Recurrent Neural Networks

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Patrick Laub
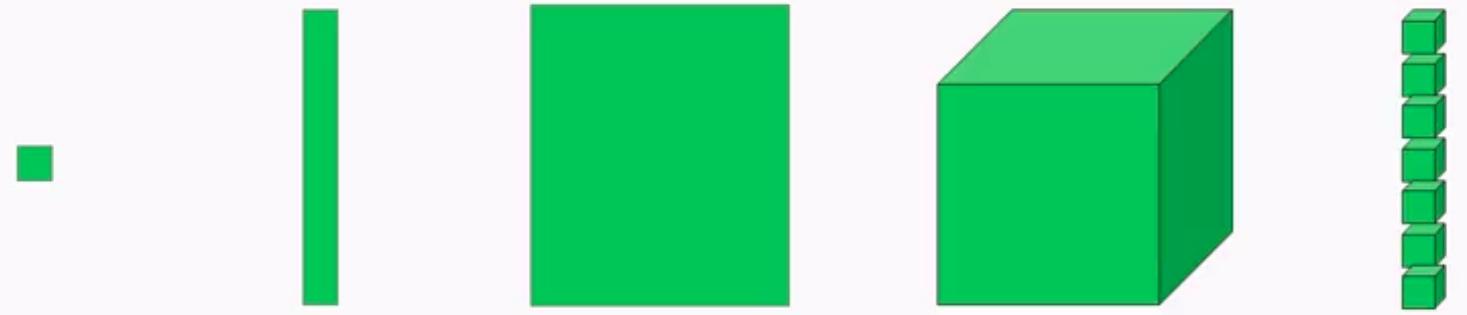
Lecture Outline

- **Tensors & Time Series**

- Some Recurrent Structures

- Recurrent Neural Networks

- CoreLogic Hedonic Home Value Index

- Splitting time series data

- Predicting Sydney House Prices

- Predicting Multiple Time Series

Shapes of data

Illustration of tensors of different rank.

Source: Paras Patidar (2019), Tensors — Representation of Data In Neural Networks, Medium article.

The `axis` argument in numpy

Starting with a $(3, 4)$-shaped matrix:

```
1  X = np.arange(12).reshape(3,4)
2  X
```

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

`axis=0`: $(3, 4) \rightsquigarrow (4,)$.

```
1  X.sum(axis=0)
```

```
array([12, 15, 18, 21])
```

`axis=1`: $(3, 4) \rightsquigarrow (3,)$.

```
1  X.prod(axis=1)
```

```
array([   0,  840, 7920])
```

The return value's rank is one less than the input's rank.

> **⚠ Important**
>
> The `axis` parameter tells us which dimension is removed.

Using `axis` & `keepdims`

With `keepdims=True`, the rank doesn't change.

```
1  X = np.arange(12).reshape(3,4)
2  X
```

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

`axis=0`: $(3, 4) \rightsquigarrow (1, 4)$.

```
1  X.sum(axis=0, keepdims=True)
```

```
array([[12, 15, 18, 21]])
```

```
1  X / X.sum(axis=1)
```

```
ValueError: operands could not be broadcast
together with shapes (3,4) (3,)
```

`axis=1`: $(3, 4) \rightsquigarrow (3, 1)$.

```
1  X.prod(axis=1, keepdims=True)
```

```
array([[   0],
       [ 840],
       [7920]])
```

```
1  X / X.sum(axis=1, keepdims=True)
```

```
array([[0.  , 0.17, 0.33, 0.5 ],
       [0.18, 0.23, 0.27, 0.32],
       [0.21, 0.24, 0.26, 0.29]])
```

The rank of a time series

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

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

If instead we had a batch of b time series'

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

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, $\mathbf{x}_t = (x_t, y_t)$.

Lecture Outline

- Tensors & Time Series
- **Some Recurrent Structures**
- Recurrent Neural Networks
- CoreLogic Hedonic Home Value Index
- Splitting time series data
- Predicting Sydney House Prices
- Predicting Multiple Time Series

Recurrence relation

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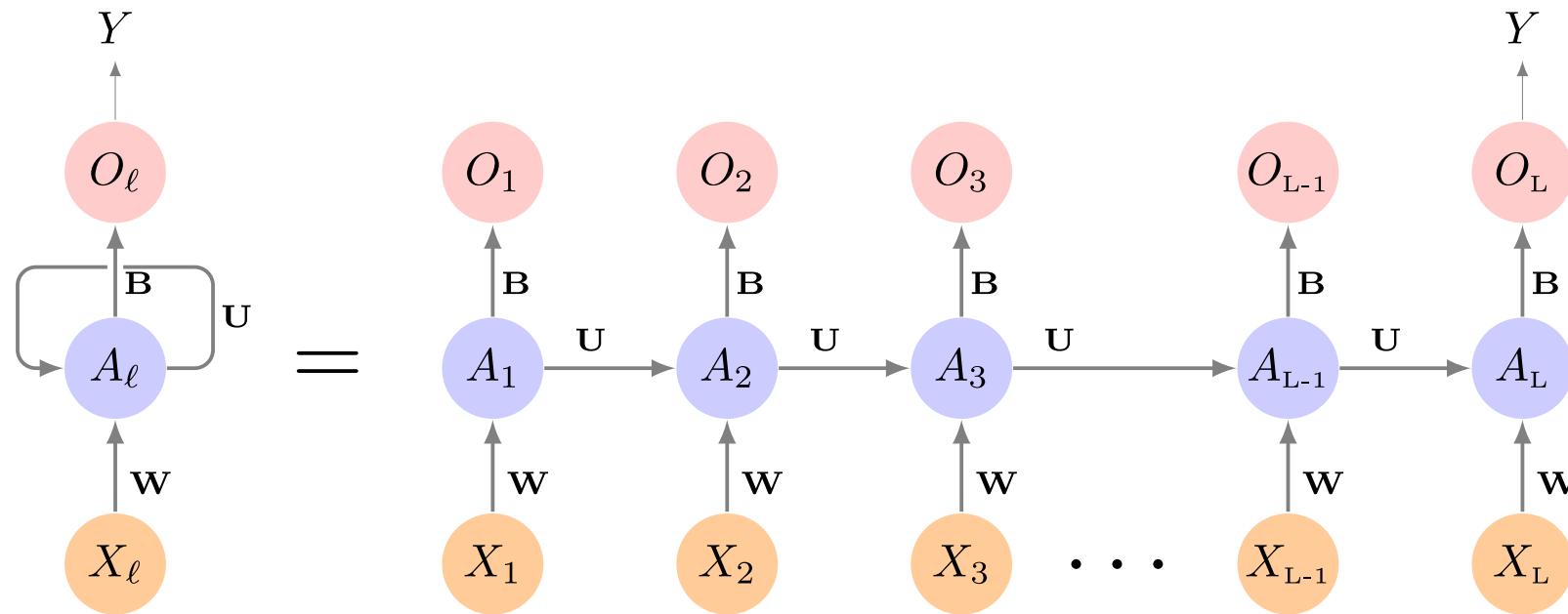
$$u_n = \psi(n, u_{n-1}) \quad \text{for} \quad n > 0.$$

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

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 simple recurrent neural network. E.g. SimpleRNN, LSTM, or GRU.
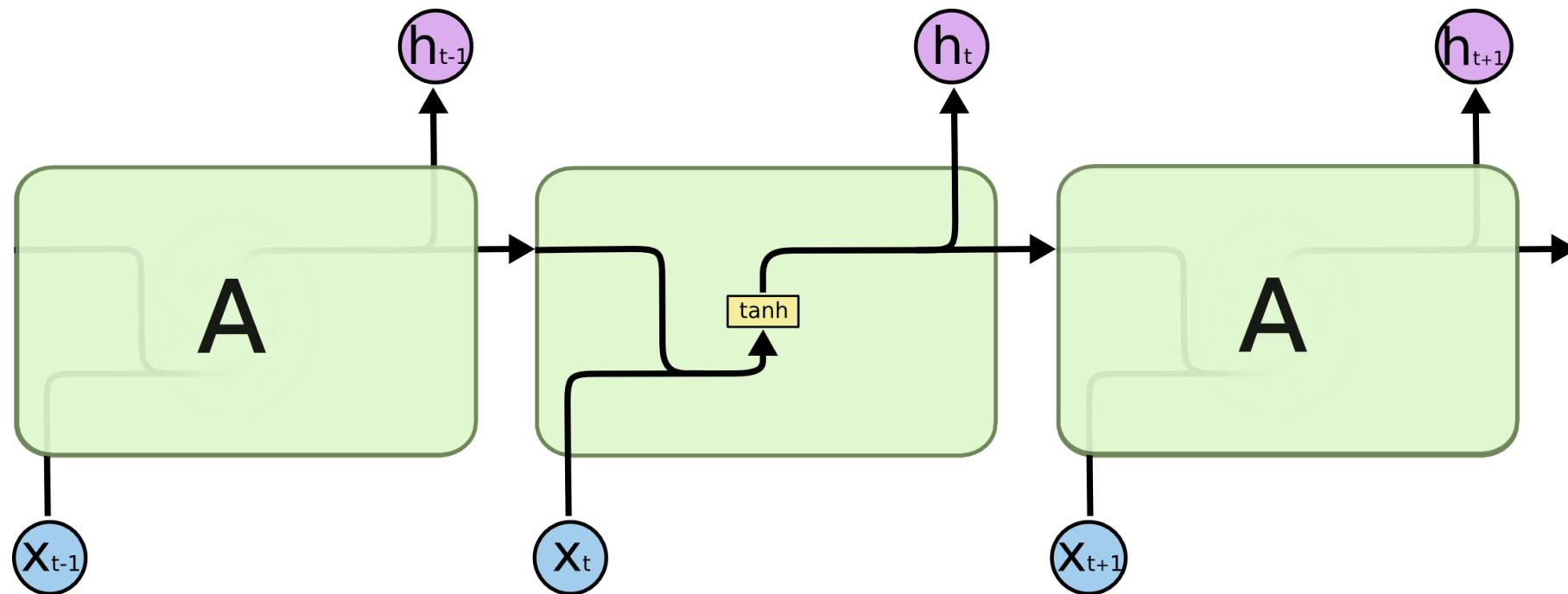
A SimpleRNN cell.

Diagram of a SimpleRNN cell.

All the outputs before the final one are often discarded.

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

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

Here,

$$\mathbf{x}_t \in \mathbb{R}^{1 \times m}, \mathbf{W}_x \in \mathbb{R}^{m \times d},$$

$$\mathbf{y}_{t-1} \in \mathbb{R}^{1 \times d}, \mathbf{W}_y \in \mathbb{R}^{d \times d}, \text{ and } \mathbf{b} \in \mathbb{R}^d.$$

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

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

Here,

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

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 `X[:, t, :]`.

Simple Keras demo

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

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

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

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

```python
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.

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

```
[8.05906867980957, 0.5]
```

The predicted probabilities on the training set are:

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

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

SimpleRNN weights
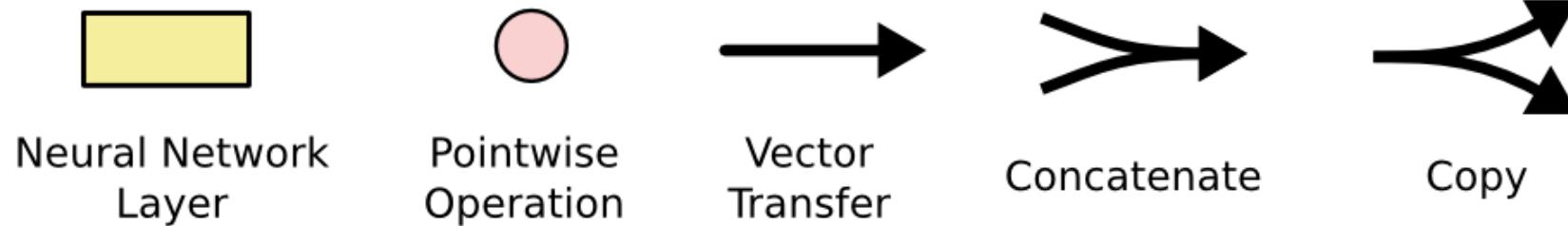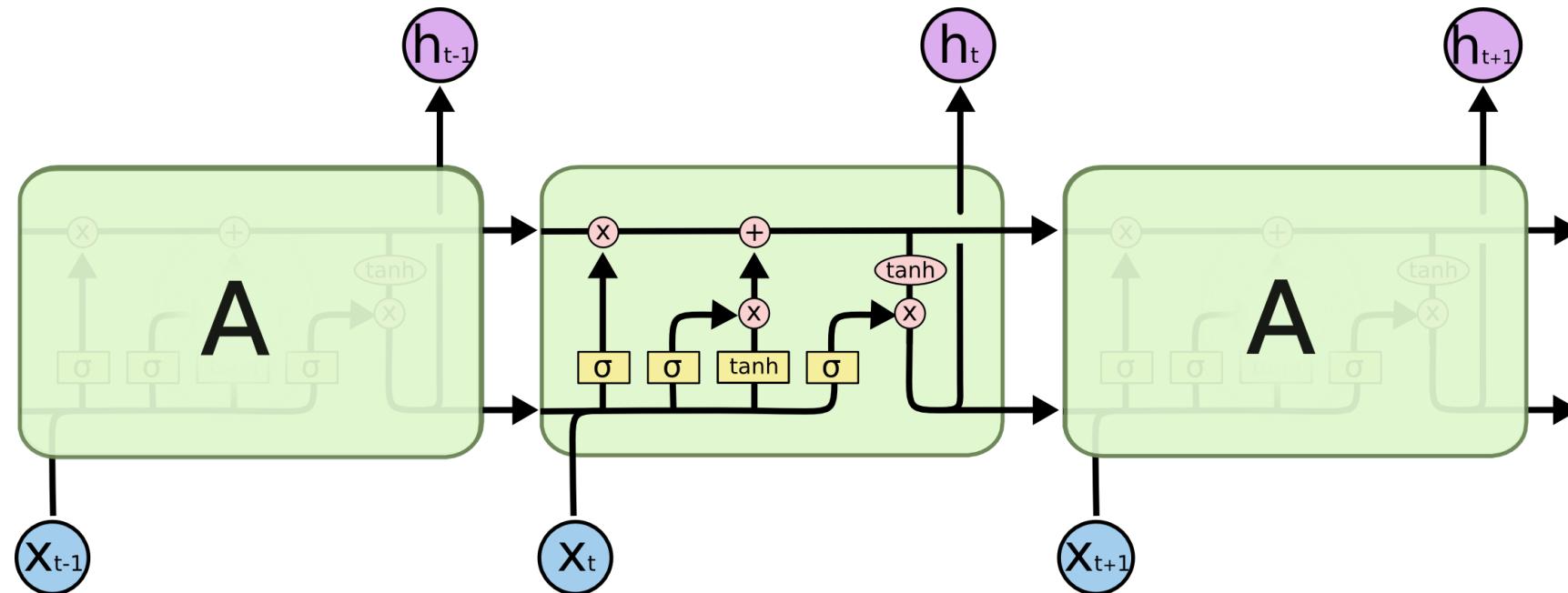
```
1  model.get_weights()
```

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

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

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

LSTM internals

GRU internals
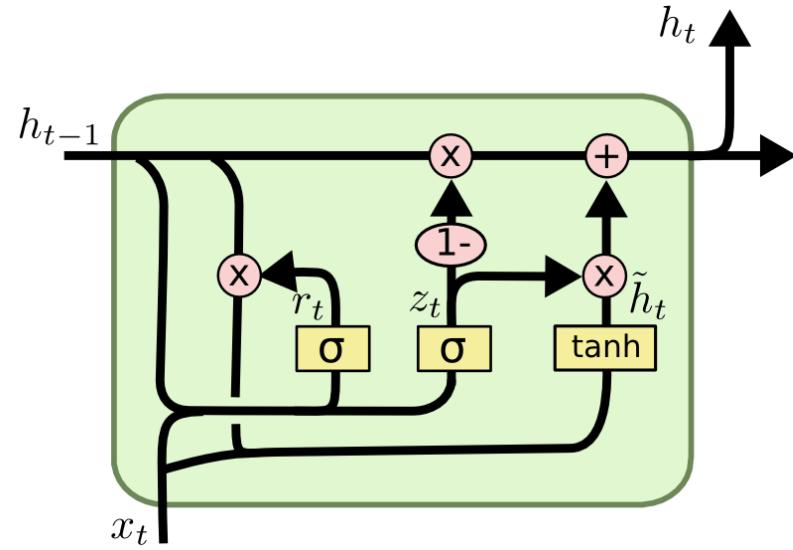
Diagram of a GRU cell.

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

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

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

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

Lecture Outline

- Tensors & Time Series

- Some Recurrent Structures

- **Recurrent Neural Networks**

- CoreLogic Hedonic Home Value Index

- Splitting time series data

- 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.
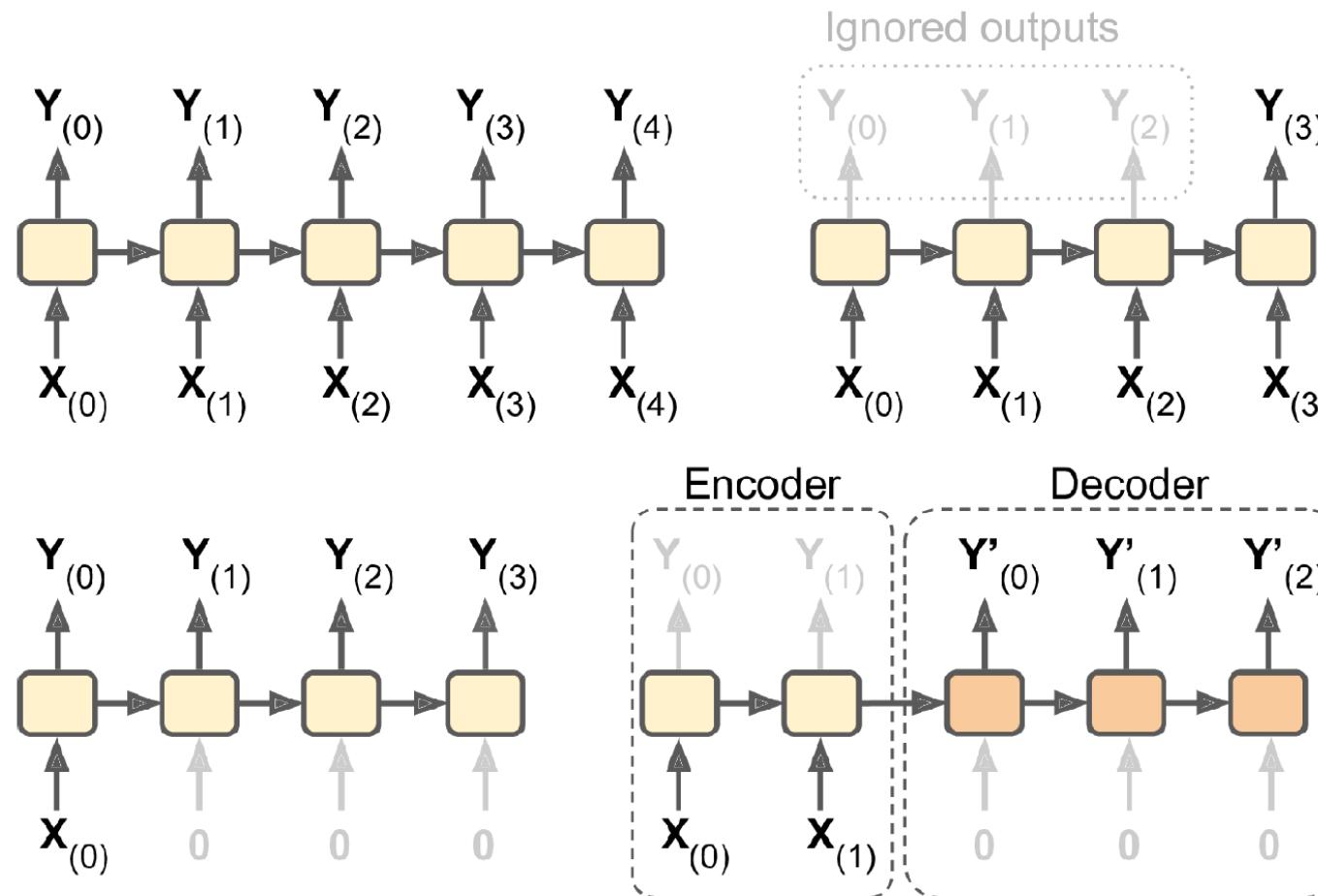
Applications

- 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.

Input and output sequences

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

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.
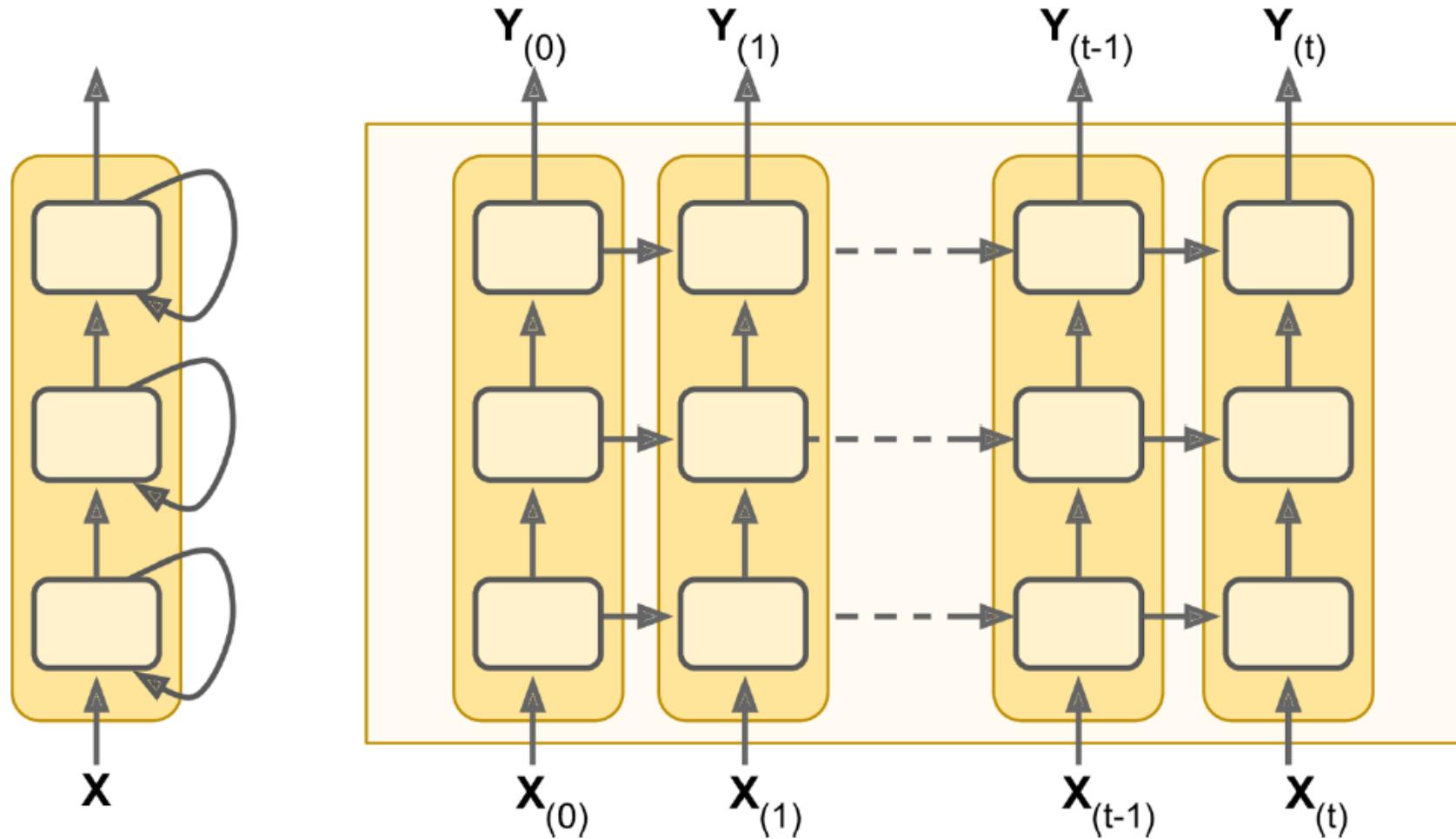
Input and output sequences

- 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.
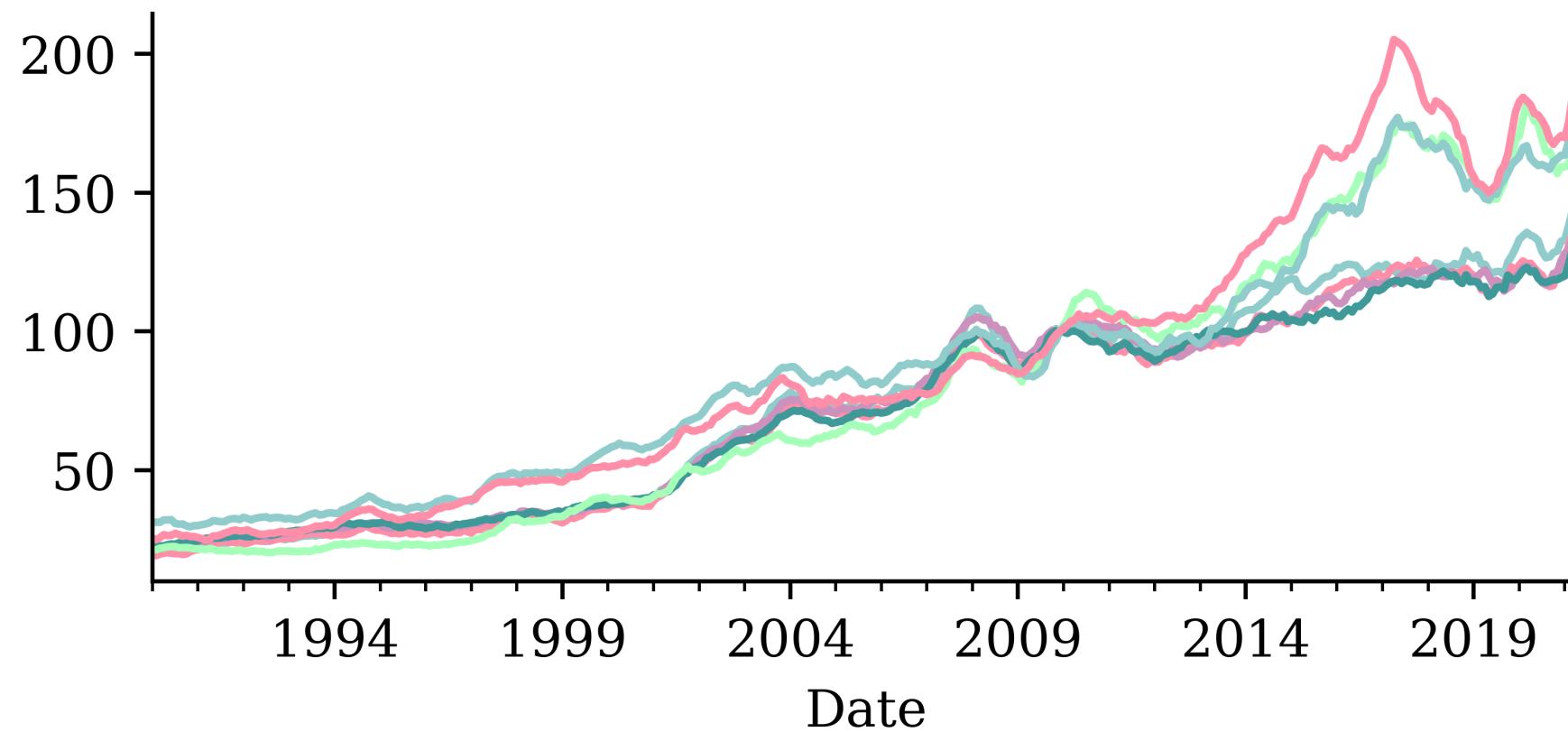
Lecture Outline

- Tensors & Time Series
- Some Recurrent Structures
- Recurrent Neural Networks
- **CoreLogic Hedonic Home Value Index**
- Splitting time series data
- Predicting Sydney House Prices
- Predicting Multiple Time Series

Australian House Price Indices

Percentage changes

```
1  changes = house_prices.pct_change().dropna()
2  changes.round(2)
```
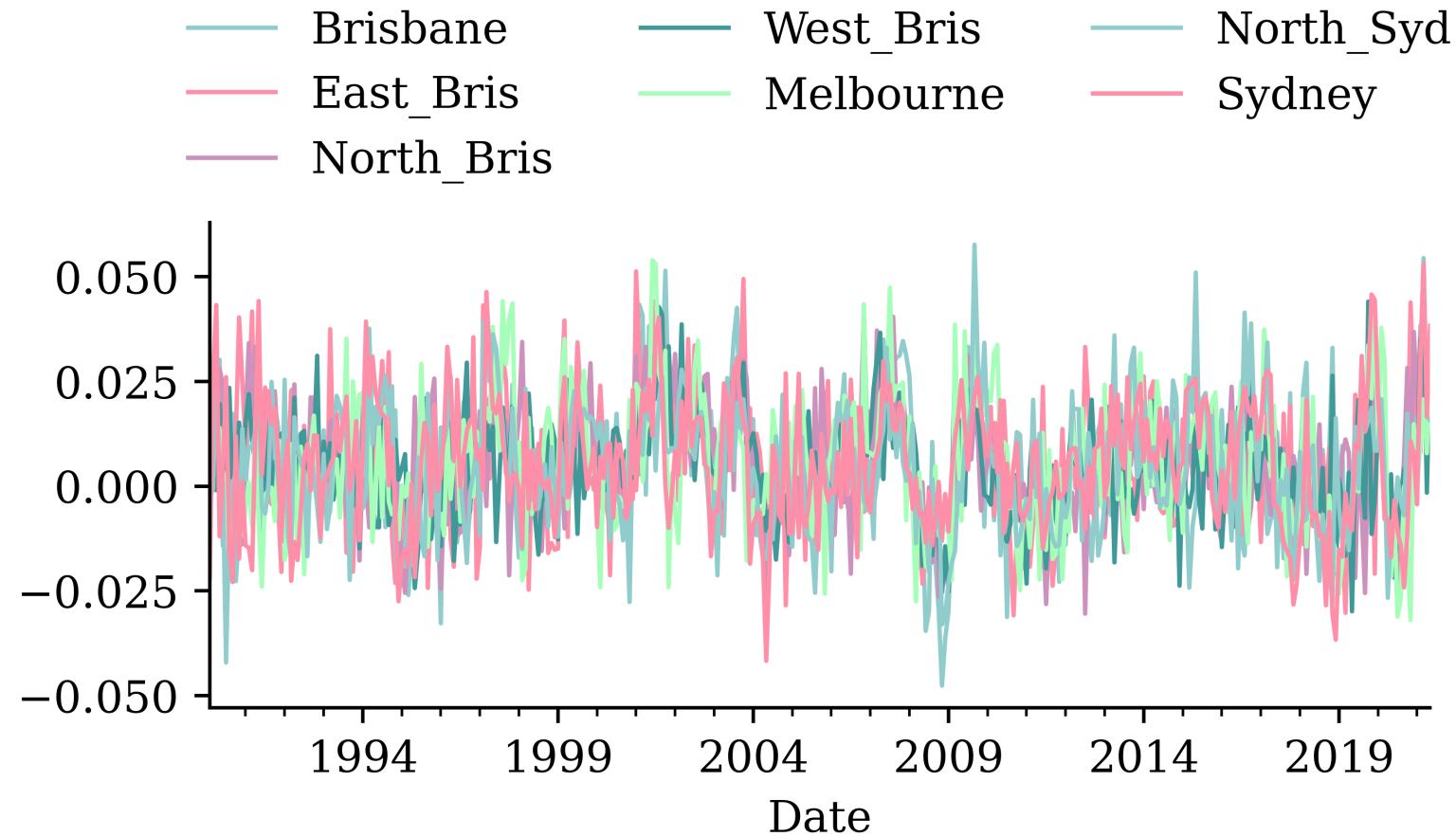
	Brisbane	East_Bris	North_Bris	West_Bris	Melbourne	North_Syd	Sydney
Date							
1990-02-28	0.03	−0.01	0.01	0.01	0.00	−0.00	−0.02
1990-03-31	0.01	0.03	0.01	0.01	0.02	−0.00	0.03
...	...	...	...	...	...	...	...
2021-04-30	0.03	0.01	0.01	−0.00	0.01	0.02	0.02
2021-05-31	0.03	0.03	0.03	0.03	0.03	0.02	0.04

376 rows × 7 columns

Percentage changes

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

The size of the changes

```
1  changes.mean()
```
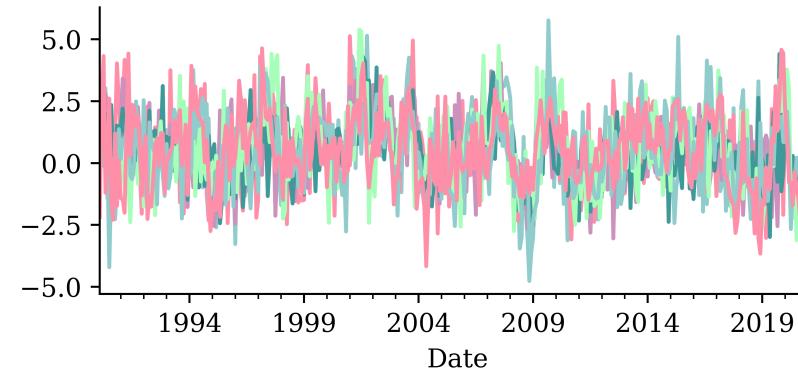
```
Brisbane      0.005496
East_Bris     0.005416
North_Bris    0.005024
West_Bris     0.004842
Melbourne     0.005677
North_Syd     0.004819
Sydney        0.005526
dtype: float64
```

```
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);
```

Lecture Outline
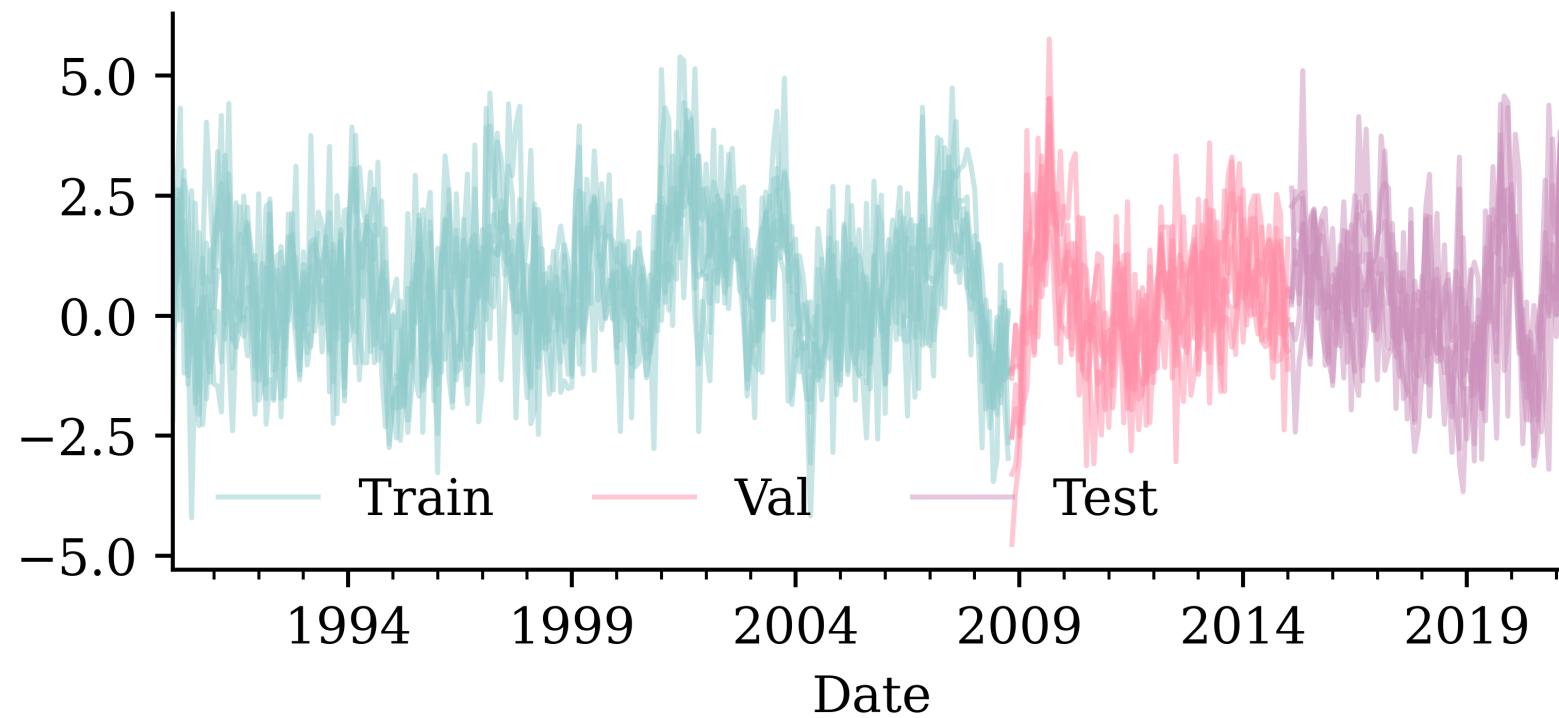
- Tensors & Time Series
- Some Recurrent Structures
- Recurrent Neural Networks
- CoreLogic Hedonic Home Value Index
- **Splitting time series data**
- Predicting Sydney House Prices
- Predicting Multiple Time Series

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.

```python
from keras.utils import timeseries_dataset_from_array

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

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

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

Source: Code snippet in Chapter 10 of Chollet.

On time series splits

If you have a lot of time series data, then use:

```python
from keras.utils import timeseries_dataset_from_array
data = range(20); seq = 3; ts = data[:-seq]; target = data[seq:]
nTrain = int(0.5 * len(ts)); nVal = int(0.25 * len(ts))
nTest = len(ts) - nTrain - nVal
print(f"# Train: {nTrain}, # Val: {nVal}, # Test: {nTest}")
```

```
# Train: 8, # Val: 4, # Test: 5
```

```python
trainDS = \
  timeseries_dataset_fr
    ts, target, seq,
      end_index=nTrain)
```

```python
valDS = \
  timeseries_dataset_fr
    ts, target, seq,
      start_index=nTrain,
      end_index=nTrain+nV
```

```python
testDS = \
  timeseries_dataset_fr
    ts, target, seq,
      start_index=nTrain+
```

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

```
Validation dataset
[8, 9, 10] 11
[9, 10, 11] 12
```

```
Test dataset
[12, 13, 14] 15
[13, 14, 15] 16
[14, 15, 16] 17
```

Adapted from: François Chollet (2021), *Deep Learning with Python*, Second Edition, Listing 10.7.

On time series splits II

If you *don't* have a lot of time series data, consider:

```python
1  X = []; y = []
2  for i in range(len(data)-seq):
3      X.append(data[i:i+seq])
4      y.append(data[i+seq])
5  X = np.array(X); y = np.array(y);
```

```python
1  nTrain = int(0.5 * X.sh
2  X_train = X[:nTrain]
3  y_train = y[:nTrain]
```

```python
1  nVal = int(np.ceil(0.25
2  X_val = X[nTrain:nTrain
3  y_val = y[nTrain:nTrain
```

```python
1  nTest = X.shape[0] - nT
2  X_test = X[nTrain+nVal:
3  y_test = y[nTrain+nVal:
```

```
Training dataset
[0, 1, 2] 3
[1, 2, 3] 4
[2, 3, 4] 5
[3, 4, 5] 6
[4, 5, 6] 7
[5, 6, 7] 8
[6, 7, 8] 9
[7, 8, 9] 10
```

```
Validation dataset
[8, 9, 10] 11
[9, 10, 11] 12
[10, 11, 12] 13
[11, 12, 13] 14
[12, 13, 14] 15
```

```
Test dataset
[13, 14, 15] 16
[14, 15, 16] 17
[15, 16, 17] 18
[16, 17, 18] 19
```

Lecture Outline

- Tensors & Time Series
- Some Recurrent Structures
- Recurrent Neural Networks
- CoreLogic Hedonic Home Value Index
- Splitting time series data
- **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  # Which suburb to predict.
2  target_suburb = changes["Sydney"]
3
4  train_ds = \
5    timeseries_dataset_from_array(
6      changes[:-delay],
7      targets=target_suburb[delay:],
8      sequence_length=seq_length,
9      end_index=num_train)
```

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

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

UNSW
SYDNEY

Converting `Dataset` to numpy

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

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

The shape of our training set is now:

```
1  X_train.shape
```

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

```
1  y_train.shape
```

```
(220,)
```

Later, we need the targets as numpy arrays:

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

A dense network

```python
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(train_ds, epochs=1_000, \
15   validation_data=val_ds, callbacks=[es], verbose=0);
```

```
This model has 3191 parameters.
Epoch 126: early stopping
Restoring model weights from the end of the best epoch: 76.
CPU times: user 6.43 s, sys: 3.52 s, total: 9.95 s
Wall time: 3.46 s
```
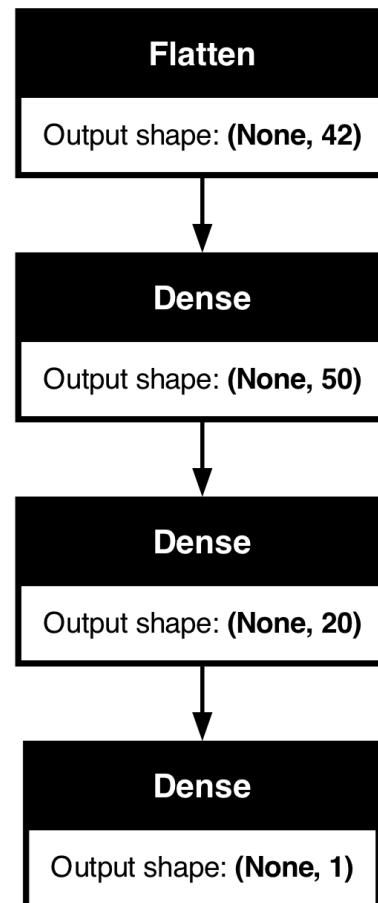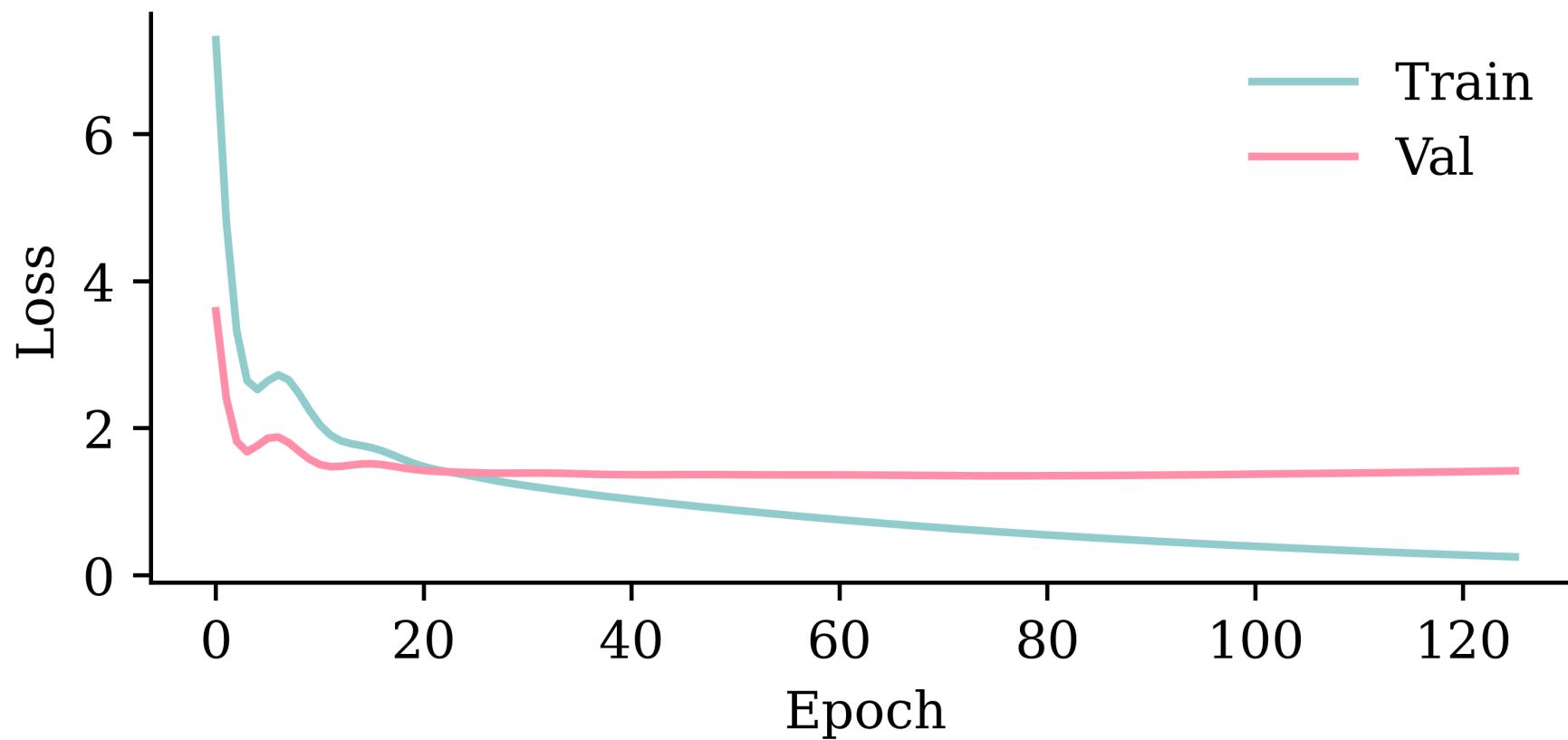
Plot the model

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

Flatten
Output shape: **(None, 42)**

↓

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

↓

Dense
Output shape: **(None, 20)**

↓

Dense
Output shape: **(None, 1)**

Assess the fits
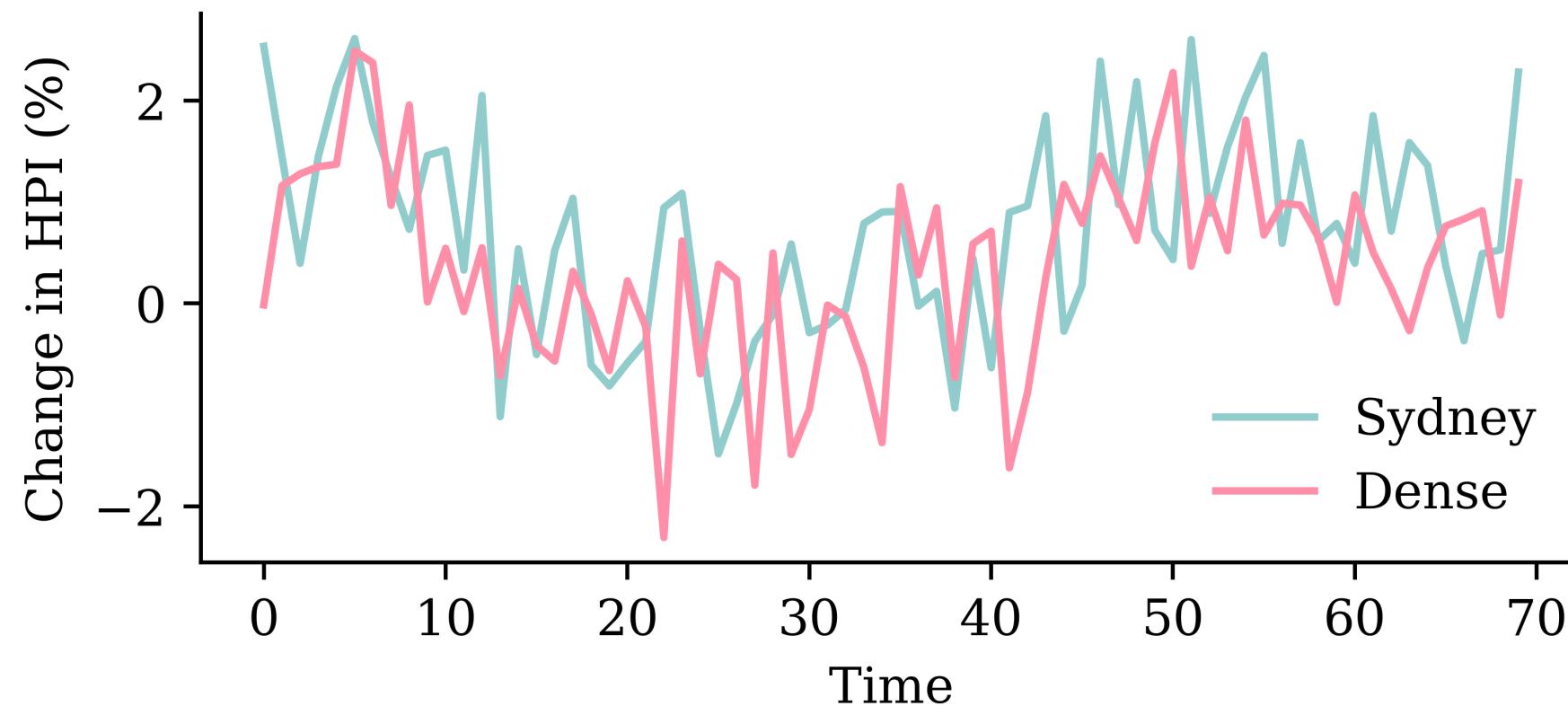

```
1 model_dense.evaluate(val_ds, verbose=0)
```

1.3521653413772583

Plotting the predictions

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(train_ds, epochs=1_000, \
13    validation_data=val_ds, callbacks=[es], verbose=0);
```

```
This model has 2951 parameters.
Epoch 57: early stopping
Restoring model weights from the end of the best epoch: 7.
CPU times: user 3.09 s, sys: 1.56 s, total: 4.65 s
Wall time: 1.74 s
```
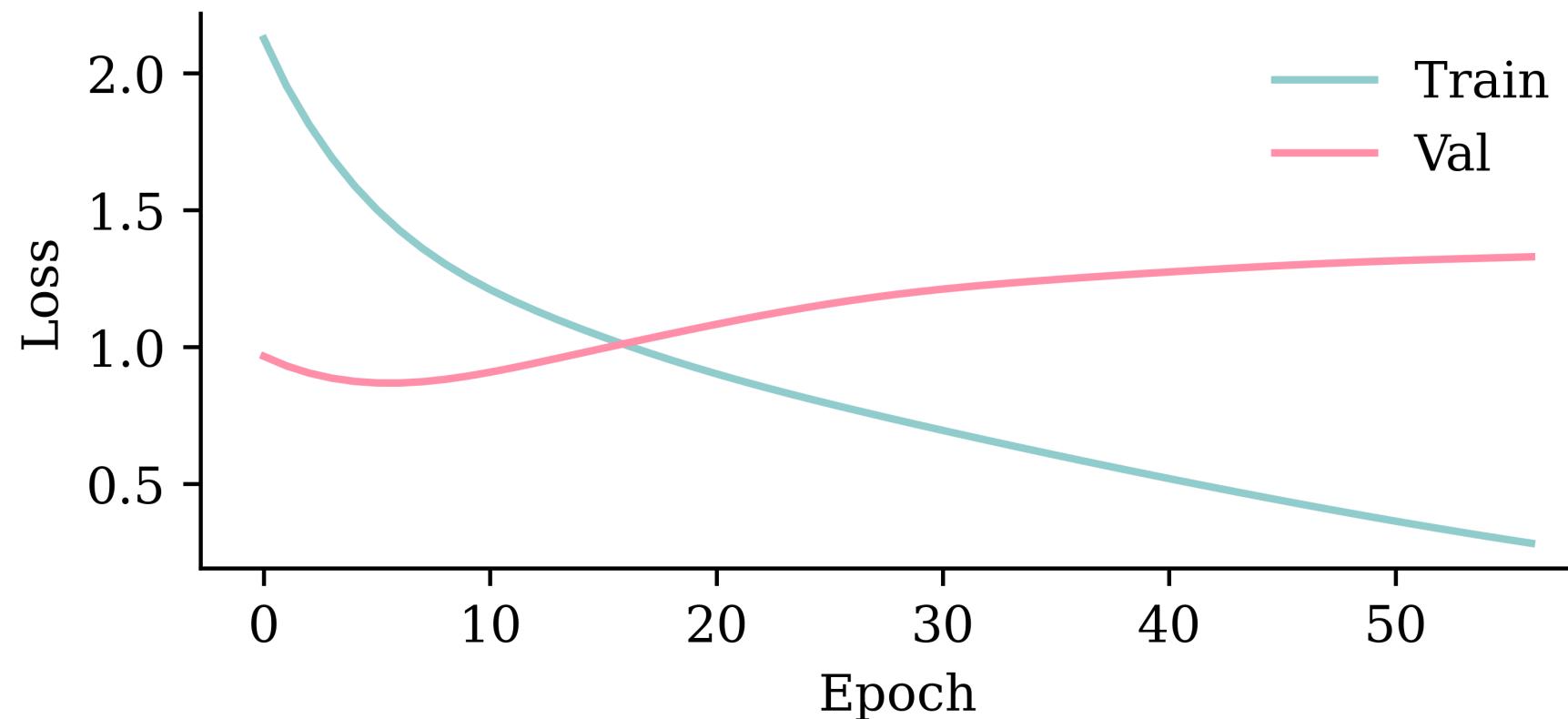
Assess the fits
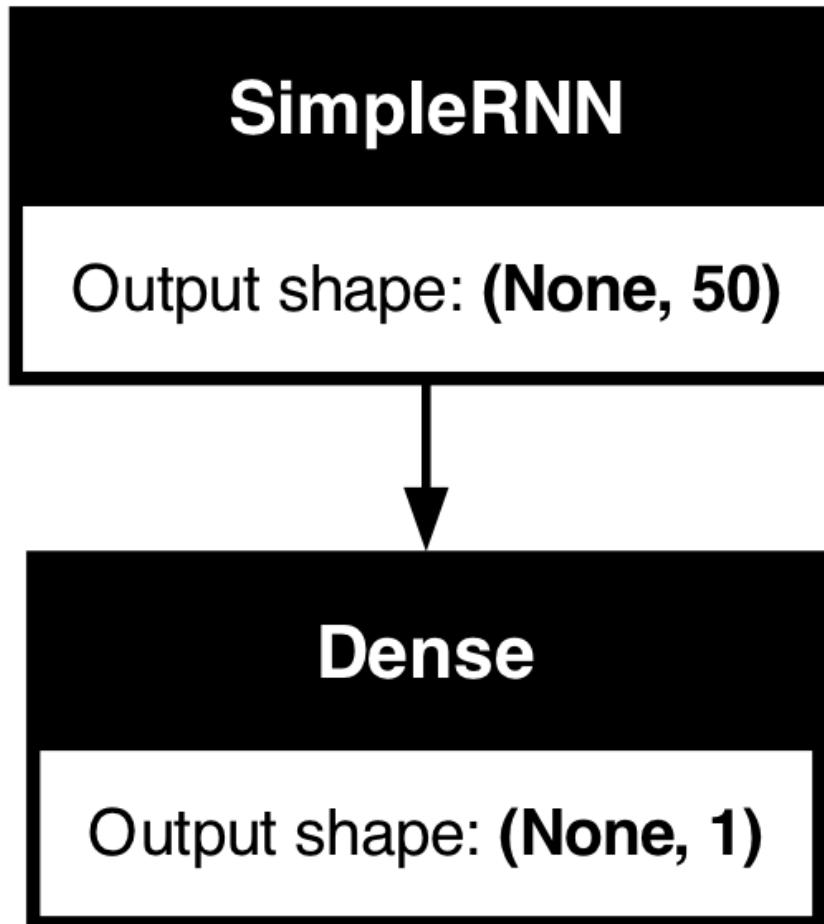

```
1  model_simple.evaluate(val_ds, verbose=0)
```

0.8697079420089722
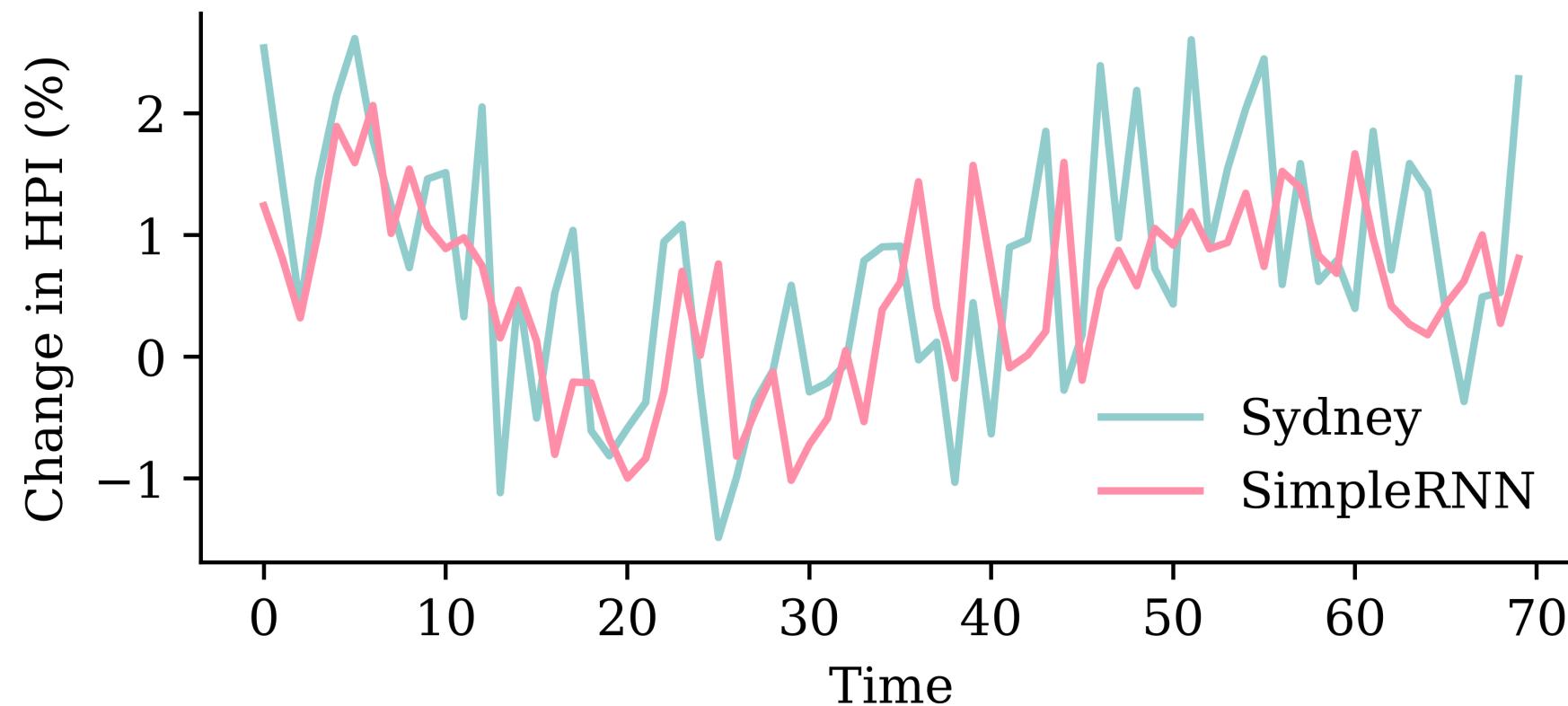
Plot the model

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

Plotting the predictions
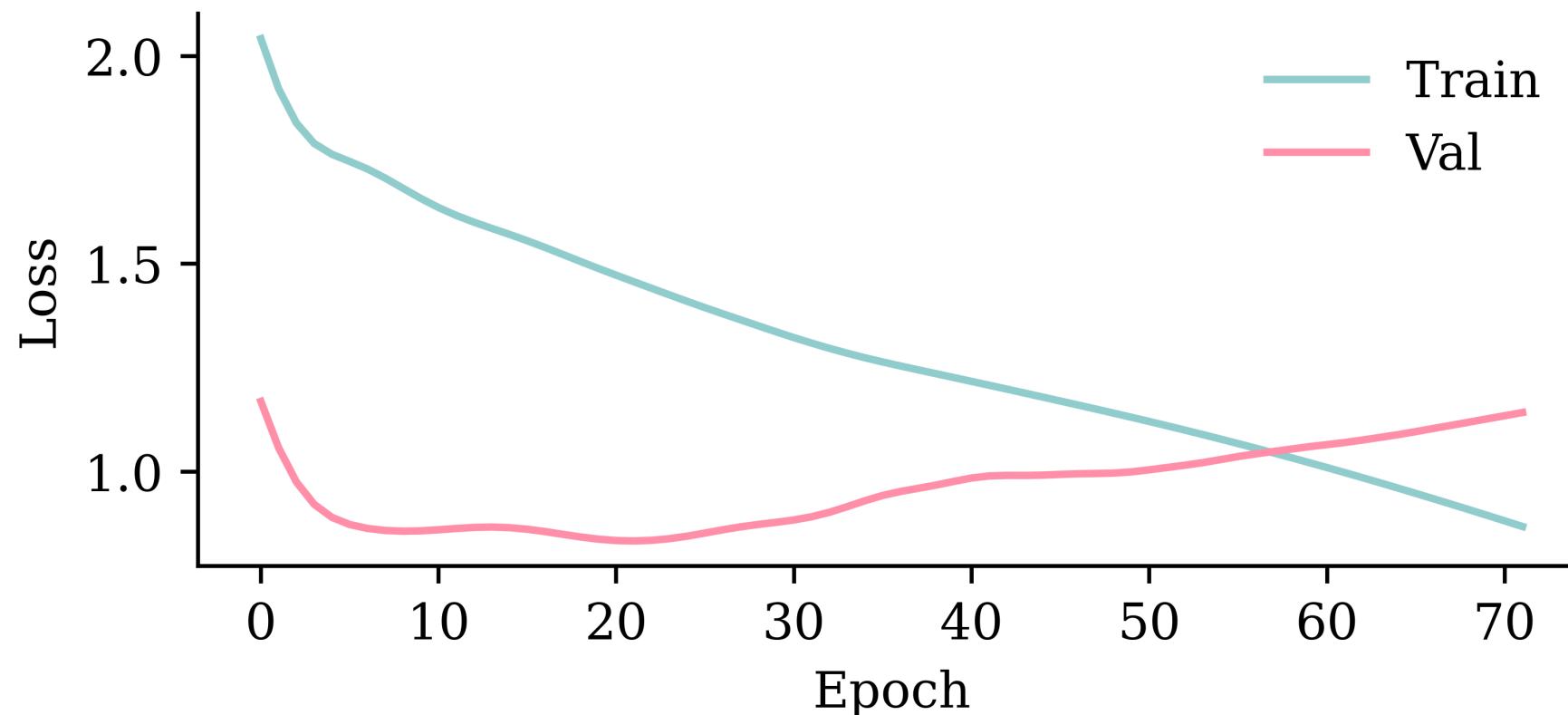
A LSTM layer

```python
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(train_ds, epochs=1_000, \
16      validation_data=val_ds, callbacks=[es], verbose=0);
```

```
Epoch 72: early stopping
Restoring model weights from the end of the best epoch: 22.
CPU times: user 4.09 s, sys: 2.1 s, total: 6.19 s
Wall time: 2.42 s
```

Assess the fits
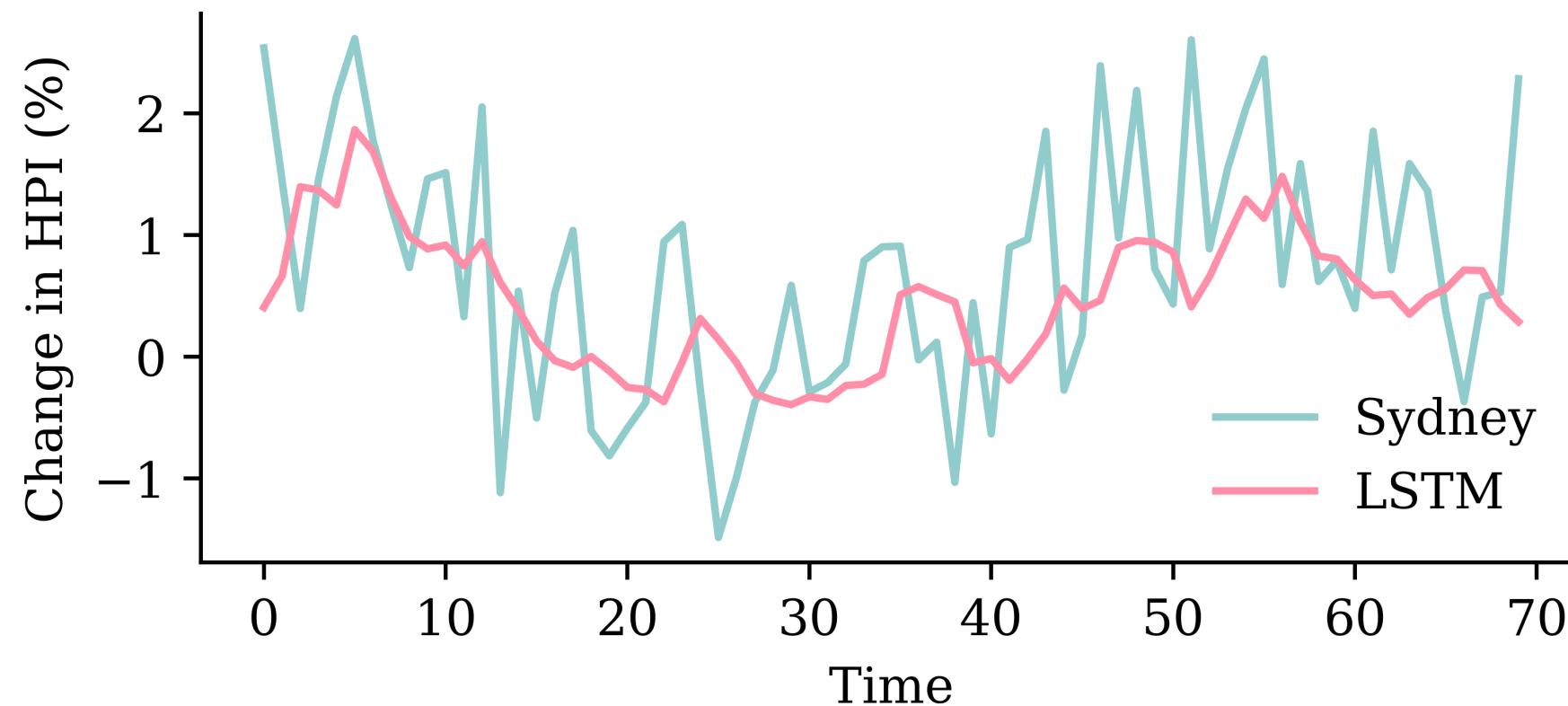

```
1  model_lstm.evaluate(val_ds, verbose=0)
```

0.833128330917358

Plotting the predictions
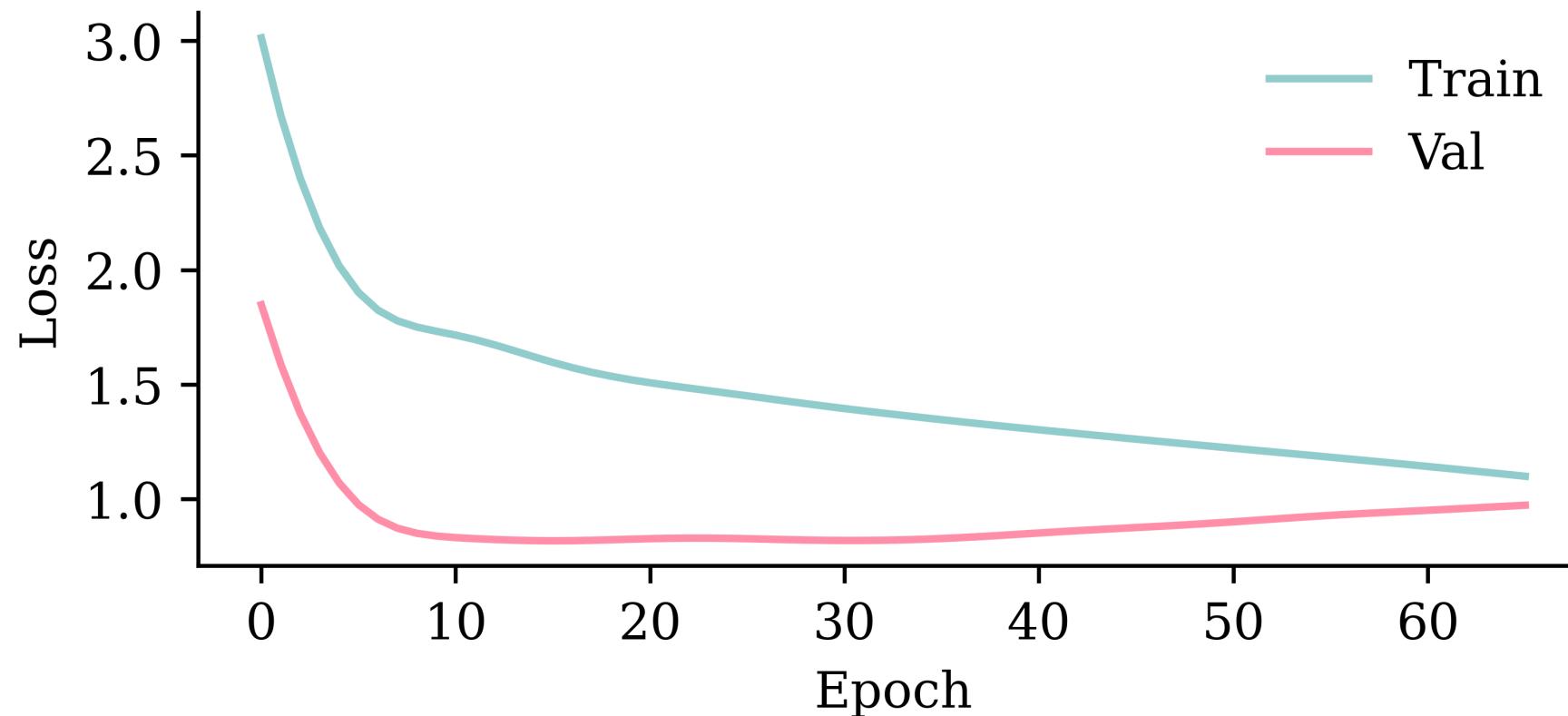
A GRU layer

```python
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(train_ds, epochs=1_000, \
16   validation_data=val_ds, callbacks=[es], verbose=0)
```

```
Epoch 66: early stopping
Restoring model weights from the end of the best epoch: 16.
CPU times: user 3.76 s, sys: 1.87 s, total: 5.63 s
Wall time: 2.25 s
```

Assess the fits
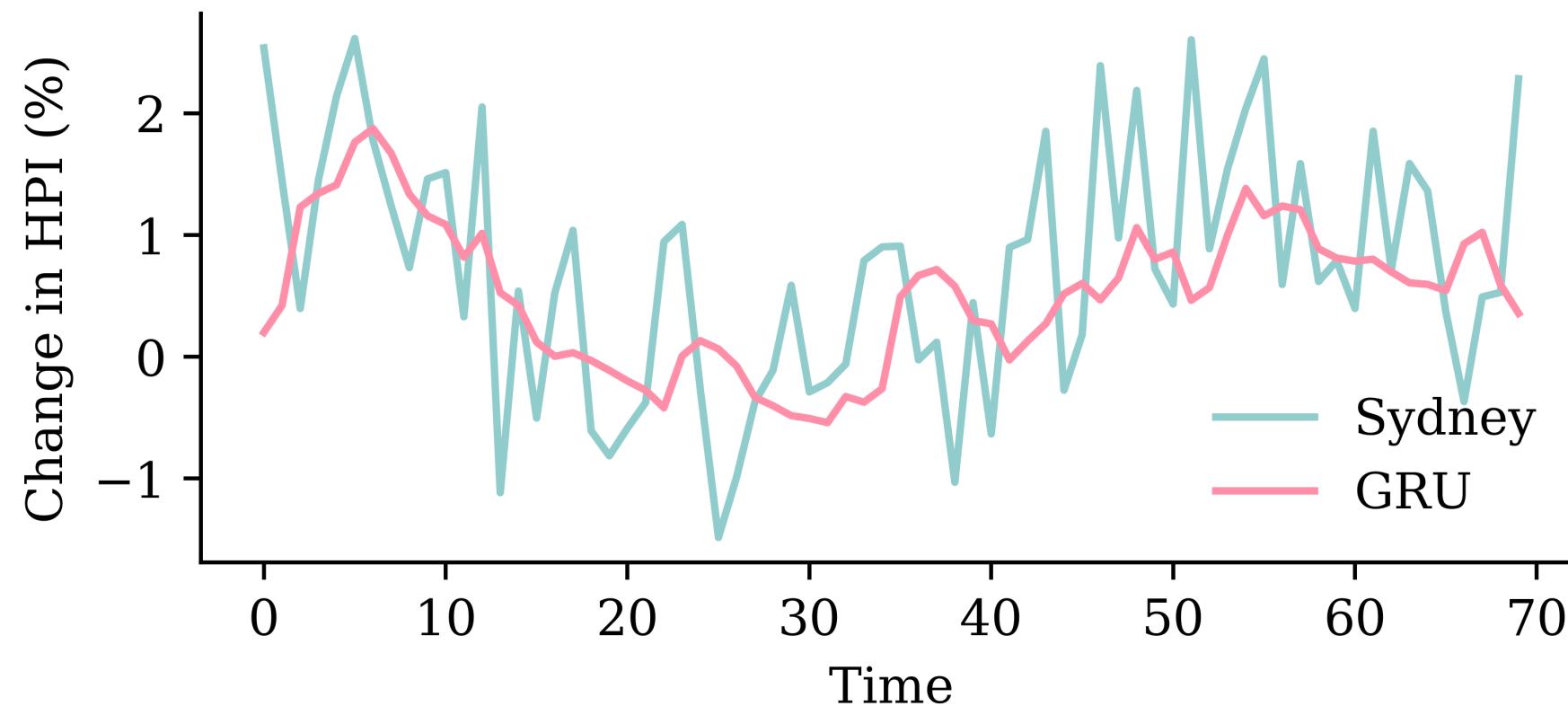

```
1  model_gru.evaluate(val_ds, verbose=0)
```

0.8188608884811401

Plotting the predictions

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(train_ds, epochs=1_000, \
15      validation_data=val_ds, callbacks=[es], verbose=0)
```

```
Epoch 69: early stopping
Restoring model weights from the end of the best epoch: 19.
CPU times: user 4.5 s, sys: 1.98 s, total: 6.49 s
Wall time: 2.92 s
```
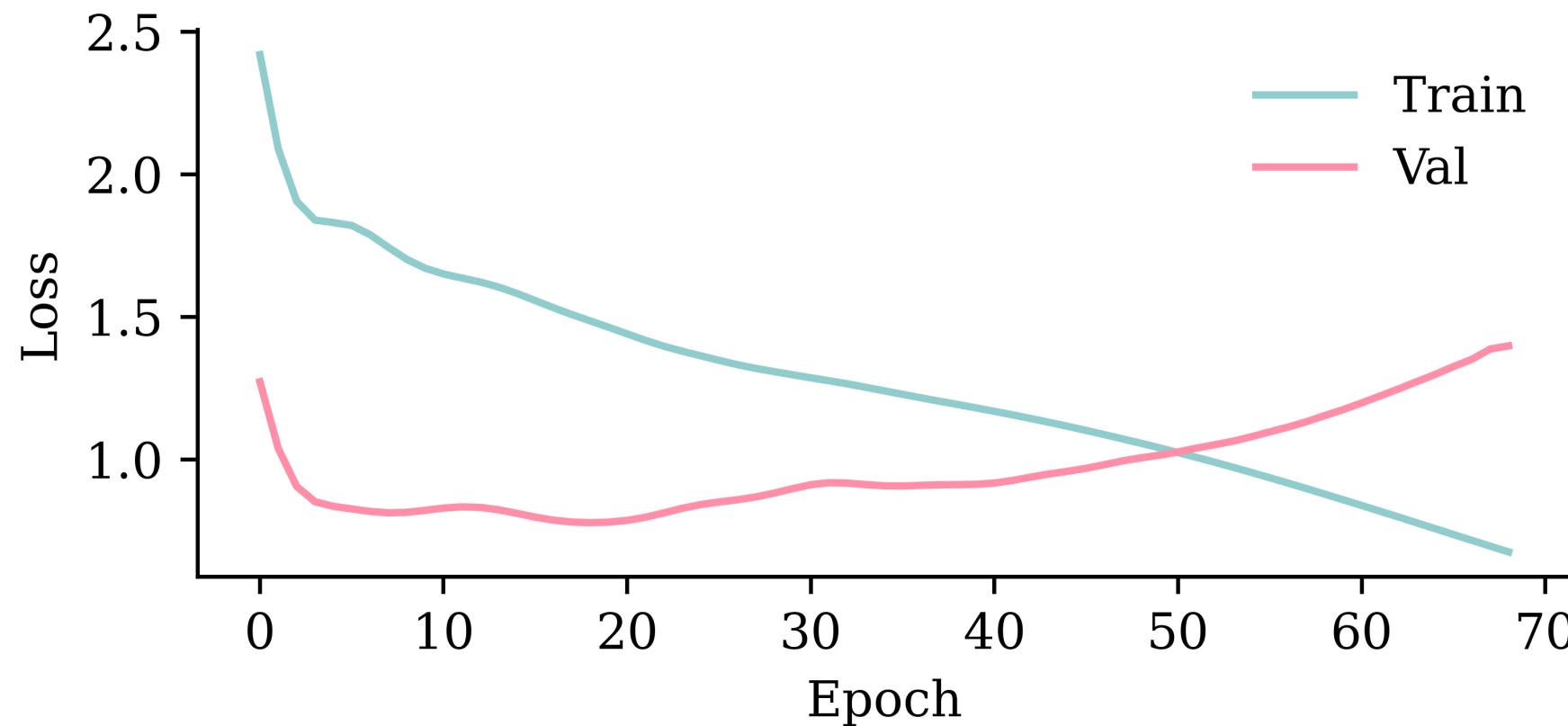
Assess the fits
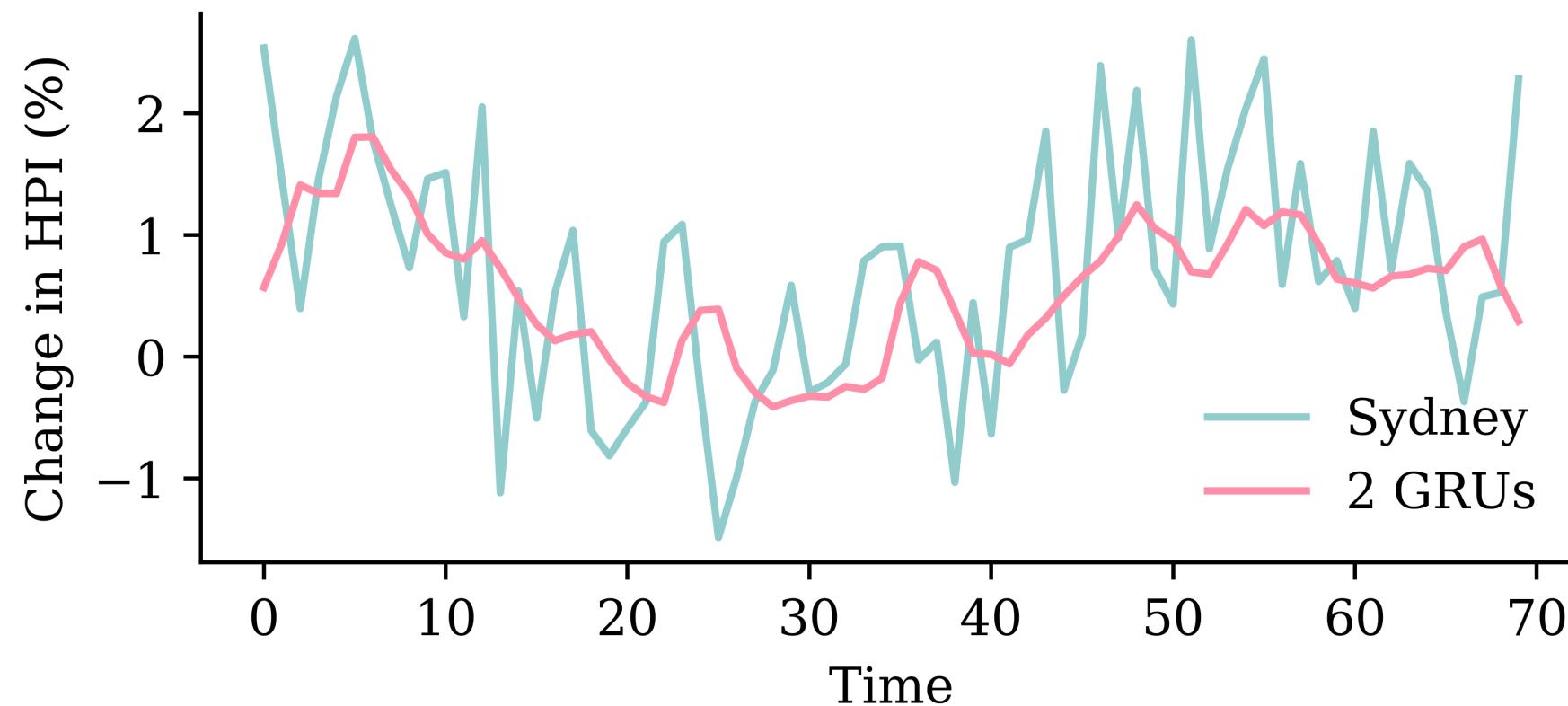

```
1  model_two_grus.evaluate(val_ds, verbose=0)
```

0.777770459651947

Plotting the predictions

Compare the models

	Model	MSE
0	Dense	1.352165
1	SimpleRNN	0.869708
2	LSTM	0.833128
3	GRU	0.818861
4	2 GRUs	0.777770
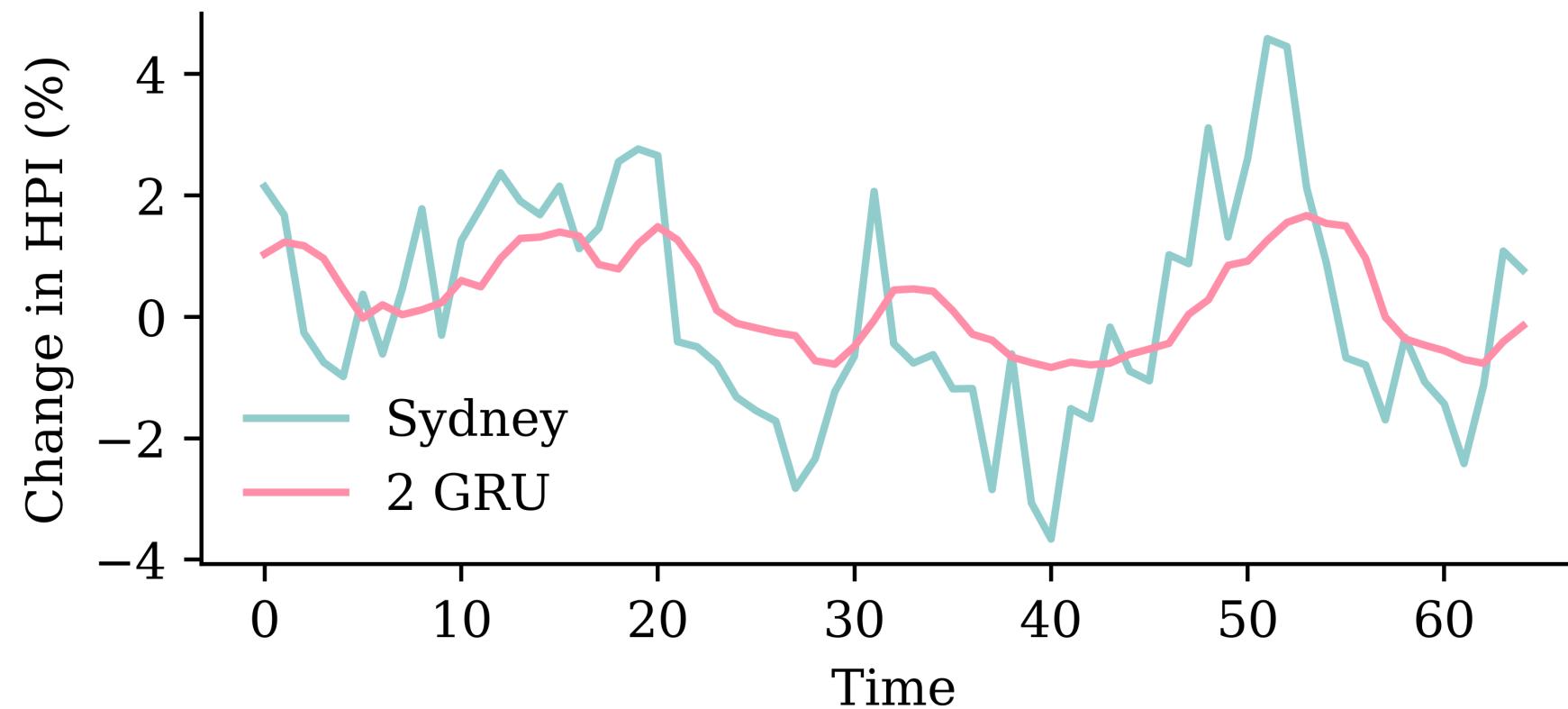
The network with two GRU layers is the best.

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

1.9774343967437744

Test set

Lecture Outline

- Tensors & Time Series

- Some Recurrent Structures

- Recurrent Neural Networks

- CoreLogic Hedonic Home Value Index

- Splitting time series data

- Predicting Sydney House Prices

- **Predicting Multiple Time Series**

Creating dataset objects

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

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

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

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

Converting `Dataset` to numpy

The shape of our training set is now:

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

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

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

```
(220, 7)
```

Later, we need the targets as numpy arrays:

```
1  Y_train = np.concatenate(list(train_ds.map(lambda x, y: y)))
2  Y_val = np.concatenate(list(val_ds.map(lambda x, y: y)))
3  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(train_ds, epochs=1_000, \
14     validation_data=val_ds, callbacks=[es], verbose=0);
```

```
This model has 3317 parameters.
Epoch 96: early stopping
Restoring model weights from the end of the best epoch: 46.
CPU times: user 4.94 s, sys: 2.72 s, total: 7.66 s
Wall time: 2.67 s
```
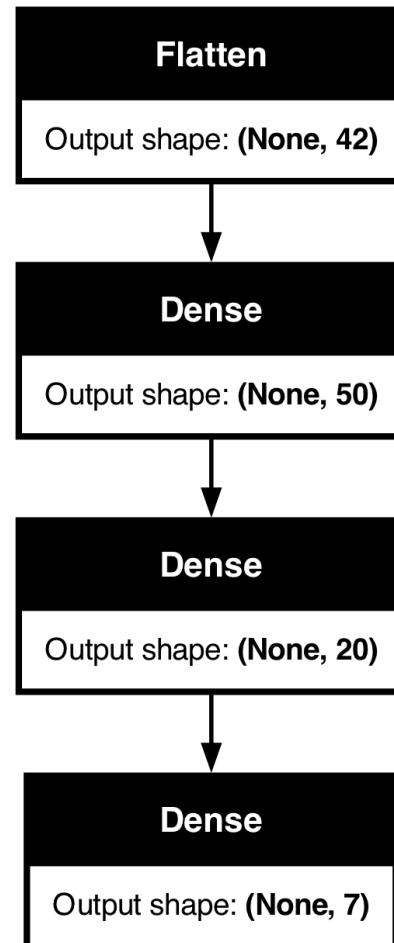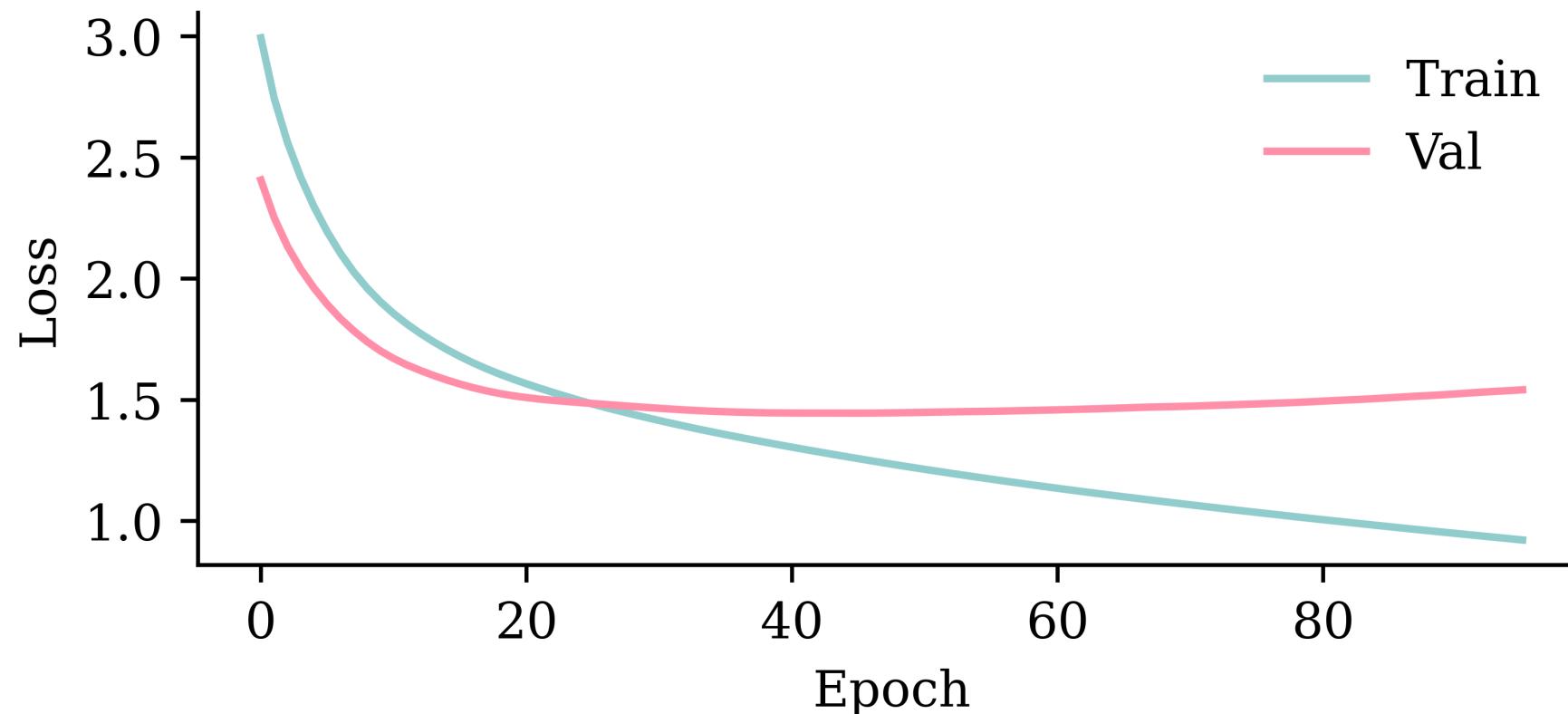
Plot the model

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

Flatten

Output shape: **(None, 42)**

Dense

Output shape: **(None, 50)**

Dense

Output shape: **(None, 20)**

Dense

Output shape: **(None, 7)**

Assess the fits
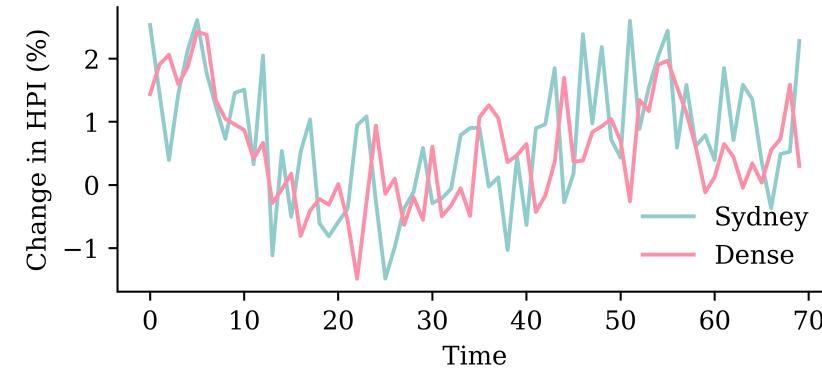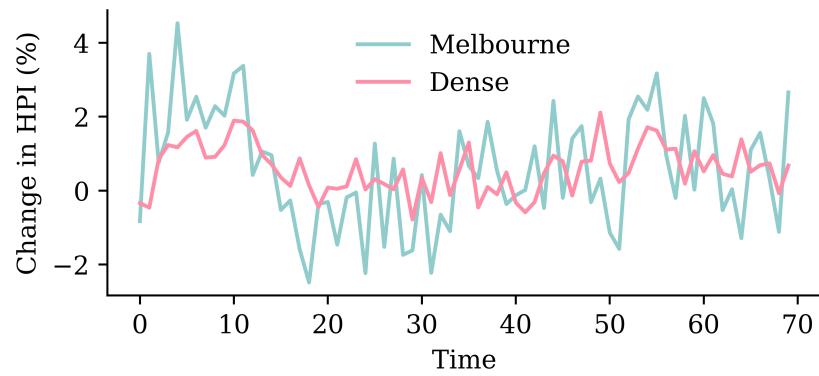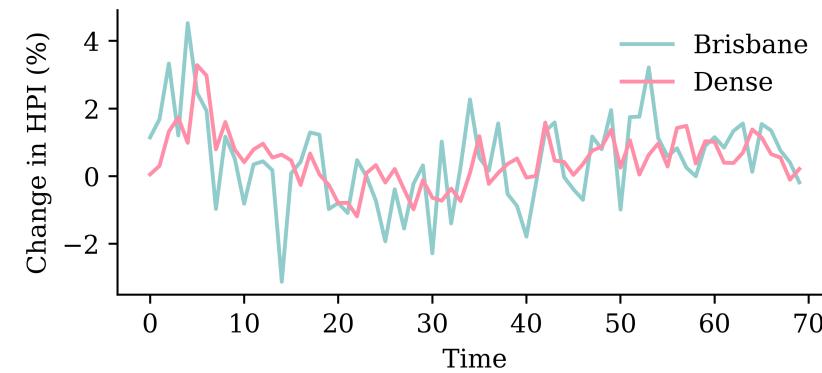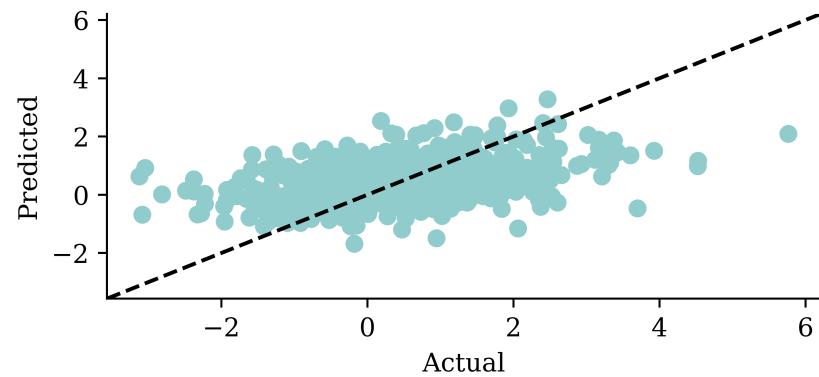


```
1  model_dense.evaluate(val_ds, verbose=0)
```

1.445832371711731

Plotting the predictions

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(train_ds, epochs=1_000, \
13    validation_data=val_ds, callbacks=[es], verbose=0);
```

```
This model has 3257 parameters.
Epoch 93: early stopping
Restoring model weights from the end of the best epoch: 43.
CPU times: user 5.02 s, sys: 2.45 s, total: 7.47 s
Wall time: 2.84 s
```
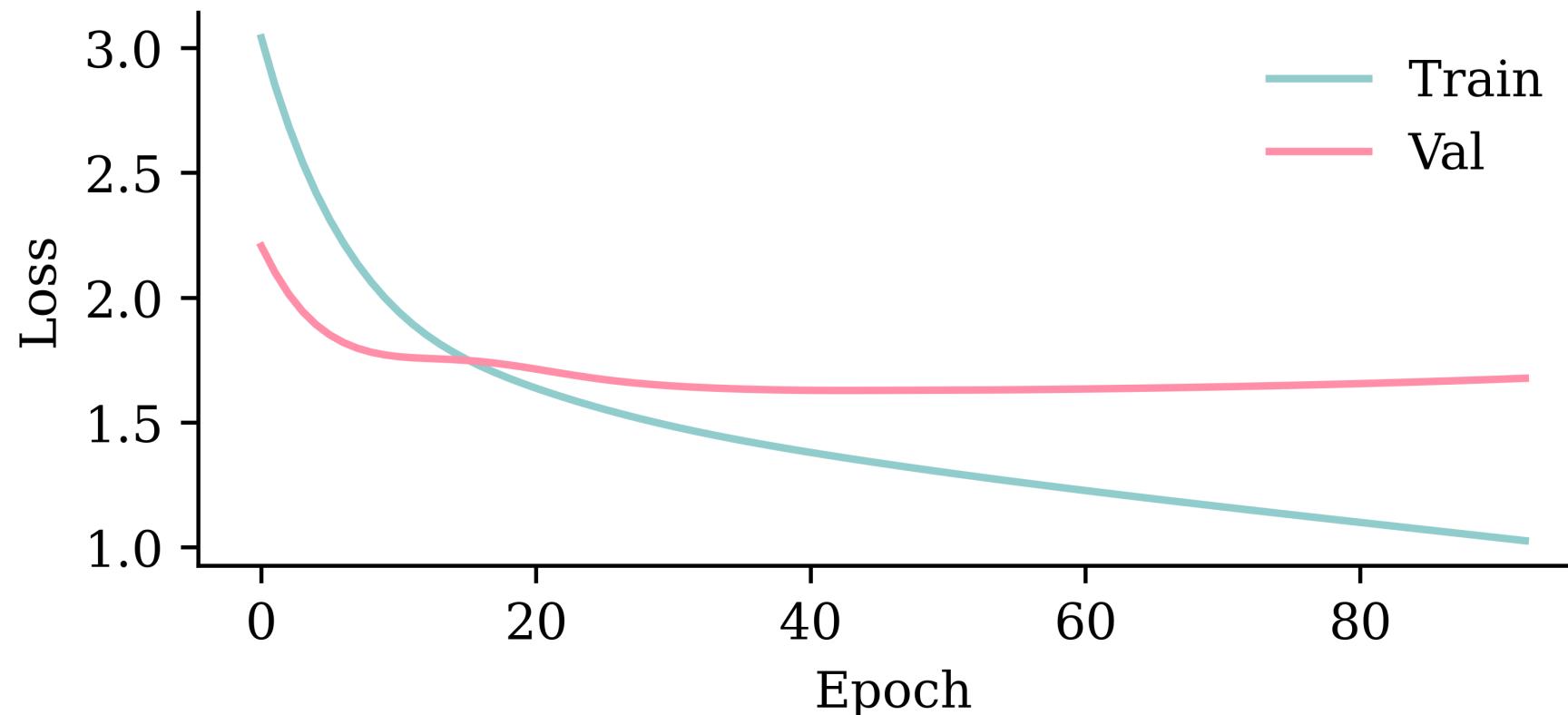
Assess the fits
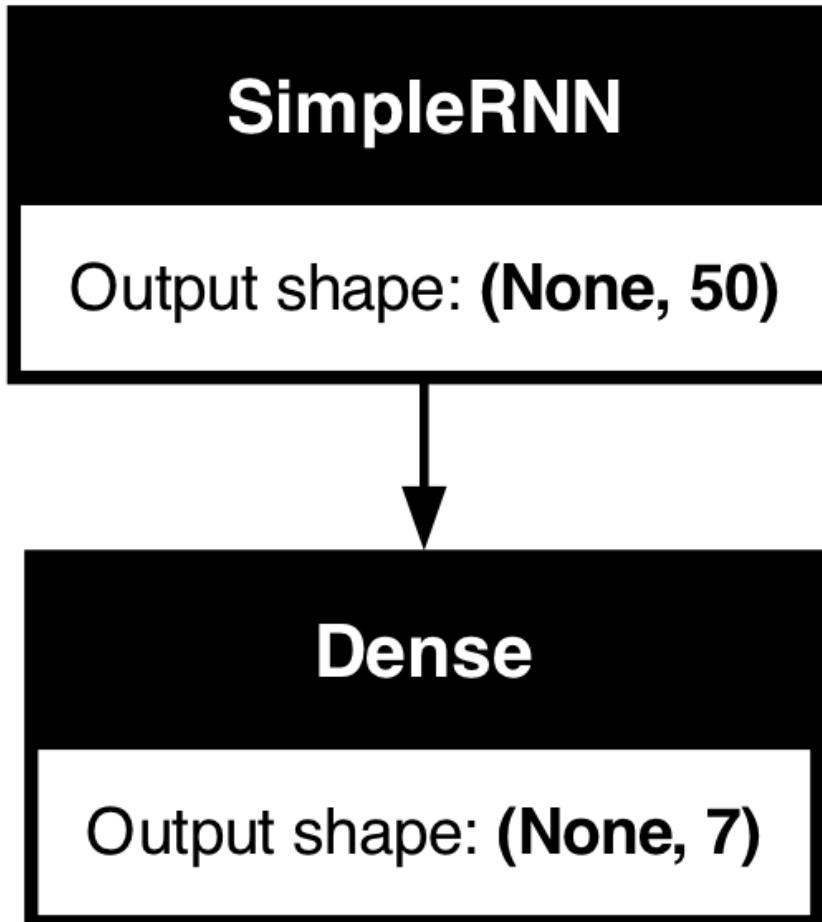

```
1  model_simple.evaluate(val_ds, verbose=0)
```

1.6287589073181152
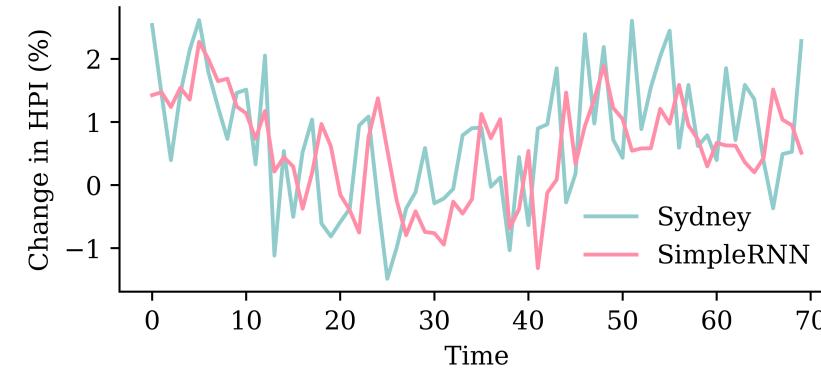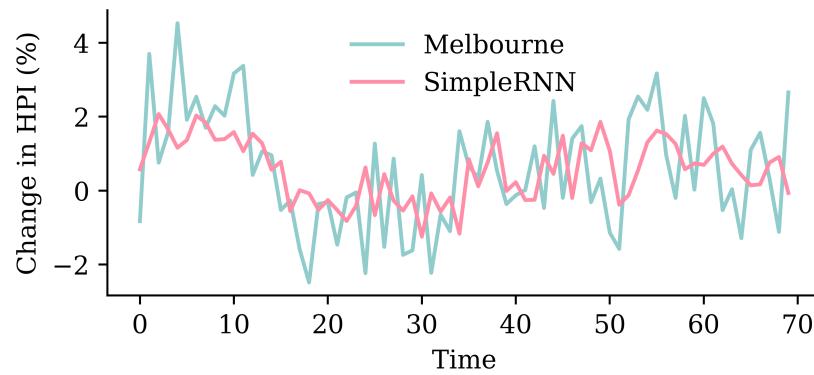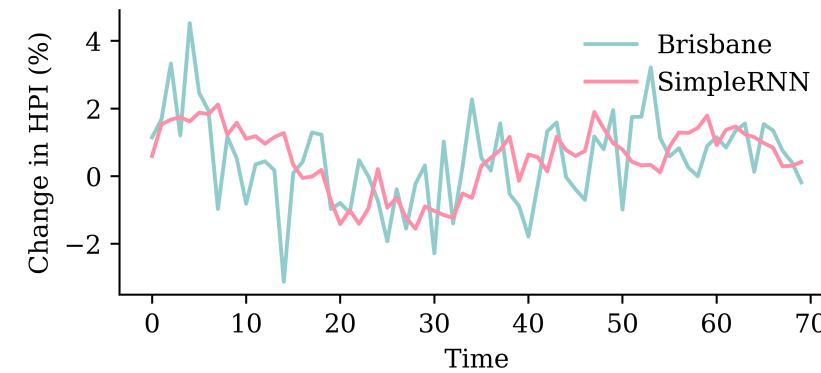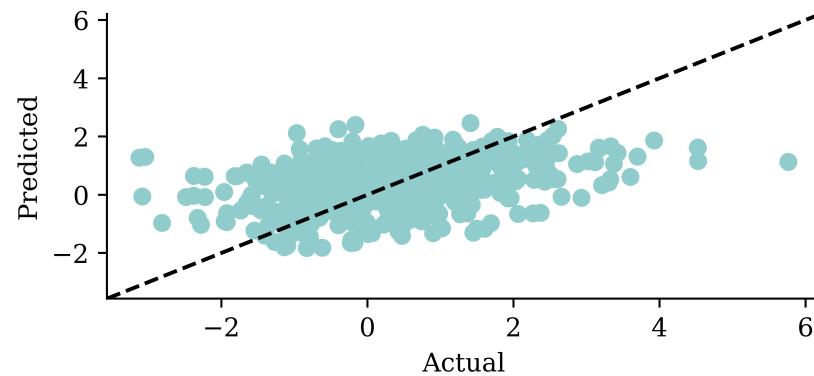
Plot the model

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

Plotting the predictions
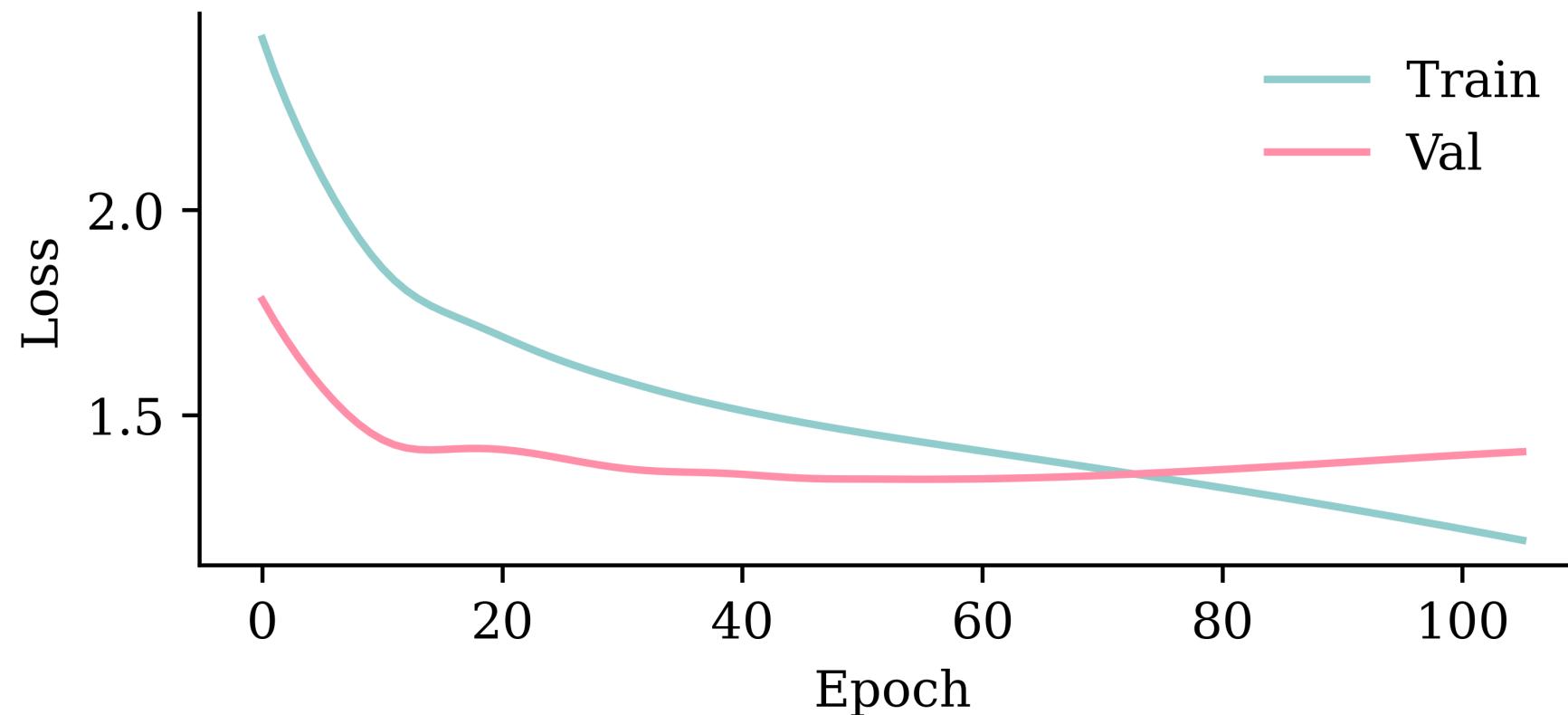
A LSTM layer

```
 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(train_ds, epochs=1_000, \
14     validation_data=val_ds, callbacks=[es], verbose=0);
```

```
Epoch 106: early stopping
Restoring model weights from the end of the best epoch: 56.
CPU times: user 6.03 s, sys: 2.87 s, total: 8.89 s
Wall time: 3.57 s
```

Assess the fits
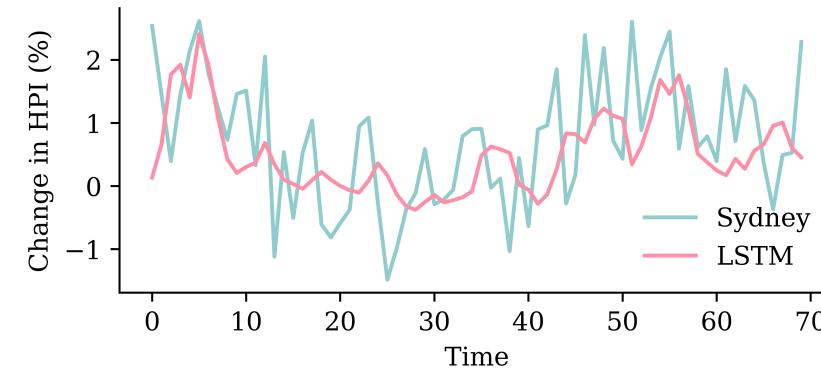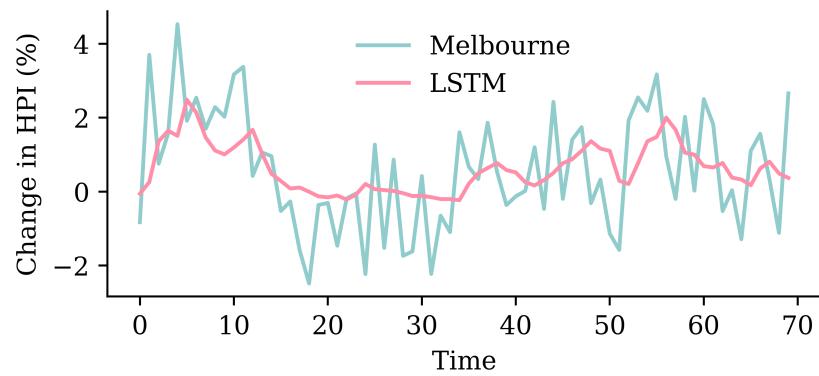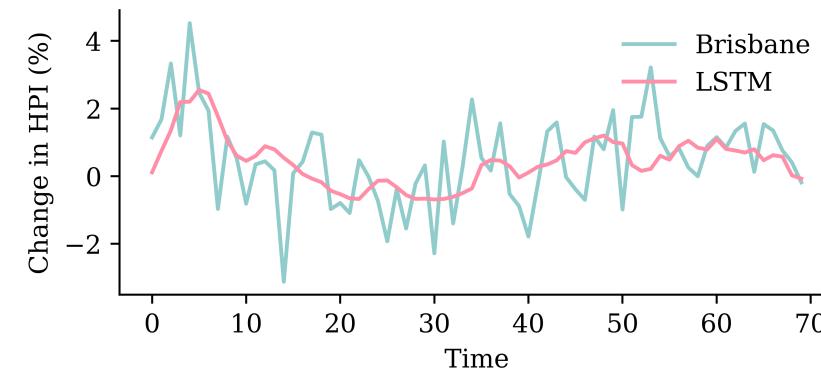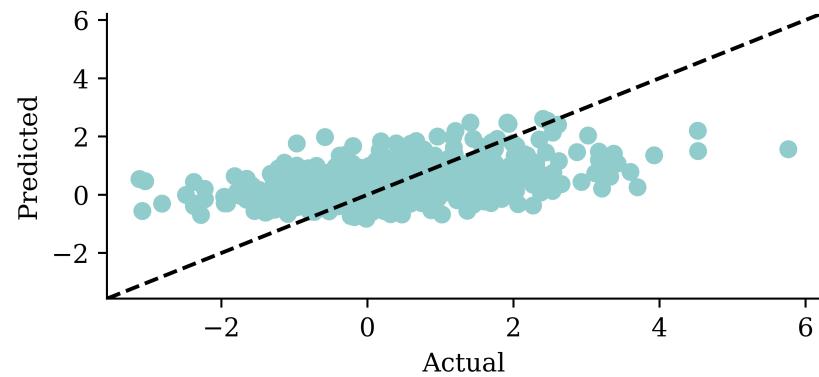

```
1  model_lstm.evaluate(val_ds, verbose=0)
```

1.3439931869506836

Plotting the predictions
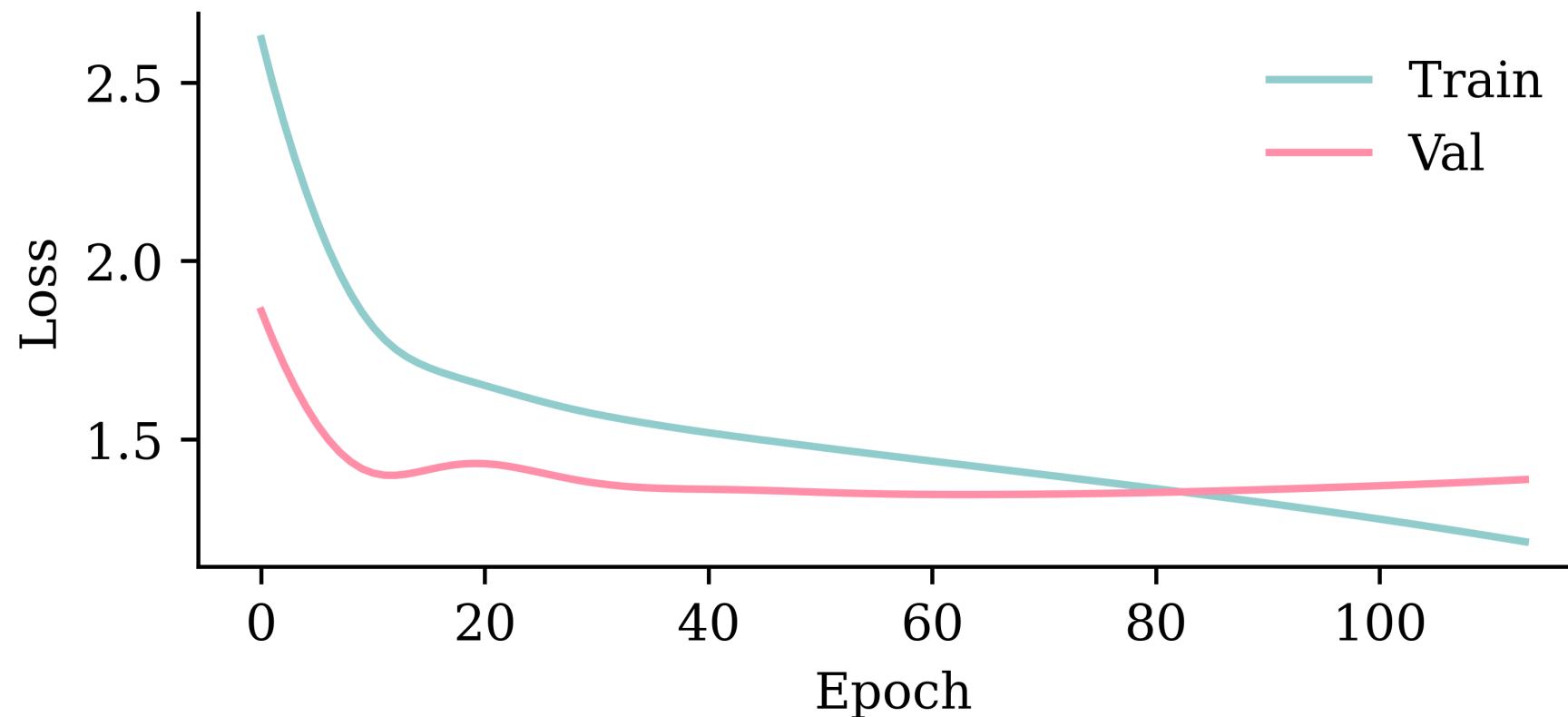
A GRU layer

```
 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(train_ds, epochs=1_000, \
14    validation_data=val_ds, callbacks=[es], verbose=0)
```

```
Epoch 114: early stopping
Restoring model weights from the end of the best epoch: 64.
CPU times: user 6.51 s, sys: 3.18 s, total: 9.69 s
Wall time: 3.9 s
```

Assess the fits
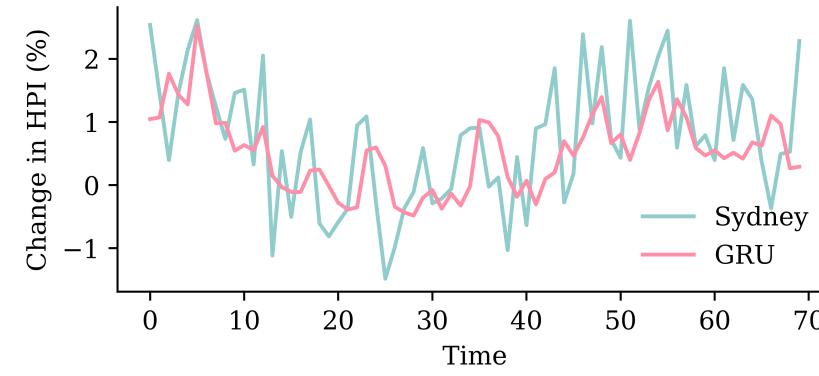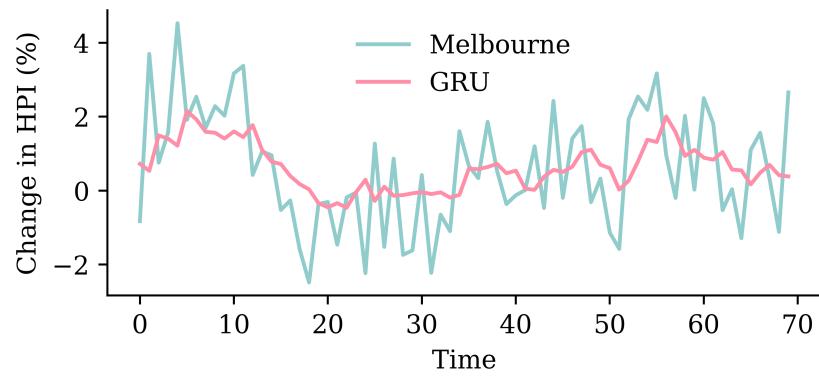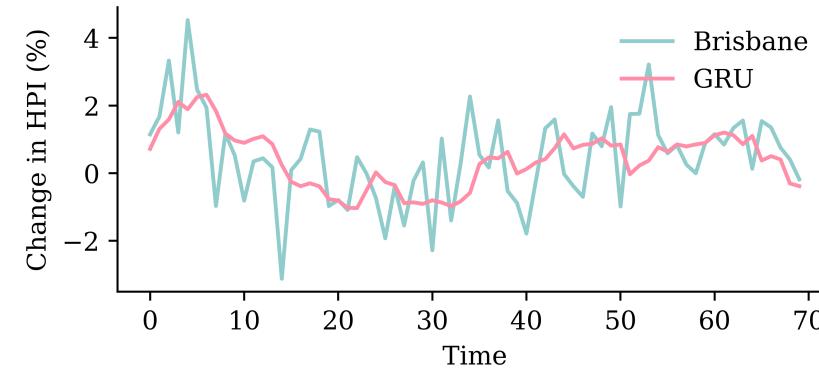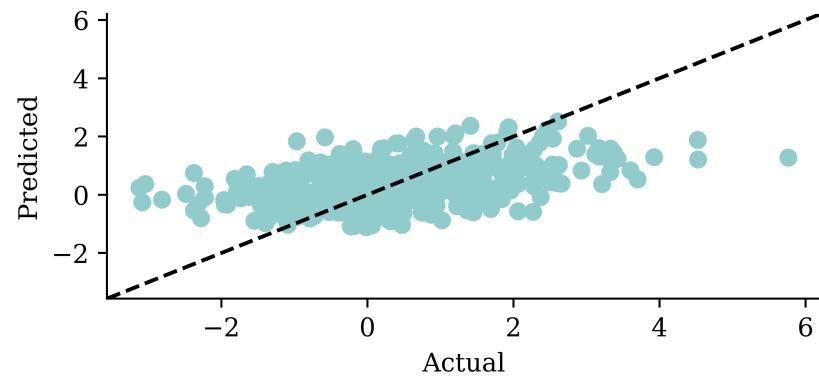

```
1  model_gru.evaluate(val_ds, verbose=0)
```

```
1.3462404012680054
```

Plotting the predictions

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(train_ds, epochs=1_000, \
15    validation_data=val_ds, callbacks=[es], verbose=0)
```
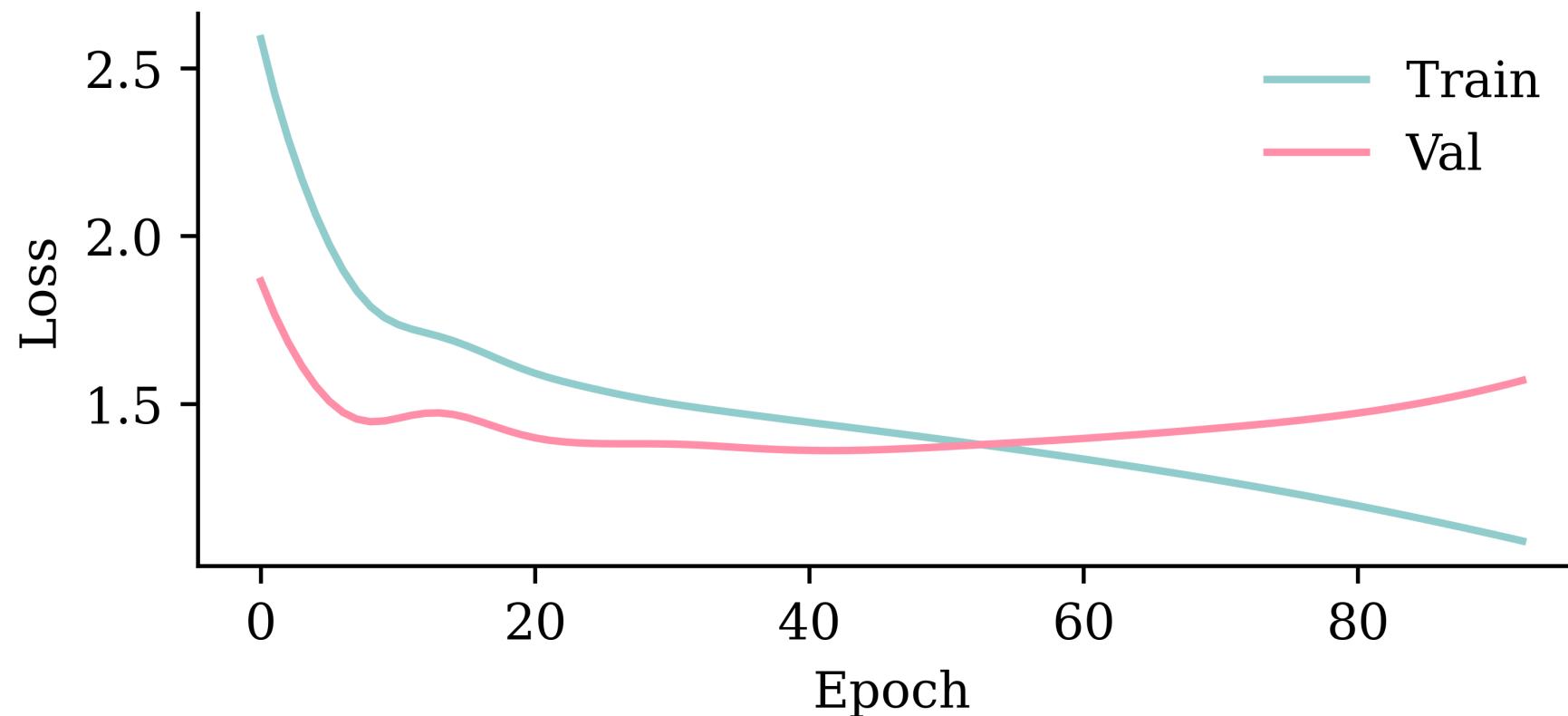
```
Epoch 93: early stopping
Restoring model weights from the end of the best epoch: 43.
CPU times: user 6.1 s, sys: 2.6 s, total: 8.7 s
Wall time: 3.92 s
```

Assess the fits
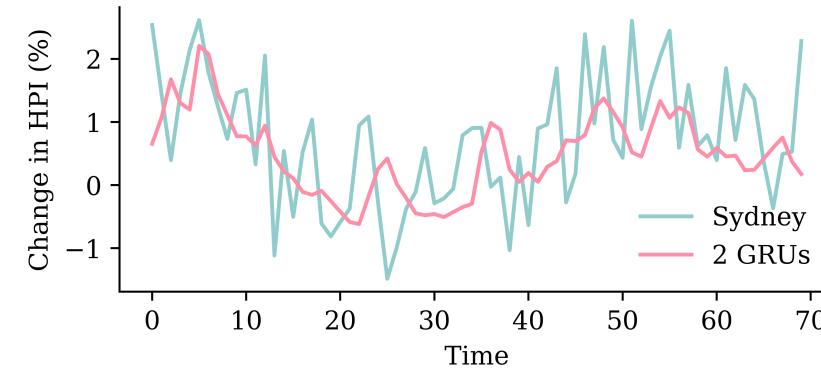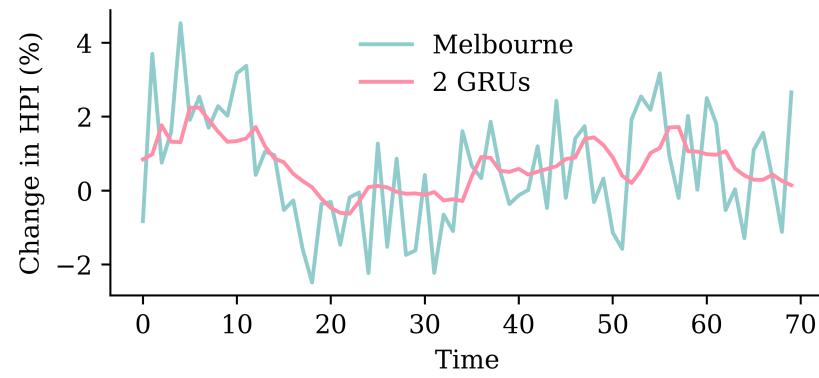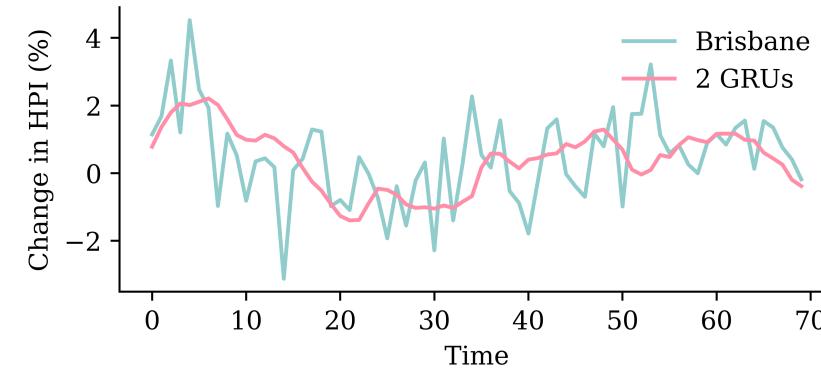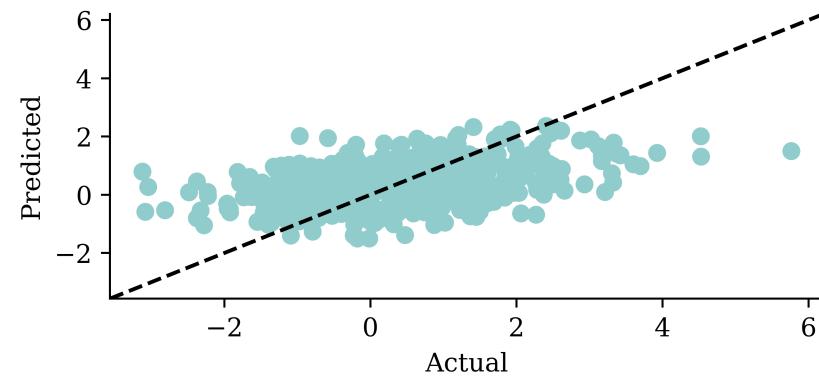

```
1   model_two_grus.evaluate(val_ds, verbose=0)
```

1.3611823320388794

Plotting the predictions

Compare the models

|	Model	MSE
1	SimpleRNN	1.628759
0	Dense	1.445832
4	2 GRUs	1.361182
3	GRU	1.346240
2	LSTM	1.343993
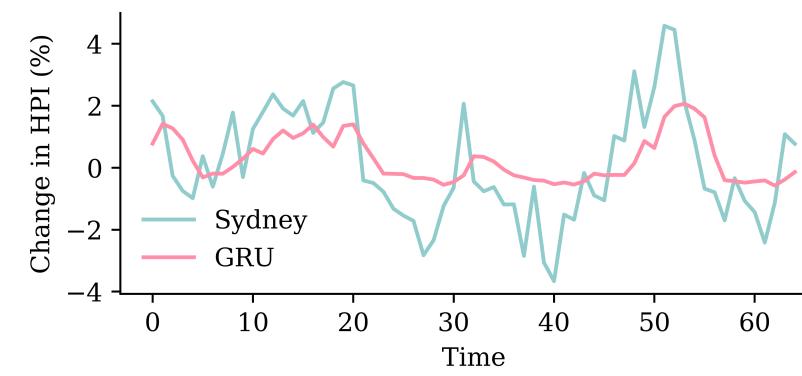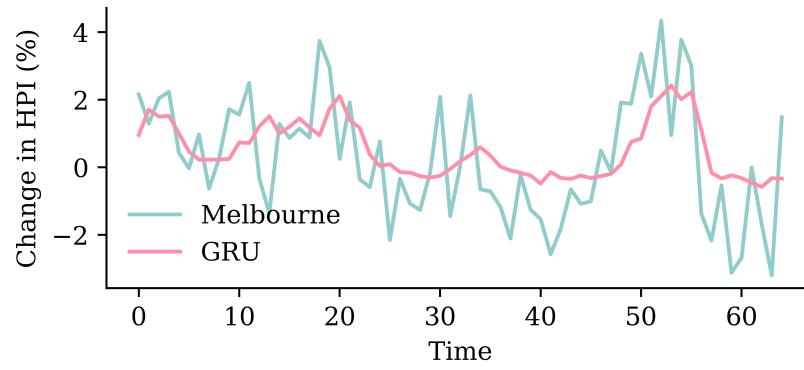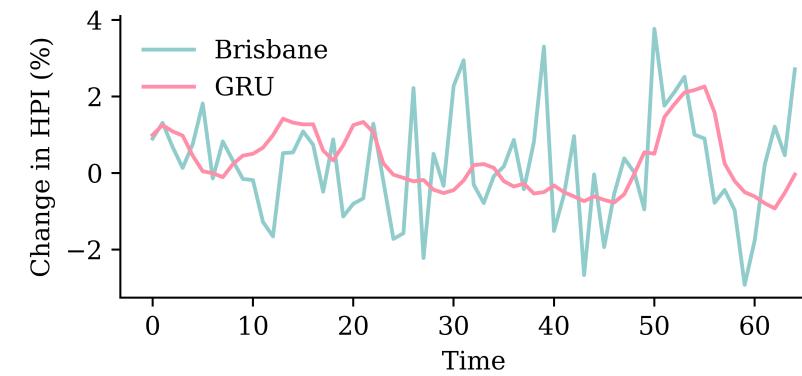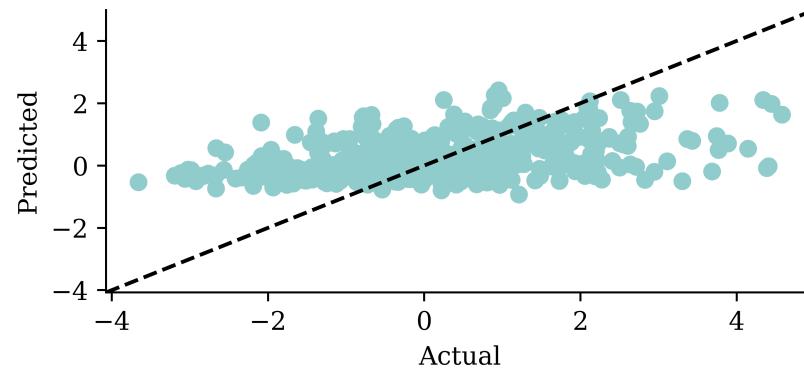
The network with an LSTM layer is the best.

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

```
1.879453420639038
```

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       : 3.11.8
IPython version      : 8.23.0

keras     : 3.2.0
matplotlib: 3.8.4
numpy     : 1.26.4
pandas    : 2.2.1
seaborn   : 0.13.2
scipy     : 1.11.0
torch     : 2.2.2
tensorflow: 2.16.1
tf_keras  : 2.16.0
```

Glossary

- GRU

- LSTM

- recurrent neural networks

- SimpleRNN