Intelligent Systems: Reasoning and Recognition

James L. Crowley

ENSIMAG 2 — Winter Semester 2020-2021
Lessons 13 — 24 March 2021

Recurrent Neural Networks

Outline

Sources

1) Goodfellow, I., Bengio, Y., and Courville, A., Deep learning. MIT press, 2016.
2) Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J. L., editors, Parallel Distributed Processing, volume 1, pages 318–362. MIT Press.
3) Understanding LSTM Networks - Christopher Olah (https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Notation

$\vec{h}^{(t)}$	The hidden recurrent activation vector of the network. Note that in previous lectures we used $\vec{a}^{(t)}$ for activation.
$f(-)$	A process equation that computers $\vec{h}^{(t+1)}$ from $\vec{h}^{(t)}$
$\vec{X}^{(t)}$	A sequence of τ input vectors. Equivalent to $\{\vec{X}_1,...,\vec{X}_\tau\}$ in earlier lectures.
$\vec{o}^{(t)}$	The network output vector.
U	a weight matrix from the input to the hidden unit.
W	a hidden-to-hidden layer recurrent weight matrix
V	a hidden-layer-to-output weight matrix
$\vec{b}$	bias vector for hidden units
$\vec{c}$	bias vector for the output units.
$\vec{X}_m^{(t)}$	A training set of *M* input sequences.
$\vec{y}_m^{(t)}$	A set of *M* target output sequences for the training set.

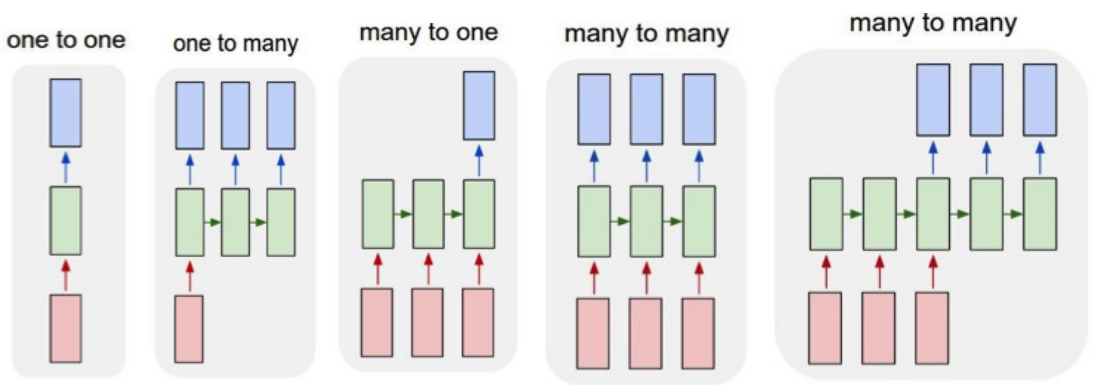
For LSTMs:

f_t	Forget gate activation vector.	$f_t = \sigma_g(W_f\vec{X}^{(t)} + U_f h_{t-1} + b_f)$
i_t	Input/update gate activation vector	$i_t = \sigma_g(W_i\vec{X}_t + U_i h_{t-1} + b_i)$
o_t	Output gate activation vector.	$o_t = \sigma_g(W_o\vec{X}_t + U_o h_{t-1} + b_o)$
$\tilde{c}_t$	Cell Input activation vector.	$\tilde{c}_t = \tanh_c(W_c\vec{X}_t + U_c h_{t-1} + b_c)$
c_t	Cell state vector:	$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_{t-1}$
$\vec{h}_t$	Hidden State/Output activation vector:	$\vec{h}_t = o_t \circ o_h(c_t)$

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are used to discriminate and generate data that have an intrinsic order relation (sequences). Examples of sequences that may be discriminated and generated with RNNs include Speech, Music, Text, and Time Series data. RNNs can be combined with convolutional networks to recognize and generate video sequences of actions. RNNs have been traditionally used for natural language processing including for understanding written text and machine translation, although they are rapidly being replaced with Transformer using Self-Attention.

Recurrent Networks are Turing Universal, which means that any function that can be computed by a Turing machine can be computed by a recurrent network.



Copied from Andrej Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks", http://karpathy.github.io/2015/05/21/rnn-effectiveness/

History

In the early days of neural networks (1980's), a frequent criticism was that networks have no memory, other than the parameter learning. It was said that because networks did not maintain temporal state, they could not be suitable for tasks involving temporal or spatial sequences.

In the late 1980s, Rumelhart addressed this question by building on a class of completely connected networks proposed by Hopfield, leading to the idea of "unfolding" the network over time. Such networks are now called recurrent neural networks.

A recurrent neural network (RNN) is a neural network where connections between nodes form a directed graph along a temporal sequence. This enables the network to exhibit temporal dynamic behavior. RNNs can use internal state (memory) to process variable length sequences of inputs. This makes them applicable to tasks such as handwriting recognition or speech recognition.

Finite vs Infinite impulse networks

The term “recurrent neural network” refers to two broad classes of networks finite impulse and infinite impulse. Both classes exhibit temporal dynamic behavior.

Finite Impulse: A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feed-forward neural network. The temporal dynamics are similar to a Finite Impulse Response (FIR) digital filter. In digital signal processing, FIR filters are known to be easy to design, stable, but limited in the duration of their response.

Infinite impulse: An infinite impulse recurrent network is a directed cyclic graph that cannot be unrolled because of internal feedback. These have similar temporal dynamics to Infinite Impulse Response (IIR) digital filters. In digital signal processing, IIR filters are known to be difficult to design, unstable, but very powerful and efficient. The classic Infinite Impulse Recurrent network is the LSTM (Long-Short-Term Memory) architecture.

Both finite impulse and infinite impulse recurrent networks can have additional states, and storage can be under direct control of the network. The storage can also be replaced by another network or graph. Such controlled states are referred to as gated states or gated memory, and are a key part of gated recurrent units including long short-term memory (LSTMs) networks.

Finite Impulse Recurrent Networks

The classic model for a dynamic process is a function, $f(-)$, that predicts the state, *s(t)* of a system at time *t*, from the state *s(t-1)* at time *t-1*, using parameters $\vec{w}$. Such as process is known as a "markov" process.

$$\begin{tikzcd} {} \arrow[r, "f_W(-)"] & S^{(t-1)} \arrow[r, "f_W(-)"] & S^{(t)} \arrow[r, "f_W(-)"] & S^{(t-2)} \arrow[r, "f_W(-)"] & S^{(t+3)} \arrow[r, "f_W(-)"] & {} \end{tikzcd}$$

$$s^{(t)} = f_{\vec{w}}(s^{(t-1)})$$

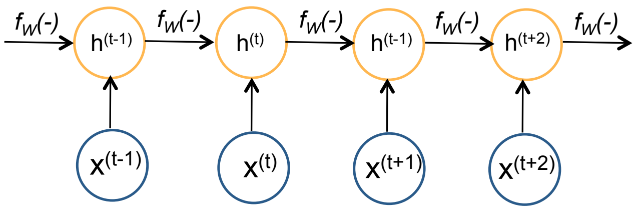
In the case of a recurrent network, the "state" is the activation (or vector of activations) of one or more "hidden" units. In previous lectures we represented the activation state of a cell with the symbol *a*. In the recurrent network literature, activation is generally represented with a state variable $h^{(t)}$

$$\begin{tikzcd} {} \arrow[r, "f_W(-)"] & h^{(t-1)} \arrow[r, "f_W(-)"] & h^{(t)} \arrow[r, "f_W(-)"] & h^{(t-1)} \arrow[r, "f_W(-)"] & h^{(t+2)} \arrow[r, "f_W(-)"] & {} \end{tikzcd}$$

$$h^{(t)} = f_{\vec{w}}(h^{(t-1)})$$

The time variable is traditionally represented with a superscript, to keep it apart from the unit indices at each level.

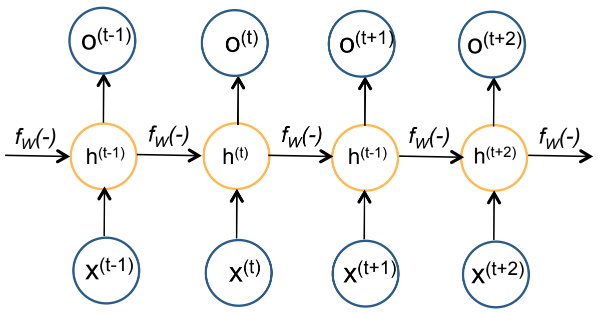
We can model the effects of an external input by adding an additional term, $x^{(t)}$, to the temporal transition function.



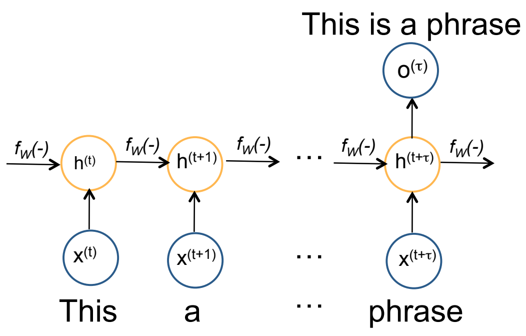
$$h^{(t)} = f_{\vec{w}}(h^{(t-1)}, x^{(t)})$$

The temporal duration of the network is typically represented the variable τ, so that the network is said to operate on a temporal sequence $x^{(t)}$ from $t=1$ to τ.

Normally, the network generates an output represented by an output variable, $o^{(t)}$.

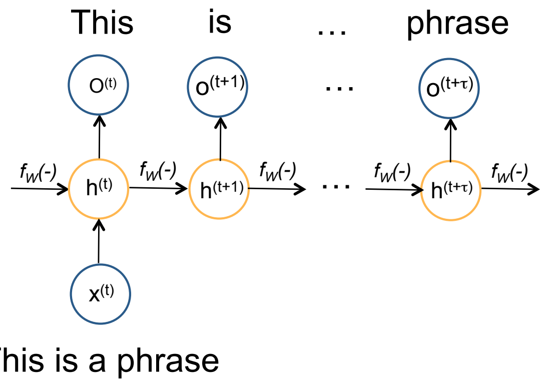


For example, in a many-to-one network, the network would produce a single output after τ time steps. For example, the following network assembles the words "This", "is", "a", and "phrase", into a single output "This is a phrase". In this case, t is the number of words in the phrase, 4.



A one-to-many network would produce a sequence of τ outputs from a single input.

For example, a single symbol for "This is a phrase" can be expanded into a sequence of outputs, where τ = 4.



Tokenizing Word Data

In the above examples, RNNs are used for processing words. RNNs are the widely used in Natural Language Processing (NLP). In order to process words and text with a network, it is necessary to "tokenize" the text by substituting natural numbers for fundamental entities of the text. These entities can be letters, syllables, word roots with prefixes and suffixes, or words and punctuation symbols or sequences of words. Generally a dictionary is created for a domain in which each word and punctuation mark are be given a unique integer ID.

A tokenizer uses a word index to convert each sentence to a vector.

For example, here is a text and its token table:

Sentence: "This is an example of a sentence in English."

Tokenized sentence: (1,2,3,4,5,6,7,8,9,10)

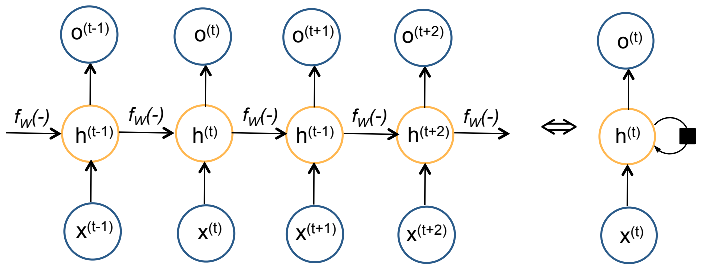
Produced using the token table:

{'This': 1, 'is': 2, 'an': 3, 'example': 4, 'of': 5, 'a': 6, 'sentence': 7, 'in': 8, 'English': 9, '.':10}

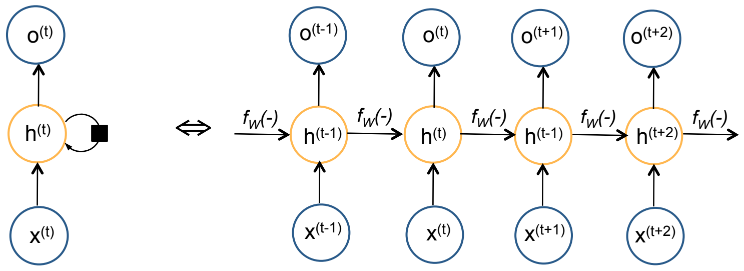
Normally a token table is produced for an entire language and not a sample sentence. We can tokenize symbolic input from any domain.

Folding and Unfolding

Recurrent networks are classically "folded" into a recurrent structure:



Where the black square represents a time delay of 1 time unit. The recurrent structure can be unfolded to see the network as a 2-D structure.



Forward Propagation Equations

Networks parameters include:

U	a weight matrix,
W	a hidden-to-hidden recurrent weight matrix
V	a hidden-to-output weight matrix
$\vec{b}$	bias vector for hidden units
$\vec{c}$	bias vector for the output units.

The classic forward propagation equations are:

$$\vec{z}^{(t)} = W\ \vec{h}^{(t)} + U\ \vec{X}^{(t)} + \vec{b}$$

$$\vec{h}^{(t)} = f(\vec{z}^{(t)}) = f(W\ \vec{h}^{(t-1)} + U\ \vec{X}^{(t-1)} + \vec{b})$$

$$\vec{o}^{(t)} = V\ \vec{h}^{(t)} + \vec{c}$$

where_ $\vec{X}^{(t)}$, $\vec{h}^{(t)}$, $\vec{z}^{(t)}$ and $\vec{o}^{(t)}$ are all vectors *U, V* and *W* are matrices and $\vec{b}$ and $\vec{c}$ are bias vectors. Such networks typically use a *tanh()* (hyperbolic tangent) activation function. The equations should be read as expressing summations over as a set of units at the same level. For example for the *i=1* to *N* units of $h^{(t)}$ and $h^{(t-1)}$:

$$\vec{z}^{(t)} = W\,\vec{h}^{(t)} + U\,\vec{X}^{(t)} + \vec{b} \qquad \Leftrightarrow \qquad z_j^{(t)} = \sum_{i=1}^{N} W_{ij}^{(t)} h_i^{(t-1)} + \sum_{i=1}^{N} U_{ij}^{(t)} X_i^{(t)} + b_j$$

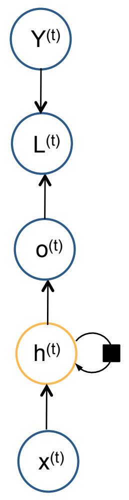
Classically recurrent networks are used to generate symbolic data such as words or characters. In this case, the output vector, $\vec{o}^{(t)}$, can be seen as an un-normalized log probability of each possible value of the discrete variable. Softmax can then be used to obtain the vector $\hat{y}^{(t)}$ of normalized probabilities for the output.

$$\hat{y}^{(t)} = \mathrm{softmax}(\vec{o}^{(t)})$$

Forward propagation starts from an initial state $\vec{h}^{(0)}$, and then computes the recurrent states from $t = 1$ to $t = \tau$.

Training

As with classic networks, a recurrent network is trained to minimize the Loss (or cost) between a target vector $\vec{y}_m^{(t)}$, and an output vector $\vec{o}_m^{(t)}$ generated from an input sequence $\vec{X}_m^{(t)}$. This is represented as $L_m^{(t)}$:



Where the Loss $L_m^{(t)}$ measures how far $\hat{y}_m^{(t)} = \mathrm{softmax}(o_m^{(t)})$ is from the target $\vec{y}_m^{(t)}$ at each time (t). The total loss L for each training sample is computed for an input sequence $\{x^{(1)},\dots,x^{(\tau)}\}$ and the resulting output sequence $\{y^{(1)},\dots,y^{(\tau)}\}$

$$L\left(\{\vec{X}^{(1)},\dots,\vec{X}^{(\tau)}\},\{y^{(1)},\dots,y^{(\tau)}\}\right) = \sum_{t=1}^{\tau} L^{(t)} = -\sum_{t=1}^{\tau} Log\left(p\left(y^{(t)} \mid \{\vec{X}^{(1)},\dots,\vec{X}^{(\tau)}\}\right)\right)$$

Computing the gradient of this loss function with respect to the parameters is an expensive operation.

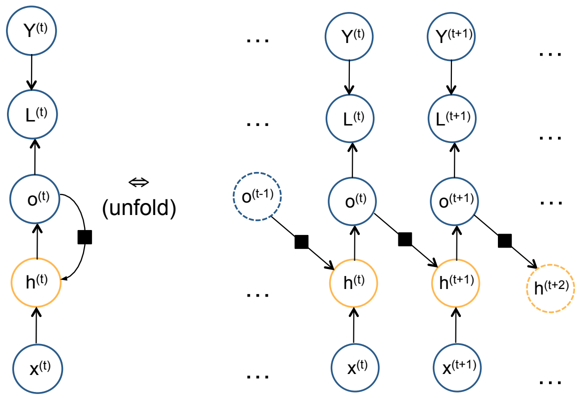
The gradient computation involves performing a forward propagation pass moving left to right through the unfolded graph of τ units, followed by a backward propagation pass moving right to left through the graph.

The algorithmic complexity is O(τ) and cannot be reduced by parallelization because the forward propagation graph is inherently sequential. Each time step may only be computed after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also O(τ) (O(-) is the order-of operator). The back-propagation algorithm applied to the unfolded graph with O(τ) cost is called back-propagation through time or BPTT.

In summary: Computing the gradient of this loss function is expensive because

1) The forward propagation followed by backward propagation operates on all τ samples in parallel.
2) Run-time cost is O(τ) and can not be implemented in parallel.
3) Memory cost is also O(τ).
4) Back-propagation must be applied to the entire unfolded graph. This is called "Back Propagation Through Time" (BPTT).

To reduce the cost of training we can use a network where the recurrence relation is from output to hidden.



The equations for the above network are

$$\vec{z}^{(t)} = W\ \vec{o}^{(t-1)} + U\ \vec{X}^{(t)} + \vec{b}$$

$$\vec{h}^{(t)} = \tanh(\vec{z}^{(t)}) = \tanh(W\ \vec{o}^{(t-1)} + U\ \vec{X}^{(t)} + \vec{b})$$

$$\vec{o}^{(t)} = V\ \vec{h}^{(t)} + \vec{c}$$

$$\hat{y}^{(t)} = \text{softmax}(\vec{o}^{(t)})$$

Such a network is less powerful, then the general network described above, but easier to train because each time step can be trained in isolation of the others, allowing for greater parallelization during training.

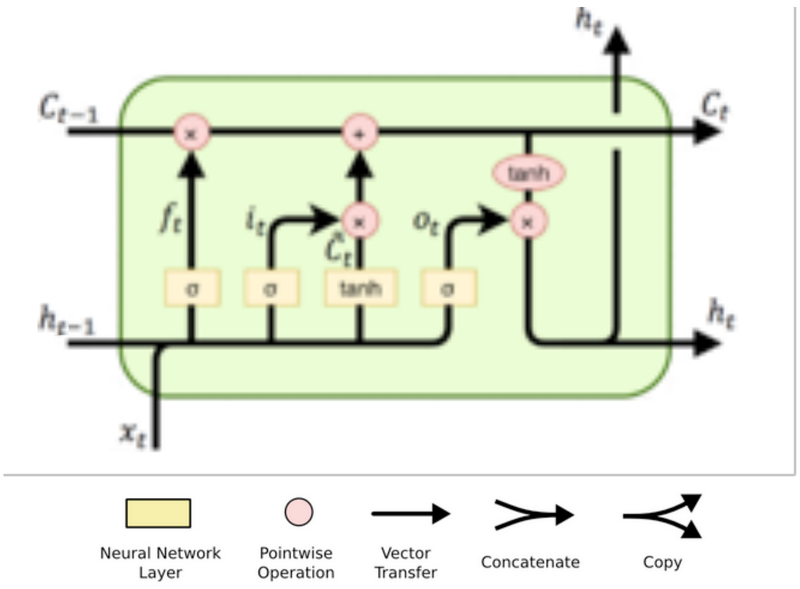
Long Short-Term Memory (LSTM)

In theory, RNNs can keep track of arbitrary long-term dependencies in an input sequences. However, this generally proves impractical because of a problem known as the "vanishing gradient" problem. When training a normal RNN using back-propagation, the gradients which are back-propagated can tend to zero (vanish) or diverge to infinity (explode), because of the accumulation of errors resulting from computation with finite-precision numbers. Long short-term memory (LSTM) provide a solution to this problem.

A long short-term memory (LSTM) is a form of RNN with a recursive memory structure. LSTM networks are well suited to classifying, processing and making predictions based on time series data, including series with lags of unknown duration between important events. LSTM are appropriate for long temporal sequences of such as speech or video, and have been used to build systems for unsegmented, connected handwriting recognition, speech recognition and anomaly detection in network traffic .

LSTMs use feedback connections. As with a recursive digital filter feedback enable design of a compact, powerful structure that can represent an arbitrarily long (potentially infinite) temporal duration, but can easily result in instability. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.

LSTMs were developed to deal with the vanishing gradient problem that can be encountered when training traditional RNNs. Relative insensitivity to gap length is an advantage of LSTM over RNNs, hidden Markov models and other sequence learning method. RNNs using LSTM units partially solve the vanishing gradient problem, because LSTM units allow gradients to also flow unchanged. However, LSTM networks can still suffer from the exploding gradient problem.



LSTM with a forget gate
Figures are taken from: Understanding LSTM Networks - Christopher Olah (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)

In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent point-wise operations, like vector addition, while the yellow boxes are learned neural network layers. Merging lines denote concatenation, while a forking line denotes copies of the vector going to different locations.

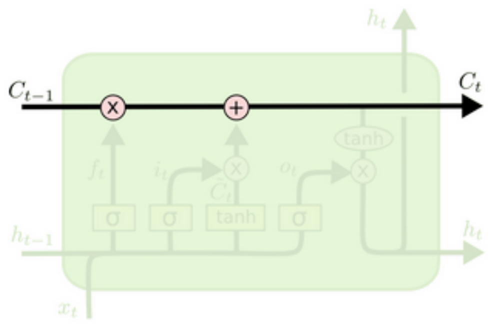
The compact forms of the equations for the forward pass of an LSTM unit with a forget gate are:

Input vector to the LSTM unit:	$\vec{X}^{(t)}$
Forget gate activation vector:	$f_t = \sigma_g(W_f \vec{X}^{(t)} + U_f \vec{h}_{t-1} + b_f)$
Input/update gate activation vector:	$\vec{i}_t = \sigma_g(W_i \vec{X}_t + U_i \vec{h}_{t-1} + b_i)$
Output gate activation vector:	$\vec{o}_t = \sigma_g(W_o \vec{X}_t + U_o \vec{h}_{t-1} + b_o)$
Cell Input activation vector:	$\tilde{c}_t = \tanh_c(W_c \vec{X}_t + U_c \vec{h}_{t-1} + b_c)$
Cell state vector:	$\vec{c}_t = f_t \circ \vec{c}_{t-1} + i_t \circ \tilde{c}_{t-1}$
Hidden State/Output activation vector:	$\vec{h}_t = o_t \circ o_h(c_t)$

where the operator ∘ denotes element-wise vector product (inner-product).

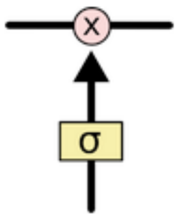
The Four layers of an LSTM unit

An LSTM unit can be seen as four interacting layers:



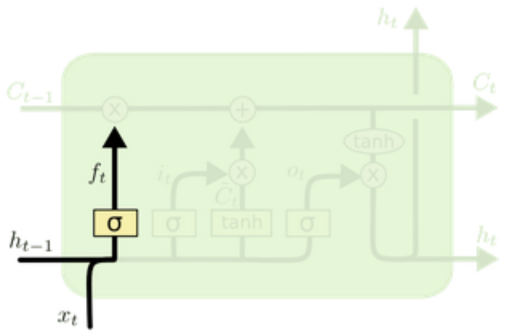
The horizontal line running through the top of the diagram represents the cell state. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

The LSTM has the ability to remove or add information to the cell state, using gates. Gates are a way to optionally pass or block information. Gates use a sigmoid activation followed by point-wise multiplication.



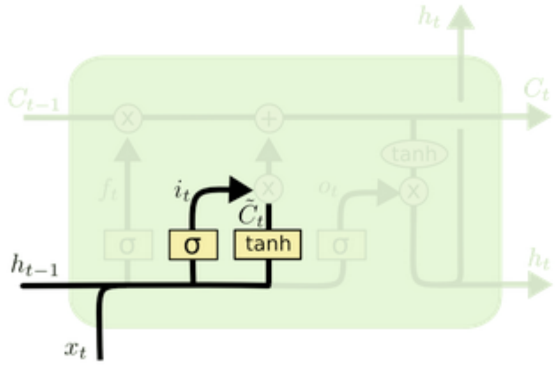
The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.

The first step in a LSTM is a sigmoid layer called the forget gate. The forget gate uses $\vec{h}_{t-1}$ and $\vec{X}^{(t)}$ to determine whether to transmit, attenuate or block an element of the cell state.



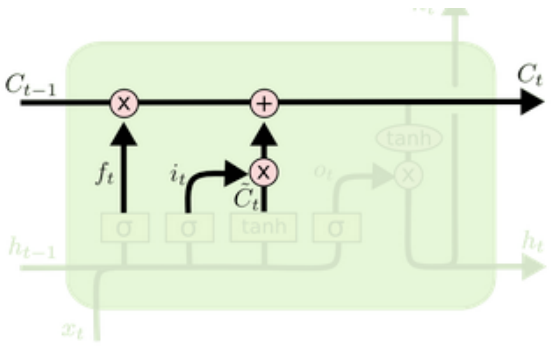
$$f_t = \sigma_g(W_f\vec{X}^{(t)} + U_f\vec{h}_{t-1} + b_f)$$

The next part determines information to be added to the cell state.



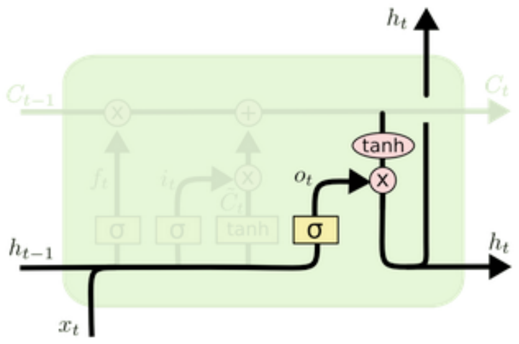
$$i_t = \sigma_g(W_i\vec{X}_t + U_i\vec{h}_{t-1} + b_i)$$
$$\tilde{c}_t = \tanh_c(W_c\vec{X}_t + U_c\vec{h}_{t-1} + b_c)$$

This has two parts. First, an "input gate layer" uses a sigmoid to determine which values to update. Then, a *tanh* layer creates a vector of new candidate values, C̃ t that may be added to the state.



$$\vec{c}_t = f_t \circ \vec{c}_{t-1} + \vec{i}_t \circ \tilde{c}_{t-1}$$

The cell state is then updated by multipling the previous state by ft to determine how much to forget (attentuate) the previous state, followed by the addition of the new state c_t .



$$\vec{o}_t = \sigma_g(W_o\vec{X}_t + U_o\vec{h}_{t-1} + b_o)$$
$$\vec{h}_t = \vec{o}_t \circ o_h(\vec{c}_t)$$

The output is then based on the a filtered version of the cell state, A sigmoid layer which decides what parts of the cell state to output. This is then modulated by a *tanh* to set scale the value between -1 and +1.