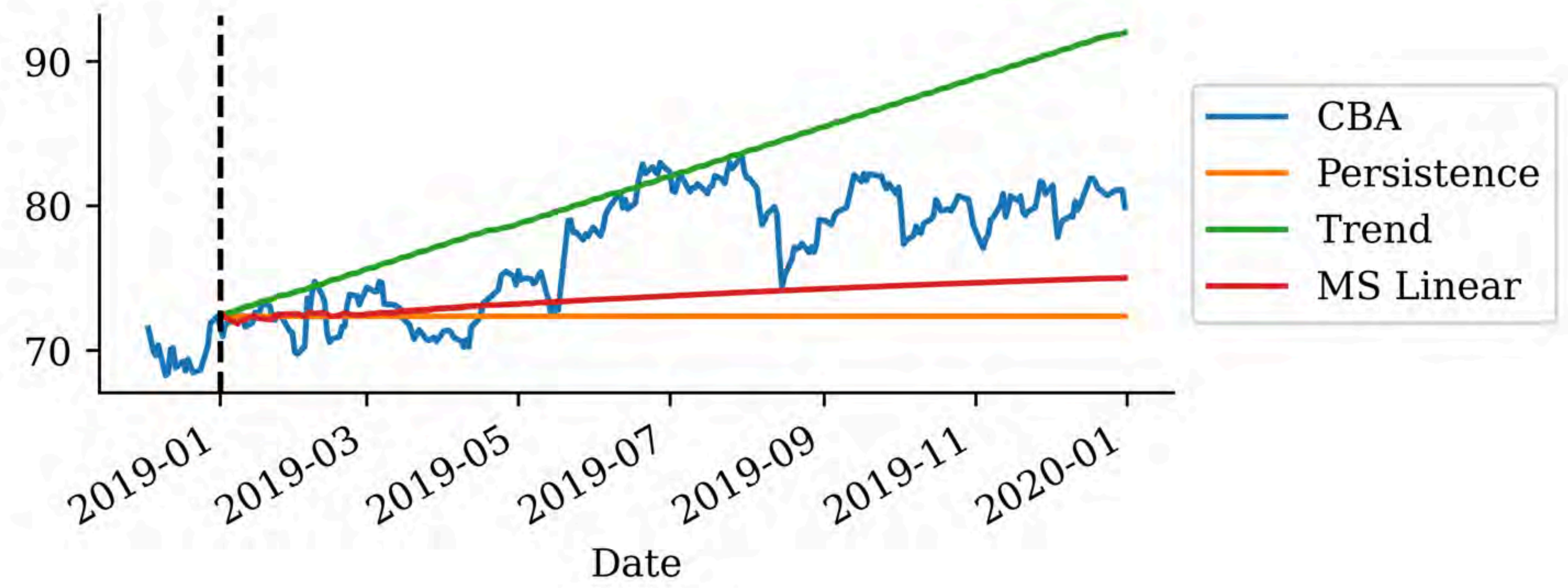
Autoregressive forecasting function

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

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

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

        multi_step.iloc[i] = next_value

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

    return multi_step
```

Look at the autoregressive linear forecasts

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

```python
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:

```
linear_mse, shifted_mse
```

```
(0.6329105517812206, 0.6367221343873524)
```

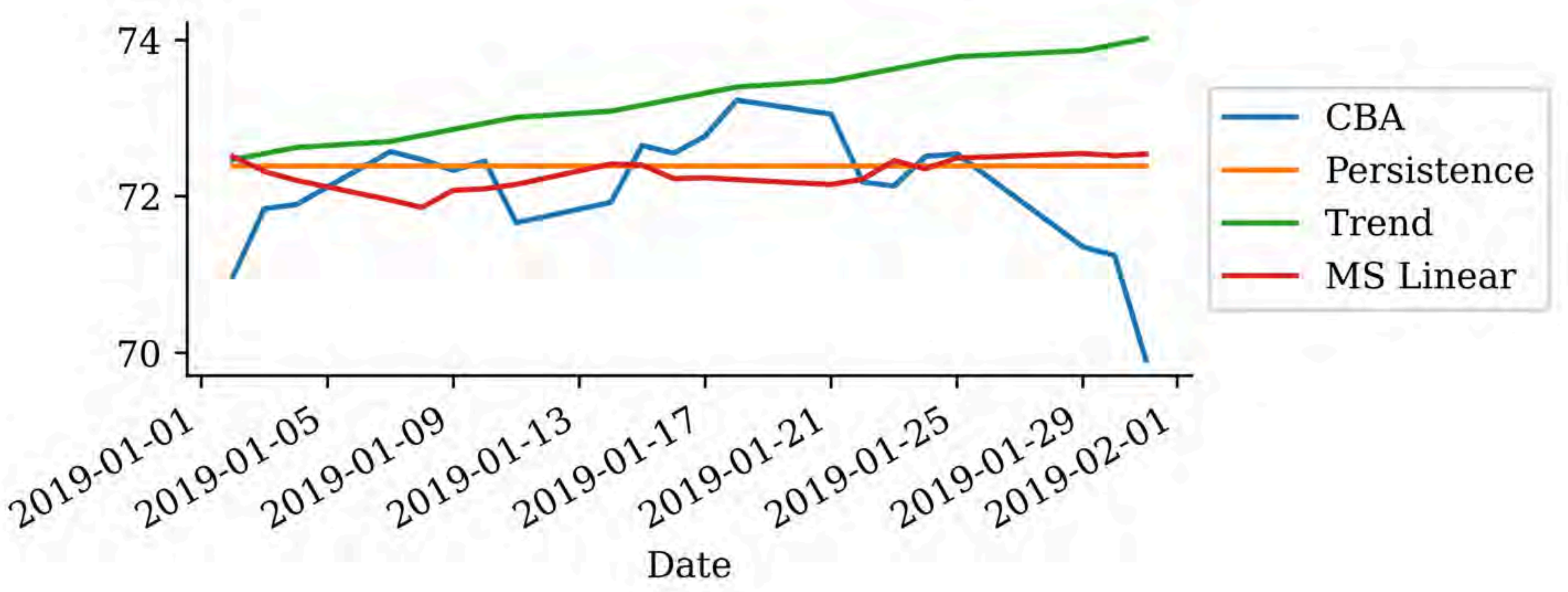
Multi-step-ahead forecasts:

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

```
(39.54629367588932, 37.87104674064297, 23.84700379112976)
```

Prefer only short windows

```
stock.loc["2019":"2019-1"].drop(["Linear", "Shifted"], axis=1).plot();
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
- RNN Demo
- SimpleRNN maths
- More complex RNNs

Simple feedforward neural network

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

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

```python
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"

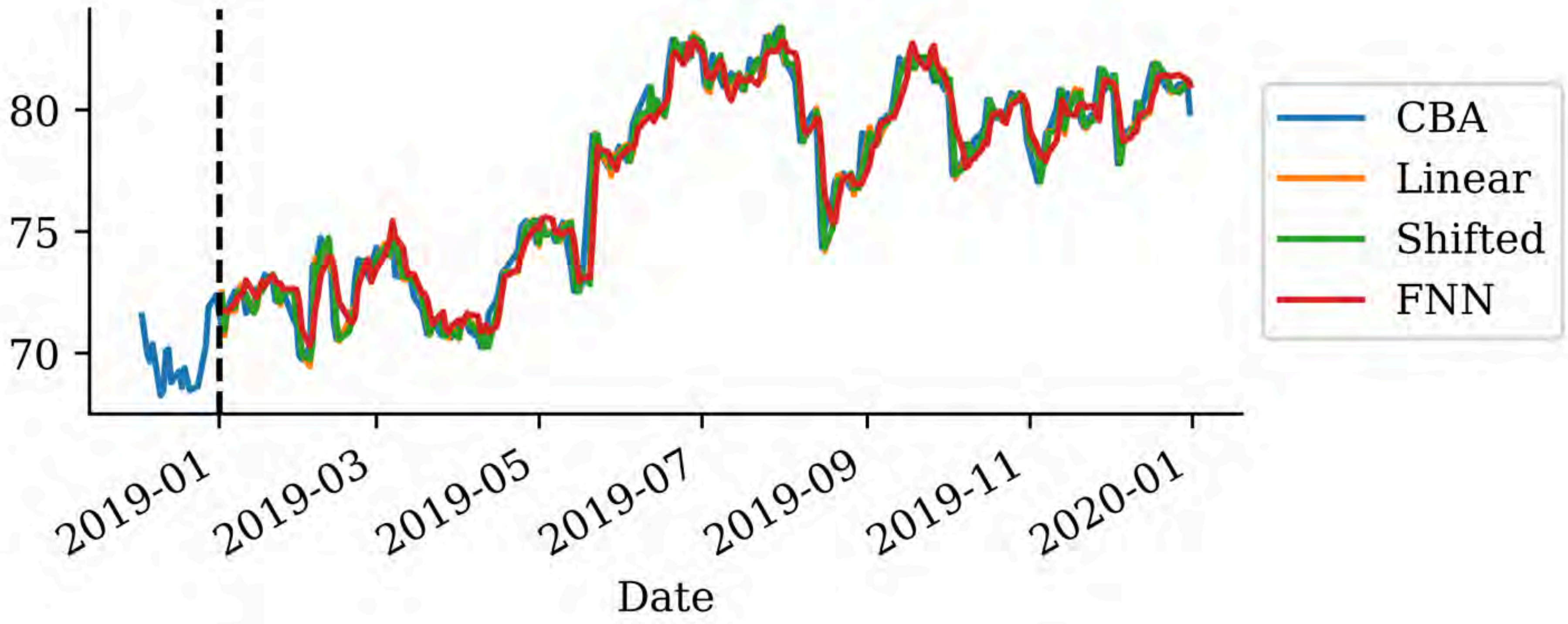
Layer (type)	Output Shape	Param #
dense (Dense)	(32, 64)	2,624
dense_1 (Dense)	(32, 1)	65

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

Forecast and plot

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

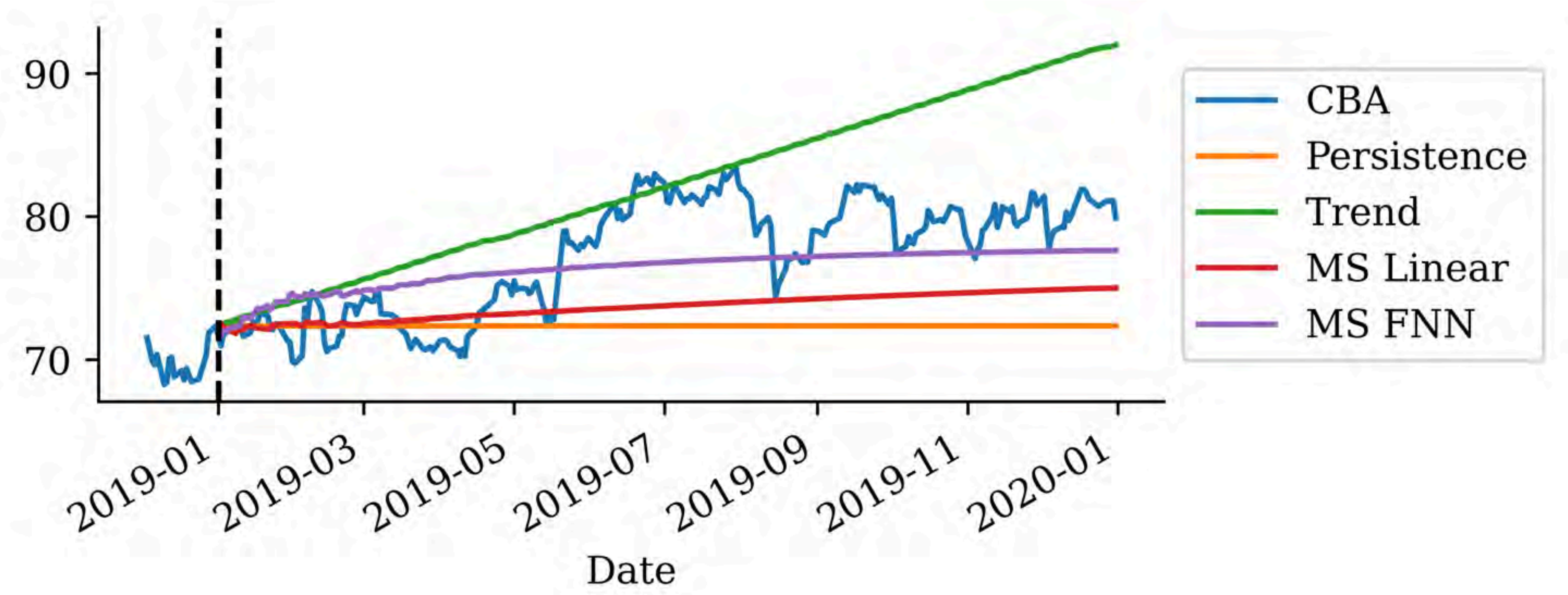
```python
stock.loc["2018-12":"2019"].drop(["Persistence", "Trend", "MS Linear"], axis=1).plot()
plt.axvline("2019", color="black", linestyle="--")
plt.legend(loc="center left", bbox_to_anchor=(1, 0.5));
```



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
```

```
(0.6329105517812206, 0.6367221343873524, 1.0445119512080592)
```

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, 23.84700379112976, 10.15084682208212)
```

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- **Recurrent Neural Networks**
- RNN Demo
- SimpleRNN maths
- More complex RNNs

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

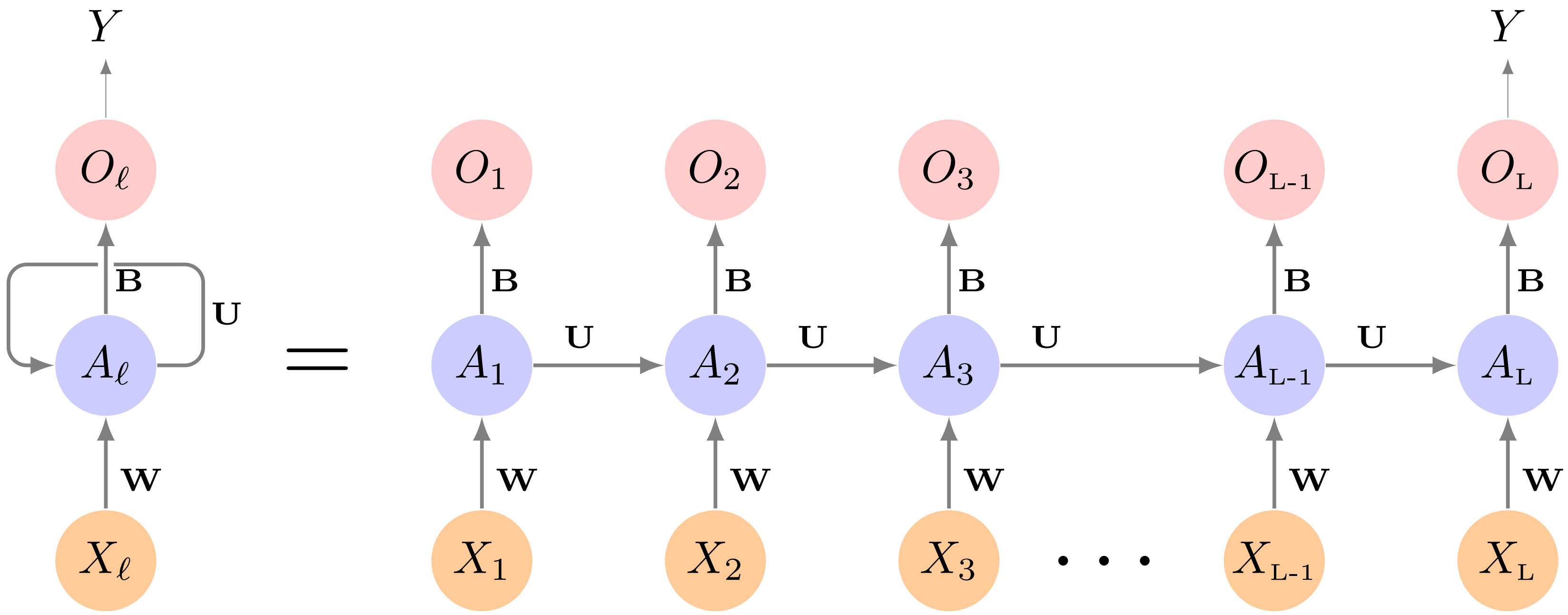
> A recurrence relation is an equation that expresses each element of a sequence as a function of the preceding ones. More precisely, in the case where only the immediately preceding element is involved, a recurrence relation has the form
>
> $$u_n = \psi(n, u_{n-1}) \quad \text{for} \quad n > 0.$$

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

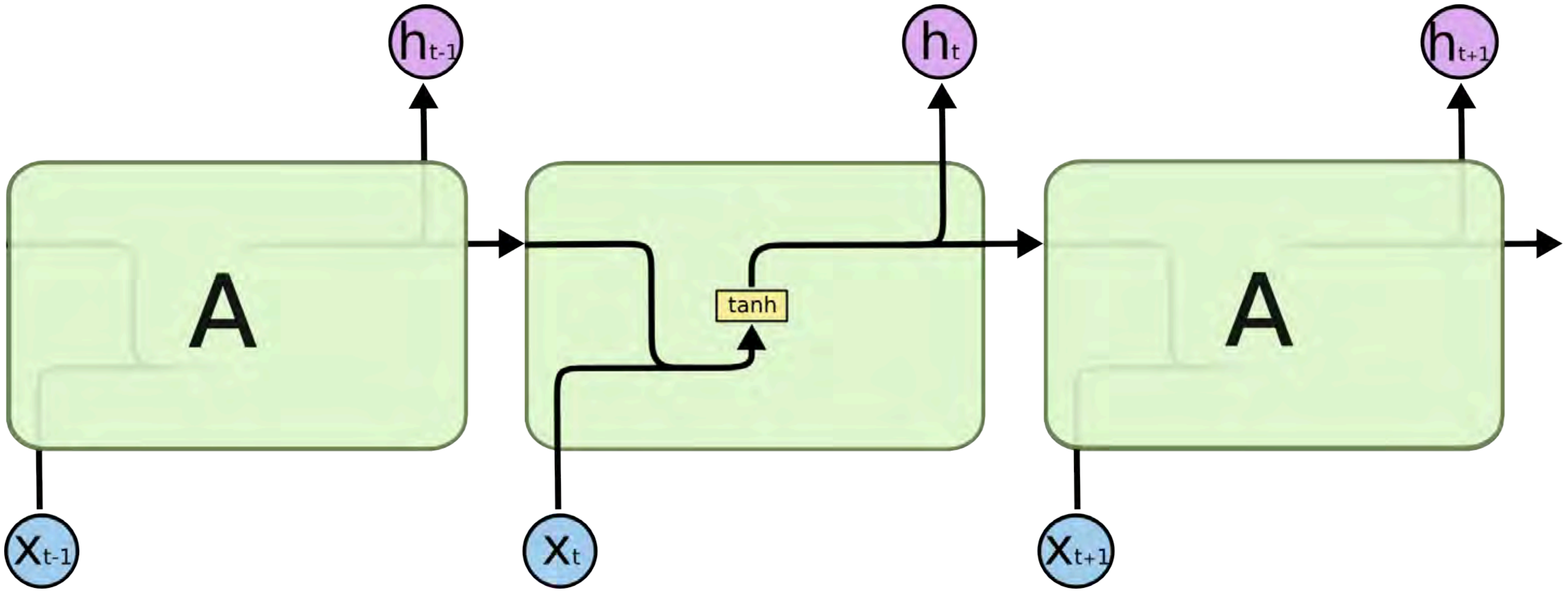
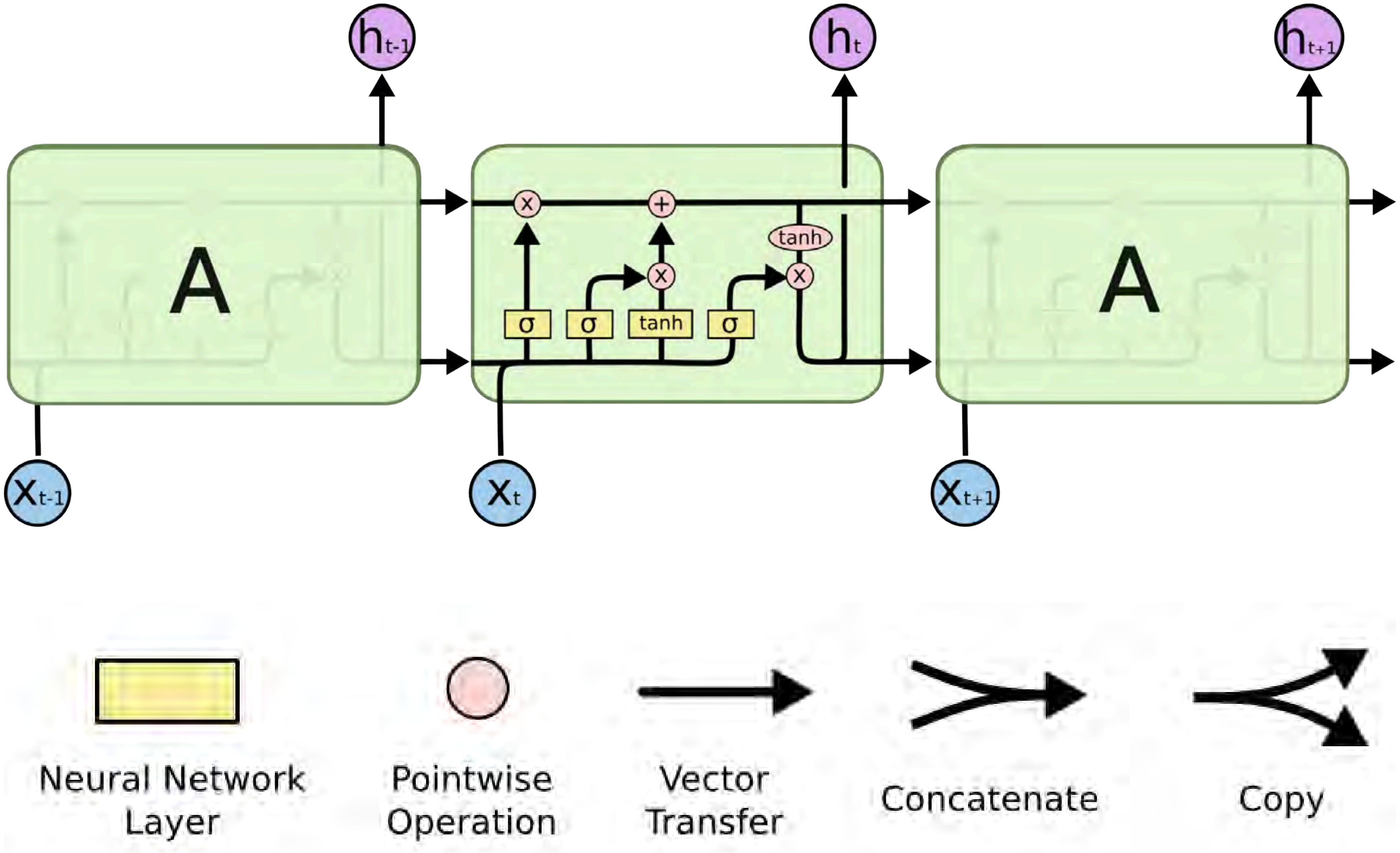


Diagram of a SimpleRNN cell.

All the outputs before the final one are often discarded.

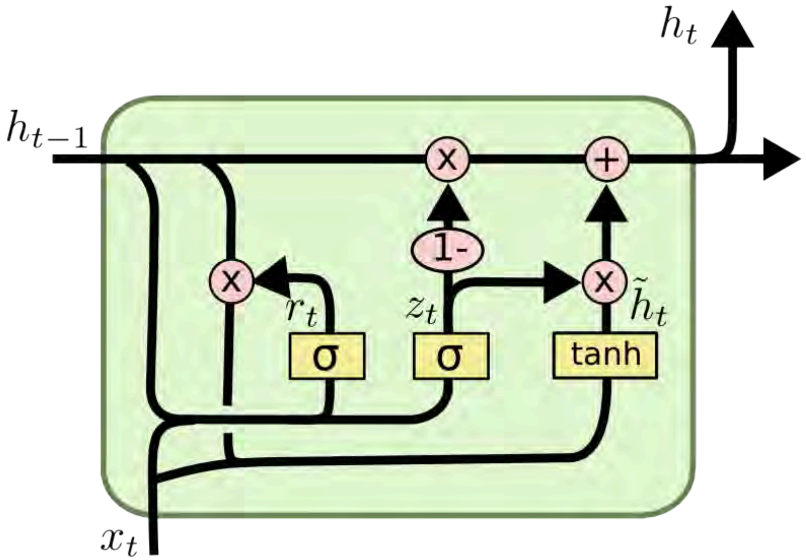
Source: Christopher Olah (2015), Understanding LSTM Networks, Colah's Blog.

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
- **RNN Demo**
- SimpleRNN maths
- More complex RNNs

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()
```

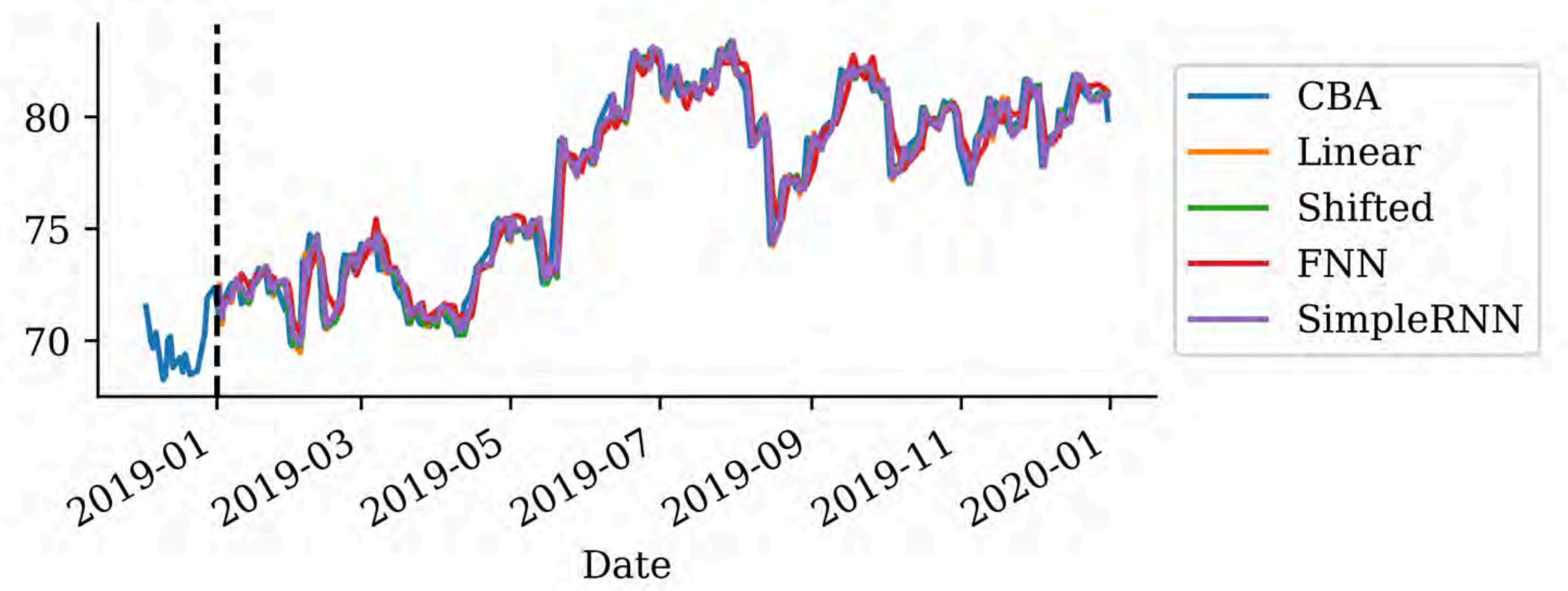
Model: "sequential_1"

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 (16.76 KB)
Trainable params: 4,289 (16.75 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 2 (8.00 B)

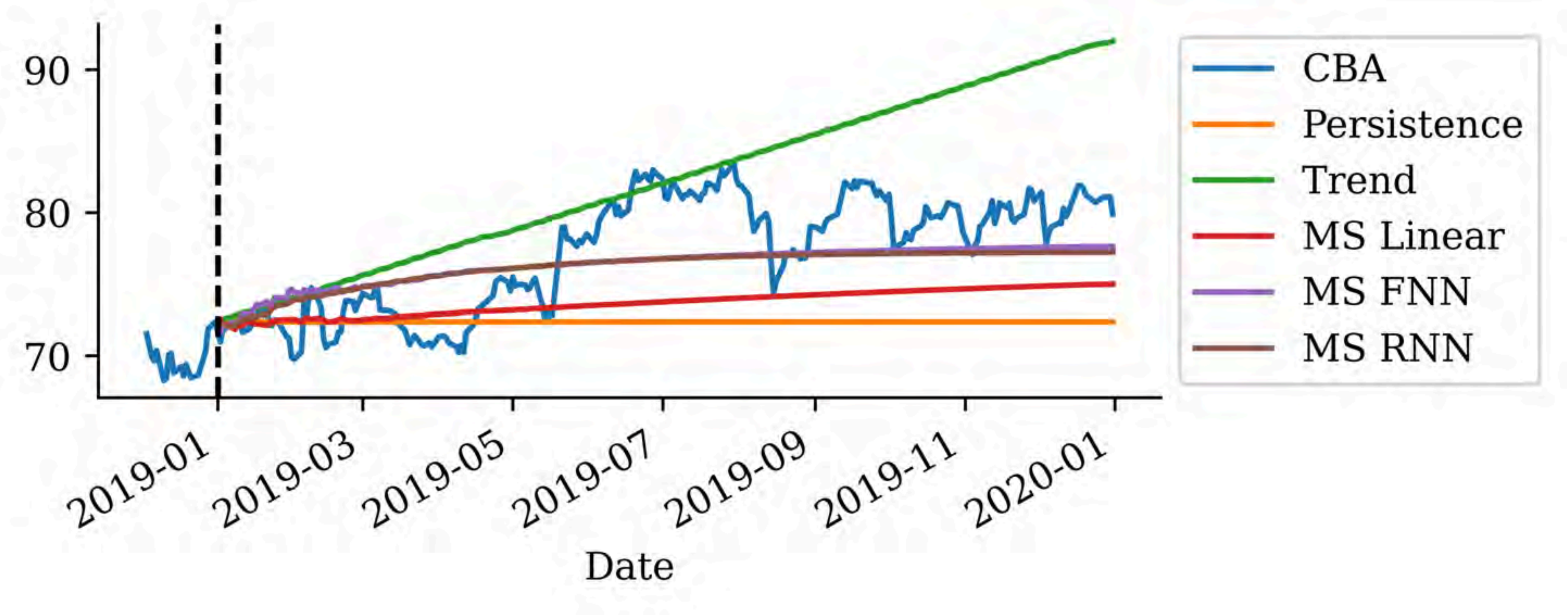
Forecast and plot

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



Multi-step forecasts

```python
rnn_forecast = autoregressive_forecast(model, X_val, True)
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
```

```
(0.6329105517812206, 0.6367221343873524, 1.0445119512080592, 0.644451203191208)
```

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,
 23.84700379112976,
 10.15084682208212,
 10.584407213912131)
```

GRU

```python
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")
```

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

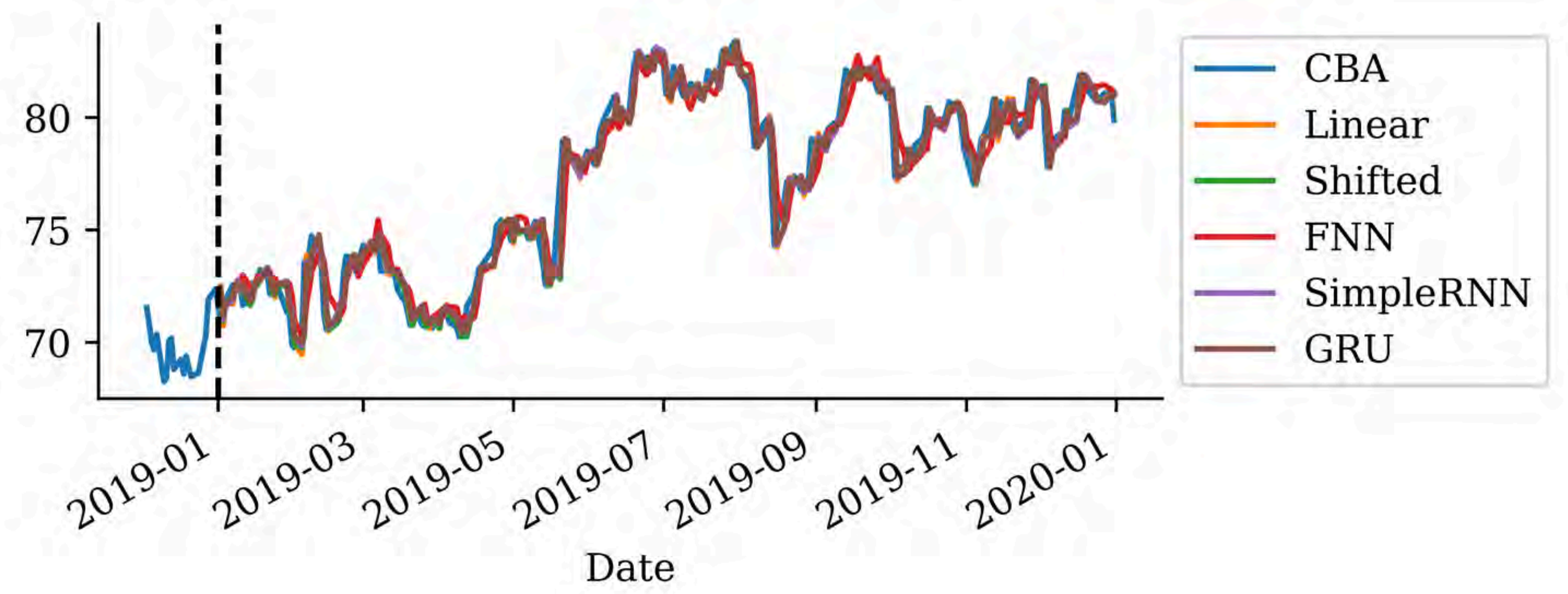
Model: "sequential_2"

Layer (type)	Output Shape	Param #
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)

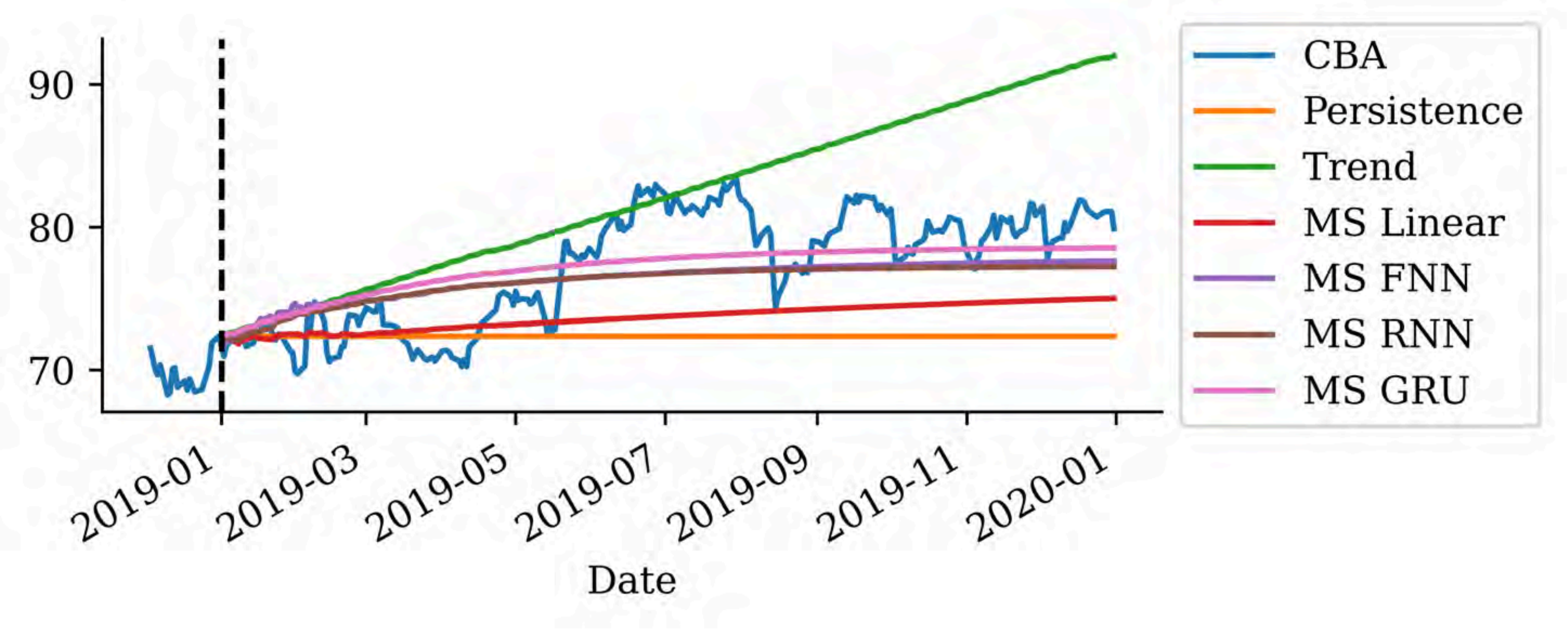
Forecast and plot

```python
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
```

```
(0.6329105517812206,
 0.6367221343873524,
 1.0445119512080592,
 0.644451203191208,
 0.6390273644339947)
```

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,
 23.84700379112976,
 10.15084682208212,
 10.584407213912131,
 8.111645234476077)
```

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- RNN Demo
- **SimpleRNN maths**
- More complex RNNs

The rank of a time series

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

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

If instead we had a batch of b time series'

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

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

Multivariate time series

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

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

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

Note

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

SimpleRNN

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

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

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

Here,

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

SimpleRNN (in batches)

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

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

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

Here,

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

Simple Keras demo

```python
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])
```

```python
X[:2]
```

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

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

```python
X[2:]
```

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

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

Keras' SimpleRNN

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

```
from keras.layers import SimpleRNN

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

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

```
[0.5405111312866211, 0.75]
```

The predicted probabilities on the training set are:

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

```
array([[0.49],
       [0.61],
       [0.72],
       [0.81]], dtype=float32)
```

SimpleRNN weights

```python
model.get_weights()
```

```
[array([[ 0.13],
        [-0.06]], dtype=float32),
 array([[0.51]], dtype=float32),
 array([-0.5], dtype=float32)]
```

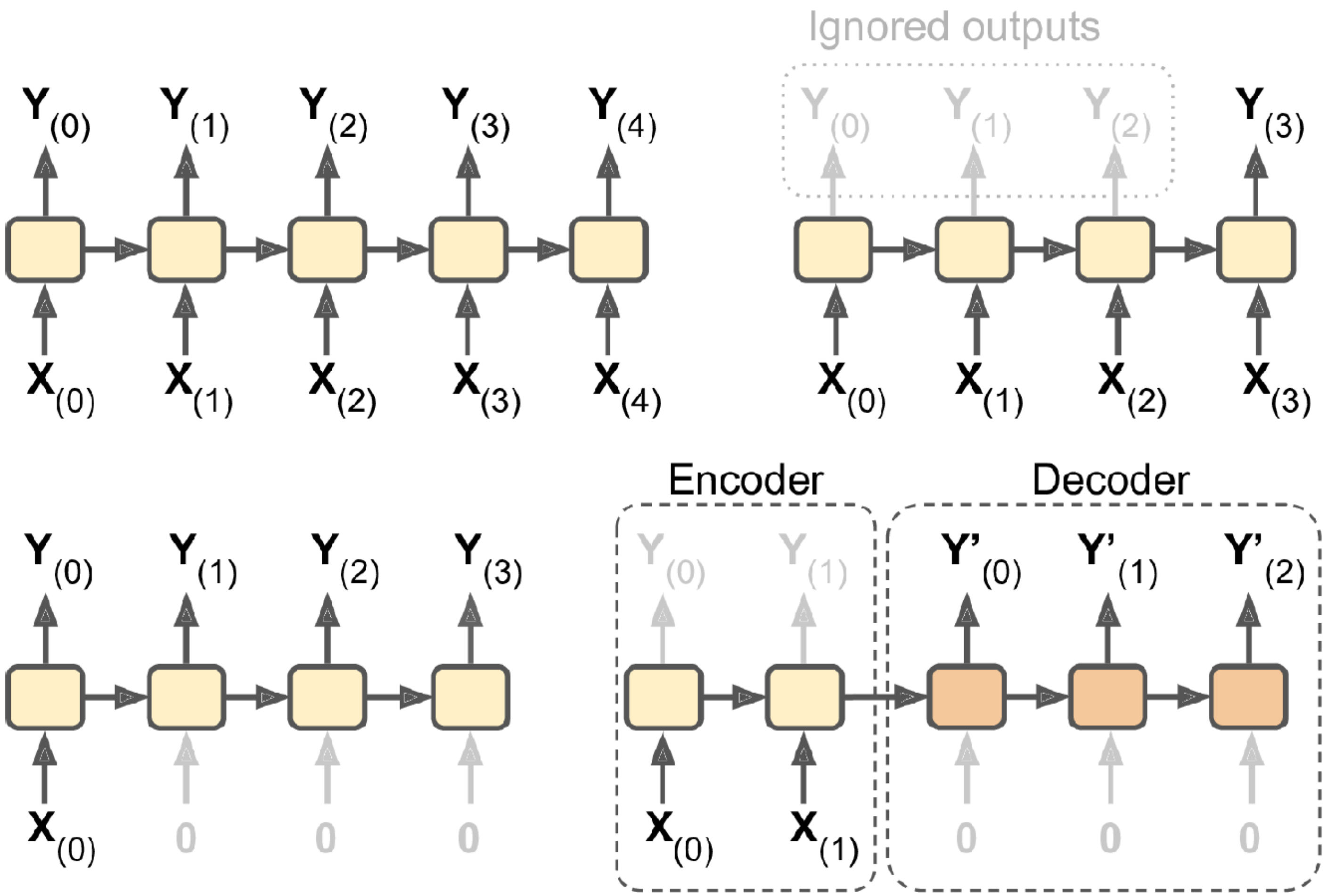
```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


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([[0.49],
       [0.61],
       [0.72],
       [0.81]], dtype=float32)
```

Lecture Outline

- Time Series
- Baseline forecasts
- Multi-step forecasts
- Neural network forecasts
- Recurrent Neural Networks
- RNN Demo
- SimpleRNN maths
- **More complex RNNs**

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.

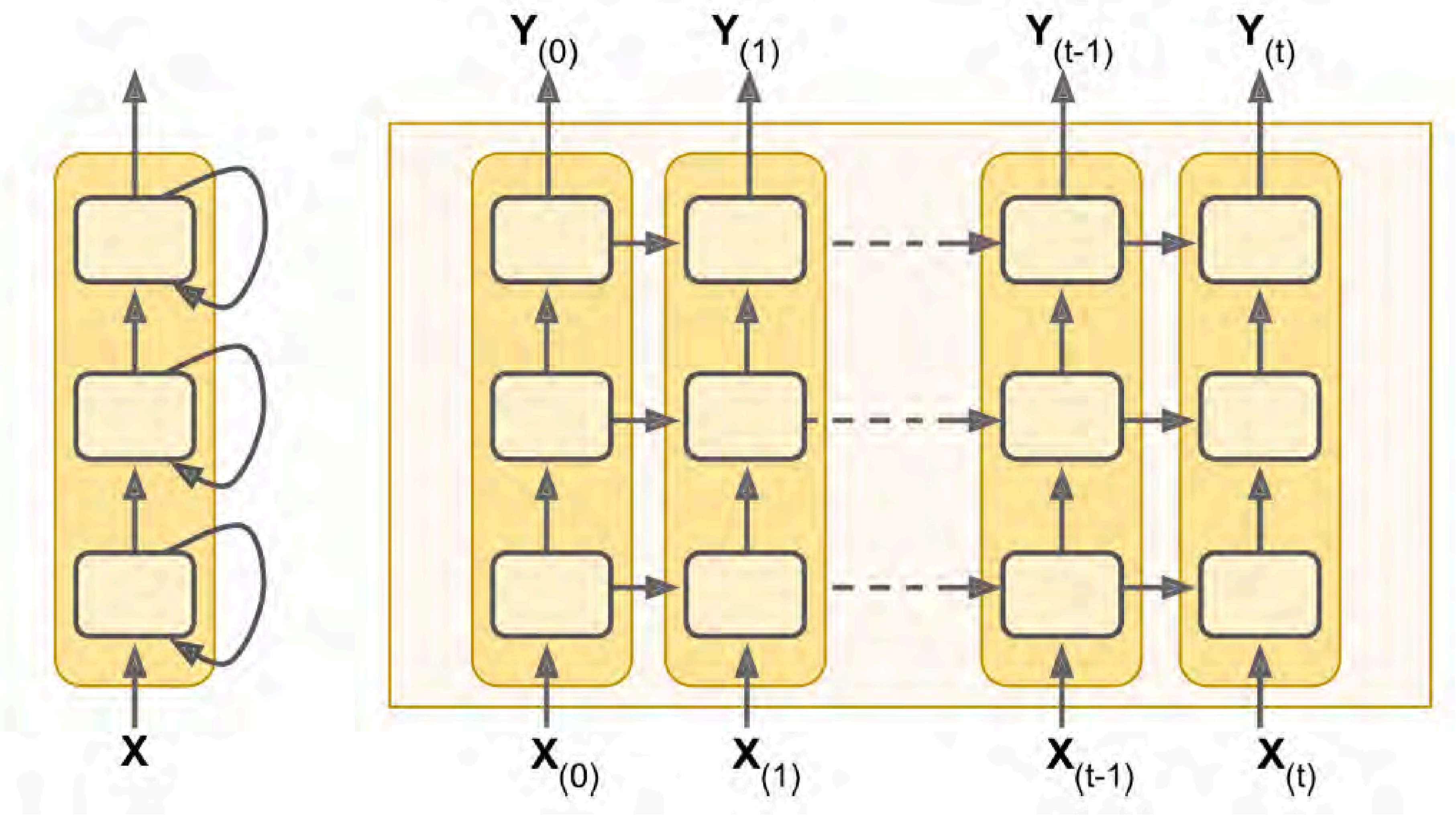
Input and output sequences

- 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.

Package Versions

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

```
Python implementation: CPython
Python version       : 3.11.9
IPython version      : 8.25.0

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

- GRU
- LSTM
- recurrent neural networks
- SimpleRNN