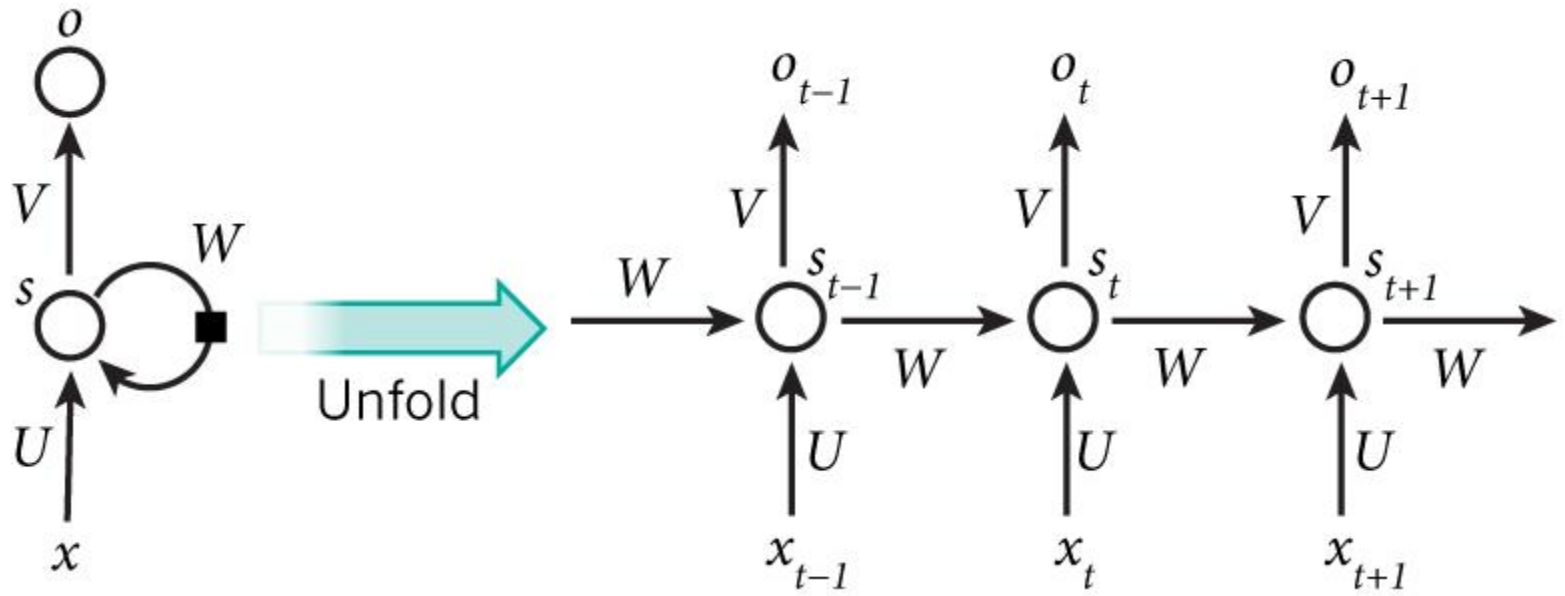
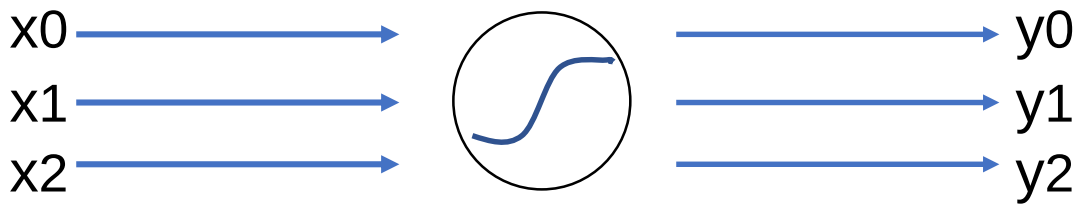
# Recurrent Neural Networks




Material from:
WildML (http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/)
Andrej Karpathy (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)
Trask (https://iamtrask.github.io/2015/11/15/anyone-can-code-lstm/)
Colah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)
Figure from *Nature*

# Recurrent Neural Networks

In a traditional neural network we assume that all inputs (and outputs) are independent of each other.




RNNs are called *recurrent* because they perform the same task for every element of a sequence, with the output being depended on the previous computations.

RNNs is have a “memory” which captures information about what has been calculated so far.

# Recurrent Neural Networks

- A glaring limitation of Vanilla Neural Networks is that they are too constrained: they accept a fixed-sized vector as input and produce a fixed-sized vector as output

- Not only that: These models perform this mapping using a fixed amount of computational steps (e.g. the number of layers in the model).

- The core reason that recurrent nets are more exciting is that they allow us to operate over *sequences* of vectors: Sequences in the input, the output, or in the most general case both.

# Enhancing Neural Nets with Memory

So how exactly can we endow our networks with the ability to memorize? To answer this question, let's recall our basic hidden layer neural network, which takes as input a vector `X`, dot products it with a weight matrix `W` and applies a nonlinearity. We'll consider the output `y` when three successive inputs are fed through the network. Note that the bias term has been eliminated so as to simplify the notation, and I've taken the liberty of coloring the equations to make certain patterns stand out.

$$y_0 = f(W_x X_0)$$

$$y_1 = f(W_x X_1)$$

$$y_2 = f(W_x X_2)$$

Given the simple API above, it's pretty clear that each output is solely determined by its input, i.e. there is no trace of past inputs in the calculation of its value. So let's alter the API by allowing our hidden layer to use a combination of both the current input and the previous input, and visualize what happens.

$$y_0 = f(W_x X_0)$$

$$y_1 = f(W_x X_1 + W_h X_0)$$

$$y_2 = f(W_x X_2 + W_h X_1)$$

Nice! By introducing recurrence into the formula, we've managed to obtain a mix of 2 colors in each hidden layer. Intuitively, our network now has a memory depth of 1, equivalent to "seeing" one step backwards in time. Remember though that our goal is to be able to capture information across **all** previous timesteps, so this does not cut it.

Hmm… What if we feed in a combination of the current input and the previous hidden layer?

$$y_0 = f(W_x X_0)$$

$$y_1 = f\big(W_x X_1 + W_h\ f(W_x X_0)\big)$$

$$y_2 = f\Big(W_x X_2 + W_h\ f\big(W_x X_1 + W_h\ f(W_x X_0)\big)\Big)$$

Much better! Our layer at each timestep is now a blend of all the colors that have come before it, allowing our network to take into account all its past history when computing its output. This is the power of recurrence in all its glory: creating a loop where information can persist across timesteps.

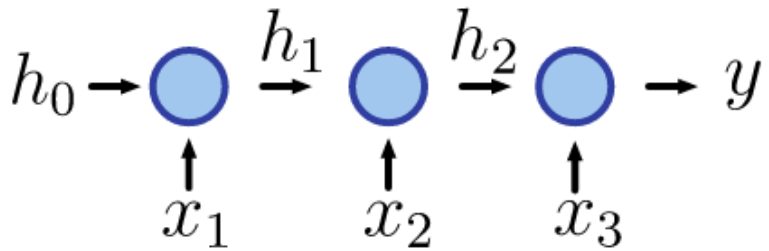


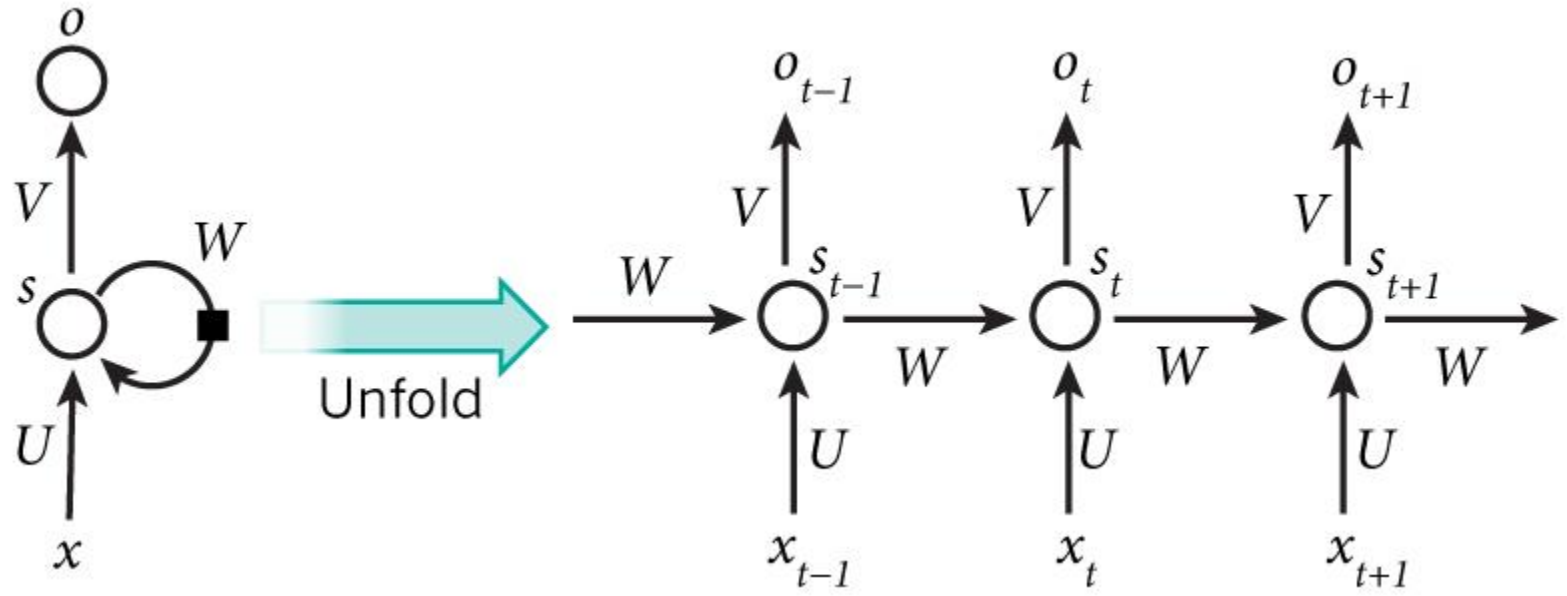

Image Courtesy

At its core, an RNN can be represented by an internal, hidden state `h` that gets updated with every timestep and from which an output `y` can be optionally derived[3]. This update behavior is governed by the following equations:

$$\begin{cases} h_t = f\left(W_{xh}x_t + W_{hh}h_{t-1} + b_1\right) \\ y_t = g\left(W_{hy}h_t + b_2\right) \end{cases}$$

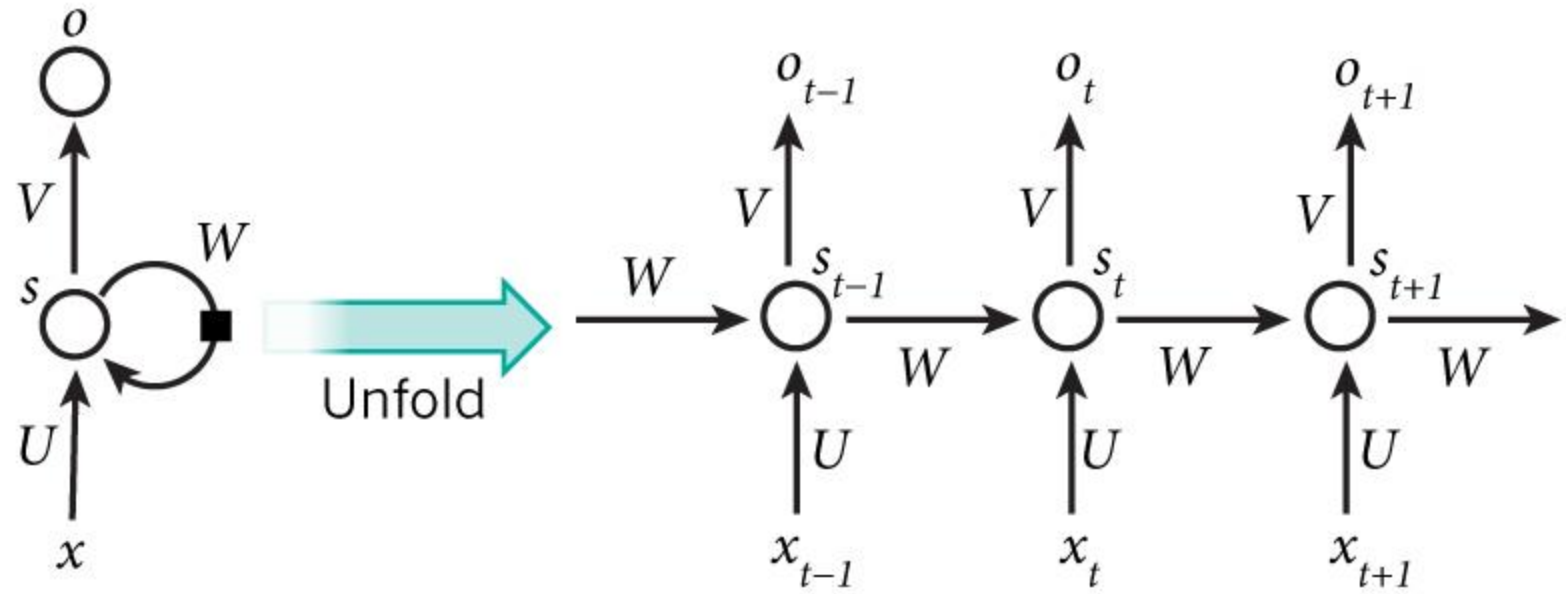
$$\begin{cases} h_t = f\left(W_{xh}x_t + W_{hh}h_{t-1} + b_1\right) \\ y_t = g\left(W_{hy}h_t + b_2\right) \end{cases}$$

- $W_{xh}x_t$ - we're multiplying the input $x_t$ by a weight matrix $W_{xh}$. You can think of this dot product as a way for the hidden layer to extract information out of the input.
- $W_{hh}h_{t-1}$ - this dot product is allowing the network to extract information from an entire history of past inputs which it will use in conjunction with information gathered from the current input, to compute its output. This is the crucial, self-defining property of RNNs.
- $f$ and $g$ are activation functions that squash the dot products to a specific range. The function $f$ is usually `tanh` or `ReLU`. $g$ can be a `softmax` when we want to output class probabilities.
- $b_1$ and $b_2$ are biases that help offset the outputs away from the origin (similar to the b in your typical $ax + b$ line).




- The above diagram shows a RNN being *unrolled* (or unfolded) into a full network. By unrolling we simply mean that we write out the network for the complete sequence.

- For example, if the sequence we care about is a sentence of 5 words, the network would be unrolled into a 5-layer neural network, one layer for each word.




- $x_t$ is the input at time step $t$. For example, $x_1$ could be a one-hot vector corresponding to the second word of a sentence.
- $s_t$ is the hidden state at time step $t$. It's the "memory" of the network. $s_t$ is calculated based on the previous hidden state and the input at the current step: $s_t = f(Ux_t + Ws_{t-1})$. The function $f$ usually is a nonlinearity such as tanh or ReLU. $s_{-1}$, which is required to calculate the first hidden state, is typically initialized to all zeroes.
- $o_t$ is the output at step $t$. For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary. $o_t = \text{softmax}(Vs_t)$.

# Language generation/modeling

- Let’s start by giving the RNN a huge chunk of text and ask it to model the probability distribution of the next character in the sequence given a sequence of previous characters.

- This will then allow us to generate new text one character at a time.

# Language generation/modeling

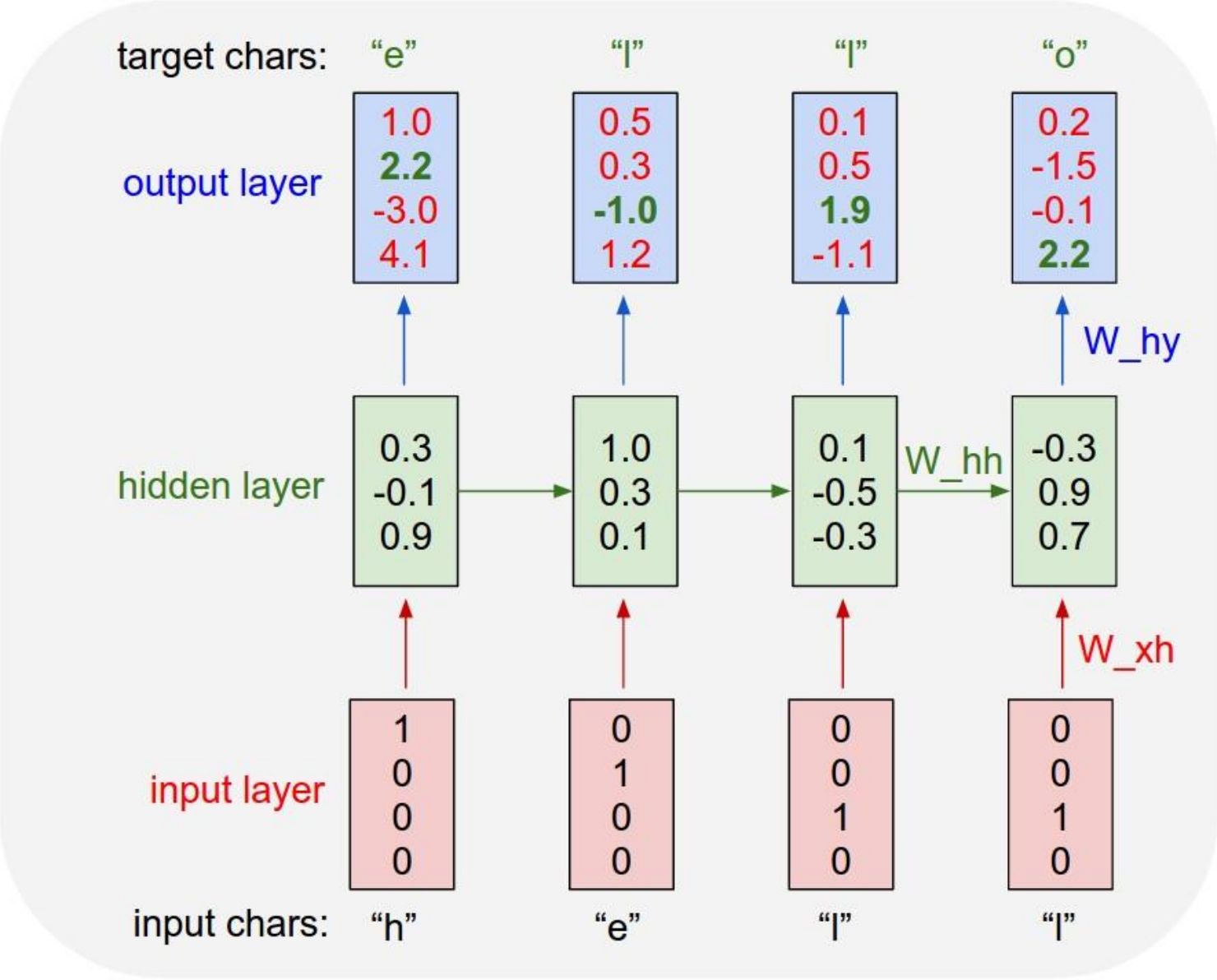
- As a working example, suppose we only had a vocabulary of four possible letters “helo”, and wanted to train an RNN on the training sequence “hello”.

- This training sequence is in fact a source of 4 separate training examples:
  - 1. The probability of “e” should be likely given the context of “h”
  - 2. “l” should be likely in the context of “he”
  - 3. “l” should also be likely given the context of “hel”
  - 4. “o” should be likely given the context of “hell”.

# Language generation/modeling

- 1. we will encode each character into a vector using 1-of-k encoding (i.e. all zero except for a single one at the index of the character in the vocabulary)
- 2. and feed them into the RNN one at a time with the step function
- 3. We will then observe a sequence of 4-dimensional output vectors (one dimension per character), which we interpret as the confidence the RNN currently assigns to each character coming next in the sequence.

# Language generation/modeling



# Language generation/modeling

- For example, we see that in the first time step when the RNN saw the character “h” it assigned confidence of 1.0 to the next letter being “h”, 2.2 to letter “e”, -3.0 to “l”, and 4.1 to “o”.

- Since in our training data (the string “hello”) the next correct character is “e”, we would like to increase its confidence (green) and decrease the confidence of all other letters (red).

- Similarly, we have a desired target character at every one of the 4 time steps that we’d like the network to assign a greater confidence to.

# Language generation/modeling

- Since the RNN consists entirely of differentiable operations we can run the backpropagation algorithm to figure out in what direction we should adjust every one of its weights to increase the scores of the correct targets (green bold numbers).

- We can then perform a parameter update, which nudges every weight a tiny amount in this gradient direction.

- If we were to feed the same inputs to the RNN after the parameter update we would find that the scores of the correct characters (e.g. “e” in the first time step) would be slightly higher (e.g. 2.3 instead of 2.2), and the scores of incorrect characters would be slightly lower. We then repeat this process over and over many times until the network converges.

# Language generation/modeling

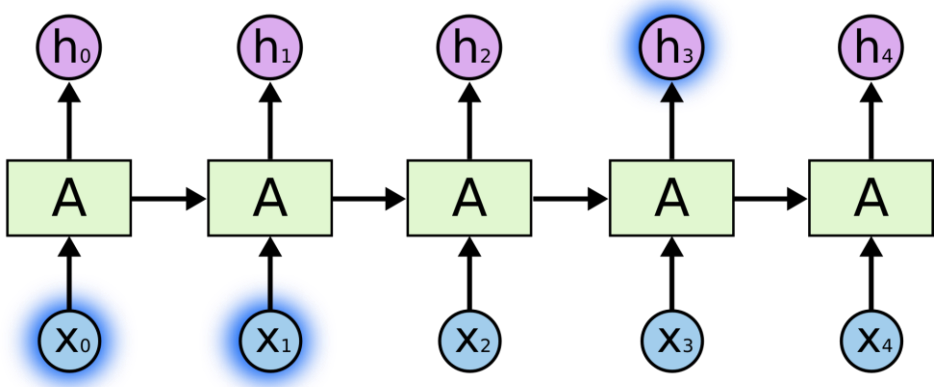
- A more technical explanation is that we use the standard Softmax classifier (also commonly referred to as the cross-entropy loss) on every output vector simultaneously. The RNN is trained with mini-batch Stochastic Gradient Descent and I like to use RMSProp or Adam (per-parameter adaptive learning rate methods) to stablilize the updates.

- Notice also that the first time the character “l” is input, the target is “l”, but the second time the target is “o”. The RNN therefore cannot rely on the input alone and must use its recurrent connection to keep track of the context to achieve this task.

- At test time, we feed a character into the RNN and get a distribution over what characters are likely to come next. We sample from this distribution, and feed it right back in to get the next letter. Repeat this process and you’re sampling text!

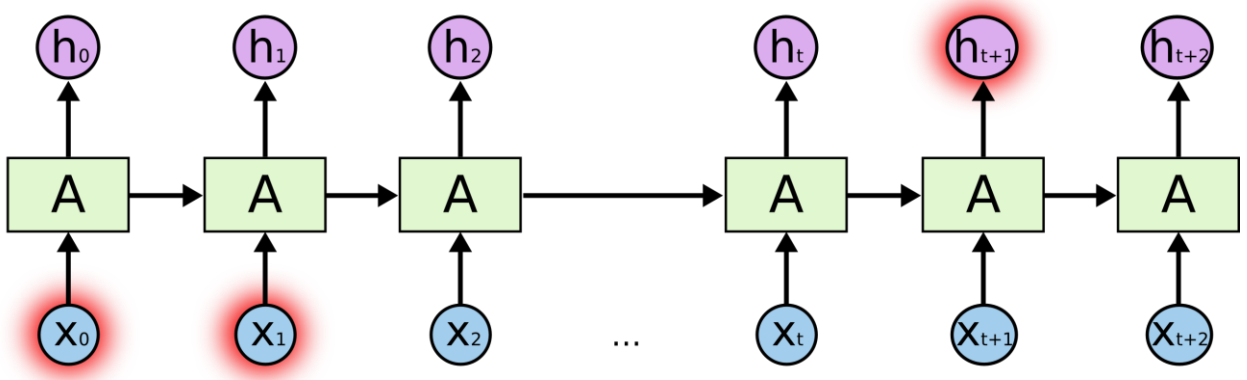
# LSTM RNNs

- Sometimes, we only need to look at recent information to perform the present task.
  - For example, consider a language model trying to predict the next word based on the previous ones.
  - If we are trying to predict the last word in “the clouds are in the sky,” we don’t need any further context – it’s pretty obvious the next word is going to be sky.
  - In such cases, where the gap between the relevant information and the place that it’s needed is small, RNNs can learn to use the past information.



# LSTM RNNs

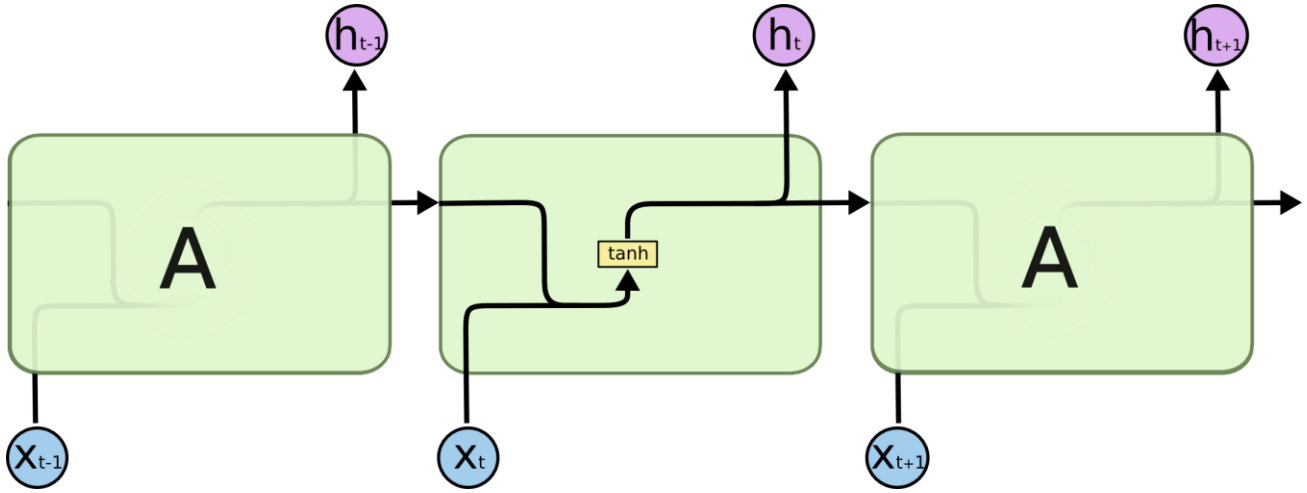
- But consider trying to predict the last word in the text “I grew up in France… I speak fluent French.”
- Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back.
- It’s entirely possible for the gap between the relevant information and the point where it is needed to become very large.
- Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



# LSTM RNNs

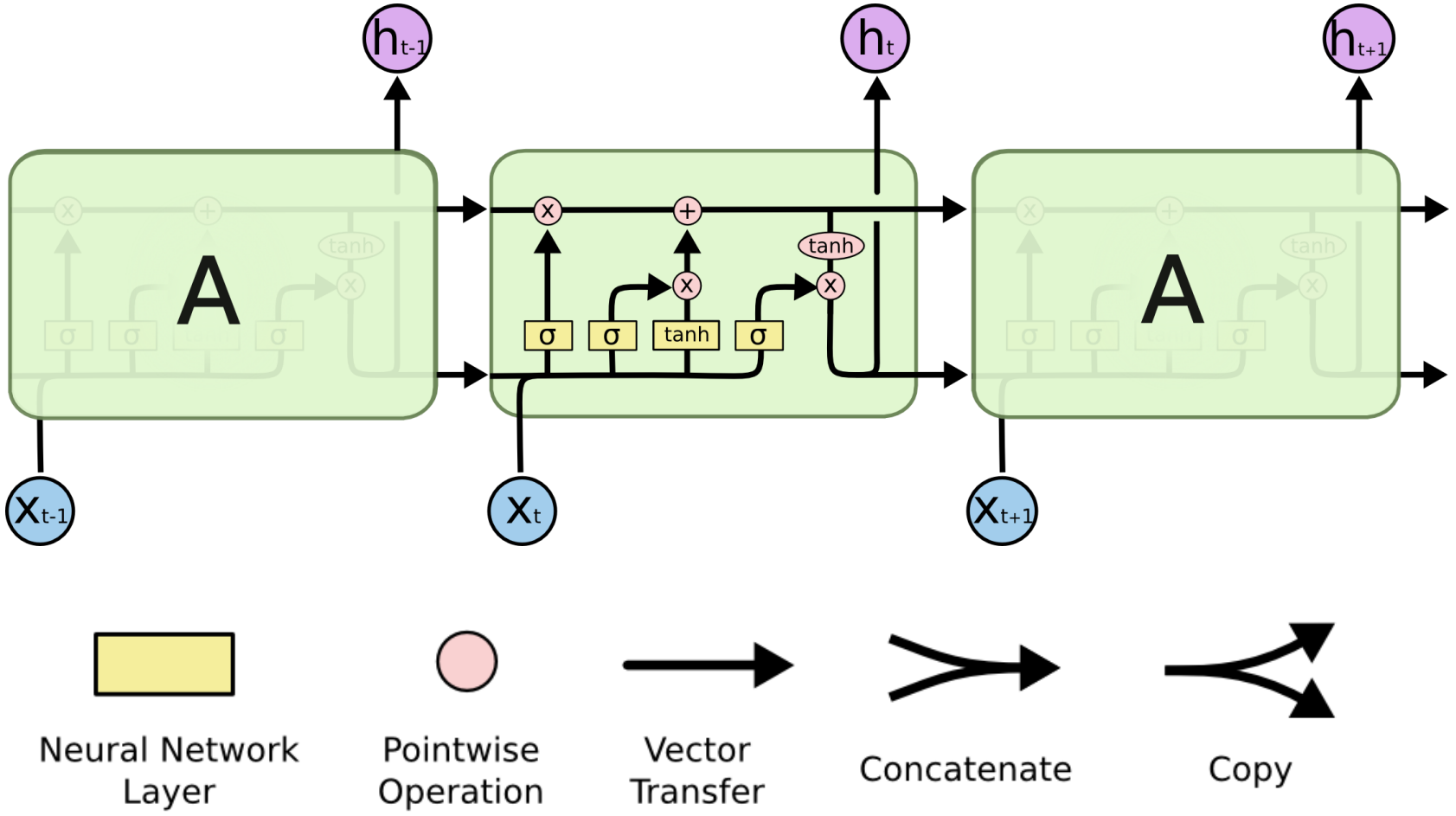
- Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies.
- LSTMs are explicitly designed to avoid the long-term dependency problem.

- All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer:



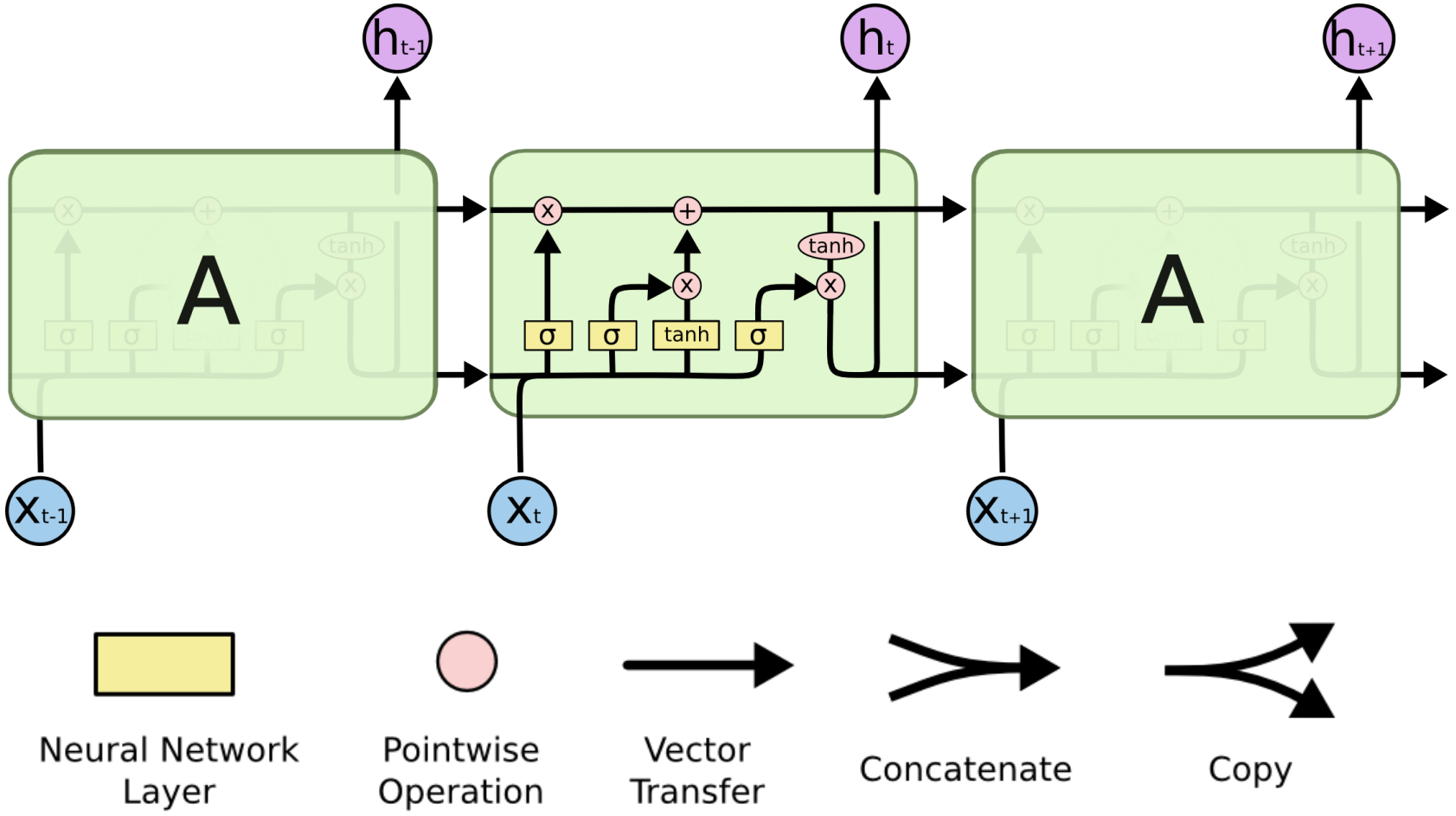
# LSTM RNNs

- LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



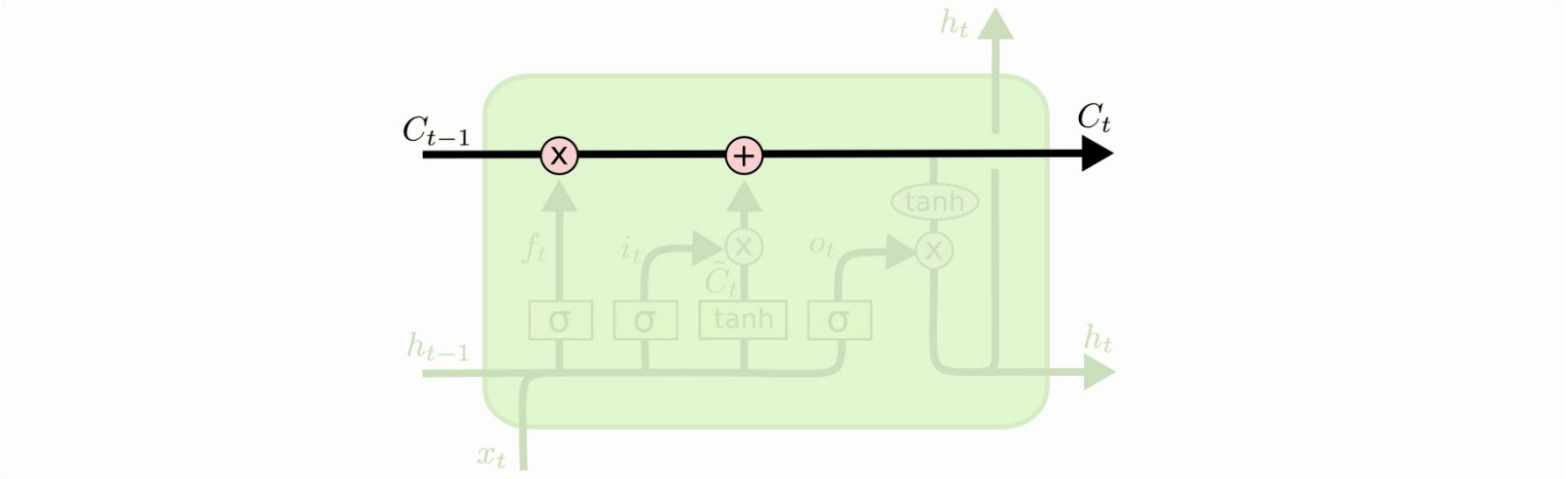
# LSTM RNNs

- LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



# LSTM RNNs

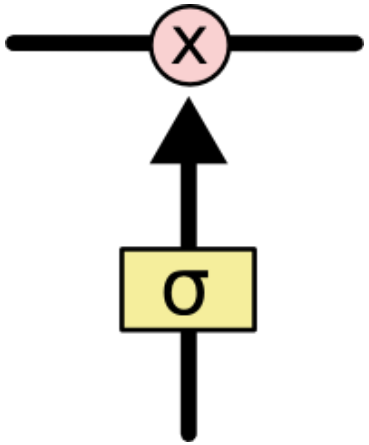
- The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.
- The cell state is kind of like a conveyor belt. It's very easy for information to just flow along it unchanged.




- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.
- **c serves as the memory, while h acts a copy of it that can be passed to further process the inputs in the next time-step.**

# LSTM RNNs

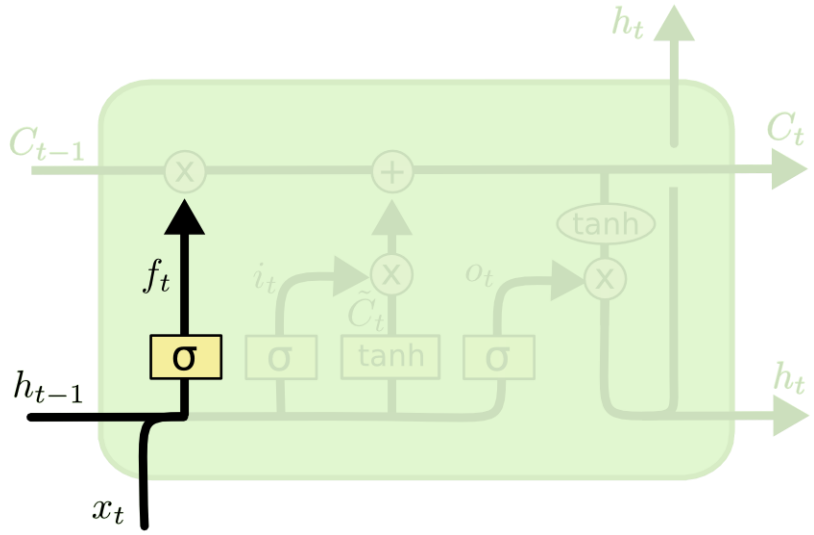
- Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.
- The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”




- An LSTM has three gates.

# LSTM RNNs – Gate 1

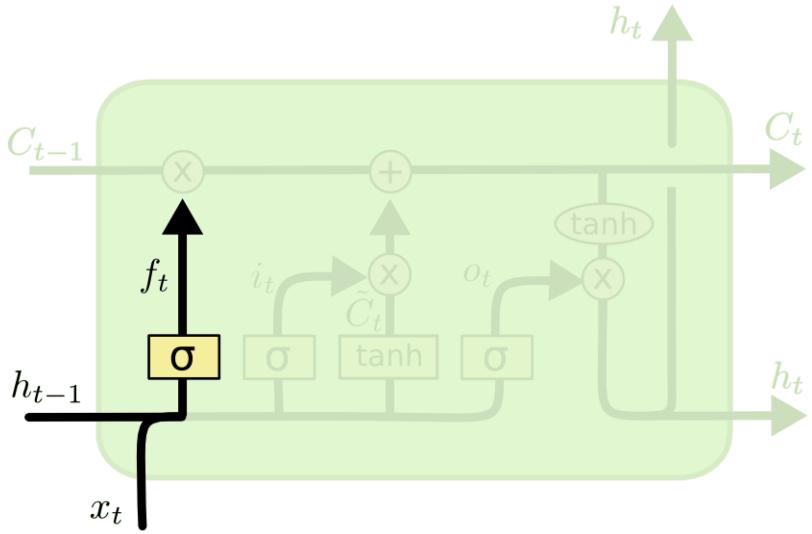
- The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at $h_{t-1}$ and $x_t$ and outputs a number between 0 and 1 for each number in the cell state $C_{t-1}$
- 1 represents "completely keep this" while a 0 represents "completely get rid of this."




$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

# LSTM RNNs – Gate 1 - forget

- Let’s go back to our example of a language model trying to predict the next word based on all the previous ones:
  - The cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

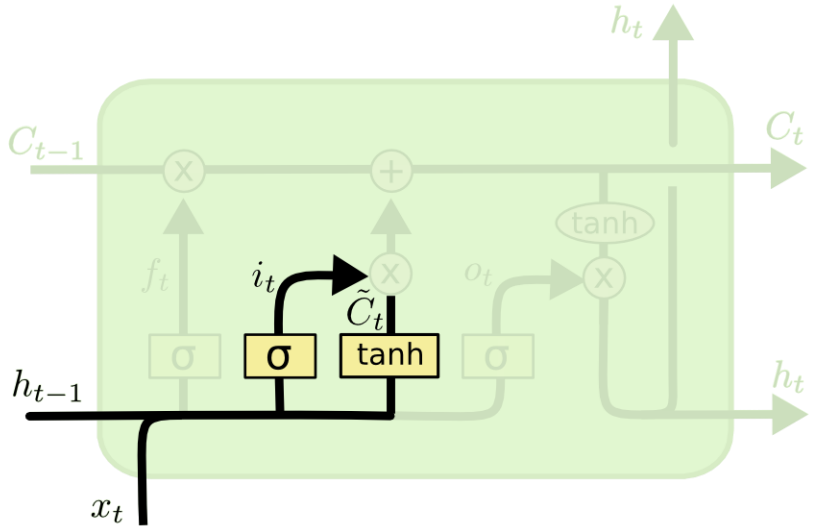



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

# LSTM RNNs – Gate 2 - input

- The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, $\sim C_t$ , that could be added to the state. In the next step, we'll combine these two to create an update to the state.

- In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.

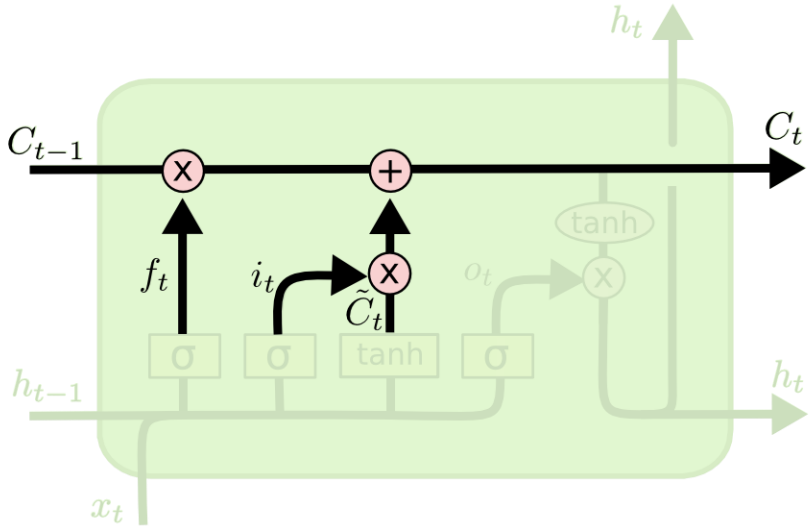



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

# LSTM RNNs – Update

- It's now time to update the old cell state, $C_{t-1}$ into the new cell state $C_t$
- We multiply the old state by $f_t$, forgetting the things we decided to forget earlier. Then we add $i_t * \sim C_t$ . This is the new candidate values, scaled by how much we decided to update each state value.
- In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.

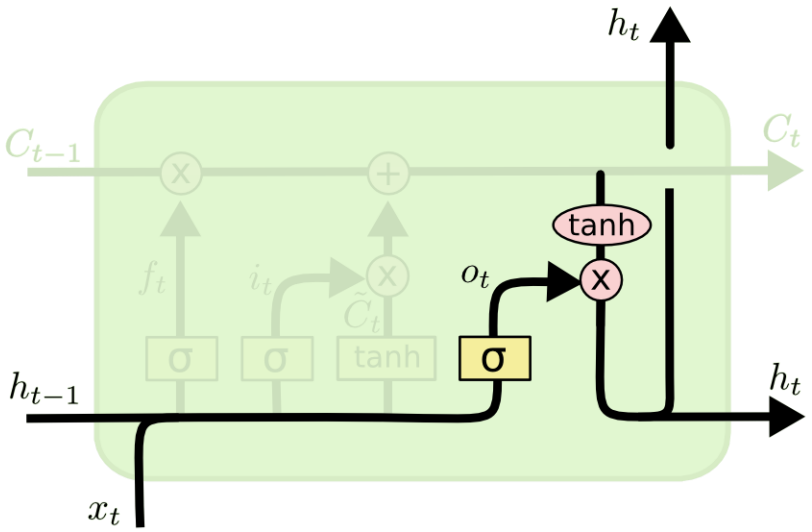



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

# LSTM RNNs – Gate 3 - output

- Finally, we need to decide what we're going to output. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

- For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next.




$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

# LSTM RNNs

- The previous diagrams can be summarized with this pseudocode block:

```
input_gate = tanh(dot(input_vector, W_input) + dot(prev_hidden,
U_input) + b_input)
forget_gate = tanh(dot(input_vector, W_forget) + dot(prev_hidden,
U_forget) + b_forget)
output_gate = tanh(dot(input_vector, W_output) + dot(prev_hidden,
U_output) + b_output)

candidate_state = tanh(dot(x, W_hidden) + dot(prev_hidden, U_hidden)
+ b_hidden)

memory_unit = prev_candidate_state * forget_gate + candidate_state *
input_gate

new_hidden_state = tanh(memory_unit) * output_gate
```