Recurrent Neural Network

- In this chapter we will learn about
  - Fundamental concepts in RNNs
  - The main problem RNNs face
  - And the solution to the problems
  - How to implement RNNs
  - Variations of the RNNs
  - Convolution with RNN

Recurrent Neural Network

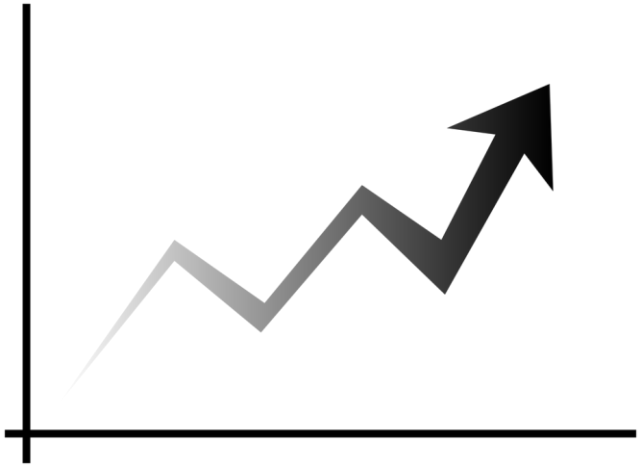
- Predicting the future is what we do all the time
  - Finishing a friend’s sentence
  - Anticipating the smell of coffee at the breakfast or
  - Catching the ball in the field

Recurrent Neural Network

- Unlike all the nets we have discussed so far
  - RNN can work on sequences of arbitrary lengths
  - Rather than on fixed-sized inputs

Recurrent Neural Network – Applications

- RNN can analyze time series data
  - Such as stock prices, and
  - Tell you when to buy or sell

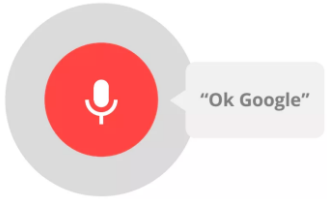


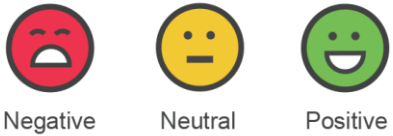
Recurrent Neural Network – Applications

- In autonomous driving systems, RNN can
  - Anticipate car trajectories and
  - Help avoid accidents

Recurrent Neural Network – Applications

- RNN can take sentences, documents, or audio samples as input and
  - Make them extremely useful
  - For natural language processing (NLP) systems such as
    - Automatic translation
    - Speech-to-text or
    - Sentiment analysis





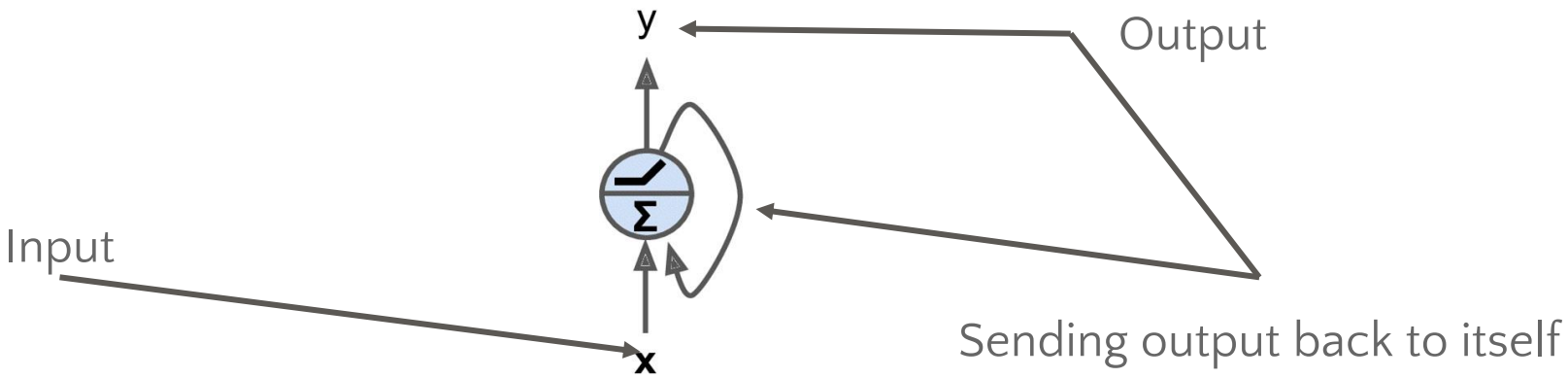
Recurrent Neurons

Recurrent Neurons

- RNN looks much like a feedforward neural network
  - Except it has connections pointing backward

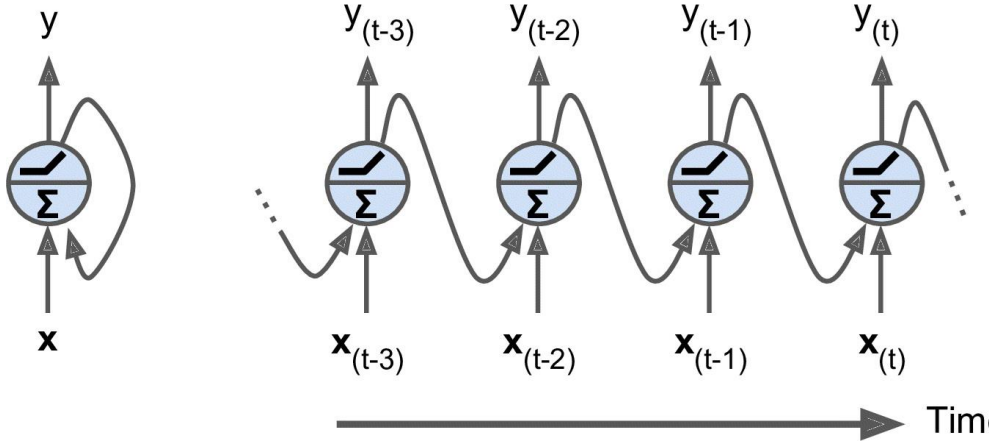
Recurrent Neurons

The simplest possible RNN



Recurrent Neurons

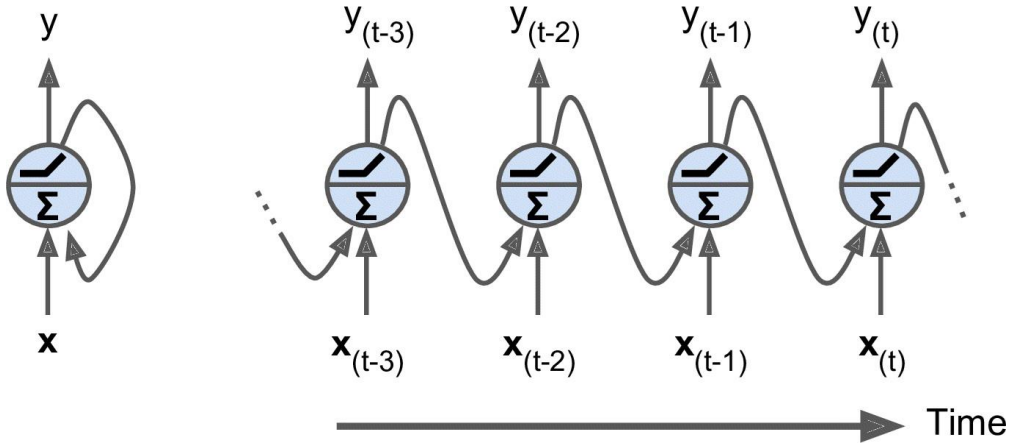
- At each time step **t** (also called a frame)
  - This recurrent neuron receives the inputs $\mathbf{x}_{(t)}$
  - As well as its own output from the previous time step $y_{(t-1)}$



A recurrent neuron (left), unrolled through time (right)

Recurrent Neurons

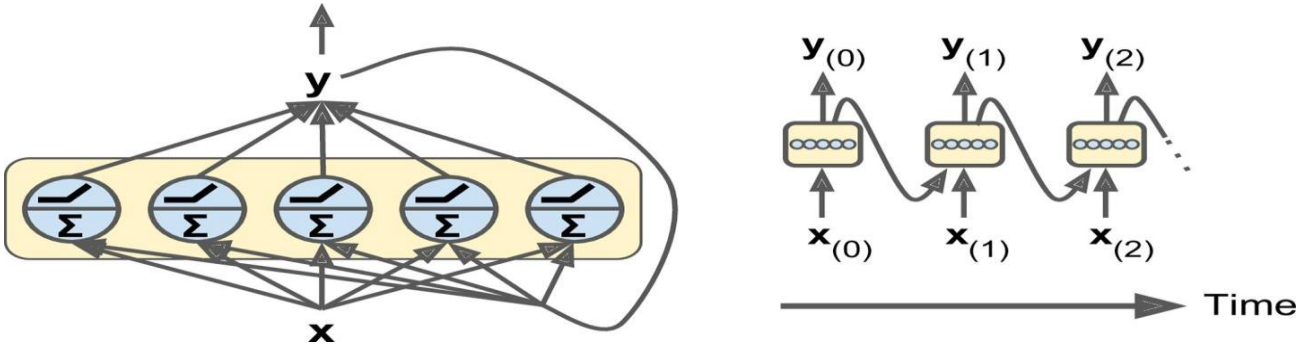
- We can represent this tiny network against the time axis (See below figure)
- This is called ***unrolling the network through time***



A recurrent neuron (left), unrolled through time (right)

Recurrent Neurons

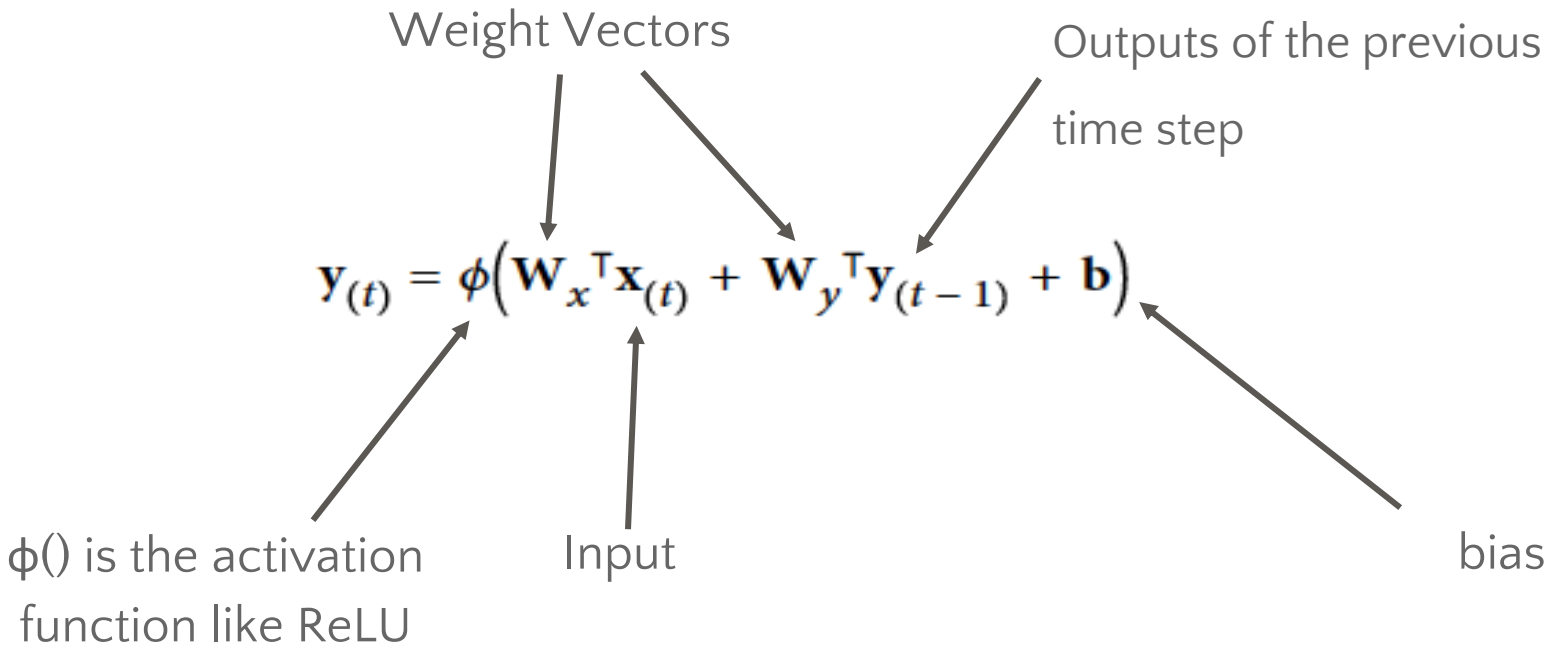
- We can easily create a layer of recurrent neurons
- At each time step t, every neuron receives both the
  - Input vector $x_{(t)}$ and
  - Output vector from the previous time step $y_{(t-1)}$



A layer of recurrent neurons (left), unrolled through time(right)

Recurrent Neurons

Output of a single recurrent neuron for a single instance

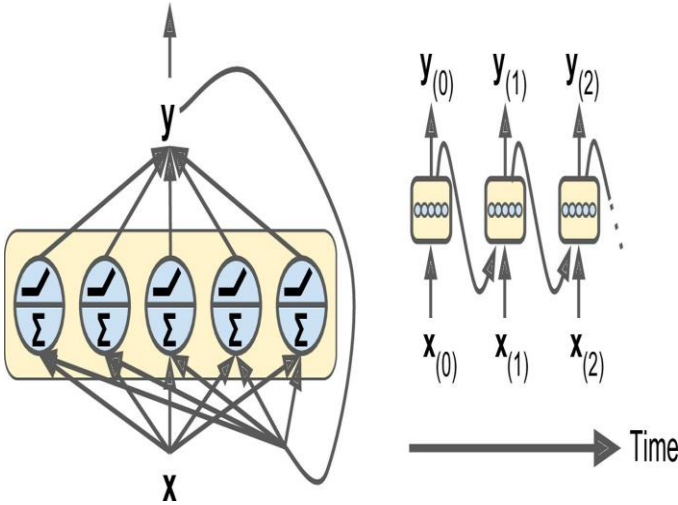


Recurrent Neurons

Vectorized form of previous equation

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

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



Used to compute a whole layer's output in one-shot for a whole mini-batch

Recurrent Neurons

Outputs of a layer of recurrent neurons for all instances in a mini-batch

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

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

- $\mathbf{Y}_{(t)}$ is an m **x** $n_{neurons}$ matrix containing the
  - Layer's outputs at time step t for each instance in the minibatch
  - m is the number of instances in the mini-batch
  - $n_{neurons}$ is the number of neurons

Recurrent Neurons

Outputs of a layer of recurrent neurons for all instances in a mini-batch

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

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

- $\mathbf{X}_{(t)}$ is an $m \times n_{inputs}$ matrix containing the inputs for all instances
  - n_{inputs} is the number of input features

Recurrent Neurons

Outputs of a layer of recurrent neurons for all instances in a mini-batch

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

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

- $\mathbf{W}_x$ is an $n_{inputs} \times n_{neurons}$ matrix containing the connection weights for the inputs of the current time step
- $\mathbf{W}_y$ is an $n_{neurons} \times n_{neurons}$ matrix containing the connection weights for the outputs of the previous time step

Recurrent Neurons

Outputs of a layer of recurrent neurons for all instances in a mini-batch

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

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

- The weight matrices $\mathbf{W}_x$ and $\mathbf{W}_y$ are often concatenated into a single weight matrix $\mathbf{W}$ of shape $(n_{inputs} + n_{neurons}) \times n_{neurons}$
- $\mathbf{b}$ is a vector of size $n_{neurons}$ containing each neuron's bias term

Memory Cells

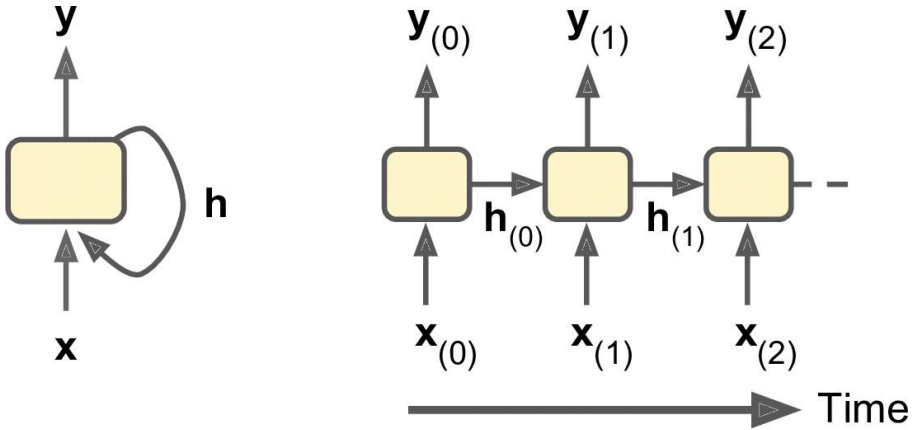
- Since the output of a recurrent neuron at time step t is a
  - Function of all the inputs from previous time steps
  - We can say that it has a form of ***memory***
- A part of a neural network that
  - Preserves some state across time steps is called a **memory cell**

Memory Cells

- In general a cell's state at time step t, denoted $h_{(t)}$ is a
  - Function of some inputs at that time step and
  - Its state at the previous time step $h_{(t)} = f(h_{(t-1)}, x_{(t)})$
- Its output at time step t, denoted $y_{(t)}$ is also a
  - Function of the previous state and the current inputs

Memory Cells

- In the case of basic cells we have discussed so far
  - The output is simply equal to the state
  - But in more complex cells this is not always the case

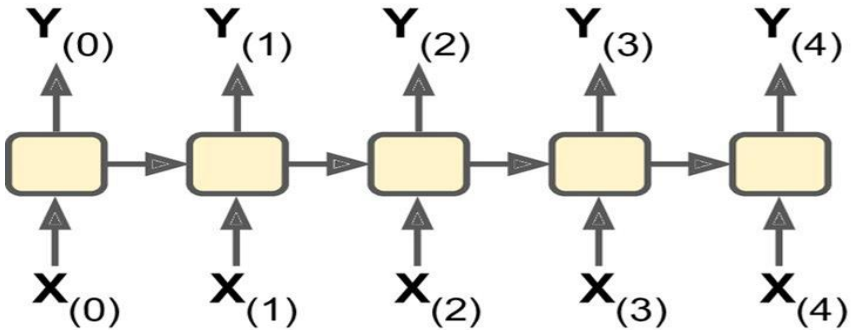


A cell's hidden state and its output may be different

Input and Output Sequences

Sequence-to-sequence Network

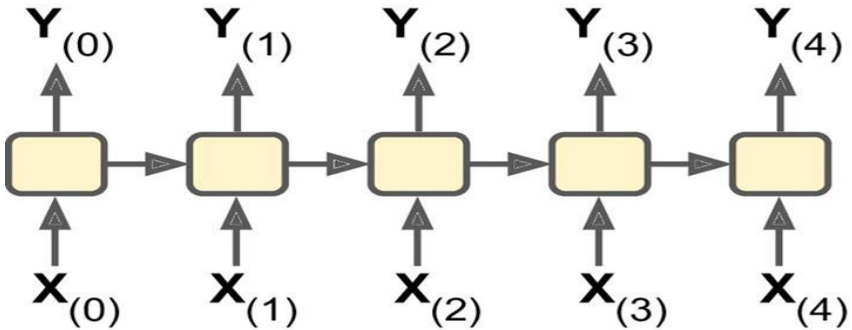
- An RNN can simultaneously take a
  - Sequence of inputs and
  - Produce a sequence of outputs



Input and Output Sequences

Sequence-to-sequence Network

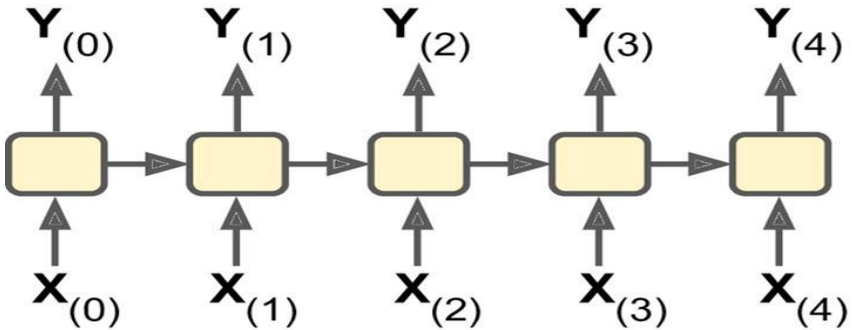
- This type of network is useful for predicting time series
  - Such as stock prices



Input and Output Sequences

Sequence-to-sequence Network

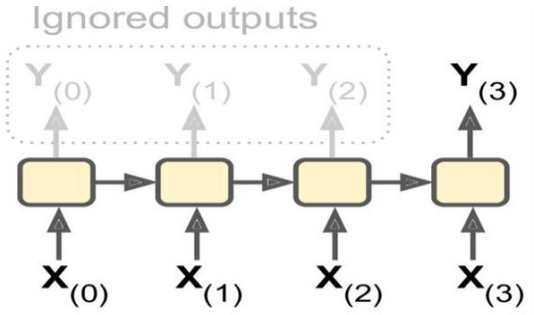
- We feed it the prices over the last N days and
  - It must output the prices shifted by one day into the future
  - i.e., from N – 1 days ago to tomorrow



Input and Output Sequences

Sequence-to-vector Network

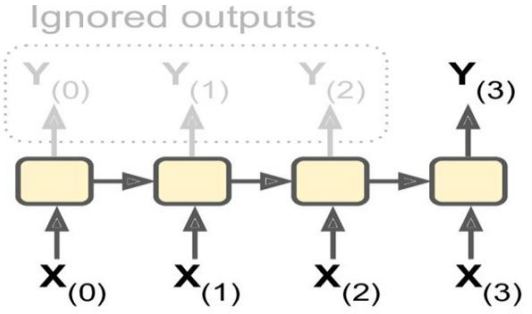
- Alternatively we could feed the network a sequence of inputs and
  - Ignore all outputs except for the last one



Input and Output Sequences

Sequence-to-vector Network

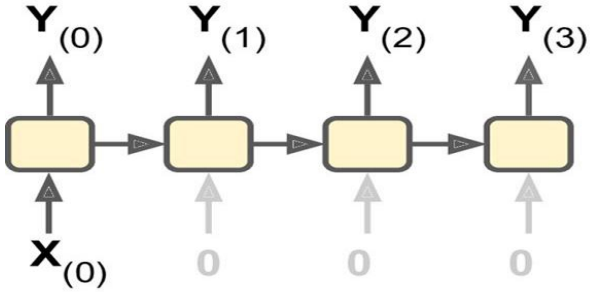
- We can feed this network a sequence of words
  - Corresponding to a movie review and
  - The network would output a sentiment score
  - e.g., from –1 [hate] to +1 [love]



Input and Output Sequences

Vector-to-sequence Network

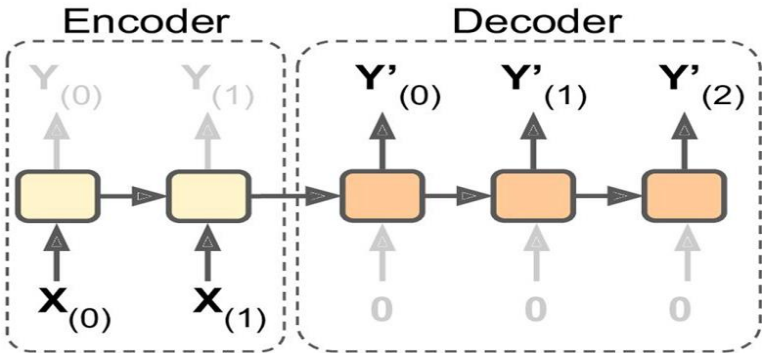
- We could feed the network a single input at the first time step and
  - Zeros for all other time steps and
  - Let it output a sequence
- For example, the input could be an image and the
  - Output could be a caption for the image



Input and Output Sequences

Encoder-Decoder

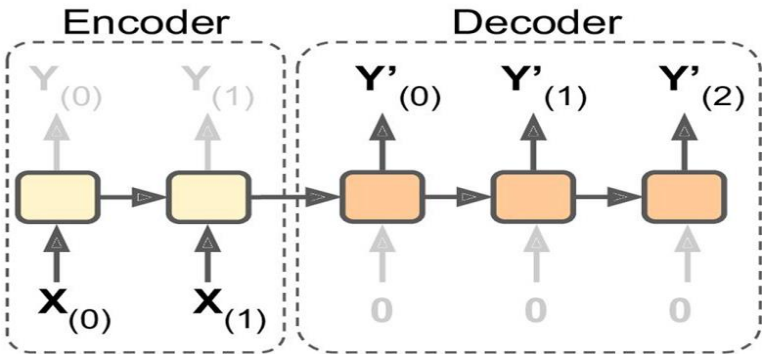
- In this network, we have
  - sequence-to-vector network, called **an encoder** followed by
  - vector-to-sequence network, called **a decoder**



Input and Output Sequences

Encoder-Decoder

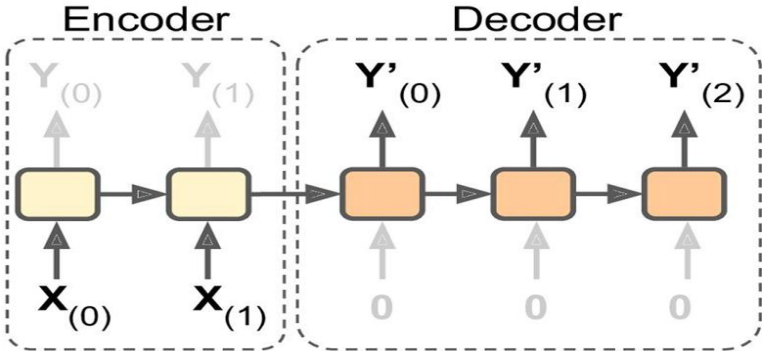
- This can be used for translating a sentence
  - From one language to another



Input and Output Sequences

Encoder-Decoder

- We feed the network sentence in one language
  - The encoder converts this sentence into single vector representation
  - Then the decoder decodes this vector into a sentence in another language



Input and Output Sequences

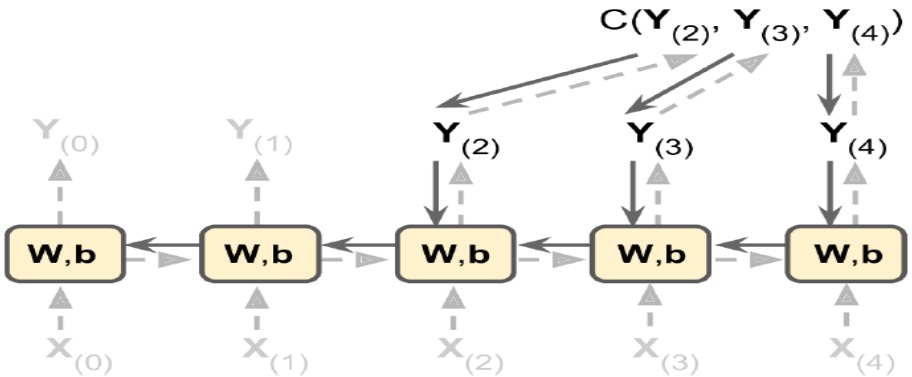
Encoder-Decoder

- This two step model works much better than
  - Trying to translate on the fly with a
  - Single sequence-to-sequence RNN
- Since the last words of a sentence can affect the
  - First words of the translation
  - So we need to wait until we know the whole sentence

Training RNNs

Training RNNs

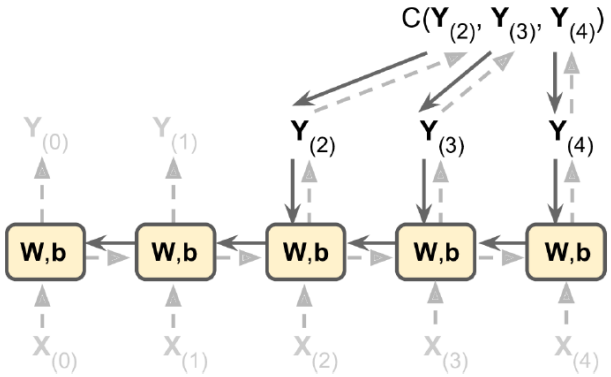
- To train an RNN, the trick is to unroll it through time and then simply use regular backpropagation. This strategy is called **backpropagation through time (BPTT)**.



- Dotted line shows forward pass.
- Solid line shows backward pass.
  //or propagation of cost in the backward pass

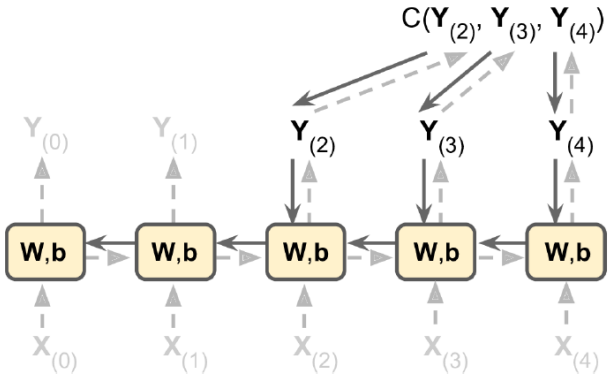
Training RNNs

- There is a first forward pass through the unrolled network (represented by the dashed arrows)
- Then the output sequence is evaluated using a cost function C(Y(0), Y(1), ...Y(T)) (where T is the max time step)



Training RNNs

- The gradients of that cost function are then propagated backward through the unrolled network
- Finally the model parameters are updated using the gradients computed during BPTT



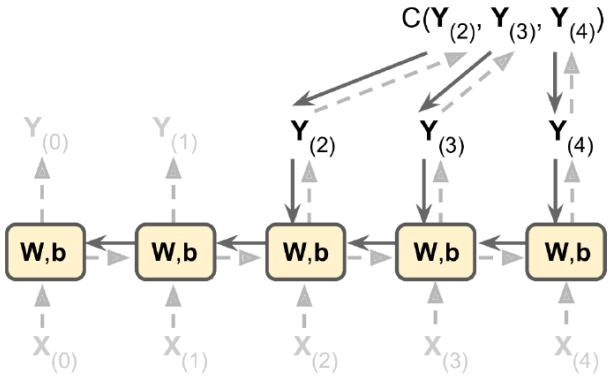
3rd type of backward arrows: from neuron-3 to nuron-2, neuron-2 to nuron-1

Ist type of backward arrows: C to Y2, Y3, Y4

2nd type of backward arrows: Y2, Y3, Y4 to the respective neuron

Training RNNs

- Note that the gradients flow backward through all the outputs used by the cost function, not just through the final output
- Since the same parameters W and b are used at each time step, backpropagation will do the right thing and sum over all time steps



Basic RNNs

Forecasting a Time Series

- Suppose you are studying
  - the number of active users per hour on your website,
  - or the daily temperature in your city,
  - or a company's financial health, measured quarterly using multiple metrics

Forecasting a Time Series

- Suppose you are studying
  - the number of active users per hour on your website,
  - or the daily temperature in your city,
  - or a company's financial health, measured quarterly using multiple metrics
- Here, the data is a sequence of one or more values per time step
- This is called a **time series**

Forecasting a Time Series – Types

Forecasting a Time Series – Univariate

The term "univariate time series" refers to a time series that consists of single (scalar) observations recorded sequentially over equal time increments

Examples:

- The **number** of active users per hour on your website,
- Or the daily **temperature** in your city

Forecasting a Time Series – Multivariate

Multivariate time series model is an extension of the univariate case and involves two or more input variables

Examples:

- A company's financial health, measured quarterly using multiple metrics

Forecasting a Time Series

What is Forecasting?
Predicting future values.

Forecasting a Time Series

Another common task is to fill in the blanks: to predict missing values from the past.

This is called **imputation**

Forecasting a Time Series

Trend is a general systematic linear or (most often) nonlinear component that changes over time and does not repeat

- For example, if you are studying the number of active users on your website, and it is growing by 10% every month

Forecasting a Time Series

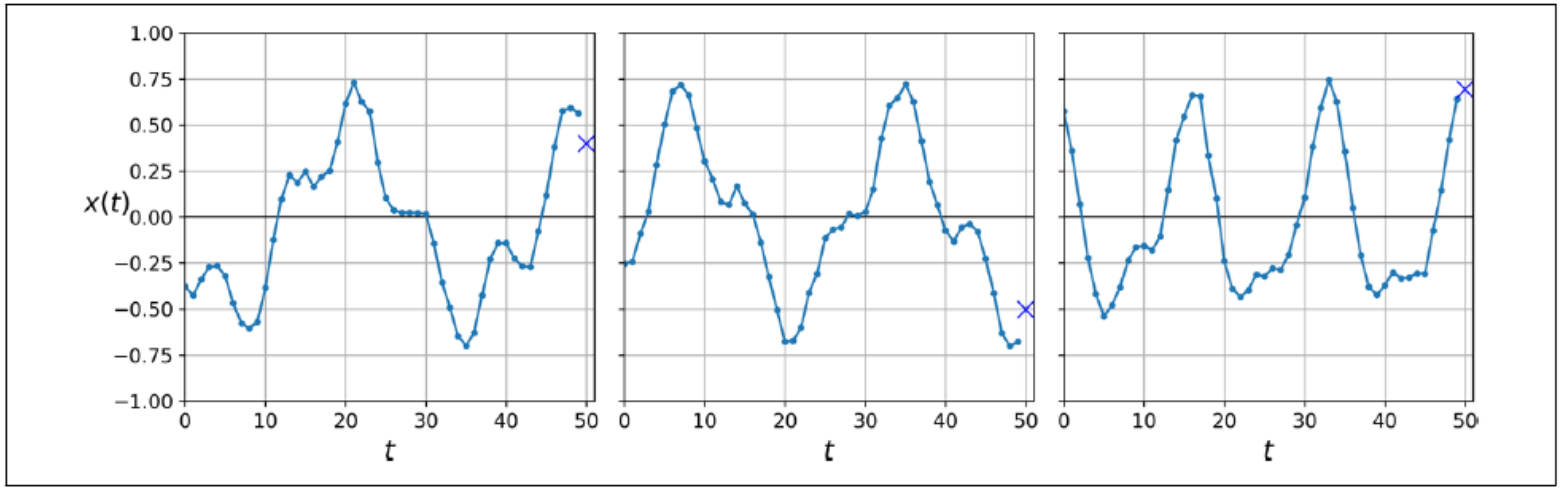
Seasonality is a general systematic linear or (most often) nonlinear component that changes over time and does repeat

- For example, if you are trying to predict the amount of sunscreen lotion sold every month, you will probably observe strong seasonality: since it sells well every summer, a similar pattern will be repeated every year

Forecasting a Time Series

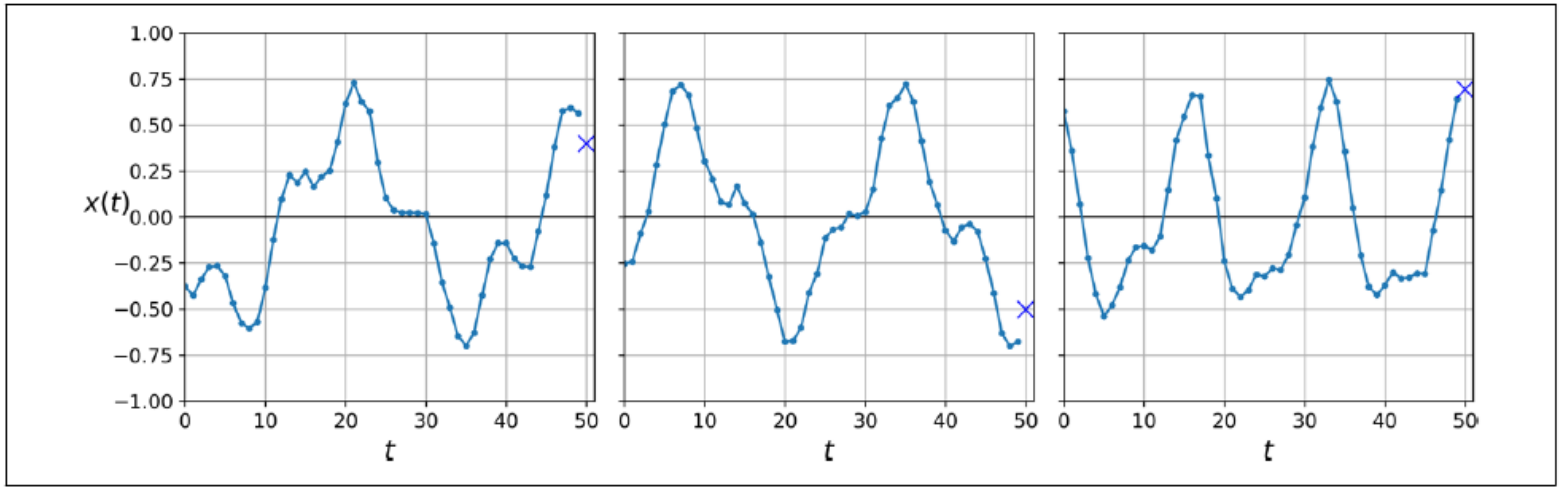
When using traditional models, we have to remove the trend and seasonality but with RNN we don't need to worry about that.

Forecasting a Time Series



The goal is to forecast value at next time step (represented by X) for each of these 50 steps long ____?____ time series. **Is it Univariate or multivariate?**

Forecasting a Time Series



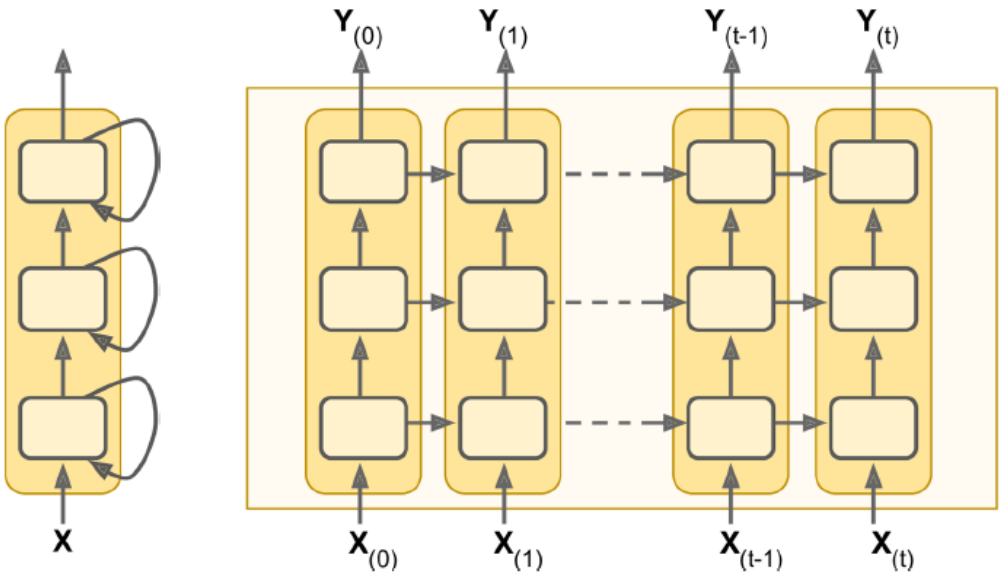
The goal is to forecast value at next time step (represented by X) for each of these 50 steps long **univariate** time series.

Deep RNNs

Deep RNNs

It is quite common to stack multiple layers of cells, as shown below.

This gives you a **deep RNN**



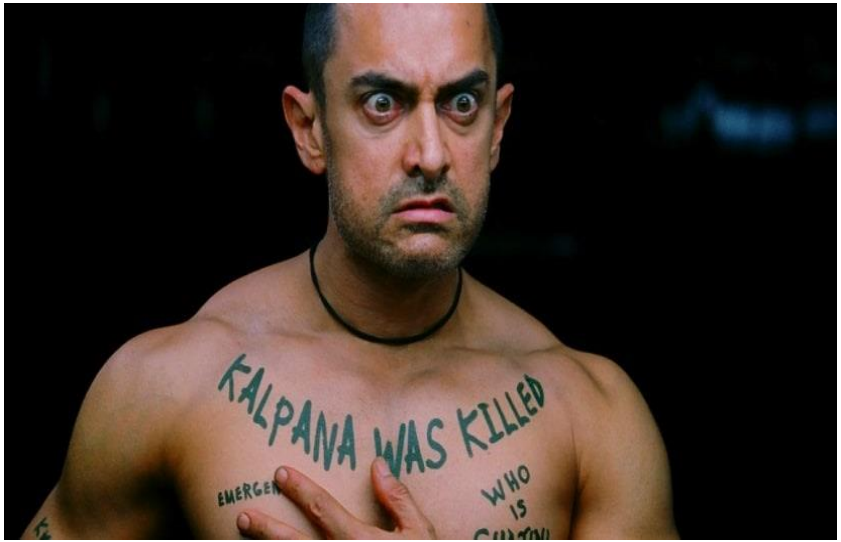
Tackling the Short-Term Memory Problem

Tackling the Short-Term Memory Problem

- Some information is lost at each time step in the RNN
- So after a while, the RNN's state contains virtually no trace of the first inputs

Tackling the Short-Term Memory Problem

This can be a showstopper



Tackling the Short-Term Memory Problem

- To tackle this problem, various types of cells with long-term memory have been introduced
- They have proven so successful that the basic cells are not used much anymore
- LSTM is one of these cells

LSTM

LSTM Cell

- The Long Short-Term Memory (LSTM) cell was proposed in 1997 by **Sepp Hochreiter** and **Jürgen Schmidhuber** and was improved by Alex Graves, Haşim Sak, Wojciech Zaremba, and many more

Sepp Hochreiter

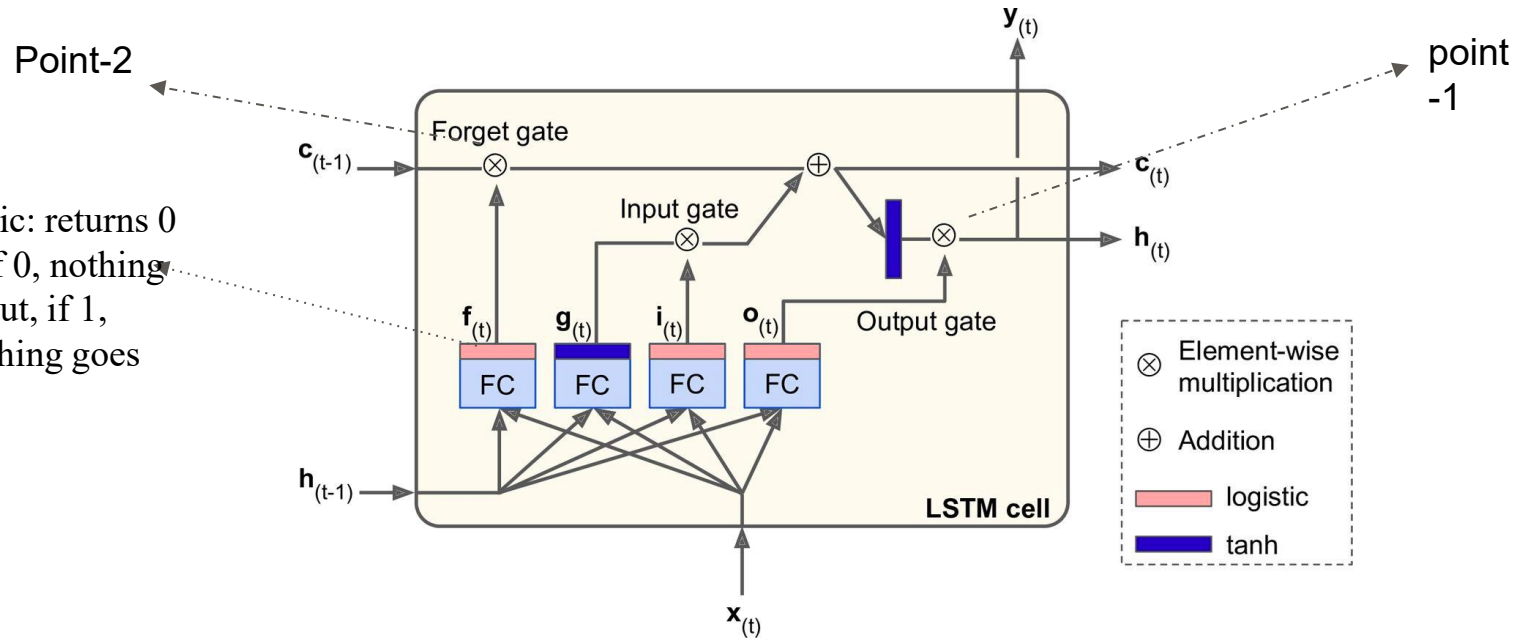
Jürgen Schmidhuber

LSTM Cell

"Can be used as a black box - behaves like basic cell"

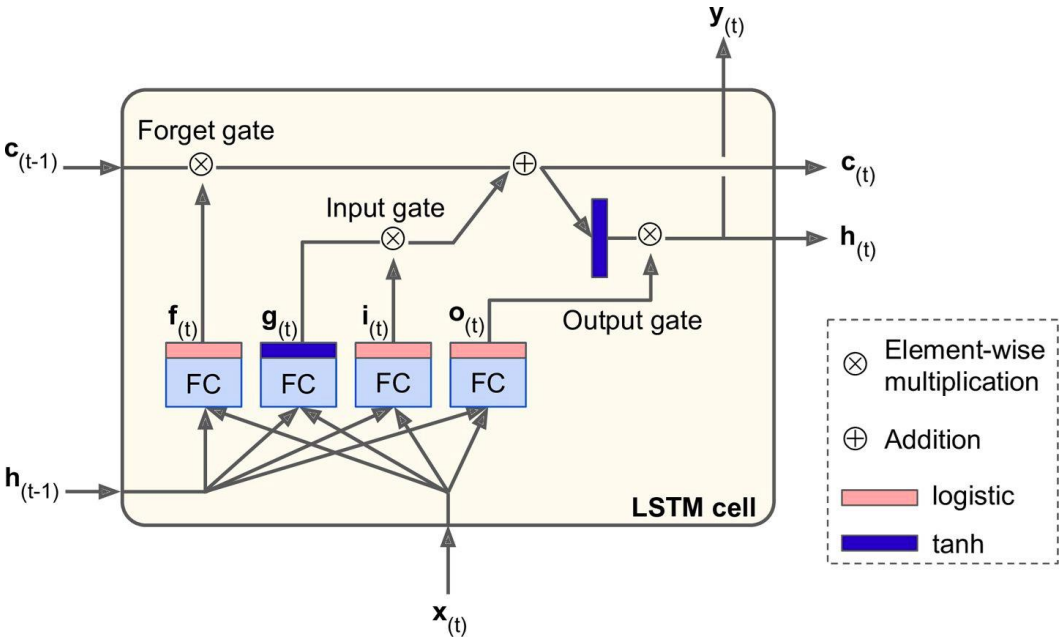
- **Except**
  - It will perform much better
  - Training will converge faster
  - And it will detect long-term dependencies in the data

Architecture of LSTM Cell



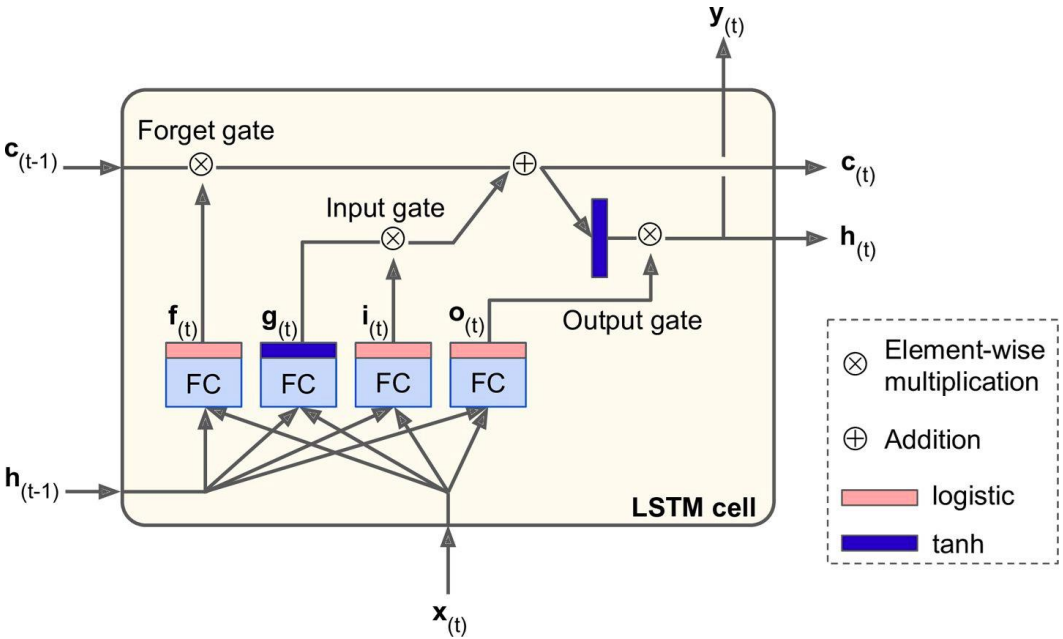
The architecture of a basic LSTM cell

Architecture of LSTM Cell



- The **LSTM cell** looks exactly like a regular cell, except that its state is split in two vectors: $h_{(t)}$ and $c_{(t)}$, here "c" stands for "cell"

Architecture of LSTM Cell



- We can think of $h_{(t)}$ as the short-term state and $c_{(t)}$ as the long-term state

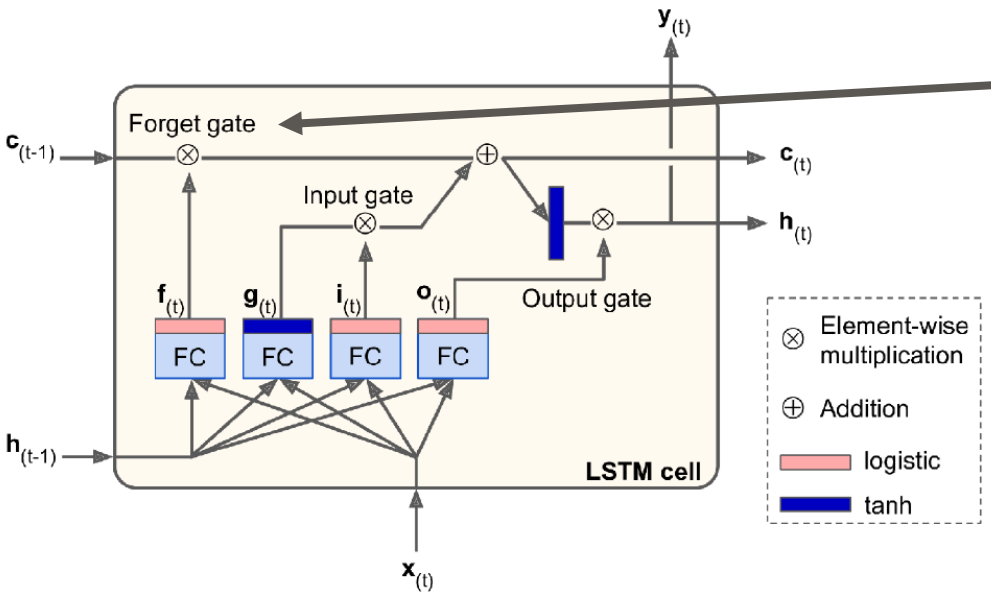
Architecture of LSTM Cell

Understanding the LSTM cell structure

- The key idea is that the network **can learn**
  - What to store in the long-term state,
  - What to throw away,
  - And what to read from it

Architecture of LSTM Cell

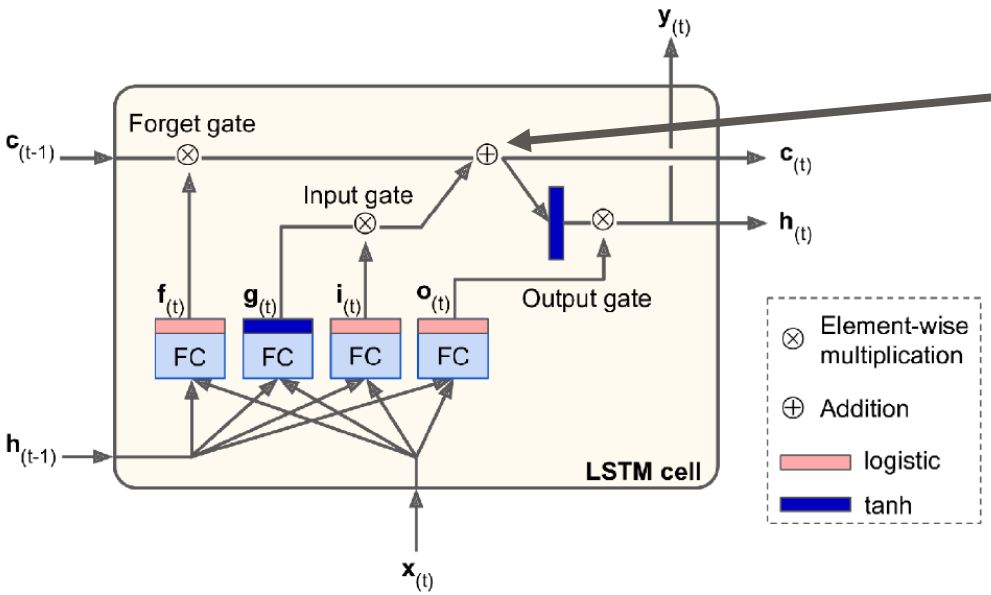
Understanding the LSTM cell structure



As the long-term state $\mathbf{c}_{(t-1)}$ traverses the network from left to right, it first goes through a **forget gate**, dropping some memories

Architecture of LSTM Cell

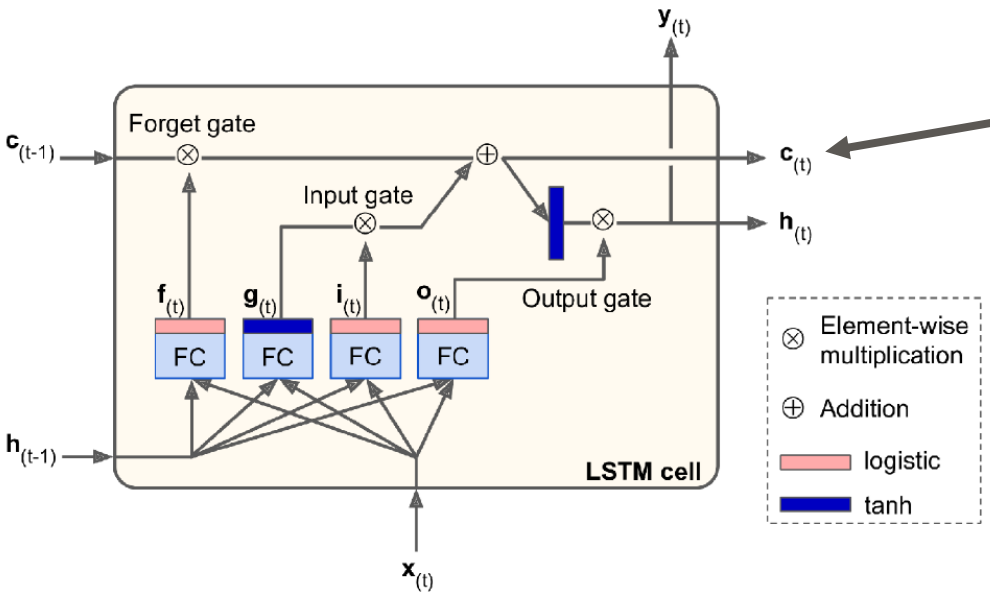
Understanding the LSTM cell structure



Then it adds some new memories via the addition operation, which adds memories that were selected by an input gate

Architecture of LSTM Cell

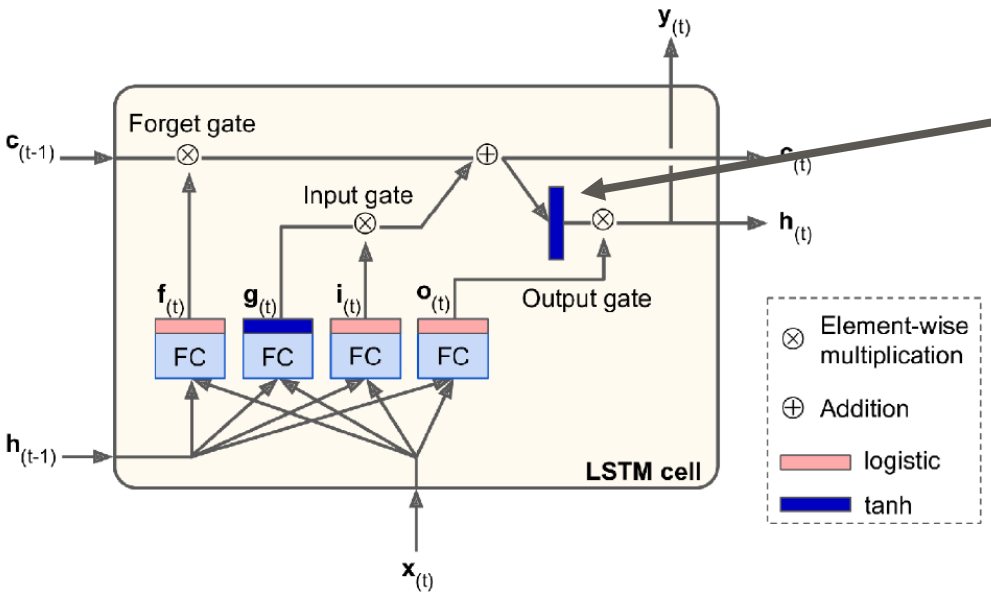
Understanding the LSTM cell structure



The result $\mathbf{c}_{(t)}$ is sent straight out, without any further transformation. So, at each time step, some memories are dropped and some memories are added

Architecture of LSTM Cell

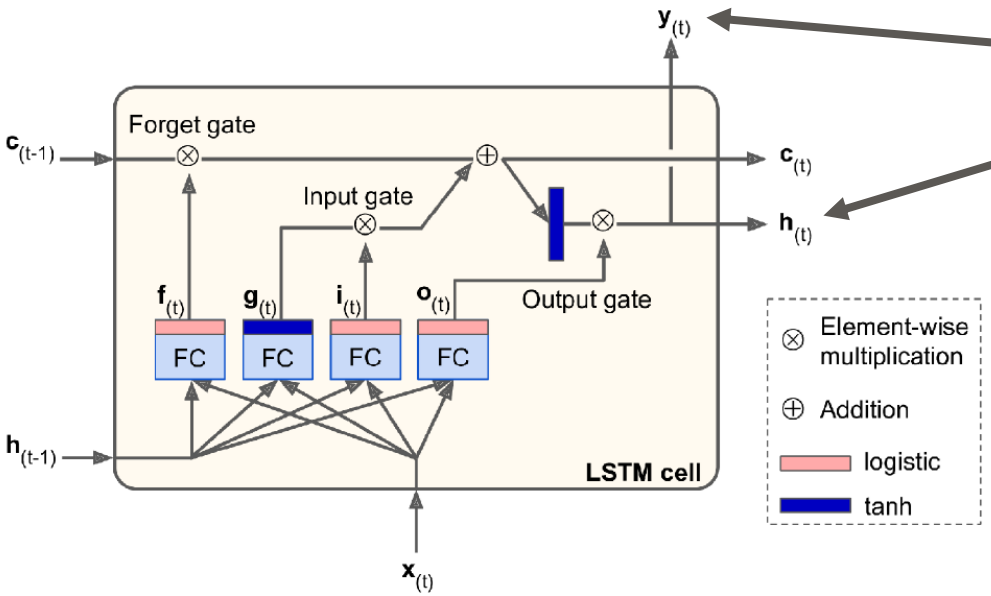
Understanding the LSTM cell structure



Moreover, after the addition operation, the long term state is copied and passed through the **tanh** function, and then the result is filtered by the output gate.

Architecture of LSTM Cell

Understanding the LSTM cell structure



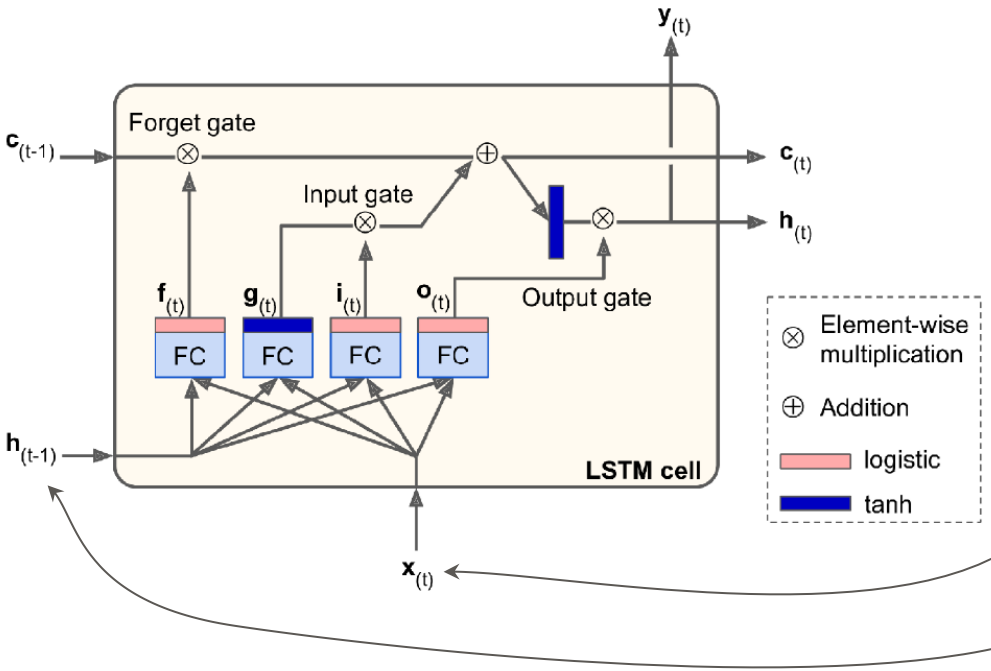
This produces the short-term state $\mathbf{h}_{(t)}$, which is equal to the cell's output for this time step $\mathbf{y}_{(t)}$

Architecture of LSTM Cell

Now let’s look at where new memories come from and how the gates work

Architecture of LSTM Cell

Understanding the LSTM cell structure



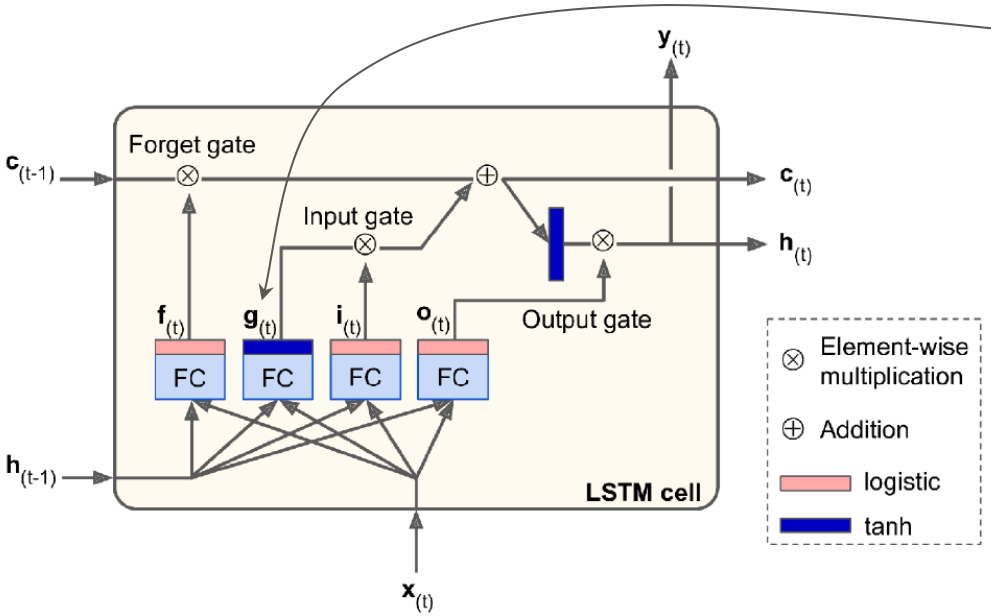
$\mathbf{x}_{(t)}$ = current input vector

$\mathbf{h}_{(t-1)}$ = previous short-term state

They are fed to four different fully connected layers

Architecture of LSTM Cell

Understanding the LSTM cell structure

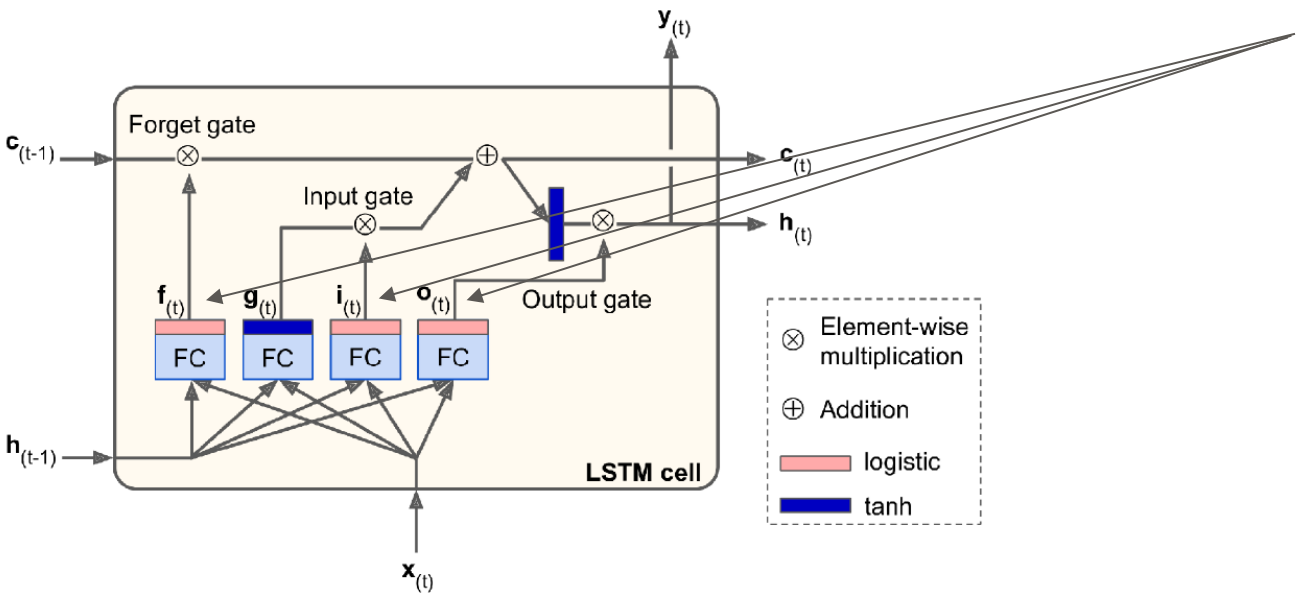


This is the main layer which analyzes $\mathbf{x}_{(t)}$ and $\mathbf{h}_{(t-1)}$

This layer's output is partially stored in the long-term state

Architecture of LSTM Cell

Understanding the LSTM cell structure



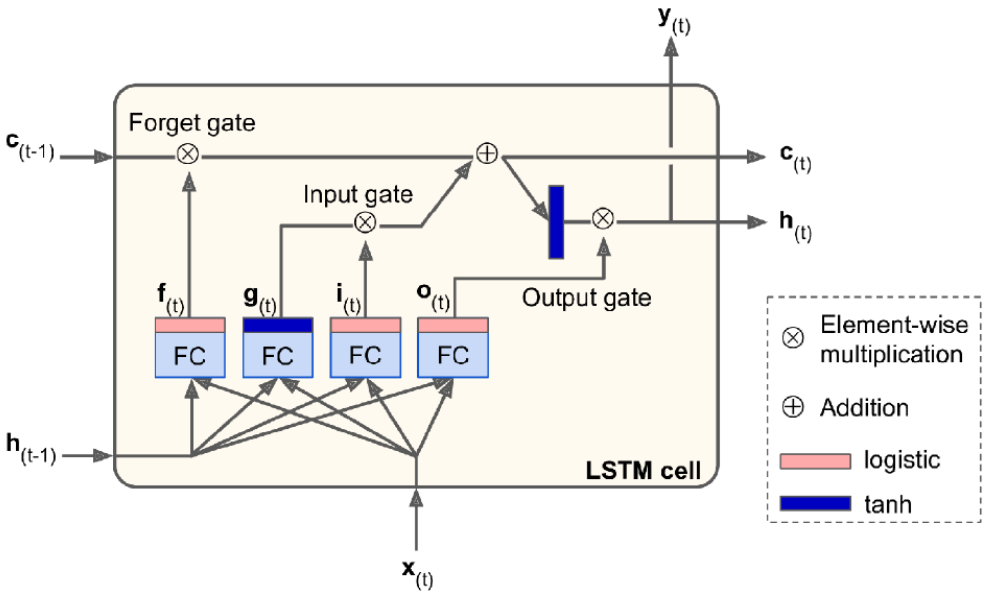
These are gate controllers

They use the logistic activation function

Their outputs range from 0 to 1

Architecture of LSTM Cell

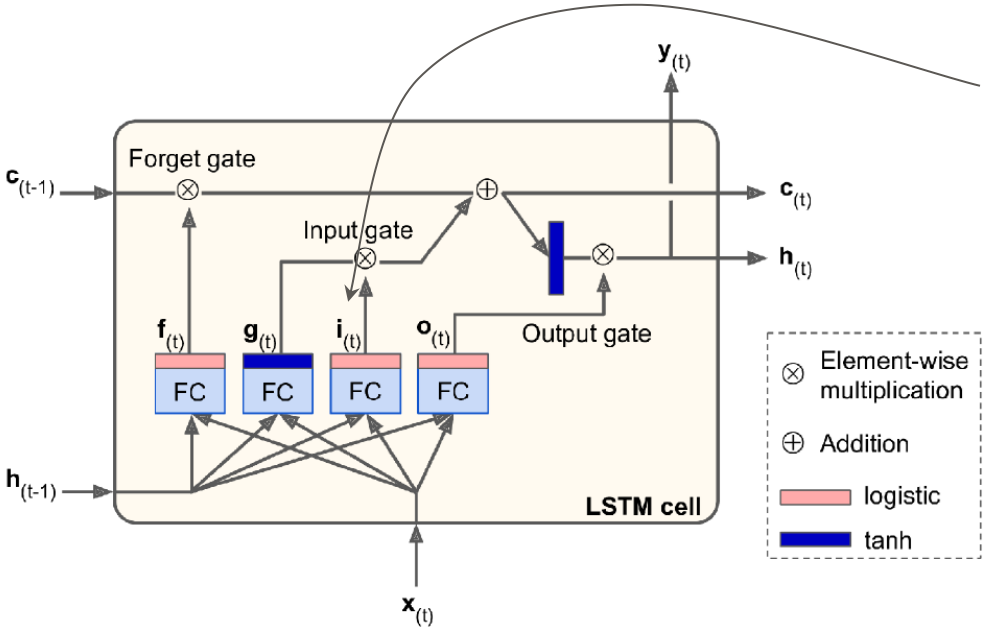
LSTM computations



- The following slides summarizes how to compute the cell's **long-term state**, its **short-term state**, and its **output** at each time step for a single instance
- The equations for a whole mini-batch are very similar

Architecture of LSTM Cell

LSTM computations

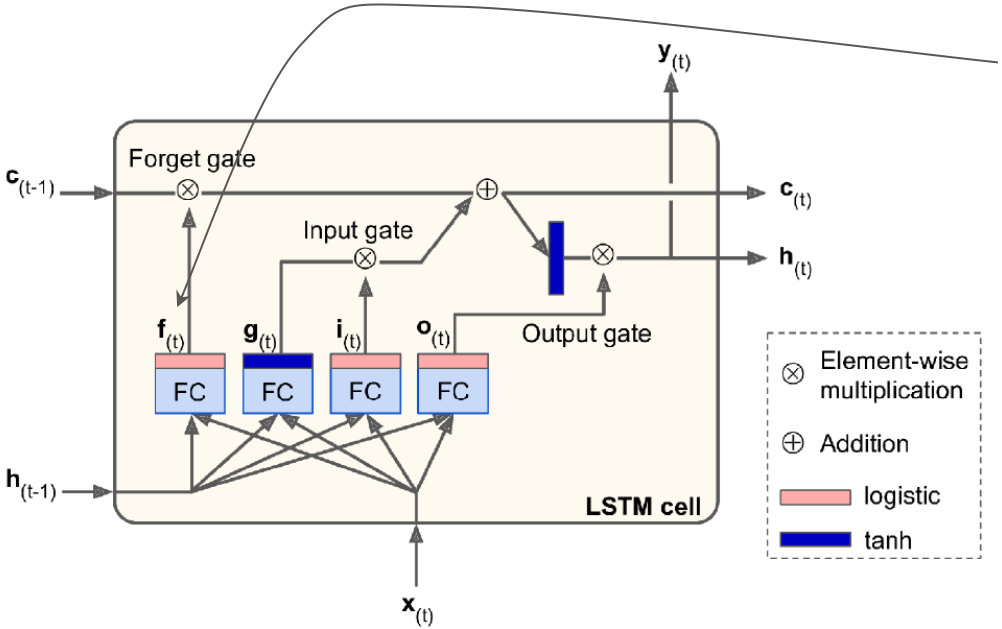


$$\mathbf{i}_{(t)} = \sigma\left(\mathbf{W}_{xi}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hi}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_i\right)$$

W_{xi} = weight matrices for the input vector $x_{(t)}$
W_{hi} = weight matrices for the previous short-term state $h_{(t-1)}$
b_i = bias term

Architecture of LSTM Cell

LSTM computations



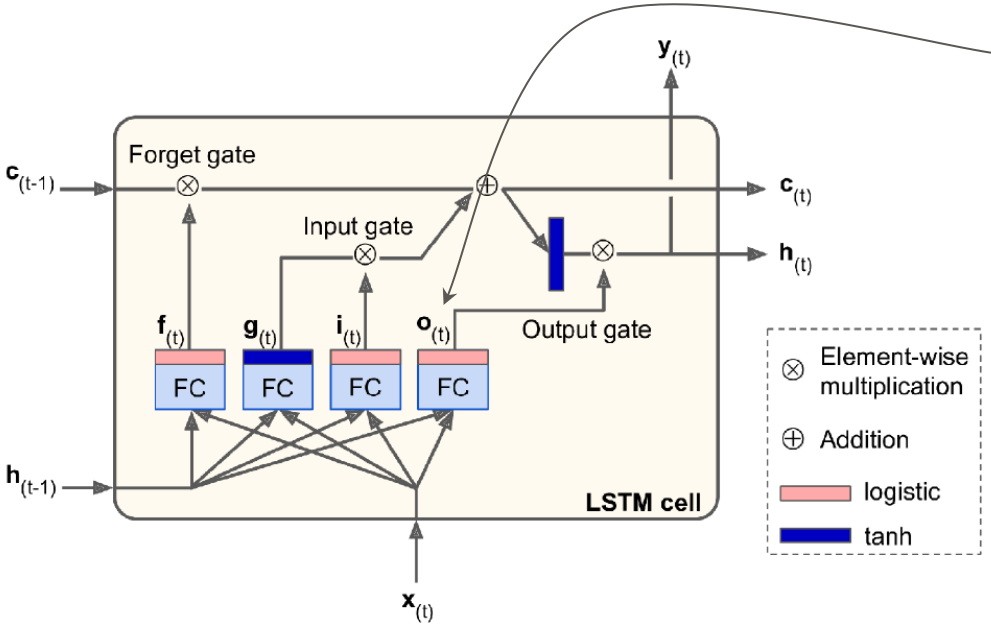
$$\mathbf{f}_{(t)} = \sigma\left(\mathbf{W}_{xf}{}^{\mathsf{T}}\mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{\mathsf{T}}\mathbf{h}_{(t-1)} + \mathbf{b}_f\right)$$

W_{xf} = weight matrices for the input vector $x_{(t)}$
W_{hf} = weight matrices for the previous short-term state $h_{(t-1)}$
b_f = bias term

Note that TensorFlow initializes b_f to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training

Architecture of LSTM Cell

LSTM computations



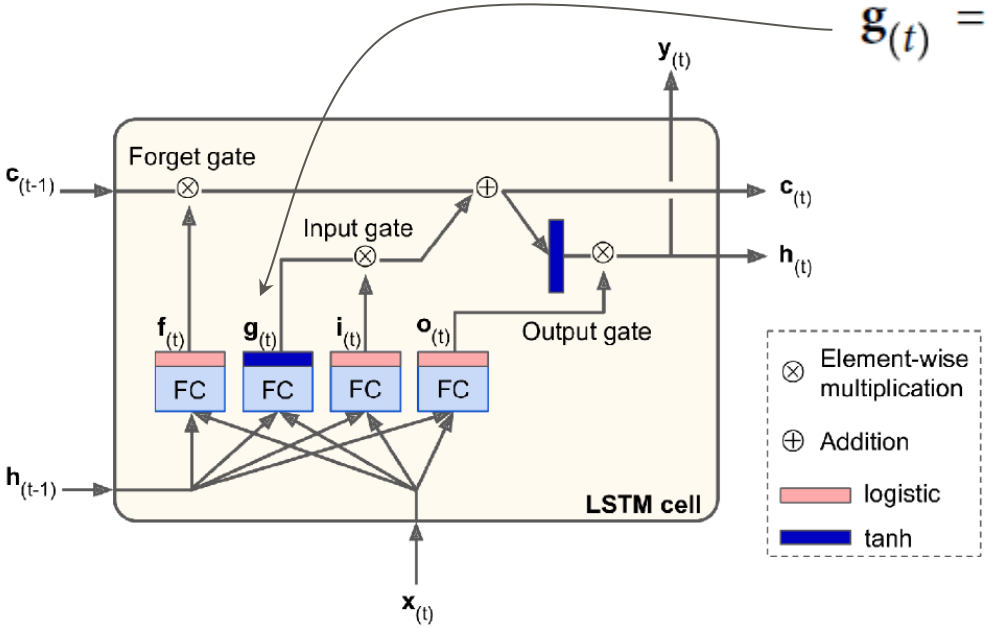
$$\mathbf{o}_{(t)} = \sigma\left(\mathbf{W}_{xo}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{ho}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_o\right)$$

W_{xo} = weight matrices for the input vector $x_{(t)}$
W_{ho} = weight matrices for the previous short-term state $h_{(t-1)}$
b_o = bias term

Architecture of LSTM Cell

LSTM computations

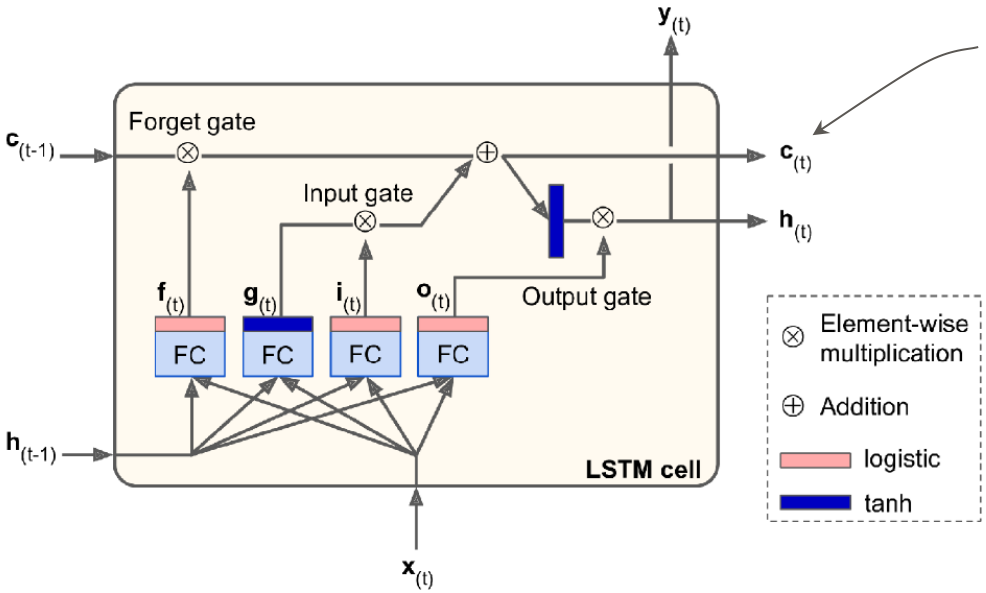
$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hg}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_{g}\right)$$



W_{xg} = weight matrices for the input vector $x_{(t)}$
W_{hg} = weight matrices for the previous short-term state $h_{(t-1)}$
b_g = bias term

Architecture of LSTM Cell

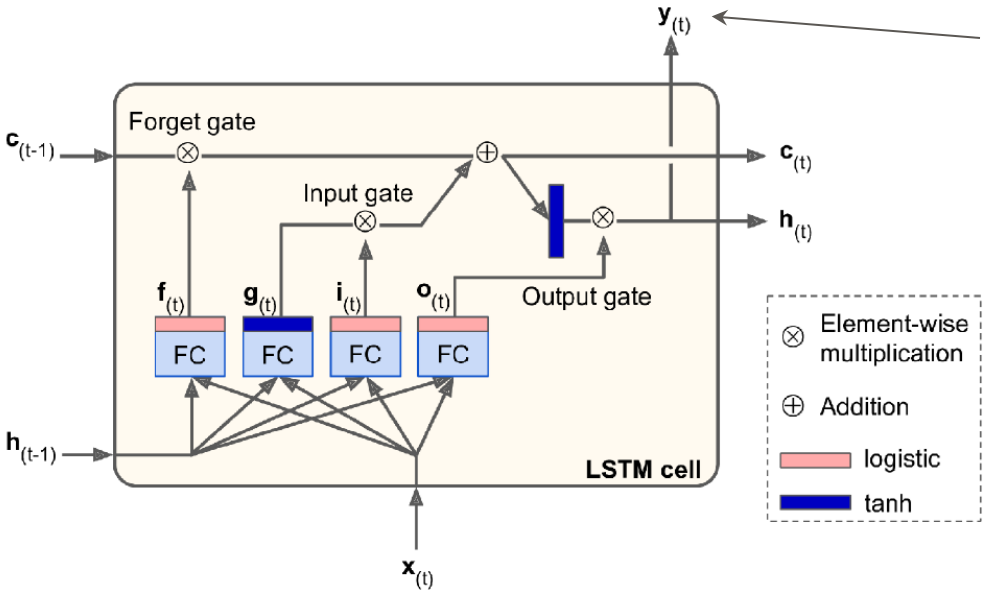
LSTM computations



$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}$$

Architecture of LSTM Cell

LSTM computations



$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)$$

LSTM Cell

Conclusion

- A **LSTM cell can learn** to
  - Recognize an important input, that's the role of the input gate,
  - Store it in the long-term state,
  - Learn to preserve it for as long as it is needed, that's the role of the forget gate,
  - And learn to extract it whenever it is needed

This explains why they have been amazingly successful at capturing long-term patterns in time series, long texts, audio recordings, and more.

Peephole Connections

Peephole Connections

- In a **basic LSTM cell**, the gate controllers can look only at the input $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$
- It may be a good idea to give them a bit more context by letting them peek at the **long-term state as well**
- This idea was proposed by **Felix Gers and Jürgen Schmidhuber** in 2000

Peephole Connections

- They proposed an **LSTM variant** with extra connections called **peephole connections**:
  - The previous long-term state $\mathbf{c}_{(t-1)}$ is added as an input to the controllers of the forget gate and the input gate,
  - And the current long-term state $\mathbf{c}_{(t)}$ is added as input to the controller of the output gate.

Peephole Connections

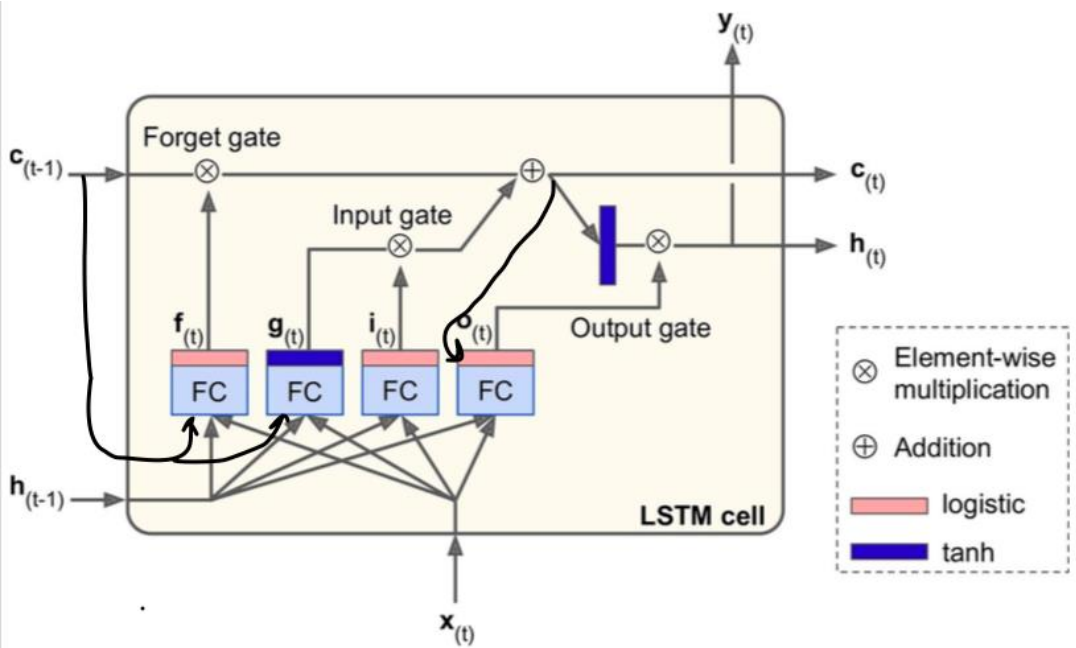


Figure 14-13. LSTM cell

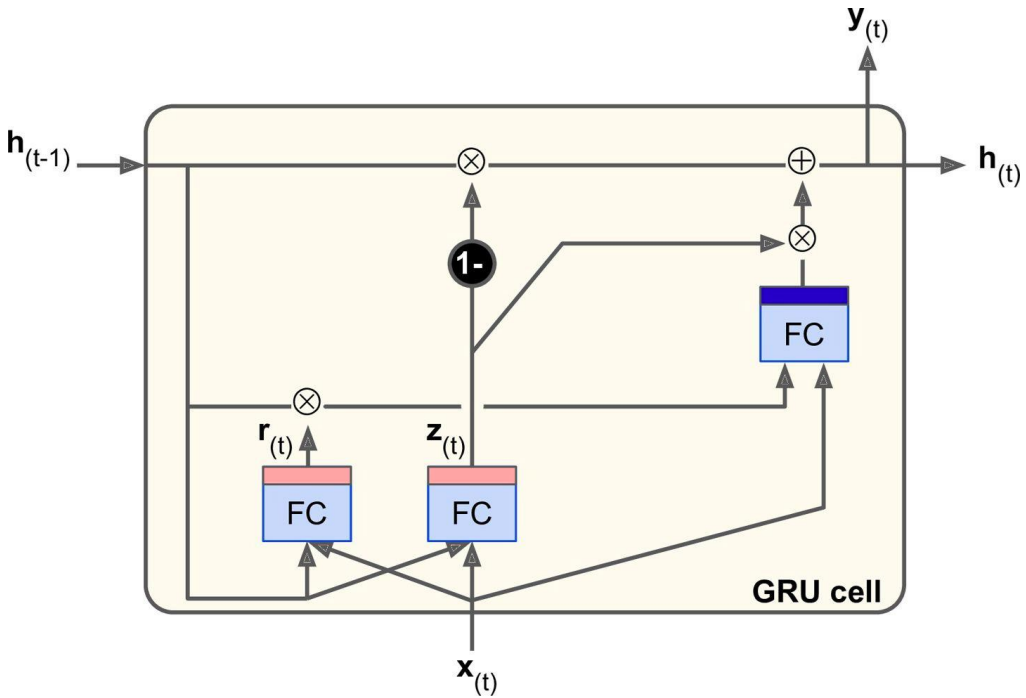
GRU Cell

GRU Cell

The Gated Recurrent Unit (GRU) cell was proposed by **Kyunghyun Cho** et al. in a 2014 paper that also introduced the Encoder–Decoder network we discussed earlier

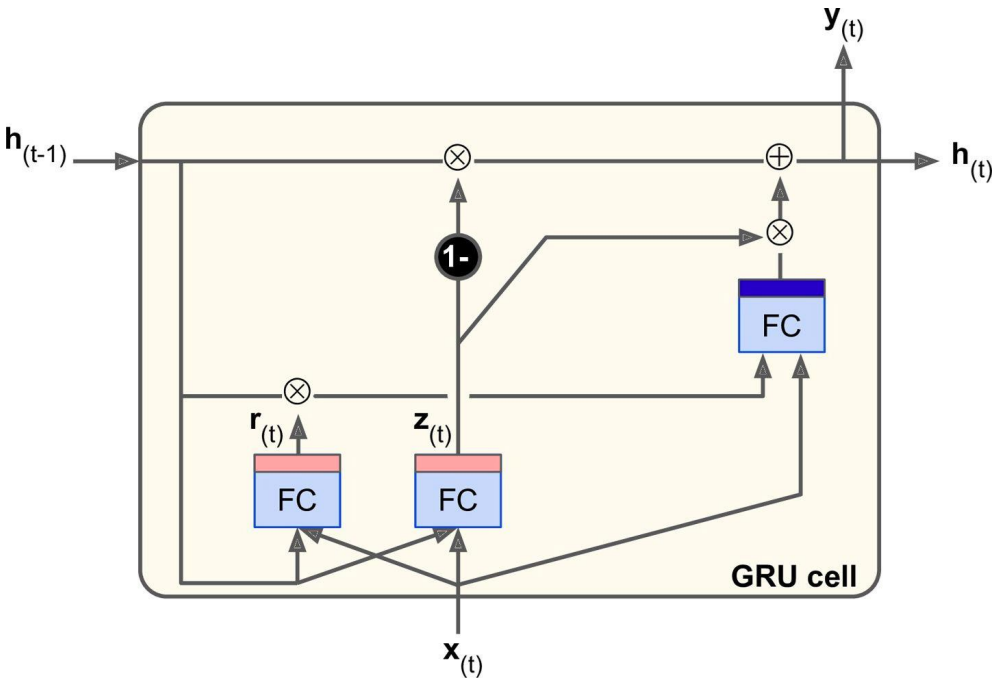
Kyunghyun Cho

GRU Cell



- The **GRU cell** is a simplified version of the **LSTM cell**
- It seems to perform just as well
- This explains its growing popularity

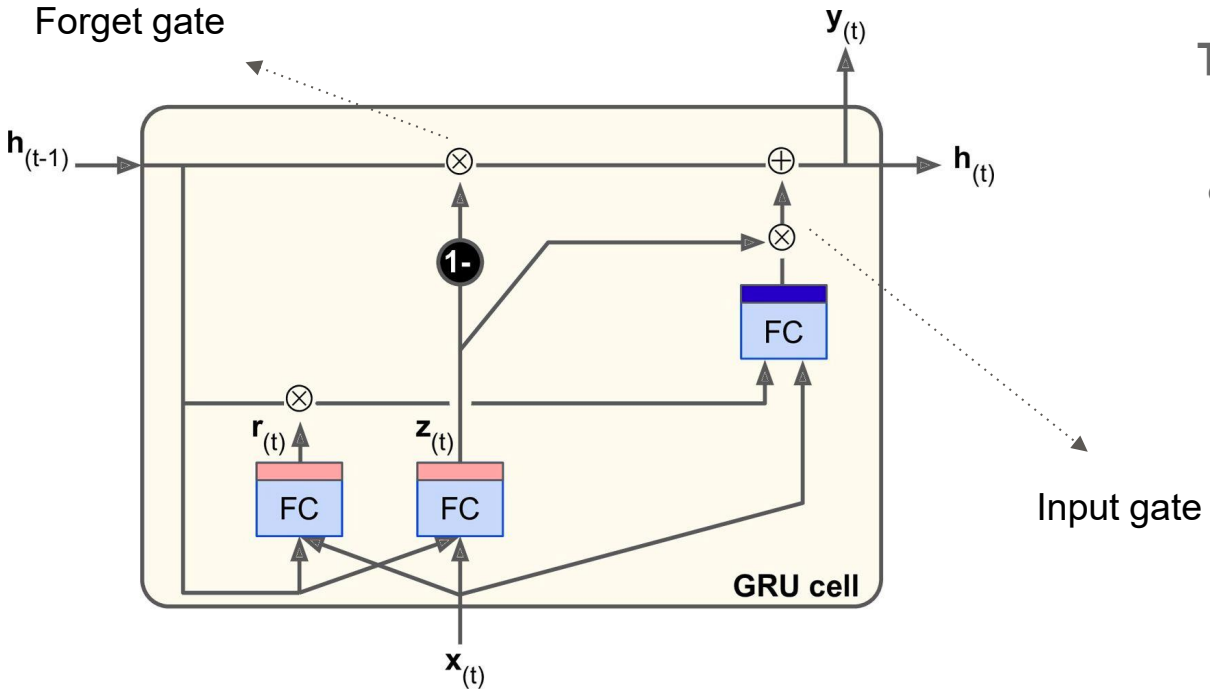
GRU Cell



The main simplifications are:

- Both state vectors are merged into a single vector $\mathbf{h}_{(t)}$

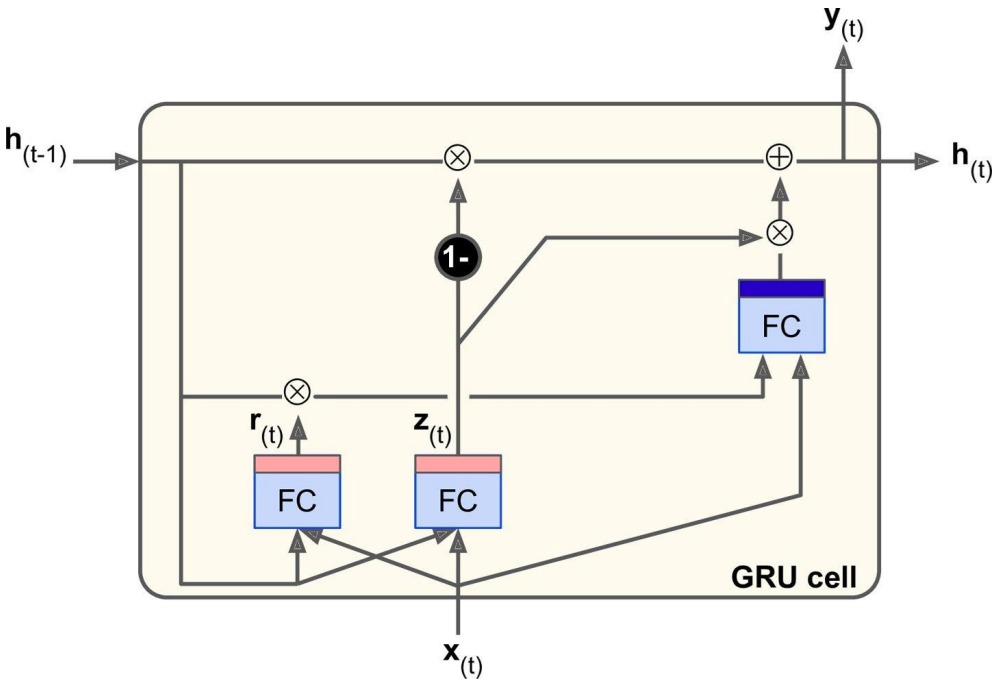
GRU Cell



The main simplifications are:

- A single gate controller controls both the forget gate and the input gate.
  If the gate controller outputs a 1, the input gate is open and the forget gate is closed.

GRU Cell

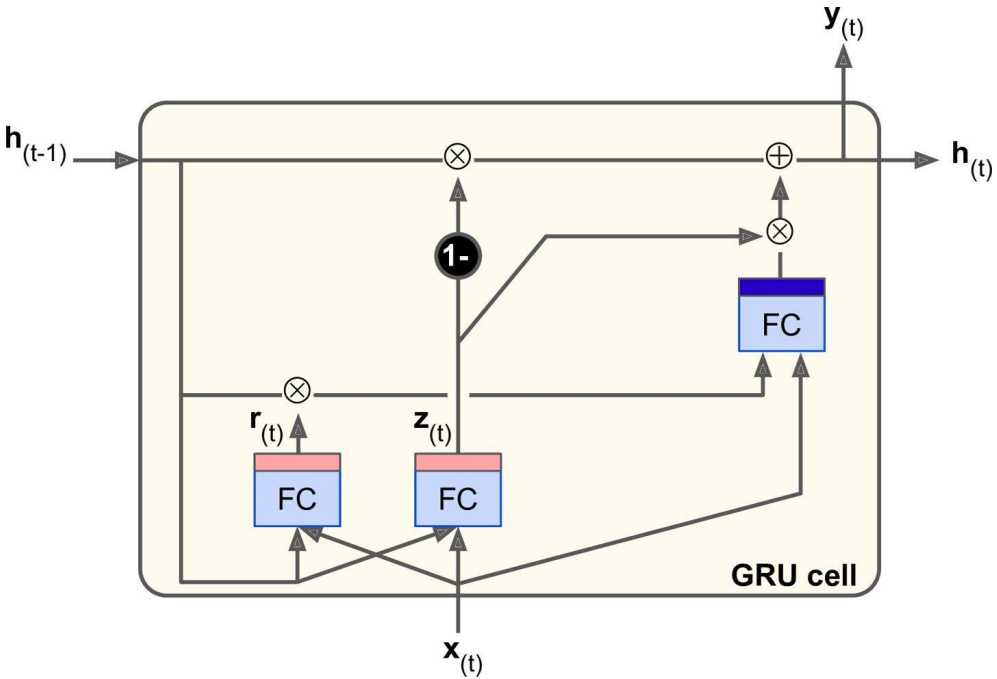


The main simplifications are:

If it outputs a 0, the opposite happens

In other words, whenever a memory must be stored, the location where it will be stored is erased first. This is actually a frequent variant to the LSTM cell in and of itself

GRU Cell

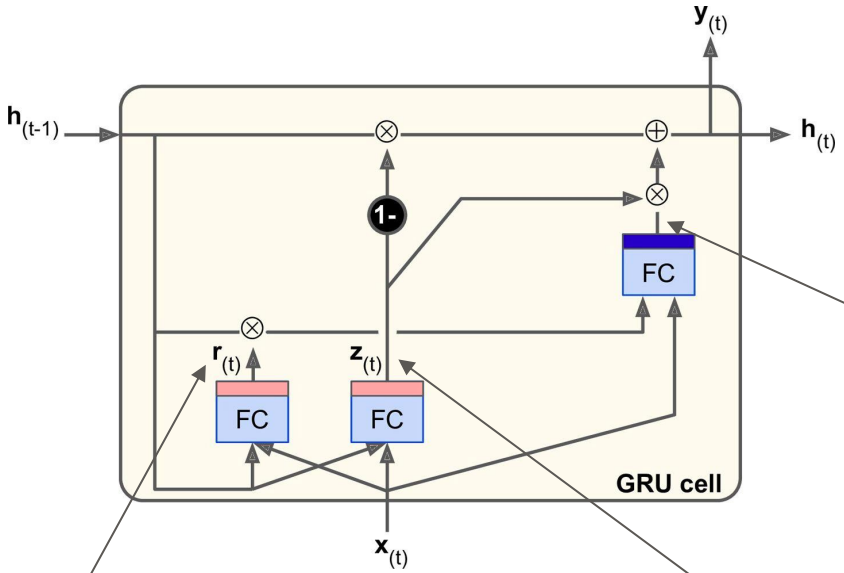


The main simplifications are:

There is no output gate; the full state vector is output at every time step. There is a new gate controller that controls which part of the previous state will be shown to the main layer.

GRU Cell

Cell's state at each time step for a single instance



$$\mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{g}_{(t)} + \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{h}_{(t-1)}$$

$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{hg}^{\intercal}\left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right) + \mathbf{b}_{g}\right)$$

$$\mathbf{r}_{(t)} = \sigma\left(\mathbf{W}_{xr}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{hr}^{\intercal}\mathbf{h}_{(t-1)} + \mathbf{b}_{r}\right)$$

$$\mathbf{z}_{(t)} = \sigma\left(\mathbf{W}_{xz}^{\intercal}\mathbf{x}_{(t)} + \mathbf{W}_{hz}^{\intercal}\mathbf{h}_{(t-1)} + \mathbf{b}_{z}\right)$$

GRU Cell

- LSTM or GRU cells are one of the main reasons behind the success of **RNNs** in recent years
- In particular for applications in **natural language processing (NLP)**