Recurrent Neural Networks

Prof. Gheith Abandah

References:

- *Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow* by Aurélien Géron (O’Reilly). 2019, 978-1-492-03264-9.
- François Chollet, *Deep Learning with Python*, Manning Pub. 2018

Outline

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
    1. Implementing a simple RNN
    2. Deep RNNs
    3. Forecasting Several Time Steps Ahead
5. Handling long sequences
    1. LSTM cell
    2. GRU cell
6. Exercises

Introduction

- YouTube Video: *Deep Learning with Tensorflow - The Recurrent Neural Network Model* from Cognitive Class

https://youtu.be/C0xoB8L8ms0

1. Introduction

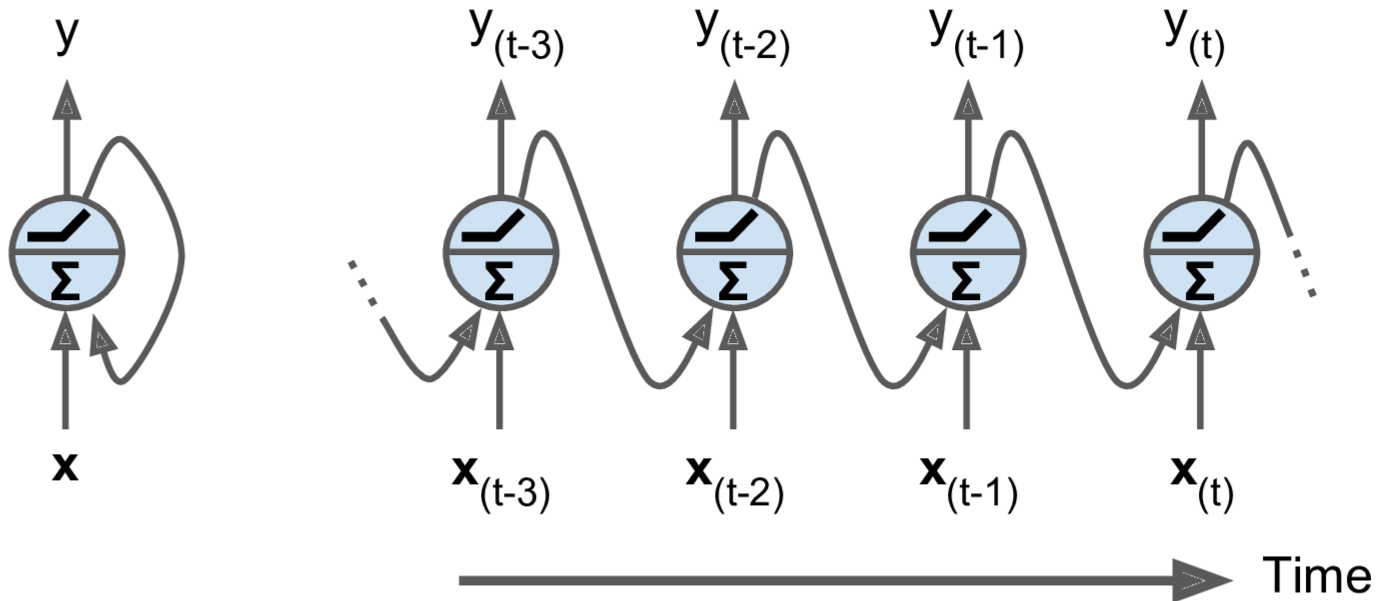
- **Recurrent neural networks (RNNs)** are used to handle time series data or sequences.
- Applications:
  - Predicting the future (stock prices)
  - Autonomous driving systems (predicting trajectories)
  - Natural language processing (automatic translation, speech-to-text, or sentiment analysis)
  - Creativity (music composition, handwriting, drawing)
  - Image analysis (image captions)

Outline

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
    1. Implementing a simple RNN
    2. Deep RNNs
    3. Forecasting Several Time Steps Ahead
5. Handling long sequences
    1. LSTM cell
    2. GRU cell
6. Exercises

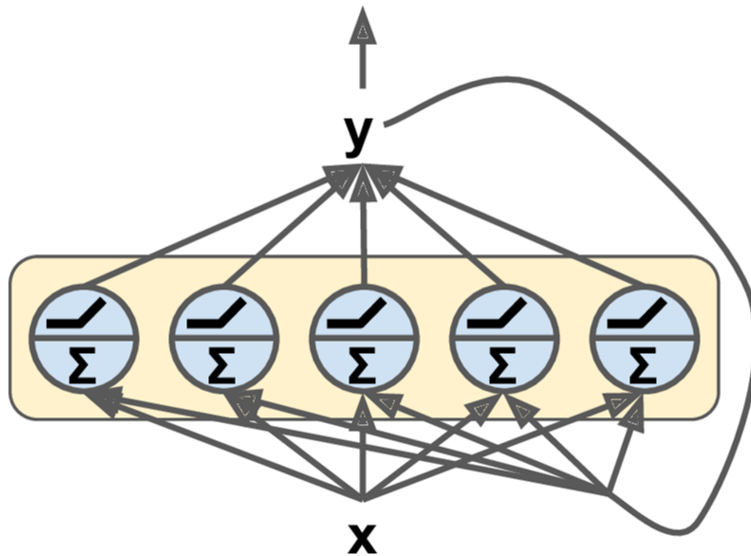
2. Recurrent Neurons and Layers

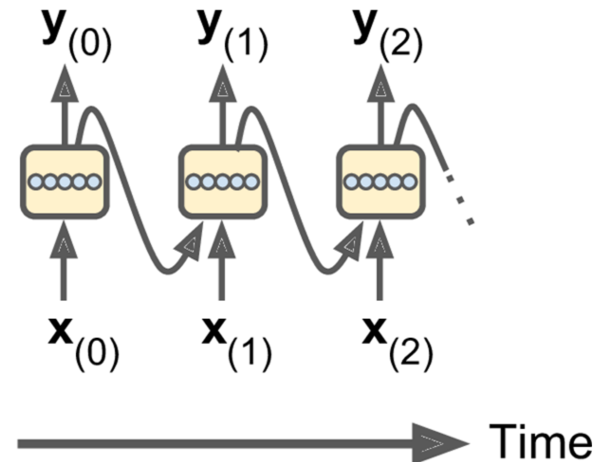
- The figure below shows a **recurrent neuron** (left), unrolled through time (right).



2. Recurrent Neurons and Layers

- Multiple recurrent neurons can be used in a **layer**.



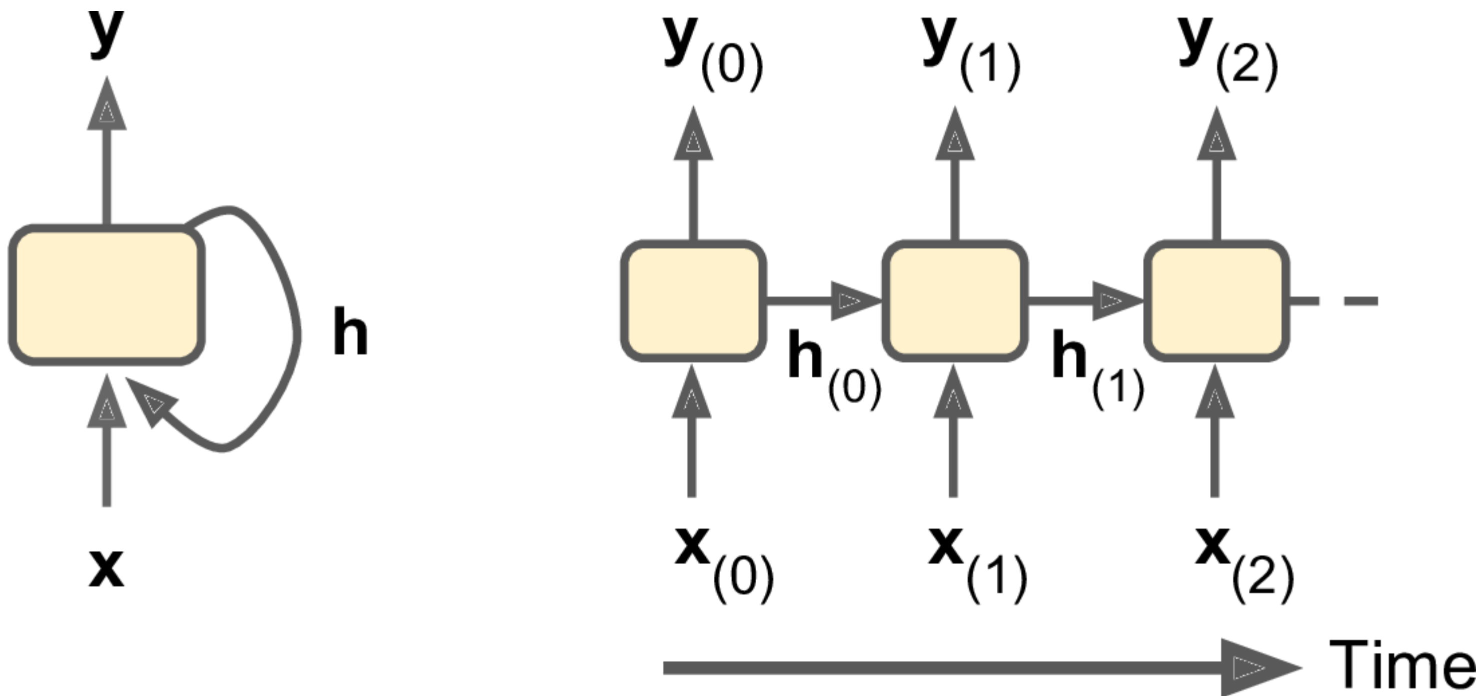


- The **output** of the layer is:

$$\mathbf{Y}_{(t)} = \phi\left(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b}\right)$$

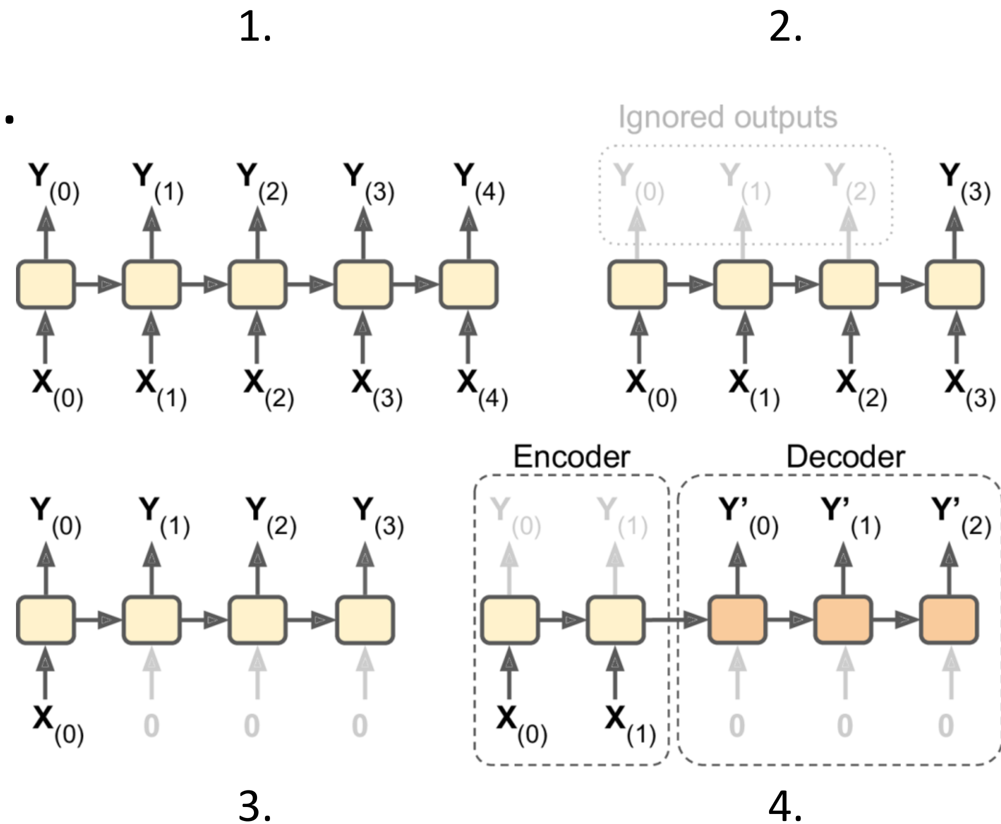
2. Recurrent Neurons and Layers

- Recurrent neurons have memory (hold state) and are called **memory cells**.
- The state $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$, not always $\equiv \mathbf{y}_{(t)}$



2. Recurrent Neurons and Layers: Input and Output Sequences

1. **Seq to seq net.**: For predicting the future.
2. **Seq to vector**: For analysis, e.g., sentiment score.
3. **Vector to seq**: For image captioning.
4. **Encoder-decoder**: For sequence transcription.

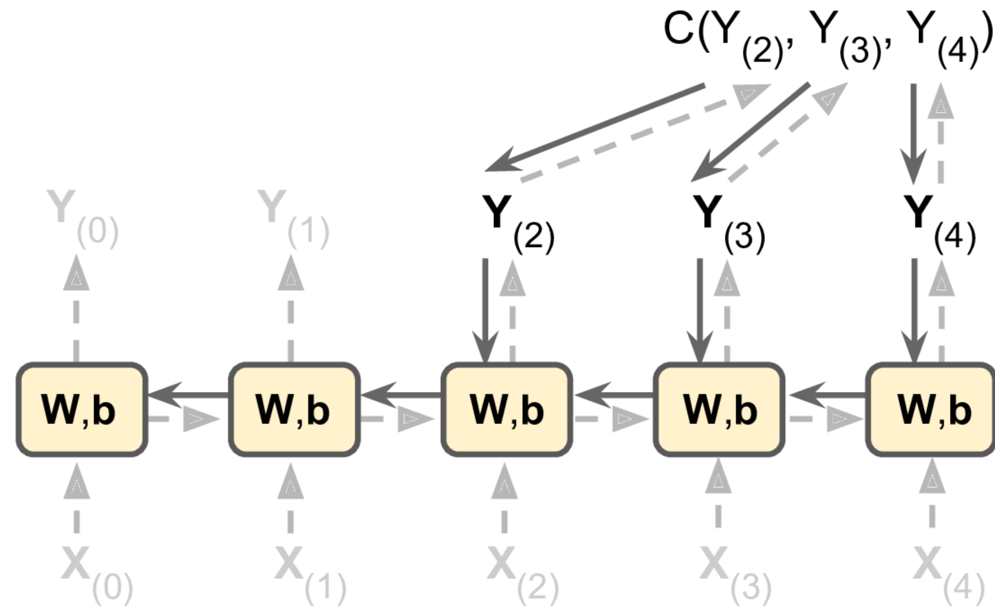


Outline

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
    1. Implementing a simple RNN
    2. Deep RNNs
    3. Forecasting Several Time Steps Ahead
5. Handling long sequences
    1. LSTM cell
    2. GRU cell
6. Exercises

3. Training RNNs

- Training using strategy called **backpropagation through time** (BPTT).
- Forward pass (dashed)
- Cost function of the not-ignored outputs.
- Cost gradients are propagated backward through the unrolled network.

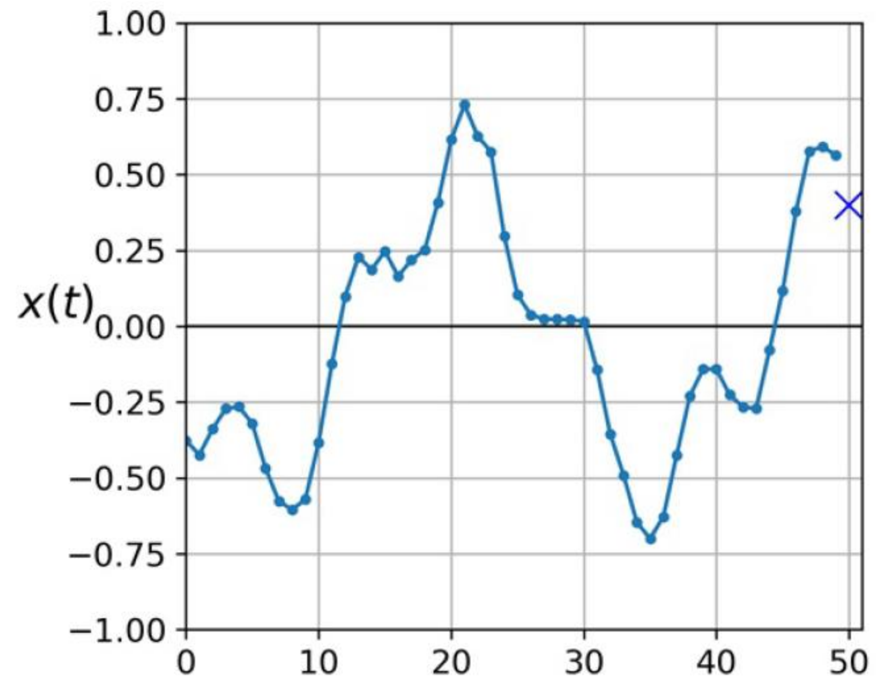


Outline

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
    1. Implementing a simple RNN
    2. Deep RNNs
    3. Forecasting Several Time Steps Ahead
5. Handling long sequences
    1. LSTM cell
    2. GRU cell
6. Exercises

4. Forecasting a Time Series

- The data is a sequence of one or more values per **time step**.
  - **Univariate** time series
  - **Multivariate** time series
- **Forecasting**: predicting future values
  - Forecast the next value
  - Forecast N next values



4.1 Implementing a Simple RNN

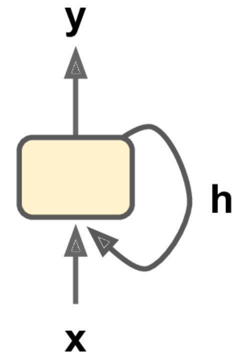
```
# Generate 10,000 time series
n_steps = 50
series = generate_time_series(10000, n_steps + 1)

# Split them 7,000 : 2,000 : 1,000
X_train, y_train = series[:7000, :n_steps],
      series[:7000, -1] # (7000, 50, 1), (7000, 1)
X_valid, y_valid = series[7000:9000, :n_steps],
      series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps],
      series[9000:, -1]
```

4.1 Implementing a Simple RNN

```
# Sequential model of one neuron
model = keras.models.Sequential([
        keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

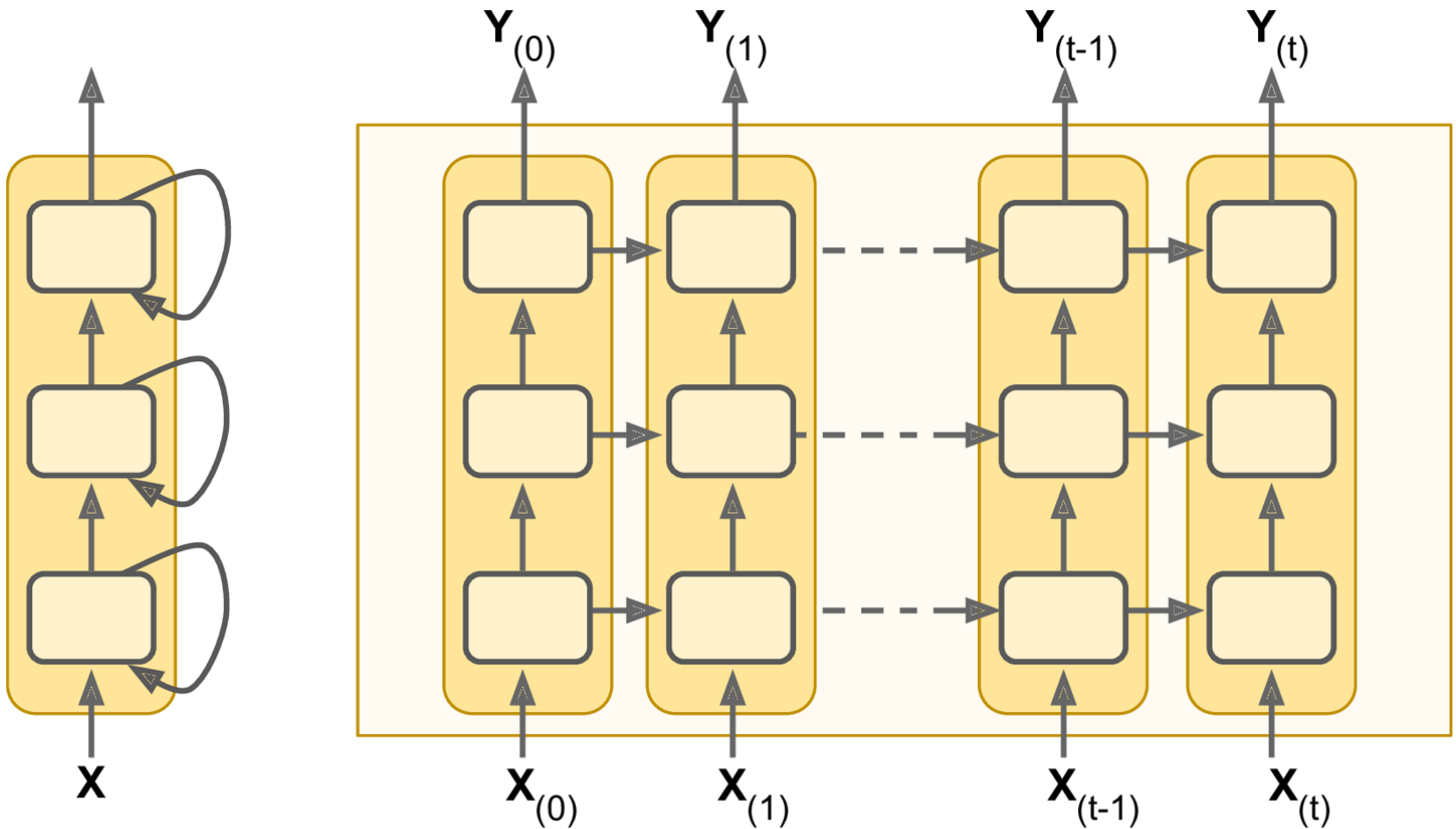
Uses tanh activation $h_t = y_t$




```
optimizer = keras.optimizers.Adam(lr=0.005)
model.compile(loss="mse", optimizer=optimizer)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))

model.evaluate(X_valid, y_valid) # MSE = 0.011
                                 # Dense achieves 0.004
```

4.2 Deep RNNs



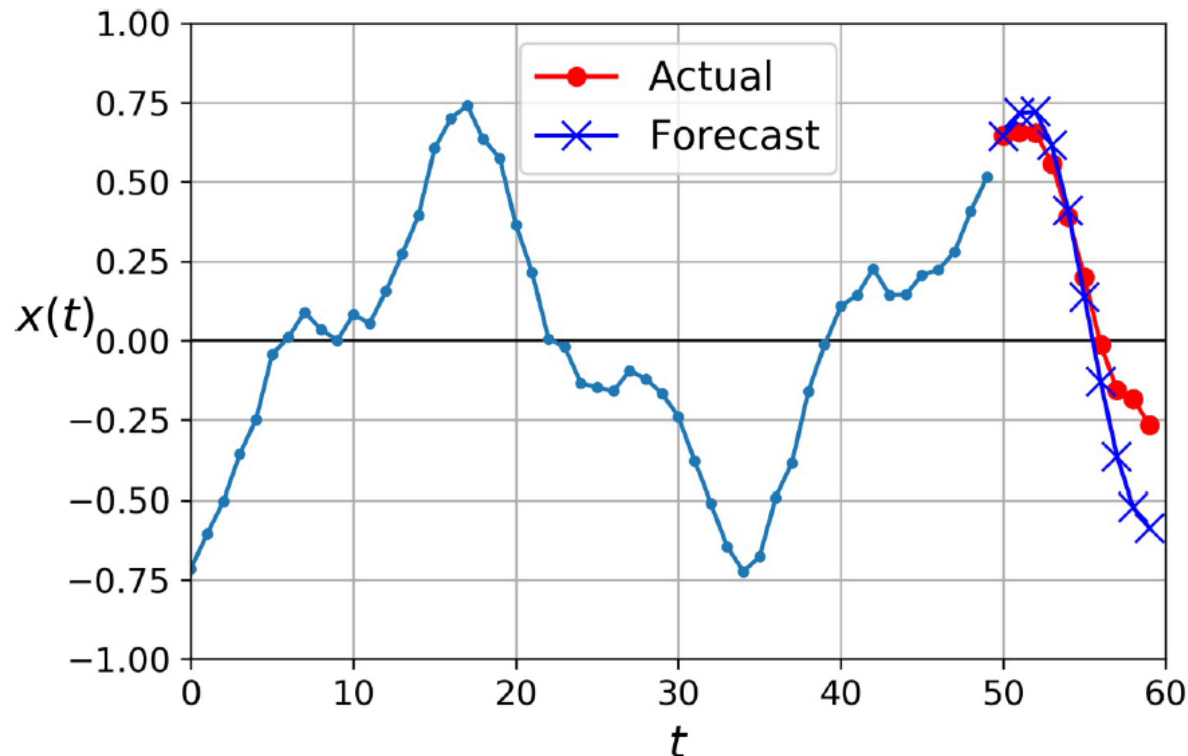
4.2 Deep RNNs

```
# Sequential model of two hidden RNN layers
model = keras.models.Sequential([
      keras.layers.SimpleRNN(20,
            return_sequences=True, # output all steps
            input_shape=[None, 1]),
      keras.layers.SimpleRNN(20),
      keras.layers.Dense(1)
])

# MSE = 0.0026
```

4.3 Forecasting Several Time Steps Ahead

- Can train an RNN to predict all **N next** values at once (sequence-to-vector model).
- The output layer should have N neurons.



4.3 Forecasting Several Time Steps Ahead

```
# Generate 10,000 time series with 10 steps ahead
series = generate_time_series(10000, n_steps + 10)

# Split them 7,000 : 2,000 : 1,000
X_train, y_train = series[:7000, :n_steps],
       series[:7000, -10, 0] # (7000, 50, 1), (7000,10)
X_valid, y_valid = series[7000:9000, :n_steps],
       series[7000:9000, -10, 0]
X_test, y_test = series[9000:, :n_steps],
       series[9000:, -10, 0]
```

4.3 Forecasting Several Time Steps Ahead

```
# Sequential model of two hidden RNN layers
model = keras.models.Sequential([
      keras.layers.SimpleRNN(20,
            return_sequences=True,
            input_shape=[None, 1]),
      keras.layers.SimpleRNN(20),
      keras.layers.Dense(10)
])

# MSE = 0.008
```

Outline

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
    1. Implementing a simple RNN
    2. Deep RNNs
    3. Forecasting Several Time Steps Ahead
5. Handling long sequences
    1. LSTM cell
    2. GRU cell
6. Exercises

5. Handling Long Sequences

- Training long sequences has two major challenges:
  - Unstable gradients
  - Forgetting the first inputs in the sequence
- For the **unstable gradients**:
  - **Does not help**: ReLU activation, batch normalization
  - **Helps**: good parameter initialization, faster optimizers, dropout

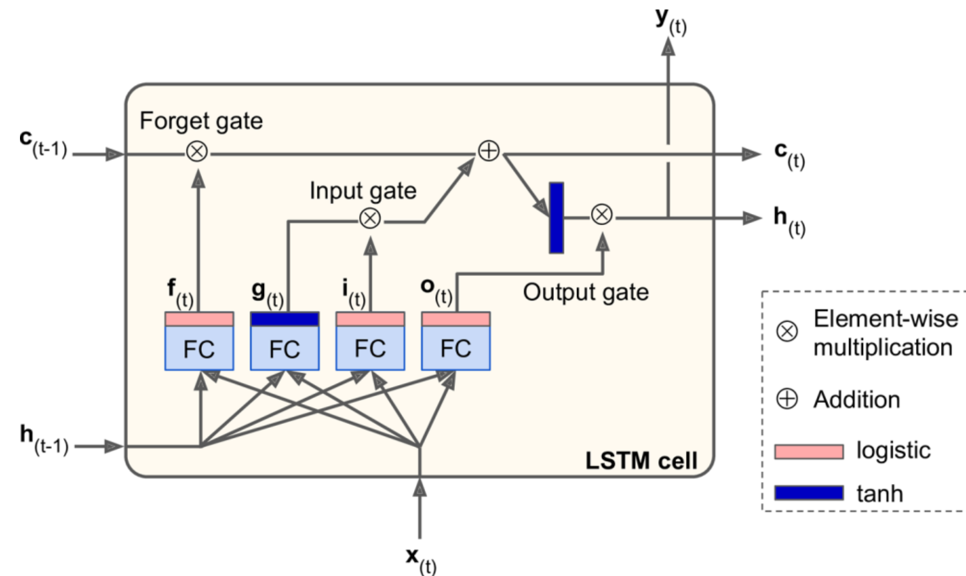
```
model = Sequential()
model.add(layers.SimpleRNN(20, dropout=0.2,
      recurrent_dropout=0.2, input_shape=[None, 1]))
model.add(layers.Dense(1))
```

5. Handling Long Sequences

- To solve the **short-term memory problem**, use
  - **LSTM cell**
  - **GRU cell**
- These cells can be used in place of SimpleRNN

5.1 LSTM Cell

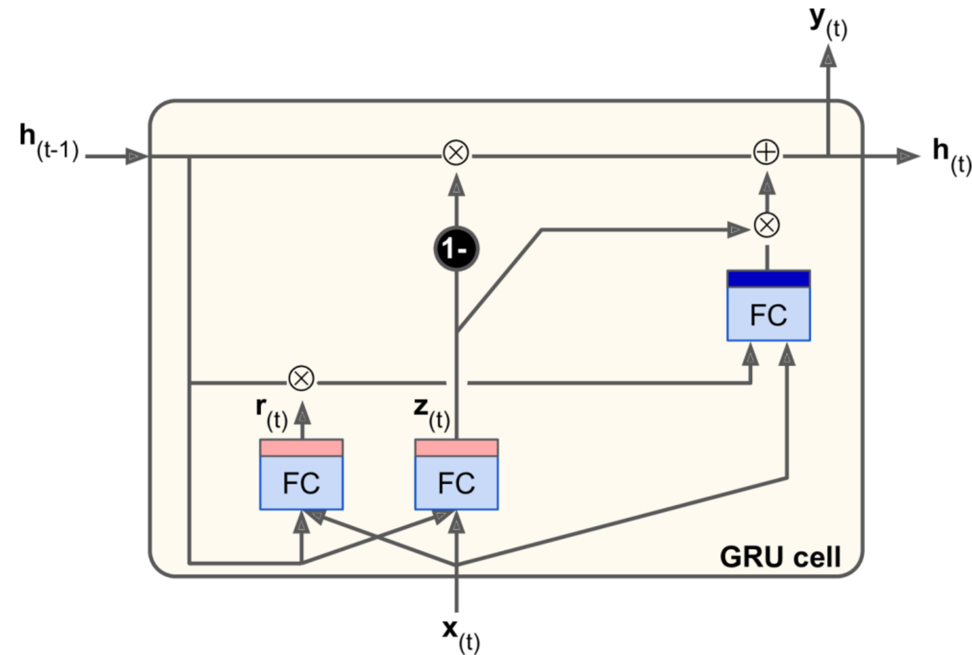
- The **Long Short-Term Memory** (LSTM) cell was proposed in 1997.
- Training converges faster and it **detects long-term dependencies** in the data.
- $\mathbf{h}_{(t)}$ as the short-term state and $\mathbf{c}_{(t)}$ as the long-term state.




```
model.add(LSTM(20))
```

5.2 GRU Cell

- The **Gated Recurrent Unit** (GRU) cell was proposed in 2014.
- **Simplified version** of the LSTM cell, performs just as well.
- A single gate controls the forget gate and the input gate.




```
model.add(GRU(20))
```

6. Exercises

From Chapter 15, solve exercises:

- 1 through 6

Summary

1. Introduction
2. Recurrent neurons and layers
3. Training RNNs
4. Forecasting a time series
    1. Implementing a simple RNN
    2. Deep RNNs
    3. Forecasting Several Time Steps Ahead
5. Handling long sequences
    1. LSTM cell
    2. GRU cell
6. Exercises