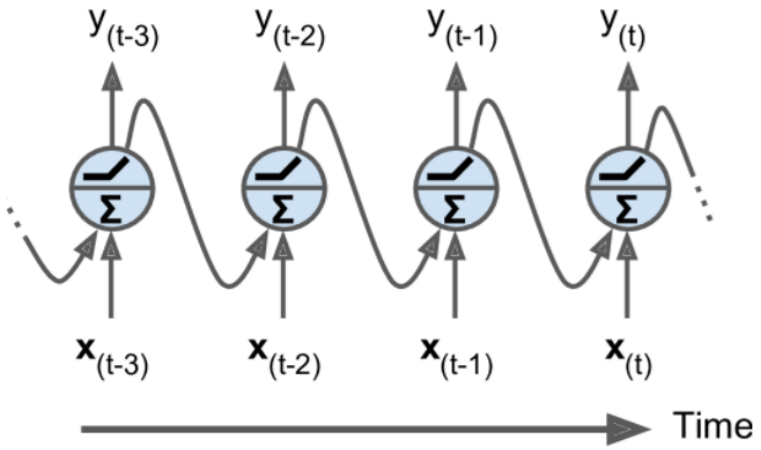
Recurrent Neural Networks (RNN)

- RNN chain multiple neuron layers together into a long sequence of layers
- It can analyze sequences of arbitrary length, like time series or natural language sentences, while all architectures to-date were fixed-length
- These can anticipate the future based on the history so far
- You can use them for:
  - time series analysis
  - automatic language translation (e.g. English to French)
  - speech-to-text
  - composing music
  - sentiment analysis

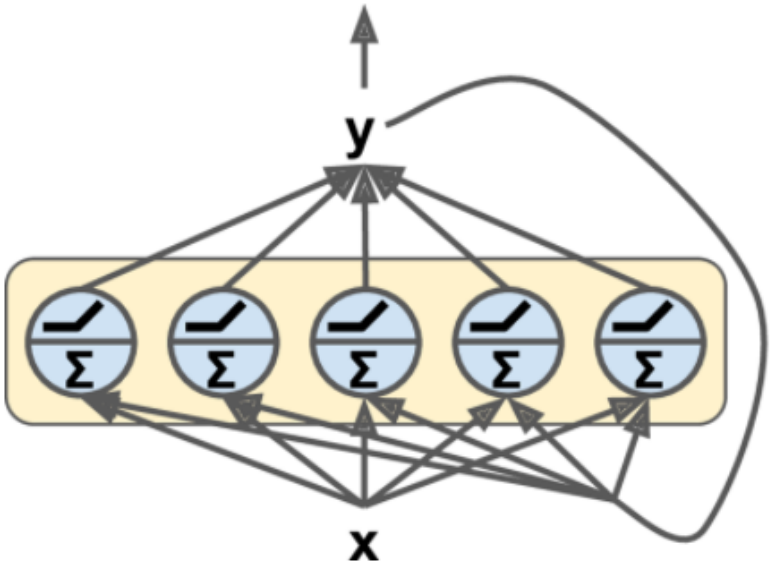
- "Recurrent neurons": compared to the "feed forward neural networks" we covered so far (where activation flows only in one direction), the recurrent neuron loops back to itself

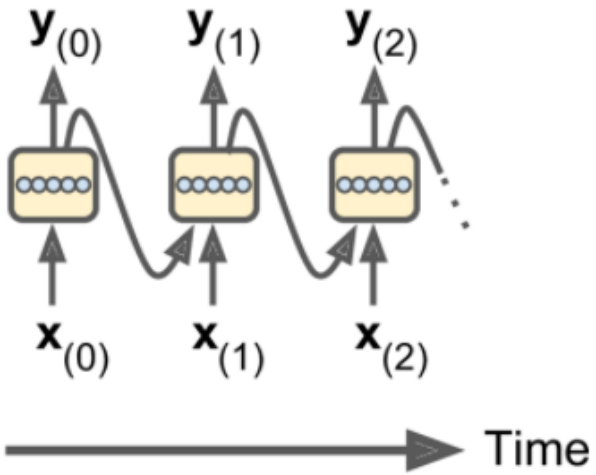
- Recurrent neuron (basic cell):

- Unrolled recurrent neuron (usually $y_{(0)} = 0$):



- A layer of recurrent neurons:





Output for a single instance and for a batch

1 instance

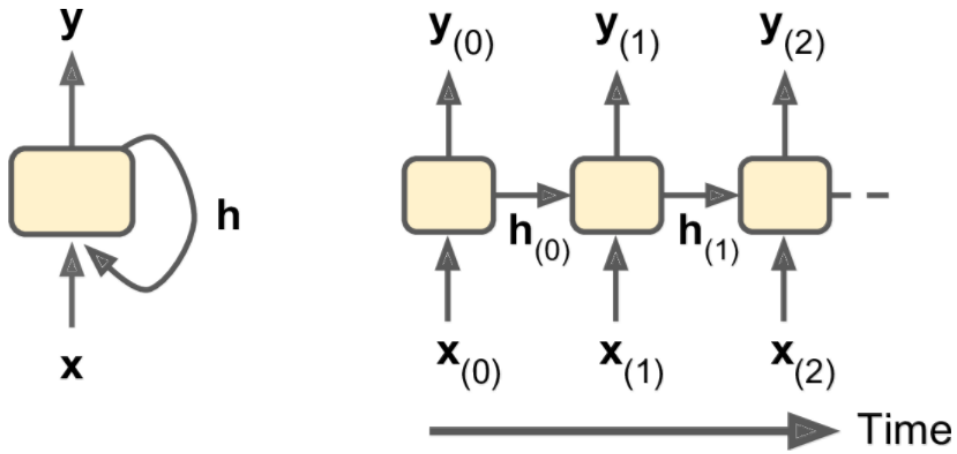
$$\mathbf{y}_{(t)} = \phi\left(\mathbf{W}_x^{\top}\mathbf{x}_{(t)} + \mathbf{W}_y^{\top}\mathbf{y}_{(t-1)} + \mathbf{b}\right)$$

m - # of batches
n_{inp} → X (inputs)
n_{out} → Y (outputs)

$$\mathbf{Y}_{(t)} = \phi\left(\mathbf{X}_{(t)}\mathbf{W}_x + \mathbf{Y}_{(t-1)}\mathbf{W}_y + \mathbf{b}\right) = \phi\left(\left[\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}\right]\mathbf{W} + \mathbf{b}\right) \text{ with } \mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}$$

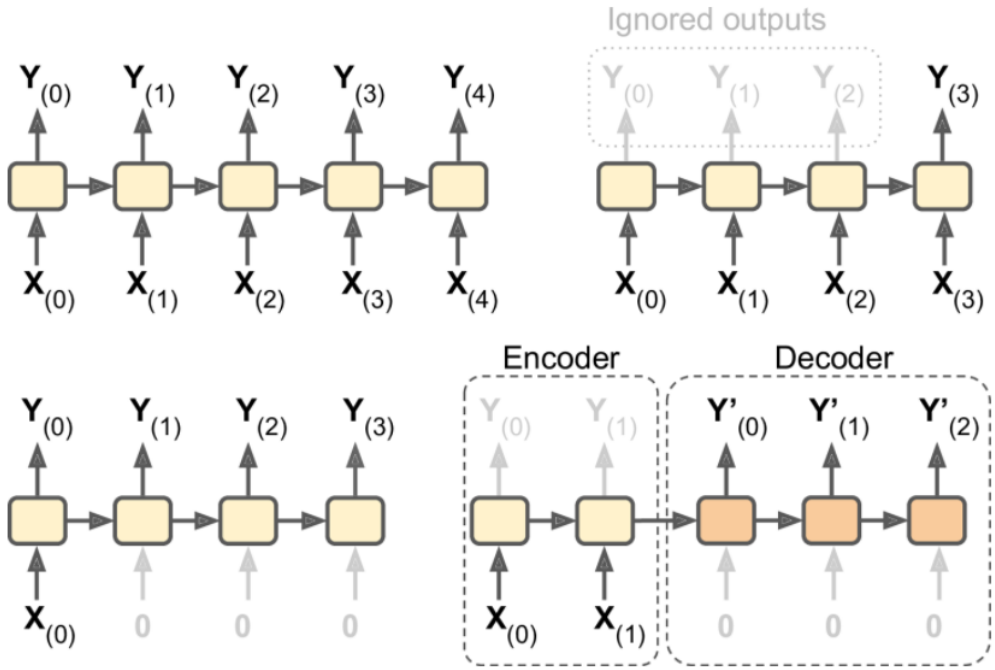
$W_x = (\quad \cdots \quad)$, $n_{inp} \times n_{out}$
neuron 1 neuron n_{out}

- A part of a neural network that preserves some state across time steps is called a memory cell
- Since the output of a recurrent neuron at time step *t* is a function of all the inputs from previous time steps, you could say it has a form of memory

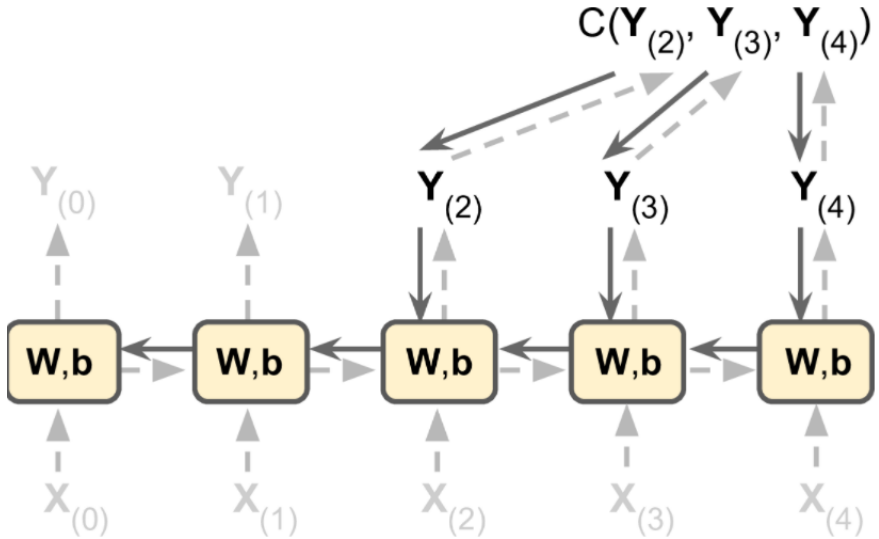


- A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (about 10 steps long)
- A cell's state at time step t, denoted $\boldsymbol{h}(t)$ (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: $\boldsymbol{h}_{(t)} = f(\boldsymbol{h}_{(t-1)}, \boldsymbol{x}_{(t)})$.

- An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs
- You could use the RNN to create:
  - Sequence to sequence (top left diagram)
  - Sequence to vector (top right)
  - Vector to sequence
  - Delayed sequence to sequence



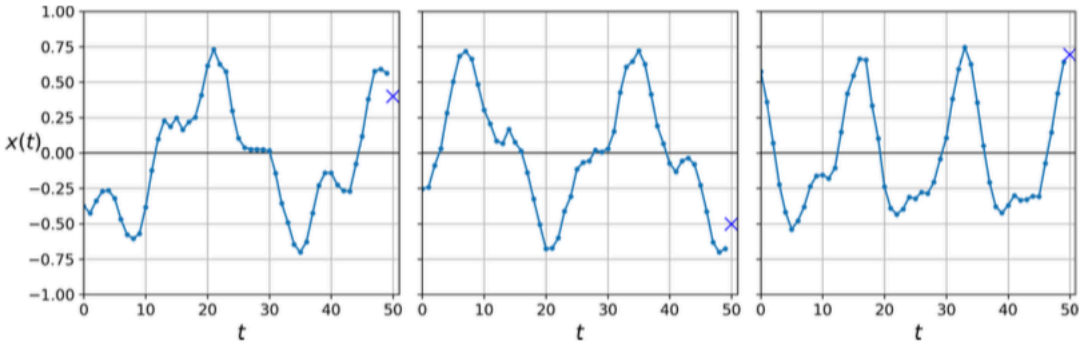
- To train RNNs, unroll it through time and then apply regular backpropagation, which is called backpropagation through time (BPTT)
- Just like in regular backpropagation:
  - forward pass through the unrolled network (dashed arrows)
  - then the output sequence is evaluated using a cost function
  - then the gradients of that cost function are propagated backward through the unrolled network (represented by the solid arrows)
  - then finally the model parameters are updated using the gradients computed during BPTT



Forecasting a Time Series

- Time series - data is a sequence of one or more values per time step. E.g. number of active users per hour on your website, or the daily temperature in your city, or your company's daily stock value.

- Typical task - forecasting



- Time Series above are generated randomly – there are 50 time steps. We generate 10,000 copies and using train, validation and test subsets in 7:2:1 proportion, will forecast the next (51^{st} value of each time series).

Baseline Models

- The simplest - predict the last value in each series. This is called naive forecasting, and it is sometimes surprisingly difficult to outperform. In this case, it gives us a mean squared error of about 0.020 (on the test set)

- Linear Regression – it has 51 parameters, $MSE \approx 0.004$

$$y_{(50)} = \alpha_0 y_{(0)} + \alpha_1 y_{(1)} + \dots$$

ARIMA (Autoregressive Integrated Moving Average) Models

- Some of them require you to first remove the trend and seasonality (to make them stationary)

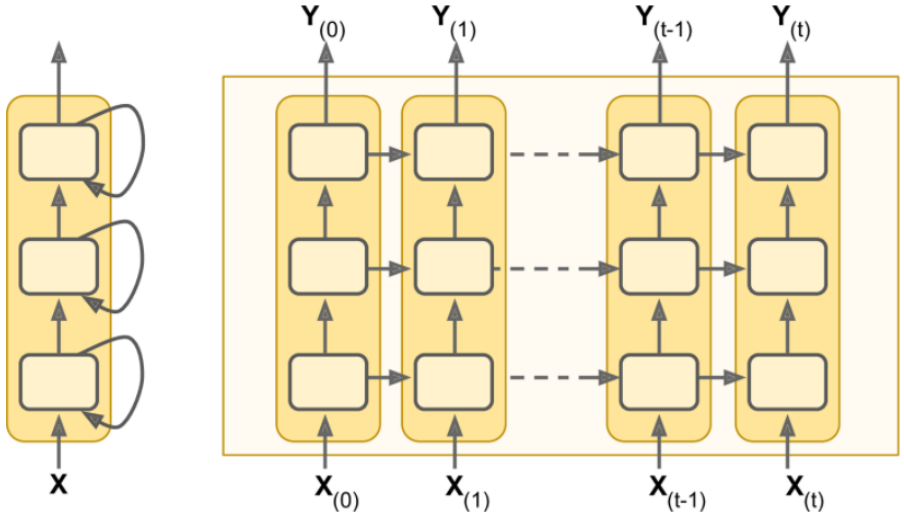
Simple RNN

- Single layer, single neuron – 3 parameters, $MSE \approx 0.014$

$$y_{(t)} = \alpha x_{(t)} + \beta y_{(t-1)} + b$$

Deep RNN

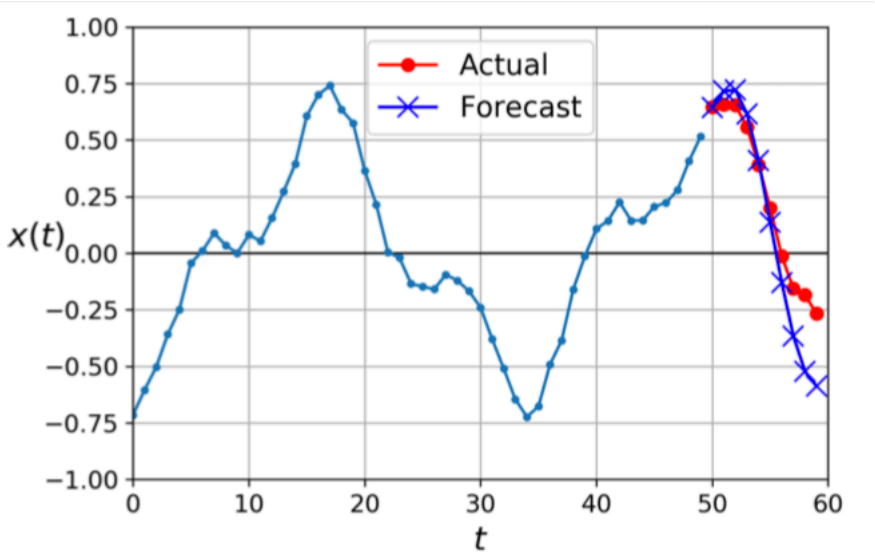
- The RNN cells themselves can have multiple layers




```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])
```

Forecasting Several Time Steps Ahead

- 1st option - use the model we already trained, make it predict the next value, then add that value to the inputs, and use the model again to predict the following value, and so on. We'll do it for 10 steps ahead – MSE=0.008



- Sequence-to-sequence RNN

```
Y = np.empty((10000, n_steps, 10)) # each target is a sequence of 10D vectors
for step_ahead in range(1, 10 + 1):
    Y[:, :, step_ahead - 1] = series[:, step_ahead:step_ahead + n_steps, 0]
Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

- Compute error only at the last time step => a validation MSE of about 0.006, which is 25% better than the previous model.

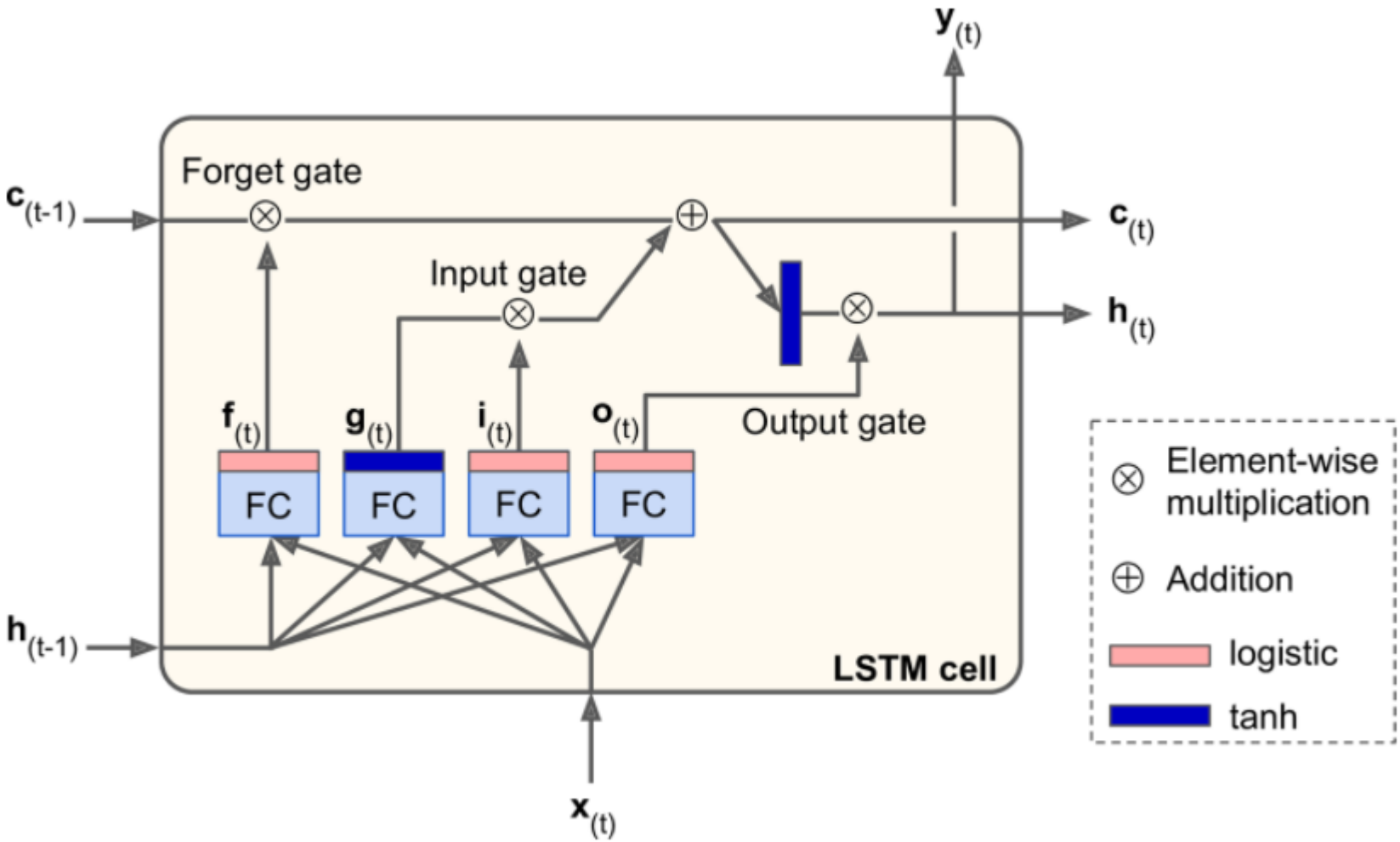
Handling Long Sequences

- Long RNNs can specially suffer from the vanishing/exploding gradients problem and can be very slow to train

- Fighting the Unstable Gradients Problem for basic RNNs
  - Saturated activation functions
  - Layer Normalization

Tackling the Short-Term Memory Problem

- You need a long term memory next to the built-in short term one
- Multiple long term memory cell designs:
  - Long Short-Term Memory (LSTM) cell
  - Gated Recurrent Unit (GRU) cell



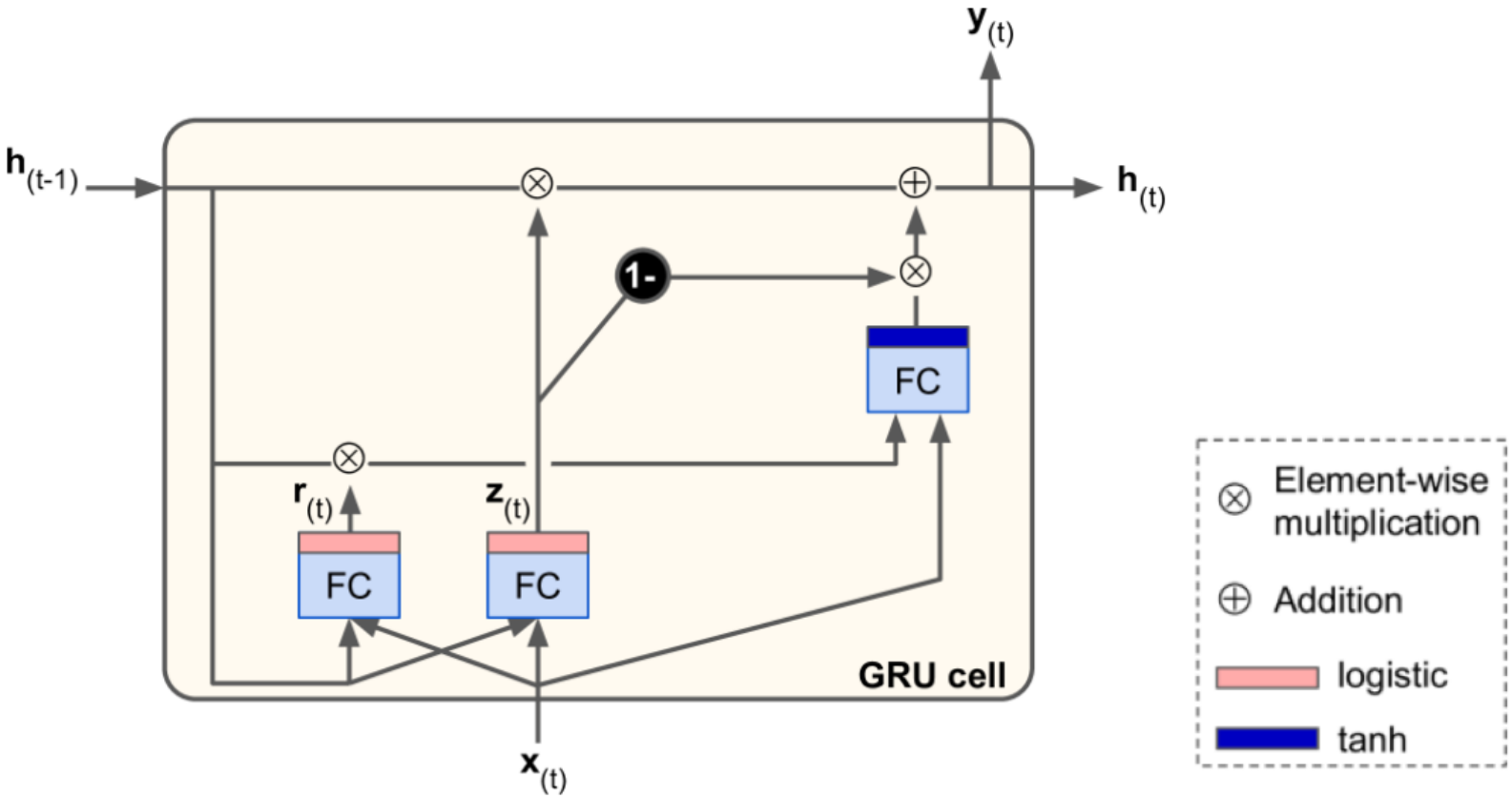
Long-short term memory (LSTM)

Equation 15-3. LSTM computations

$$
\begin{aligned}
\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \\
\mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \\
\mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{ho}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)
\end{aligned}
$$

3 layers are *gate controllers.* They use the logistic activation function => outputs in [0,1]. Their outputs are fed to element-wise multiplication operations, so if they output 0s they close the gate, and if they output 1s they open it.

- *forget gate* (controlled by **f**(*t*)) controls which parts of the long-term state should be erased.
- *input gate* (controlled by **i**(*t*)): which parts of **g**(*t*) should be added to the long-term state.
- Finally, the *output gate* (controlled by **o**(*t*)) controls which parts of the long-term state should be read and output at this time step, both to **h**(*t*) and to **y**(*t*).



Gated Recurrent Unit (GRU)

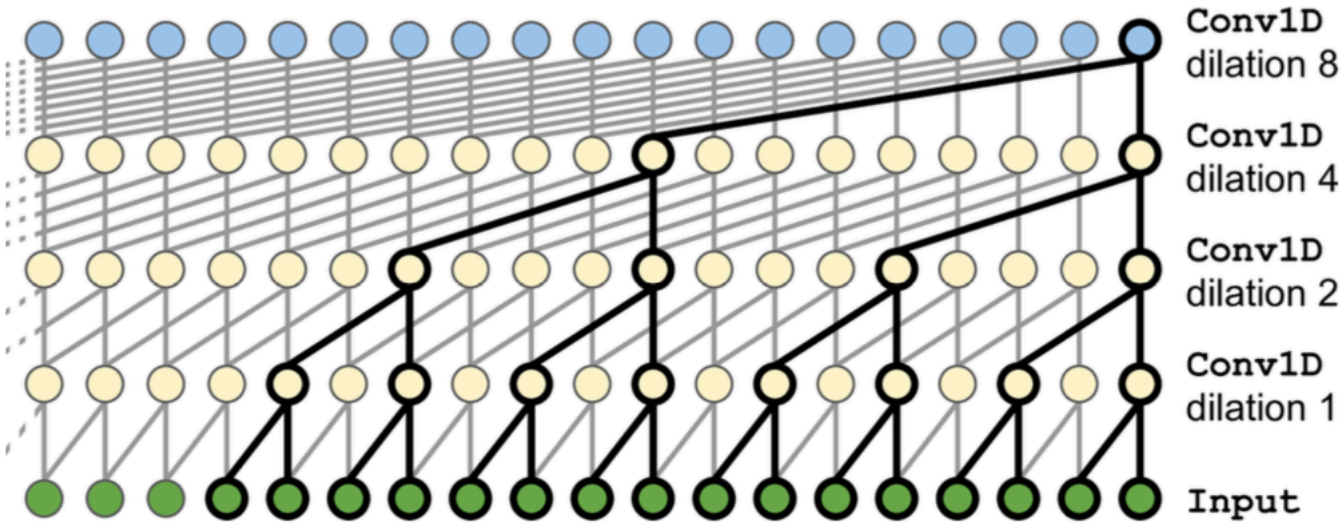
Using 1D convolutional layers to process sequences

- Apply several kernels per sequence and produce a 1D feature map per kernel – each one will learn a short pattern.
- If you apply 10 kernels, the layer's output will consist of ten 1-dim sequences (equivalently one 10-dim sequence).
- 1D convo layer + stride=1 + "same" has the same length as the input sequence. If you use "valid" or stride >1, the len(output) < len(input).
- This actually can help subsequent LSTM/GRU layers to detect longer patterns – see example in the Jupyter notebook, which is the best model so far.

  Note that it's possible to completely replace the recurrent layers by 1D convolutional layers.

WaveNet

- 1D convolutional layers are stacked, doubling the dilation rate at every layer:
  - 1st convo layer – looks at 2 time steps at a time
  - next one - 4 time steps (its receptive field is 4 time steps long) etc.
  - This way, the lower layers learn short-term patterns, while the higher layers learn long-term patterns



- WaveNet - authors stacked 10 convolutional layers with dilation rates of 1, 2, 4, 8, ..., 256, 512
- Then they stacked another 2 groups of 10 identical layers (also with dilation rates 1, 2, 4, 8, ..., 256, 512)
- A single stack of 10 convolutional layers with these dilation rates acts like a super-efficient convolutional layer with a kernel of size 1,024 (except way faster, more powerful, and using significantly fewer parameters
- Last 2 models are best so far in forecasting the example time series
- In the WaveNet paper, the authors achieved state-of-the-art performance on various audio tasks
- This is very impressive - a single second of audio can contain tens of thousands of time steps. Even LSTMs and GRUs cannot handle such long sequences.