CSCI 6521 Advanced Machine Learning I

Chapter 06: Processing Sequences using RNNs and CNNs

Md Tamjidul Hoque

Recurrent Neural Networks (RNNs)

- Recurrent neural network (RNN) is a class of nets that can predict the future (well, up to a point, of course).
- RNNs can analyze time series data such as stock prices and tell you when to buy or sell.
- In autonomous driving systems, they can anticipate car trajectories and help avoid accidents.
- More generally, they can work on sequences of arbitrary lengths, rather than on fixed-sized inputs like all other nets.
  - For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing (NLP) applications such as automatic translation or speech-to-text.

… Recurrent Neural Networks (RNNs)

- Here, we will first look at the fundamental concepts underlying RNNs and how to train them using backpropagation through time,
- then we will use them to forecast a time series.
- After that we'll explore the two main difficulties that RNNs face:
  - Unstable gradients, which can be alleviated using various techniques, including recurrent dropout and recurrent layer normalization.
  - RNNs have a (very) limited short-term memory, which can be extended using LSTM and GRU cells.
- RNNs are not the only types of NNs capable of handling sequential data:
  - for small sequences, a regular dense network can do the trick; and
  - for very long sequences, such as audio samples or text, convolutional neural networks (CNNS) can actually work quite well too -
  - we will discuss both of these possibilities.

Recurrent Neurons and Layers

- A recurrent neural network (RNN) looks very much like a feedforward neural network, except it also has connections pointing backward.

- See Figure 1 (left): a simplest possible RNN, composed of one neuron receiving inputs, producing an output, and sending that output back to itself.

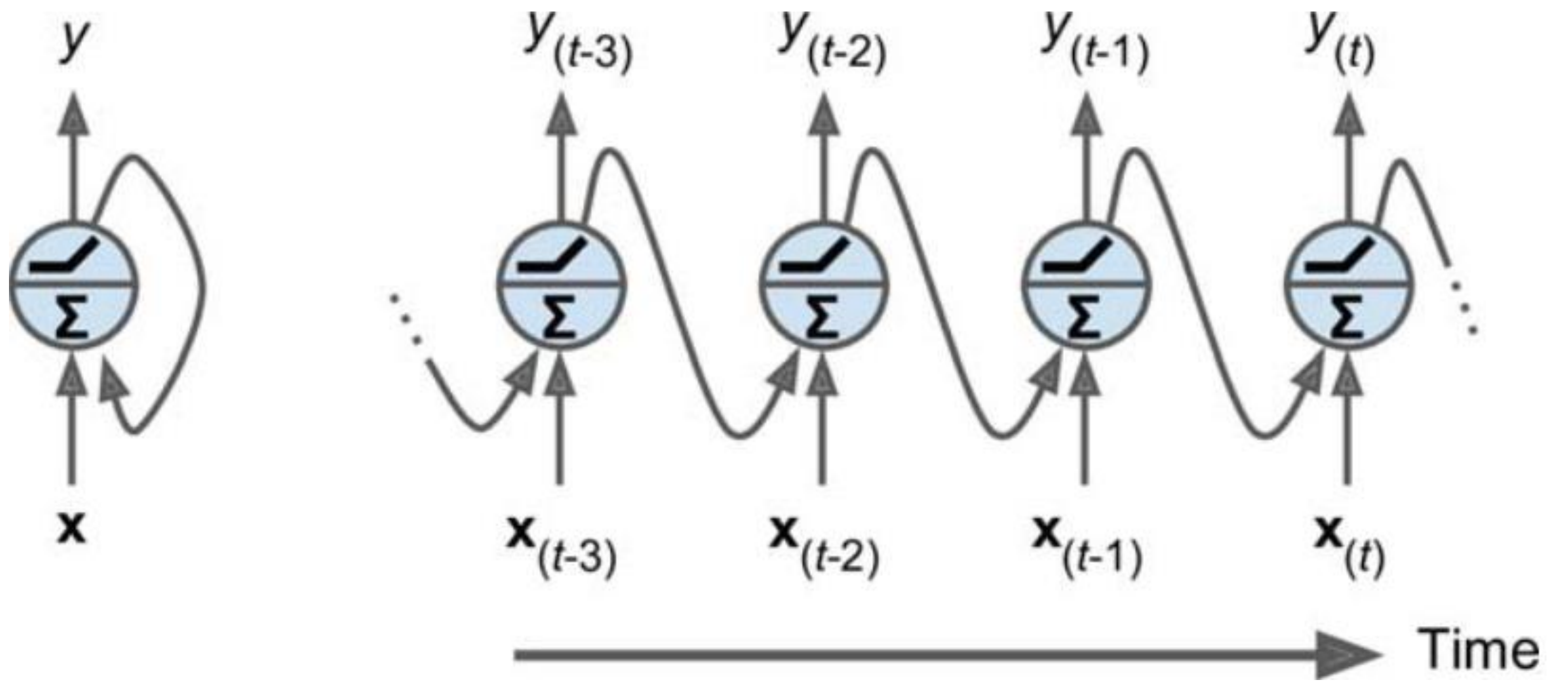


Figure 1: A recurrent neuron (left) unrolled through time (right).

… Recurrent Neurons and Layers

- At each time step ***t*** (also called a frame), this recurrent neuron receives the inputs $\boldsymbol{x}_{(t)}$ as well as its own output from the previous time step, $\boldsymbol{y}_{(t-1)}$.
- Since there is no previous output at the first time step, it is generally set to 0.
- We can represent this tiny network against the time axis, as shown in Figure 1 (right). This is called *unrolling the network through time* (it's the same recurrent neuron represented once per time step).

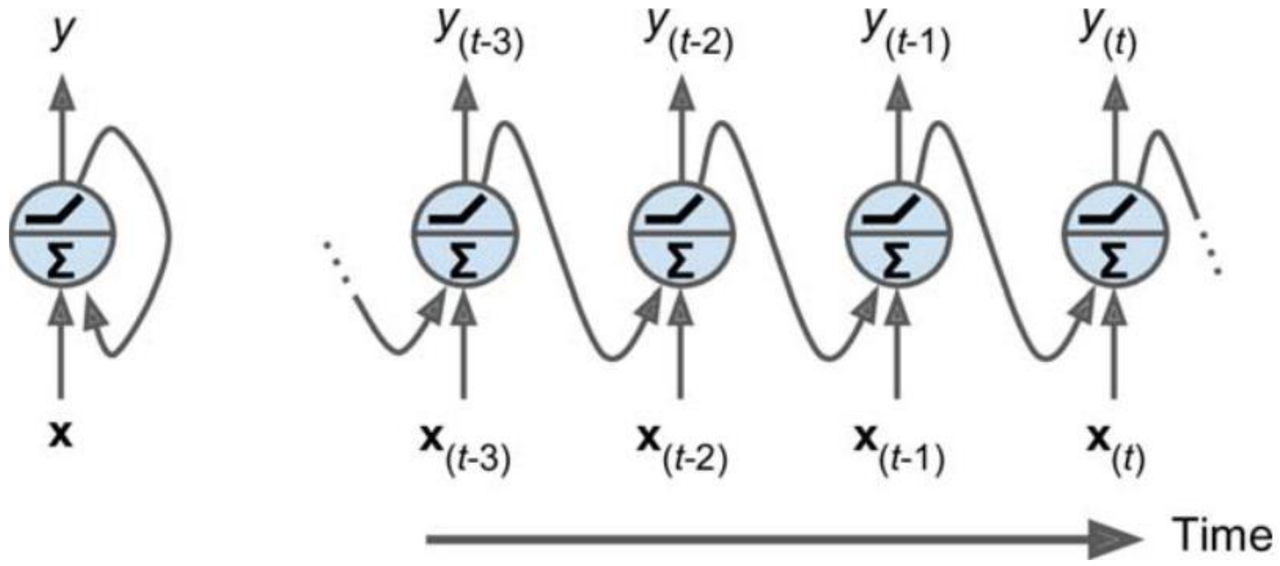


Figure 1: A recurrent neuron (left) unrolled through time (right).

… Recurrent Neurons and Layers

- We can easily create a layer of recurrent neurons.
- At each time step ***t***, every neuron receives both the input vector $\boldsymbol{x}_{(t)}$ and the output vector from the previous time step $\boldsymbol{y}_{(t-1)}$, as shown in Figure 2.
- Note that both the inputs and outputs are vectors now (when there was just a single neuron, the output was a scalar).

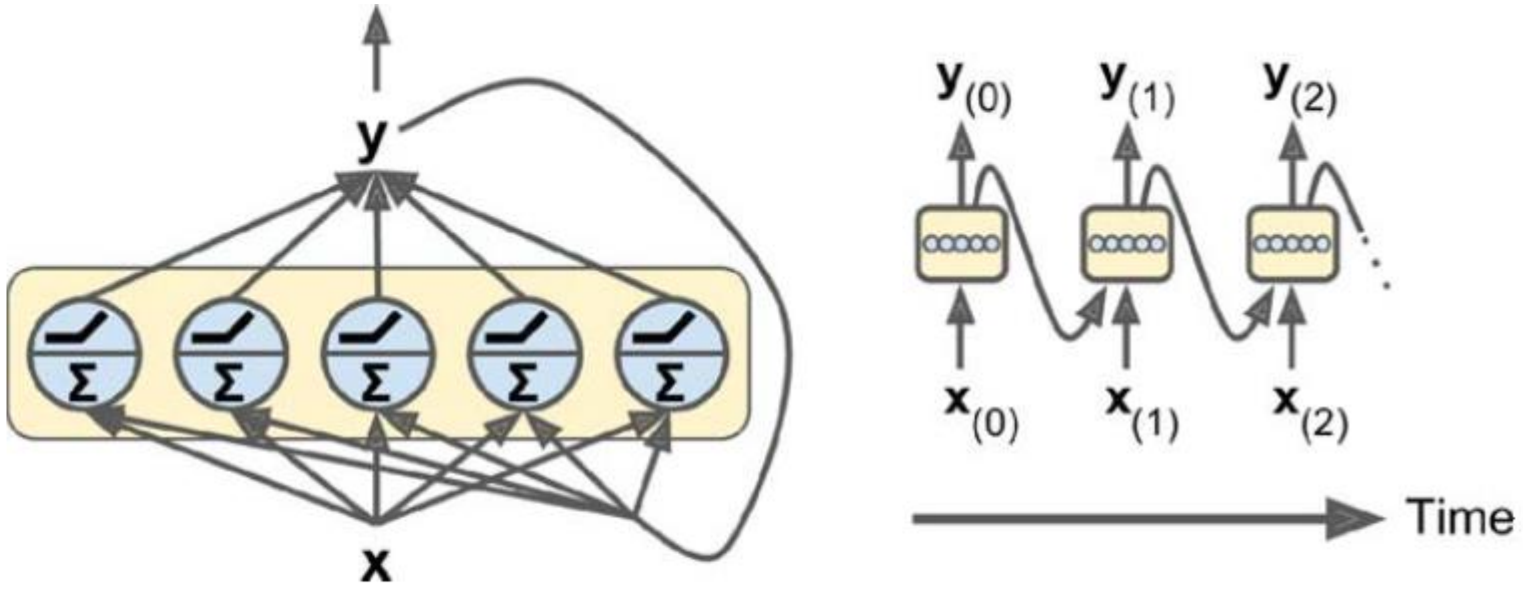


Figure 2: A layer of recurrent neurons (left) unrolled through time (right).

… Recurrent Neurons and Layers

- Each recurrent neuron has two sets of weights: one for the inputs $\boldsymbol{x}_{(t)}$ and the other for the outputs of the previous time step, $\boldsymbol{y}_{(t-1)}$.
- Let us call these weight vectors $\mathbf{w}_x$ and $\mathbf{w}_y$.
- If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices, $\mathbf{W}_x$ and $\mathbf{W}_y$.
- The output vector of the whole recurrent layer can then be computed pretty much as you might expect, as shown in Equation 1 (**b** is the bias vector and $\phi(\cdot)$ is the activation function (e.g., hyperbolic tangent (tanh) or a ReLU).
- Eq 1: Output of a recurrent layer for a single instance –

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

- We can compute a recurrent layer's output in one shot for a whole mini-batch by placing all the inputs at time step *t* in an input matrix $\mathbf{X}_{(t)}$ using Equation 2: $\mathbf{Y}_{(t)} = \emptyset\left(\mathbf{W}_x^T \, \mathbf{X}_{(t)} + \mathbf{W}_y^T \, \mathbf{Y}_{(t-1)} + \mathbf{b}\right)$

… Recurrent Neurons and Layers

- In Equation 2: $\mathbf{Y}_{(t)} = \emptyset\left(\mathbf{W}_x^T\, \mathbf{X}_{(t)} + \mathbf{W}_y^T\, \mathbf{Y}_{(t-1)} + \mathbf{b}\right)$,
- Or, $\mathbf{Y}_{(t)} = \emptyset\left(\left[\mathbf{X}_{(t)}\ \mathbf{Y}_{(t-1)}\right] \mathbf{W} + \mathbf{b}\right)$ with $\mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}$
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (m is the number of instances in the mini-batch and n_{inputs} is the number of input features).
- $\mathbf{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (n_{neurons} is the number of neurons).
- $\mathbf{W}_x$ is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $\mathbf{W}_y$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- $\mathbf{b}$ is a vector of size n_{neurons} containing each neuron's bias term.
- The weight matrices $\mathbf{W}_x$ and $\mathbf{W}_y$ are often concatenated vertically into a single weight matrix $\mathbf{W}$ of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ (see the second line of Equation 2).
- The notation $[\mathbf{X}_{(t)}\ \mathbf{Y}_{(t-1)}]$ represents the horizontal concatenation of the matrices $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$.

… Recurrent Neurons and Layers

- In Equation 2: $\mathbf{Y}_{(t)}=\emptyset\left(\mathbf{W}_x^T\,\mathbf{X}_{(t)}+\mathbf{W}_y^T\,\mathbf{Y}_{(t-1)}+\mathbf{b}\right)$,
- Or $\mathbf{Y}_{(t)}=\emptyset\left(\left[\mathbf{X}_{(t)}\ \mathbf{Y}_{(t-1)}\right]\mathbf{W}+\mathbf{b}\right)$ with $\mathbf{W}=\begin{bmatrix}\mathbf{W}_x\\ \mathbf{W}_y\end{bmatrix}$
- Notice that $\mathbf{Y}_{(t)}$ is a function of $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$, which is a function of $\mathbf{X}_{(t-1)}$ and $\mathbf{Y}_{(t-2)}$, which is a function of $\mathbf{X}_{(t-2)}$ and $\mathbf{Y}_{(t-3)}$, and so on.
- This makes $\mathbf{Y}_{(t)}$ a function of all the inputs since time $\boldsymbol{t} = 0$ (that is, $\mathbf{X}_{(0)}$, $\mathbf{X}_{(1)}$, …, $\mathbf{X}_{(t)}$).
- At the first-time step, $\boldsymbol{t} = 0$, there are no previous outputs, so they are typically assumed to be all zeros.

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 could say it has a form of memory.
- A part of a neural network that preserves some state across time steps is called a memory cell (or simply a cell).
- A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (typically about 10 steps long, but this varies depending on the task).
- Later in this chapter, we will look at some more complex and powerful types of cells capable of learning longer patterns (roughly 10 times longer, but again, this depends on the task).

… Memory Cells

- In general, a cell's state at time step *t*, denoted $\mathbf{h}_{(t)}$ (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$.

- Its output at time step *t*, denoted $\mathbf{y}_{(t)}$, is also a function of the previous state and the current inputs.

- For the basic cells, the output is simply equal to the state, but in more complex cells this is not always the case, as shown in Figure 3.

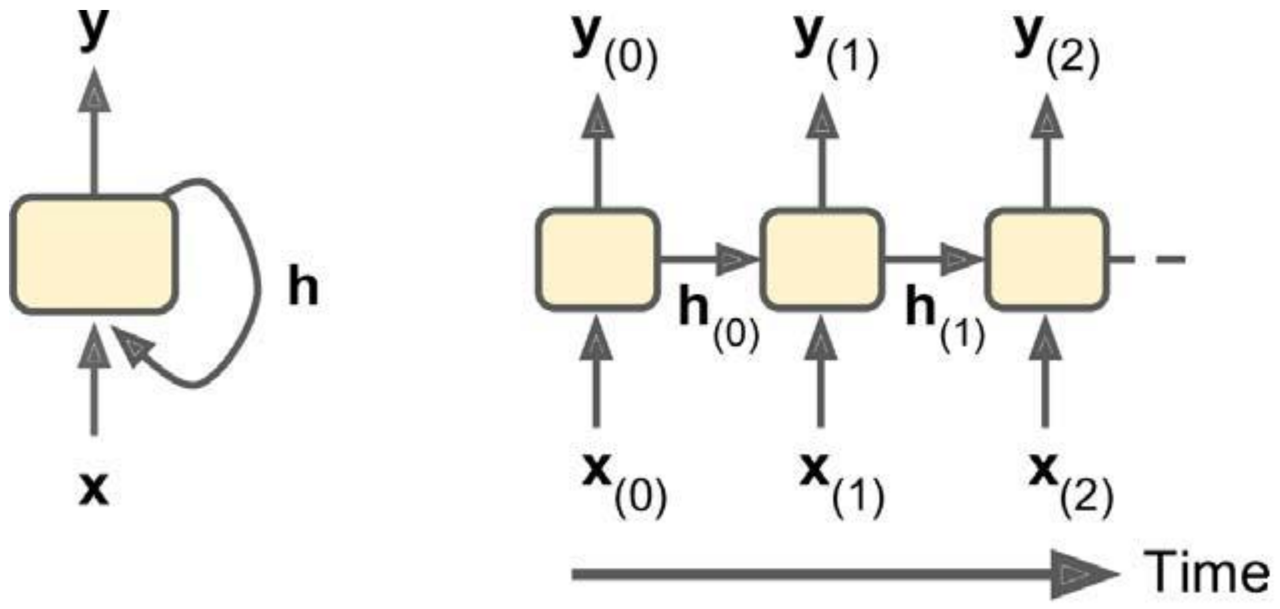
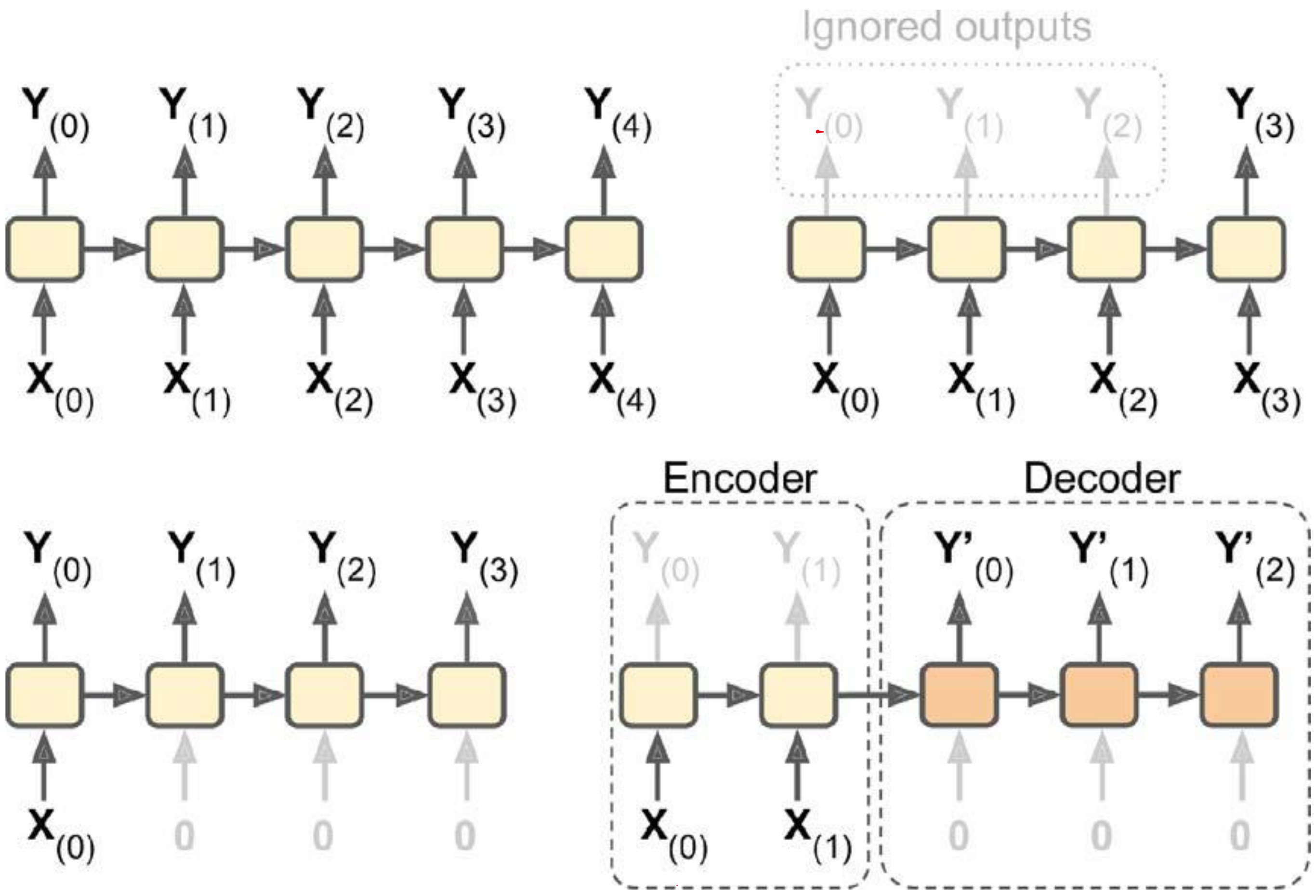


Figure 3: A cell's hidden state and its output may be different.

Input and Output Sequences

- An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs. See the top-left network in Figure 4.



Ignored outputs
$Y_{(0)}$ $Y_{(1)}$ $Y_{(2)}$ $Y_{(3)}$ $Y_{(4)}$
$X_{(0)}$ $X_{(1)}$ $X_{(2)}$ $X_{(3)}$ $X_{(4)}$
$Y_{(0)}$ $Y_{(1)}$ $Y_{(2)}$ $Y_{(3)}$
$X_{(0)}$ $X_{(1)}$ $X_{(2)}$ $X_{(3)}$
Encoder
Decoder
$Y_{(0)}$ $Y_{(1)}$ $Y_{(2)}$ $Y_{(3)}$
$X_{(0)}$ 0 0 0
$Y_{(0)}$ $Y_{(1)}$ $Y'_{(0)}$ $Y'_{(1)}$ $Y'_{(2)}$
$X_{(0)}$ $X_{(1)}$ 0 0 0

Figure 4: Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and Encoder–Decoder (bottom right) networks.

… Input and Output Sequences

- An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs. See the top-left network in Figure 4.

- This type of sequence-to-sequence network is useful for predicting time series such as stock prices:
  - you 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).

- Alternatively, you could feed the network a sequence of inputs and ignore all outputs except for the last one (see the top-right network in Figure 4).
  - In other words, this is a sequence-to-vector network.
  - For example, you could feed the 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

- Conversely, we could feed the network the same input vector over and over again at each time step and let it output a sequence (see the bottom-left network of Figure 4).

- This is a vector-to-sequence network.

- For example, the input could be an image (or the output of a CNN), and the output could be a caption for that image.

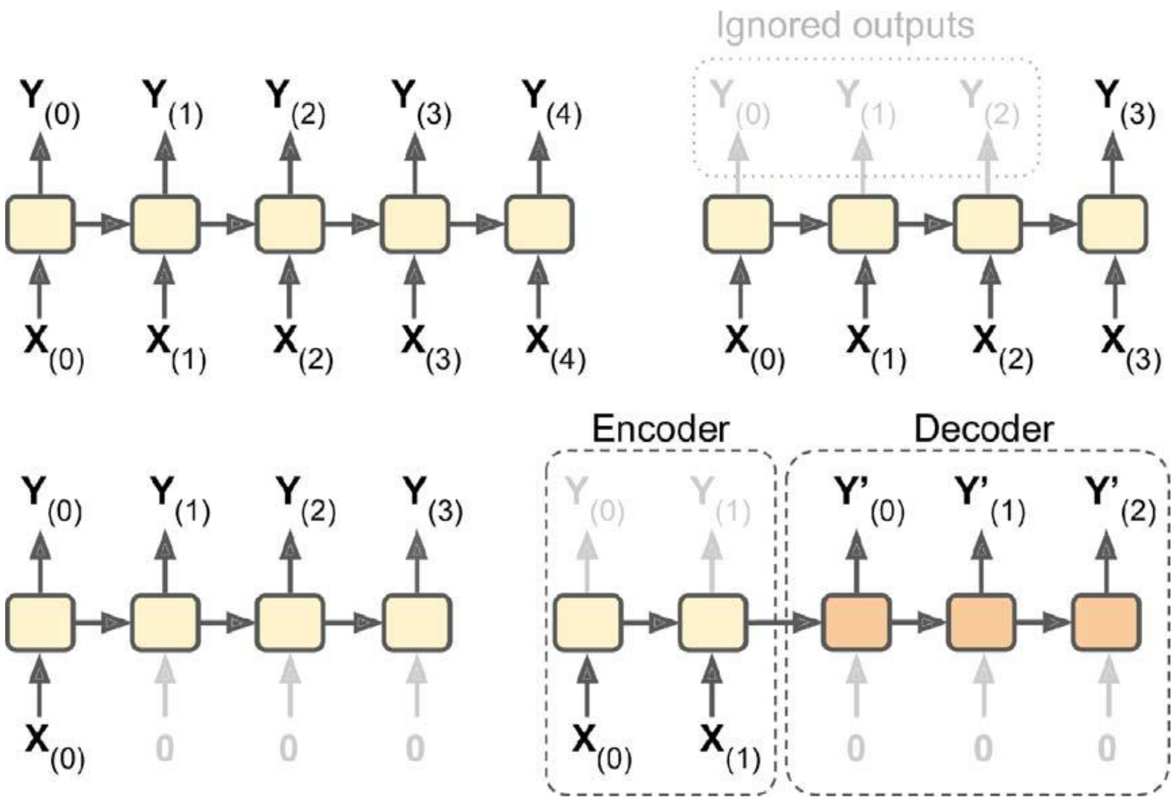


Figure 4: Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and Encoder–Decoder (bottom right) networks.

… Input and Output Sequences

- Lastly, we could have a sequence-to-vector network, called an encoder, followed by a vector-to-sequence network, called a decoder (see the bottom-right network of Figure 4).

- For example, this could be used for translating a sentence from one language to another.

- We would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language.

- This two-step model, called an Encoder–Decoder, works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented at the top left):

    - the last words of a sentence can affect the first words of the translation, so we need to wait until we have seen the whole sentence before translating it.

    - We will see how to implement an Encoder–Decoder in the next (as we will see, it is a bit more complex than in Figure 4 suggests).

Training RNNs

- To train an RNN, the trick is to unroll it through time (like we just did) and then simply use regular backpropagation (see Figure 5). This strategy is *called backpropagation through time* (BPTT).

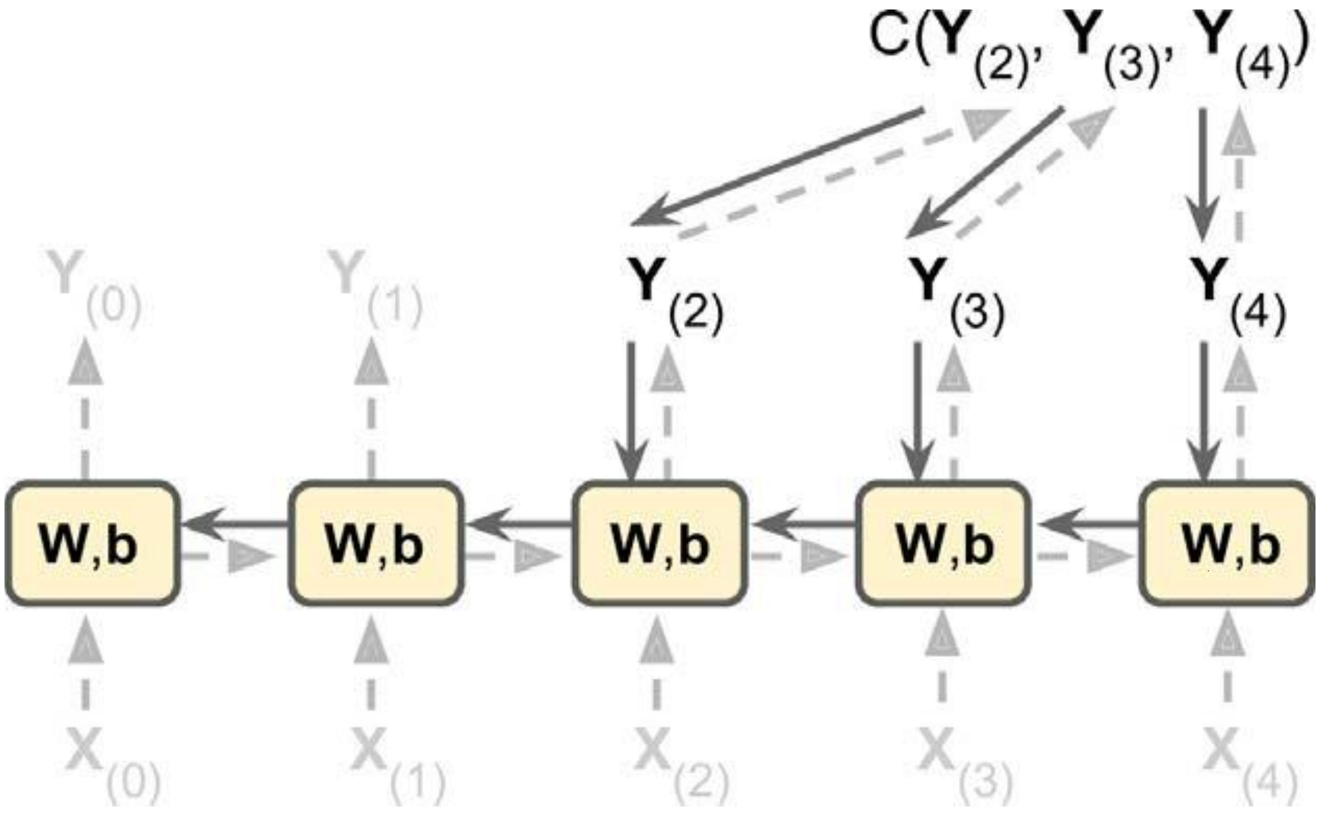


Figure 5: Backpropagation through time.

… Training RNNs

- Just like in regular backpropagation, 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 $C(\mathbf{Y}_{(0)}, \mathbf{Y}_{(1)}, \ldots \mathbf{Y}_{(T)})$ (where T is the max time step).
- Note that this cost function may ignore some outputs, as shown in Figure 5 (for example, in a sequence-to-vector RNN, all outputs are ignored except for the very last one).
- The gradients of that cost function are then propagated backward through the unrolled network (represented by the solid arrows).
- Finally the model parameters are updated using the gradients computed during BPTT.
- Note that the gradients flow backward through all the outputs used by the cost function, not just through the final output (for example, in Figure 5 the cost function is computed using the last three outputs of the network, $\mathbf{Y}_{(2)}$, $\mathbf{Y}_{(3)}$, and $\mathbf{Y}_{(4)}$, so gradients flow through these three outputs, but not through $\mathbf{Y}_{(0)}$ and $\mathbf{Y}_{(1)}$).
- Moreover, 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.
- Fortunately, tf.keras takes care of all of this complexity for us: **see exercise**.

Trend and Seasonality

- There are many other models to forecast time series, such as *weighted moving average* models or *autoregressive integrated moving average* (ARIMA) models.
- Some of them require you to first remove the trend and seasonality.
    - For example, if we are studying the number of active users on our website, and it is growing by 10% every month, we would have to remove this trend from the time series.
        - Once the model is trained and starts making predictions, you would have to add the trend back to get the final predictions.

… Trend and Seasonality

- Similarly, if we are trying to predict the amount of sunscreen lotion sold every month, we will probably observe strong seasonality:
  - since it sells well every summer, a similar pattern will be repeated every year. We would have to remove this seasonality from the time series, for example by computing the difference between the value at each time step and the value one year earlier (this technique is called *differencing*).
  - Again, after the model is trained and makes predictions, we would have to add the seasonal pattern back to get the final predictions.
- When using RNNs, it is generally not necessary to do all this, but it may improve performance in some cases, since the model will not have to learn the trend or the seasonality.
- Simple RNNs can be quite good at forecasting time series or handling other kinds of sequences, but they do not perform as well on long time series or sequences.

Handling Long Sequences

- To train an RNN on long sequences, we must run it over many time steps, making the unrolled RNN a very deep network.

- Just like any deep neural network it may suffer from the unstable gradients problem:
  - it may take forever to train, or training may be unstable.
  - Moreover, when an RNN processes a long sequence, it will gradually forget the first inputs in the sequence.

- Let us look at both these problems, starting with the unstable gradients problem.

Fighting the Unstable Gradients Problem

- Many of the tricks we used in deep nets to alleviate the unstable gradients problem can also be used for RNNs:
  - good parameter initialization,
  - faster optimizers,
  - dropout, and so on.
- However, nonsaturating activation functions (e.g., ReLU) may not help as much here; in fact, they may actually lead the RNN to be even more unstable during training.
- Why?
  - Well, suppose Gradient Descent updates the weights in a way that increases the outputs slightly at the first time step.
  - Because the same weights are used at every time step, the outputs at the second time step may also be slightly increased, and those at the third, and so on until the outputs explode—and a nonsaturating activation function does not prevent that.
  - You can reduce this risk by using a smaller learning rate, but you can also simply use a saturating activation function like the hyperbolic tangent (this explains why it is the default).
  - In much the same way, the gradients themselves can explode. If you notice that training is unstable, you may want to monitor the size of the gradients (e.g., using TensorBoard) and perhaps use Gradient Clipping.

… Fighting the Unstable Gradients Problem

- Can we try Batch Normalization (BN) to handle Unstable Gradient?
  - Batch Normalization cannot be used as efficiently with RNNs as with deep feedforward nets.
  - In fact, you cannot use it between time steps, only between recurrent layers.
  - To be more precise, it is technically possible to add a BN layer to a memory cell (see exercise) so that it will be applied at each time step (both on the inputs for that time step and on the hidden state from the previous step).
  - However, the same BN layer will be used at each time step, with the same parameters, regardless of the actual scale and offset of the inputs and hidden state. In practice, this does not yield good results (see paper #1 in the note section).
  - It was found that BN was slightly beneficial only when it was applied to the inputs, not to the hidden states (see paper #1).

… Fighting the Unstable Gradients Problem

- In other words, it was slightly better than nothing when applied between recurrent layers (i.e., vertically in Figure 7), but not within recurrent layers (i.e., horizontally).
- In Keras this can be done simply by adding a BN layer before each recurrent layer, but we don't expect too much from it.

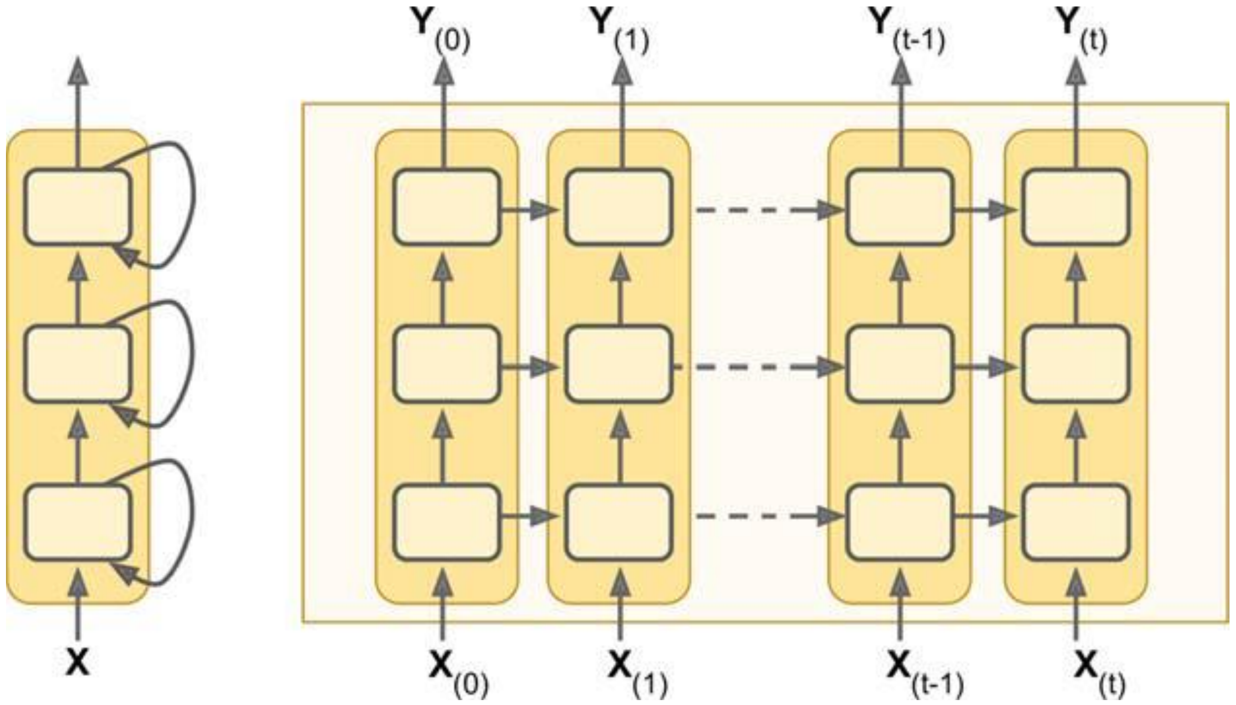


Figure 7: Deep RNN (left) unrolled through time (right).

… Fighting the Unstable Gradients Problem

- What about Layer Normalization (see paper#1) instead of BN?
  - Layer Normalization (LN) is very similar to Batch Normalization, but instead of normalizing across the batch dimension, it normalizes across the features dimension.
  - One advantage is that it can compute the required statistics on the fly, at each time step, independently for each instance.
  - This also means that it behaves the same way during training and testing (as opposed to BN), and
  - it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set.
  - Like BN, LN learns a scale and an offset parameter for each input.
  - In an RNN, LN (**see exercise**) is typically used right after the linear combination of the inputs and the hidden states.

Tackling the Short-Term Memory Problem

- Due to the transformations that the data goes through when traversing an RNN, some information is lost at each time step.
- After a while, the RNN's state contains virtually no trace of the first inputs.
- 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.
- Let us first look at the most popular of these long-term memory cells: the LSTM cell (proposed in 1997 – see paper 1-3).
- If you consider the LSTM cell as a black box, it can be used very much like a basic cell, except it will perform much better; training will converge faster, and it will detect long-term dependencies in the data.

…Tackling the Short-Term Memory Problem

- LSTM's architecture is shown in Figure 9.

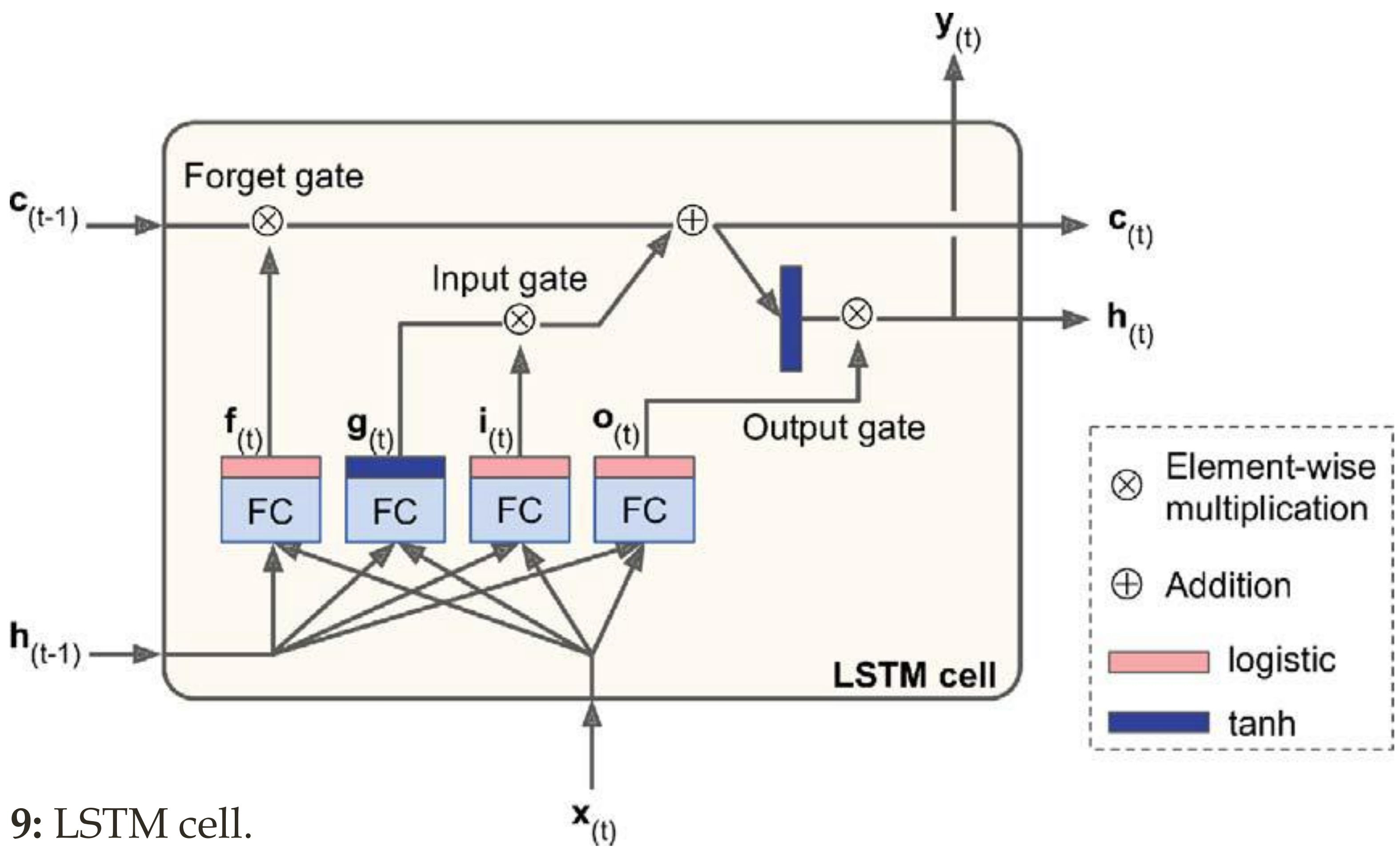


Figure 9: LSTM cell.

…Tackling the Short-Term Memory Problem

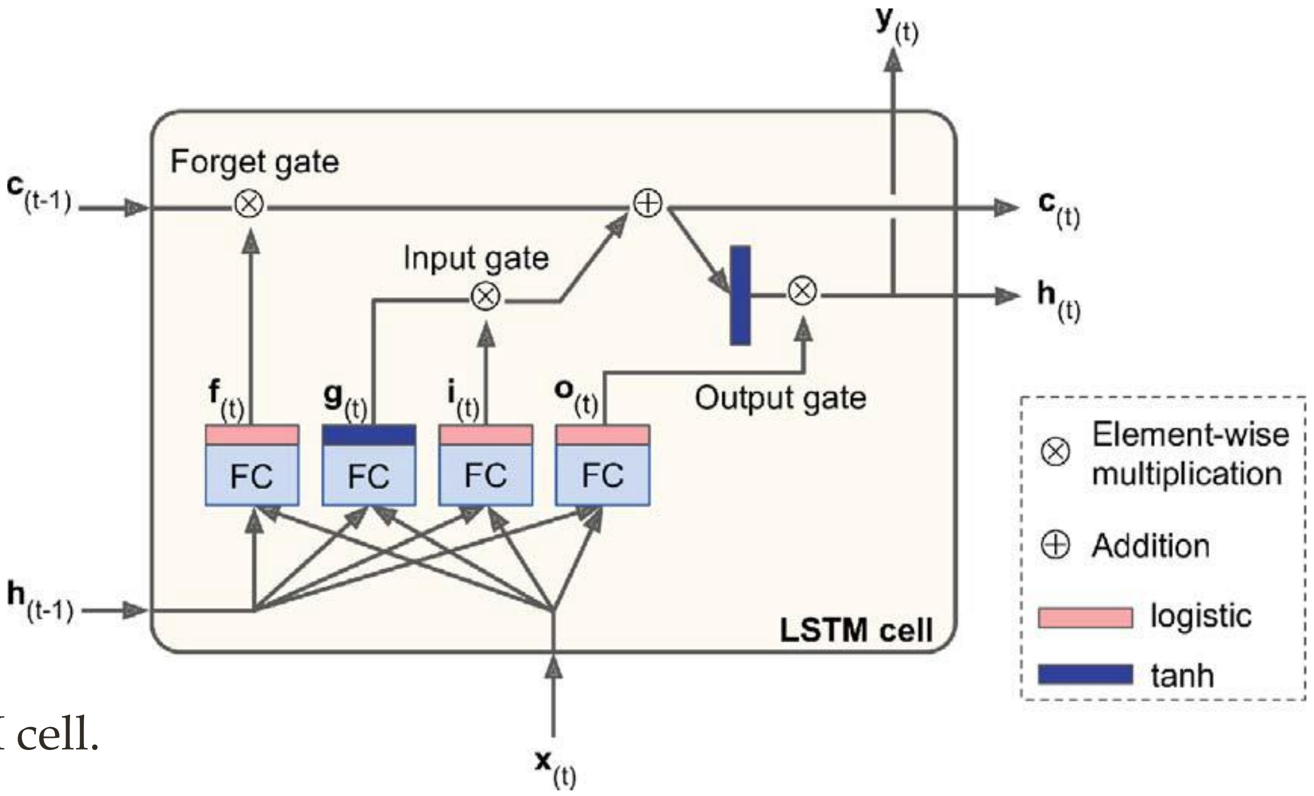


Figure 9: LSTM cell.

- If we do not look at what is inside the box, the LSTM cell looks exactly like a regular cell, -

- except that its state is split into two vectors: $\mathbf{h}_{(t)}$ and $\mathbf{c}_{(t)}$ ("c" stands for "cell").
  - We can think of $\mathbf{h}_{(t)}$ as the short-term state and
  - $\mathbf{c}_{(t)}$ as the long-term state.

…Tackling the Short-Term Memory Problem

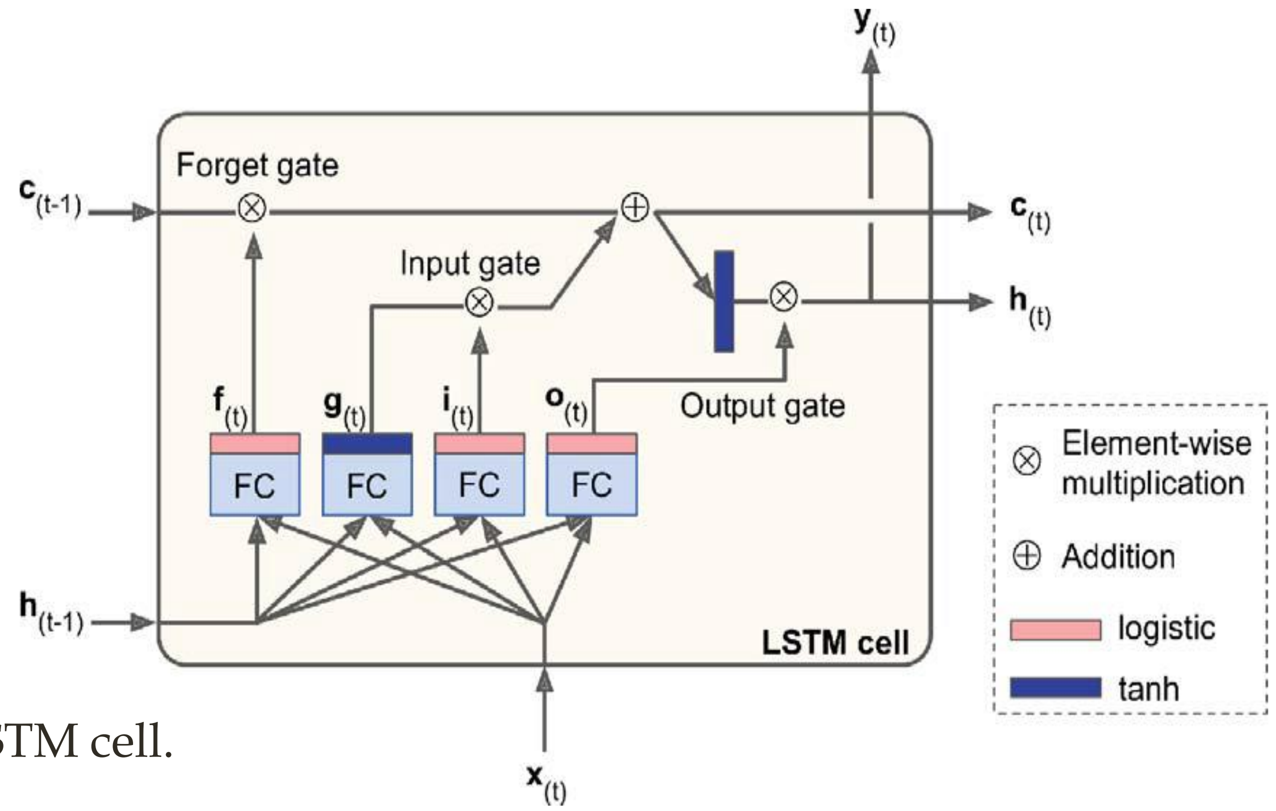


Figure 9: LSTM cell.

- 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.

…Tackling the Short-Term Memory Problem

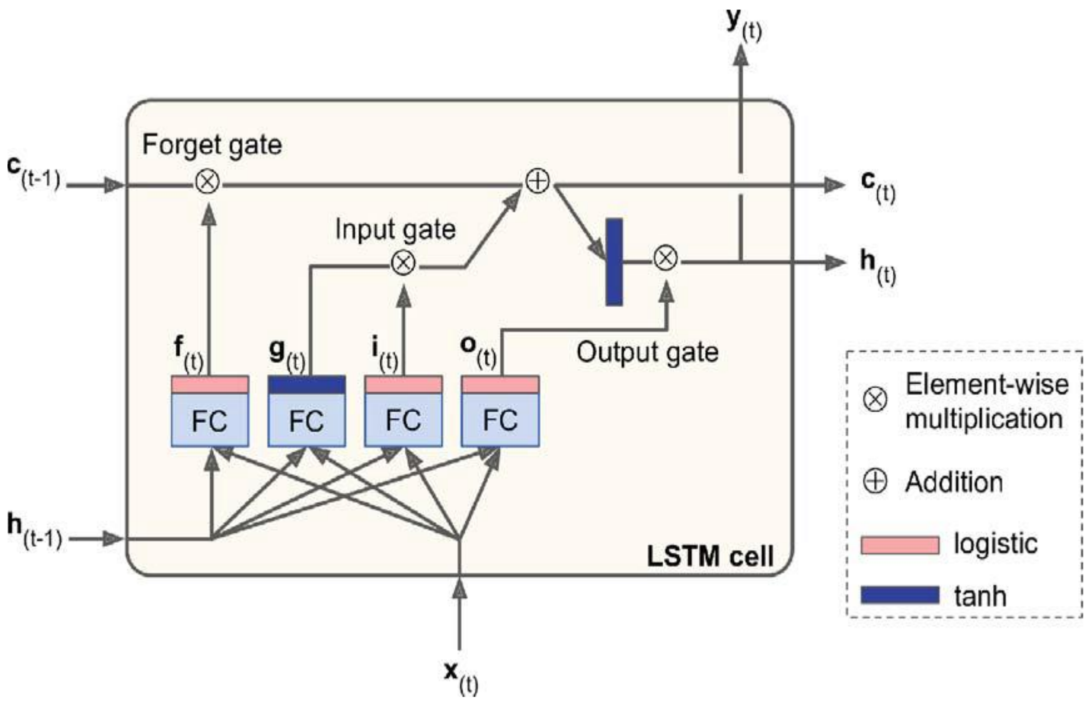


Figure 9: LSTM cell.

- As the long-term state $\mathbf{c}_{(t-1)}$ traverses the network from left to right,
- we can see that it first goes through a forget gate, dropping some memories, and
- then it adds some new memories via the addition operation (which adds the memories that were selected by an input gate).
- The result $\mathbf{c}_{(t)}$ is sent straight out, without any further transformation.

…Tackling the Short-Term Memory Problem

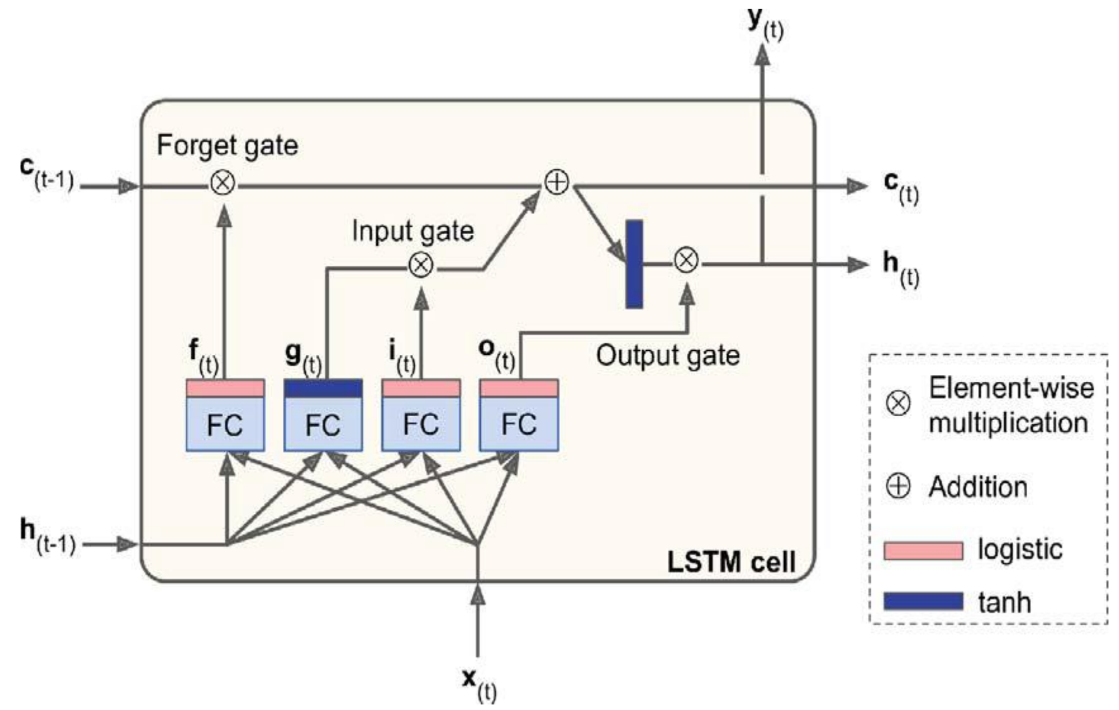


Figure 9: LSTM cell.

- ➢ So, at each time step, some memories are dropped, and some memories are added.
- ➢ 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.
- ➢ This produces the short-term state $\mathbf{h}_{(t)}$ (which is equal to the cell's output for this time step, $\mathbf{y}_{(t)}$).

Now let us look at where new memories come from and how the gates work.

…Tackling the Short-Term Memory Problem

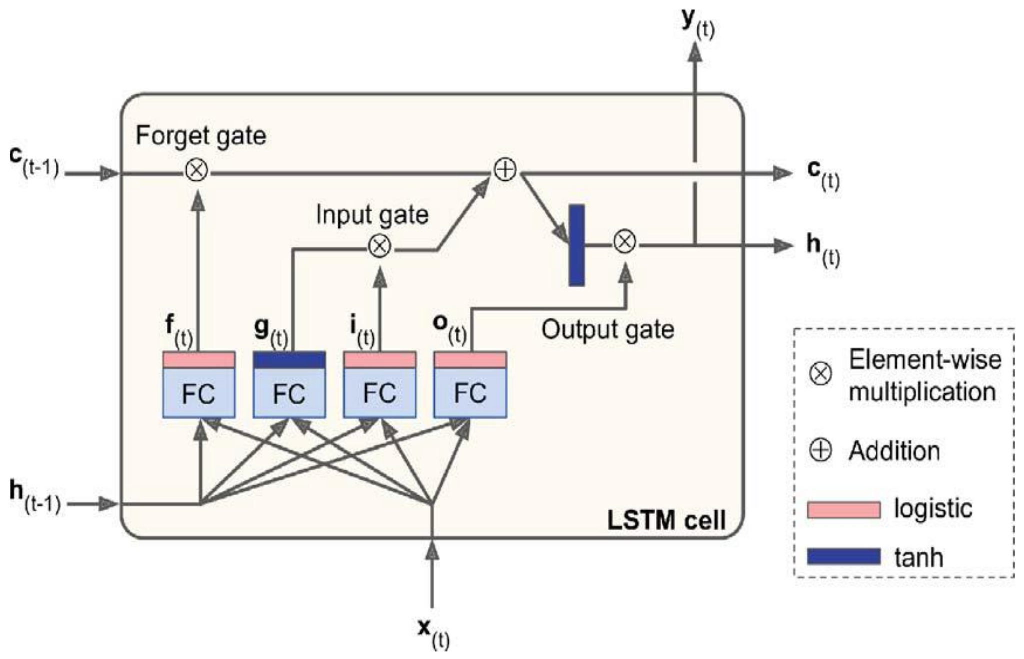


Figure 9: LSTM cell.

First, the current input vector $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$ are fed to four different fully connected layers. They all serve a different purpose:

- The main layer is the one that outputs $\mathbf{g}_{(t)}$.
  - It has the usual role of analyzing the current inputs $\mathbf{x}_{(t)}$ and the previous (short-term) state $\mathbf{h}_{(t-1)}$.
  - In a basic cell, there is nothing other than this layer, and its output goes straight out to $\mathbf{y}_{(t)}$ and $\mathbf{h}_{(t)}$.
  - In contrast, in an LSTM cell this layer's output does not go straight out, but instead its most important parts are stored in the long-term state (and the rest is dropped).

…Tackling the Short-Term Memory Problem

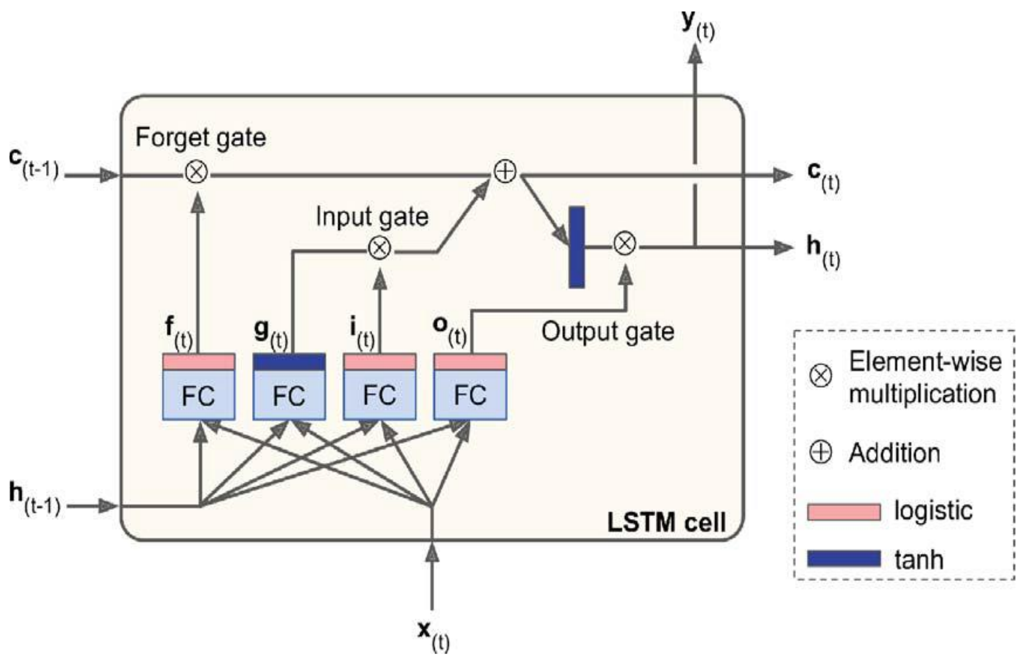


Figure 9: LSTM cell.

- The three other layers are gate controllers. Since they use the logistic (sigmoid) activation function, their outputs range from 0 to 1.
- As we can see, 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. Specifically:

— The forget gate (controlled by $\mathbf{f}_{(t)}$) controls which parts of the long-term state should be erased.

…Tackling the Short-Term Memory Problem

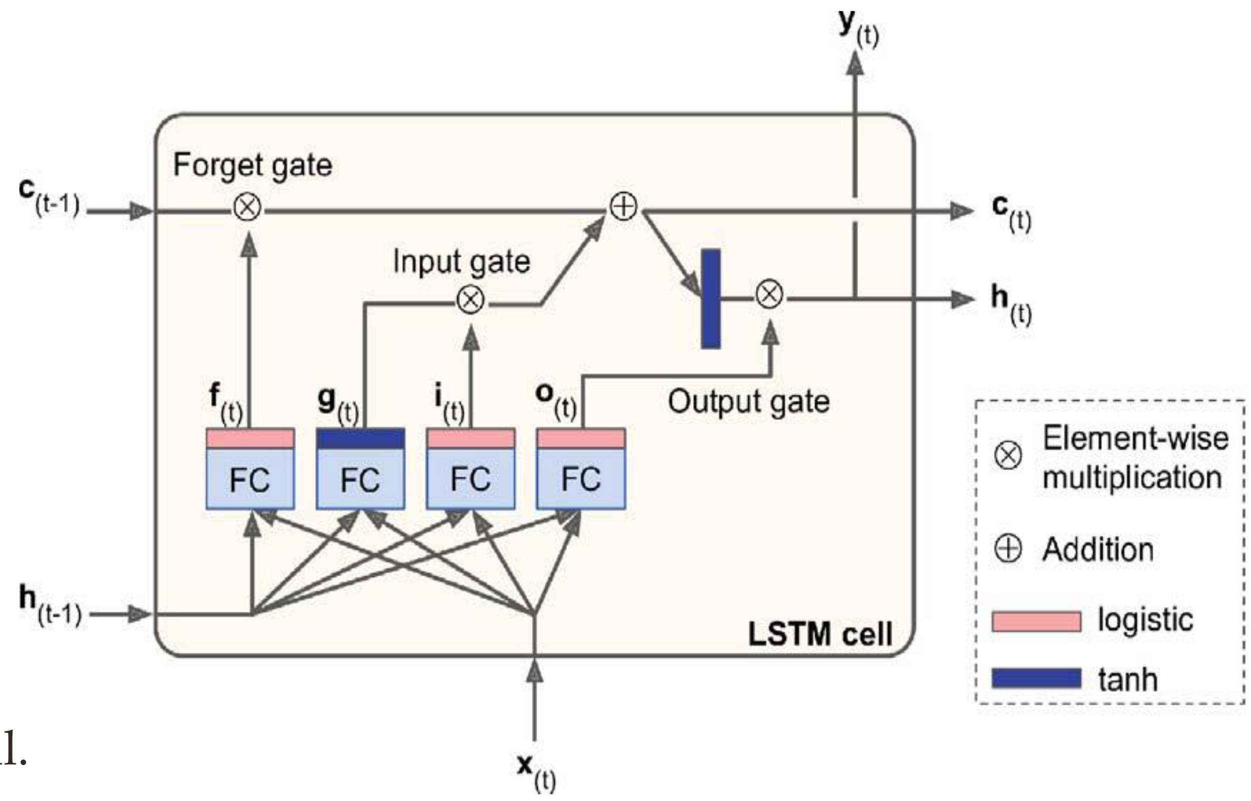


Figure 9: LSTM cell.

— The input gate (controlled by $\mathbf{i}_{(t)}$) controls which parts of $\mathbf{g}_{(t)}$ should be added to the long-term state.

— Finally, the output gate (controlled by $\mathbf{o}_{(t)}$) controls which parts of the longterm state should be read and output at this time step, both to $\mathbf{h}_{(t)}$ and to $\mathbf{y}_{(t)}$.

…Tackling the Short-Term Memory Problem

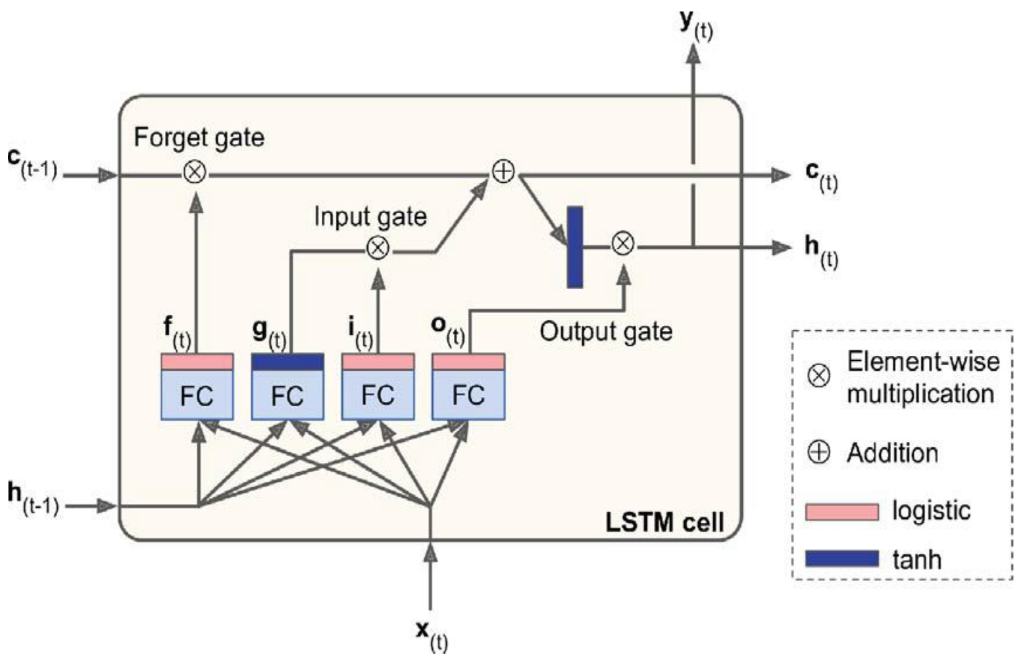


Figure 9: LSTM cell.

In short, an LSTM cell can

- learn to recognize an important input (that's the role of the input gate),
- store it in the long-term state, preserve it for as long as it is needed (that's the role of the forget gate), and
- extract it whenever it is needed.

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

…Tackling the Short-Term Memory Problem

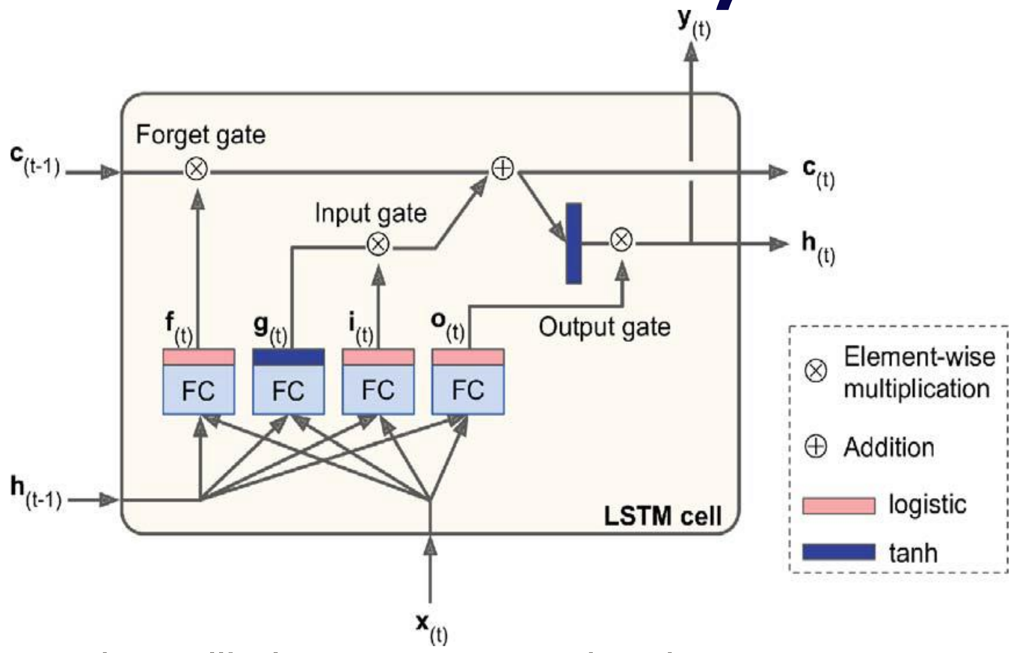


Figure 9: LSTM cell.

Equation 3 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). **Equation 3**:

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

…Tackling the Short-Term Memory Problem

Equation 3:

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

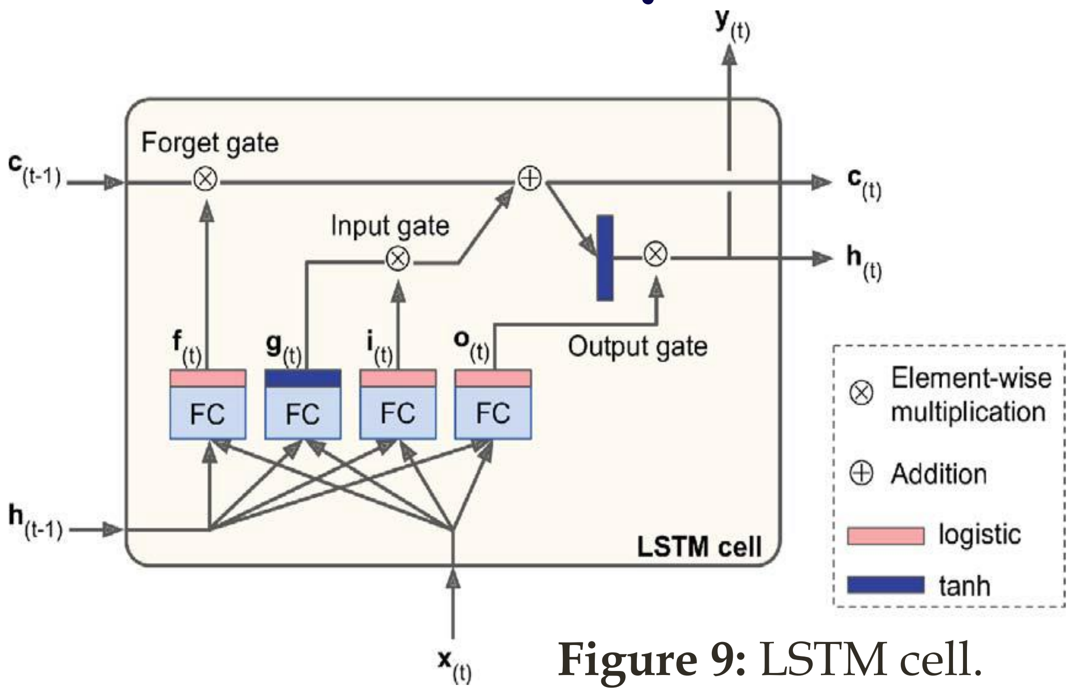


Figure 9: LSTM cell.

- $\mathbf{W}_{xi}$, $\mathbf{W}_{xf}$, $\mathbf{W}_{xo}$, $\mathbf{W}_{xg}$ are the weight matrices of each of the four layers for their connection to the input vector $\mathbf{x}_{(t)}$.
- $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$, and $\mathbf{W}_{hg}$ are the weight matrices of each of the four layers for their connection to the previous short-term state $\mathbf{h}_{(t-1)}$.
- $\mathbf{b}_{i}$, $\mathbf{b}_{f}$, $\mathbf{b}_{o}$, and $\mathbf{b}_{g}$ are the bias terms for each of the four layers. Note that TensorFlow initializes $\mathbf{b}_{f}$ to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

LSTM: Peephole Connection

- In a regular 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.
- LSTM variant was proposed (see paper 1) 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.
  - This often improves performance, but not always, and there is no clear pattern for which tasks are better off with or without them:
    - we will have to try it on our task and see if it helps.

Gated Recurrent Unit (GRU) Cell

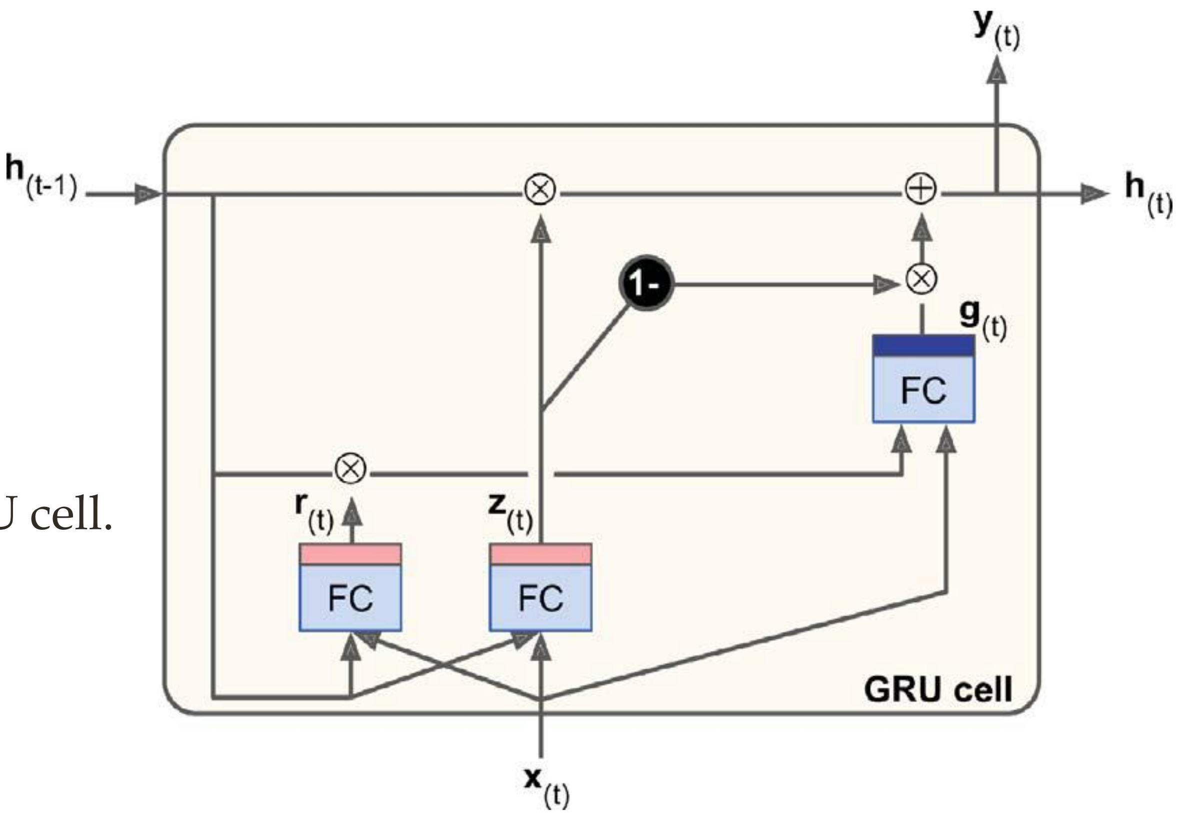


Figure 10: GRU cell.

The GRU cell is a simplified version of the LSTM cell, and it seems to perform just as well (see paper 1-2). These are the main simplifications:

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

Gated Recurrent Unit (GRU) Cell

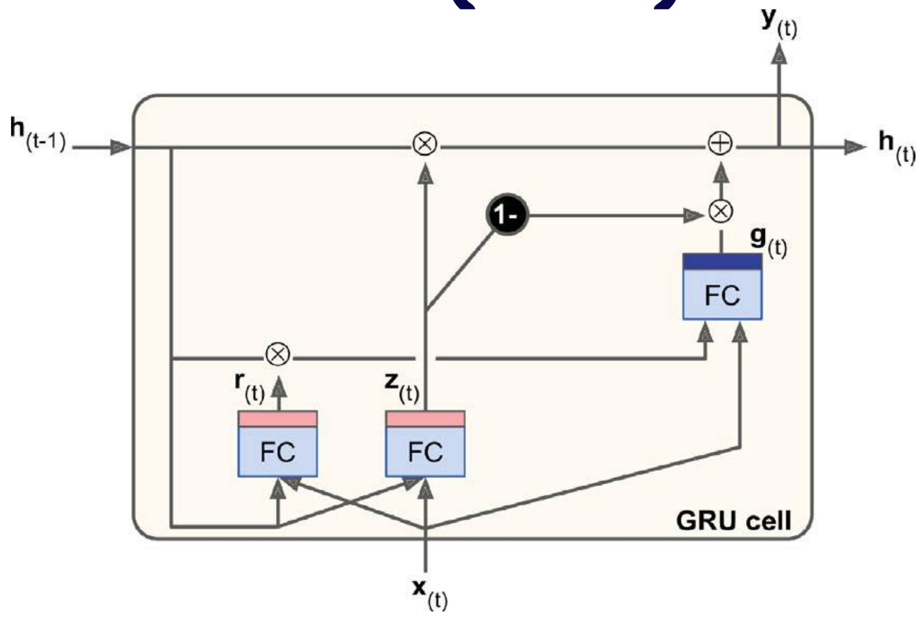


Figure 10: GRU cell.

- A single gate controller $\mathbf{z}_{(t)}$ controls both the forget gate and the input gate.
  - If the gate controller outputs a 1, the forget gate is open (= 1) and the input gate is closed (1 – 1 = 0).
  - 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.
- There is no output gate; the full state vector is output at every time step. However, there is a new gate controller $\mathbf{r}_{(t)}$ that controls which part of the previous state will be shown to the main layer ($\mathbf{g}_{(t)}$).

Gated Recurrent Unit (GRU) Cell

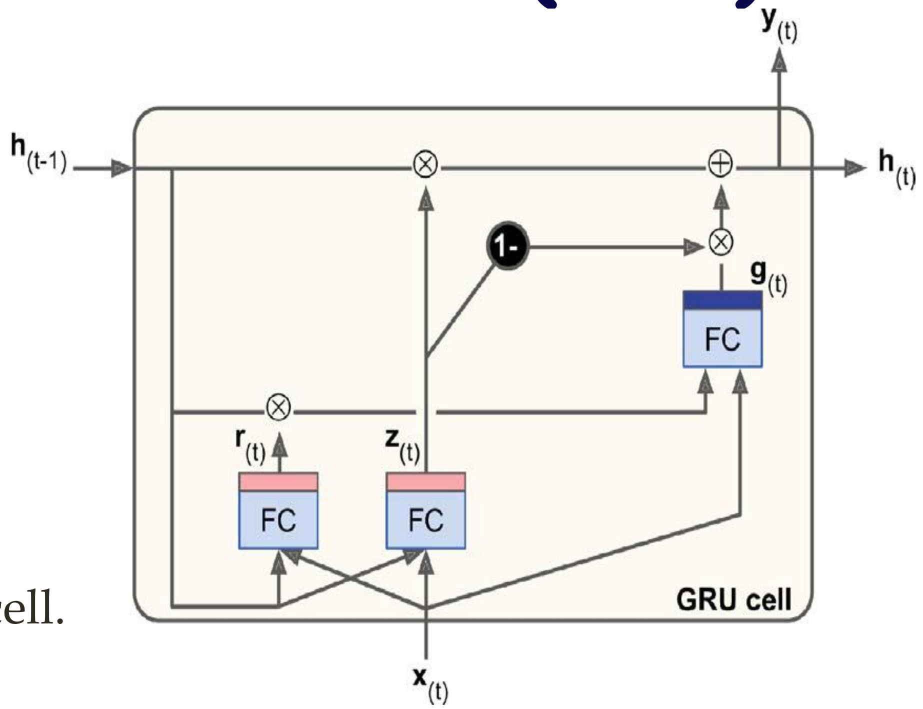


Figure 10: GRU cell.

- Equation 4 summarizes how to compute the cell's state at each time step for a single instance. Equation 4:

$$
\begin{aligned}
\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}{}^{\mathsf{T}}\mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{\mathsf{T}}\mathbf{h}_{(t-1)} + \mathbf{b}_z\right) \\
\mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}{}^{\mathsf{T}}\mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{\mathsf{T}}\mathbf{h}_{(t-1)} + \mathbf{b}_r\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{\mathsf{T}}\mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{\mathsf{T}}\left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right) + \mathbf{b}_g\right) \\
\mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{g}_{(t)}
\end{aligned}
$$

Using 1D Convolutional Layers to Process Sequences

- LSTM and GRU cells are one of the main reasons behind the success of RNNs.
- Yet while they can tackle much longer sequences than simple RNNs,
- they still have a fairly limited short-term memory, and they have a hard time learning long-term patterns in sequences of 100 time steps or more, such as audio samples, long time series, or long sentences.
- One way to solve this is to shorten the input sequences, for example using 1D convolutional layers:
  - 2D convolutional layer works by sliding several fairly small kernels (or filters) across an image, producing multiple 2D feature maps (one per kernel).
  - Similarly, a 1D convolutional layer slides several kernels across a sequence, producing a 1D feature map per kernel.

… Using 1D Convolutional Layers to Process Sequences

- Each kernel will learn to detect a single very short sequential pattern (no longer than the kernel size).
- If we use 10 kernels, then the layer's output will be composed of 10 1-dimensional sequences (all of the same length), or equivalently we can view this output as a single 10-dimensional sequence.
- This means that you can build a neural network composed of a mix of recurrent layers and 1D convolutional layers (or even 1D pooling layers).
- If we use a 1D convolutional layer with a stride of 1 and "same" padding, then the output sequence will have the same length as the input sequence.
- But if we use "valid" padding or a stride greater than 1, then the output sequence will be shorter than the input sequence, so make sure we adjust the targets accordingly.

… Using 1D Convolutional Layers to Process Sequences

- For example, the following model is the same as earlier, except it starts with a 1D convolutional layer that downsamples the input sequence by a factor of 2, using a stride of 2.
- The kernel size is larger than the stride, so all inputs will be used to compute the layer’s output, and therefore the model can learn to preserve the useful information, dropping only the unimportant details.
- By shortening the sequences, the convolutional layer may help the GRU layers detect longer patterns.
- Note that we must also crop off the first three time steps in the targets (since the kernel’s size is 4, the first output of the convolutional layer will be based on the input time steps 0 to 3), and downsample the targets by a factor of 2 (**see exercise**).

WaveNet

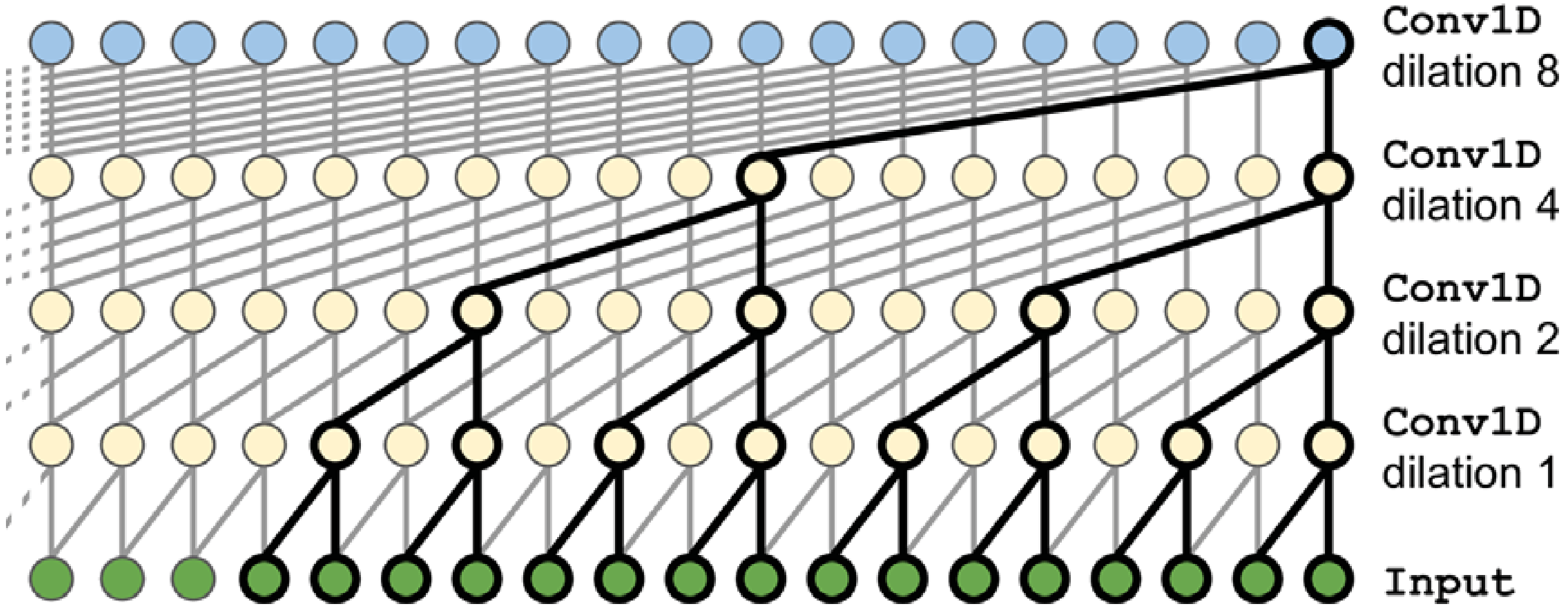


Figure 11: WaveNet architecture.

- WaveNet is a stacked 1D convolutional layers, doubling the dilation rate (how spread apart each neuron's inputs are) at every layer:
  - the first convolutional layer gets a glimpse of just two time steps at a time,
  - while the next one sees four time steps (its receptive field is four time steps long),
  - the next one sees eight time steps, and so on (see Figure 11).

… WaveNet

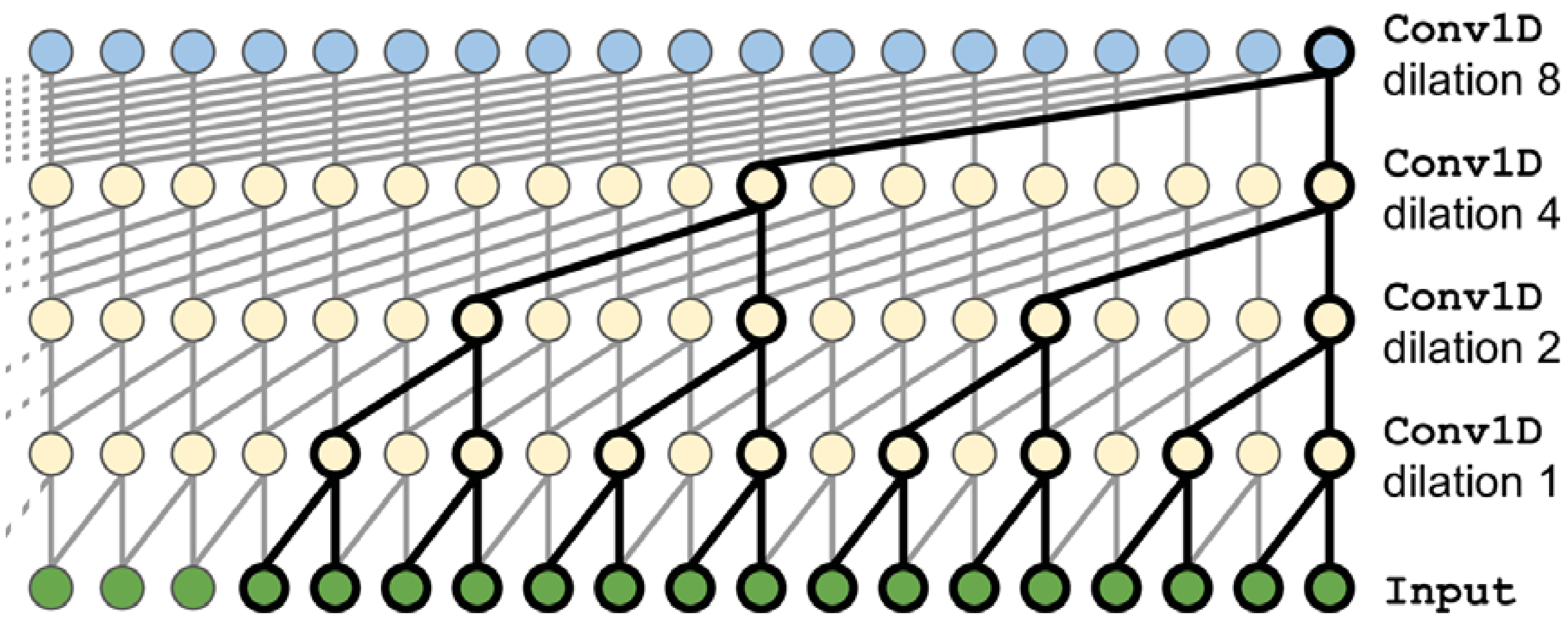


Figure 11: WaveNet architecture.

- This way, the lower layers learn short-term patterns, while the higher layers learn long-term patterns.
- Due to the doubling dilation rate, the network can process extremely large sequences very efficiently.

… WaveNet

- In the WaveNet paper (see #1) , the authors actually stacked 10 convolutional layers with dilation rates of 1, 2, 4, 8, …, 256, 512,
- then they stacked another group of 10 identical layers (also with dilation rates 1, 2, 4, 8, …, 256, 512),
- then again another identical group of 10 layers.
- They justified this architecture by pointing out that a single stack of 10 convolutional layers with these dilation rates will act like a super-efficient convolutional layer with a kernel of size 1,024 (except way faster, more powerful, and using significantly fewer parameters), which is why they stacked 3 such blocks.
- They also left-padded the input sequences with a number of zeros equal to the dilation rate before every layer, to preserve the same sequence length throughout the network.
- The complete WaveNet uses a few more tricks, such as skip connections like in a ResNet, and Gated Activation Units similar to those found in a GRU cell (**see exercise**).

… WaveNet

- The last two models offer the best performance so far in forecasting our time series!

- In the WaveNet paper, the authors achieved state-of-the-art performance on various audio tasks (hence the name of the architecture), including text-to-speech tasks, producing incredibly realistic voices across several languages.

- They also used the model to generate music, one audio sample at a time. This achievement is all the more impressive when we realize that a single second of audio can contain tens of thousands of time steps—even LSTMs and GRUs cannot handle such long sequences.

END.