Recurrent Neural Network and Long-Short Term Memory

Lê Anh Cường

2020

Outline

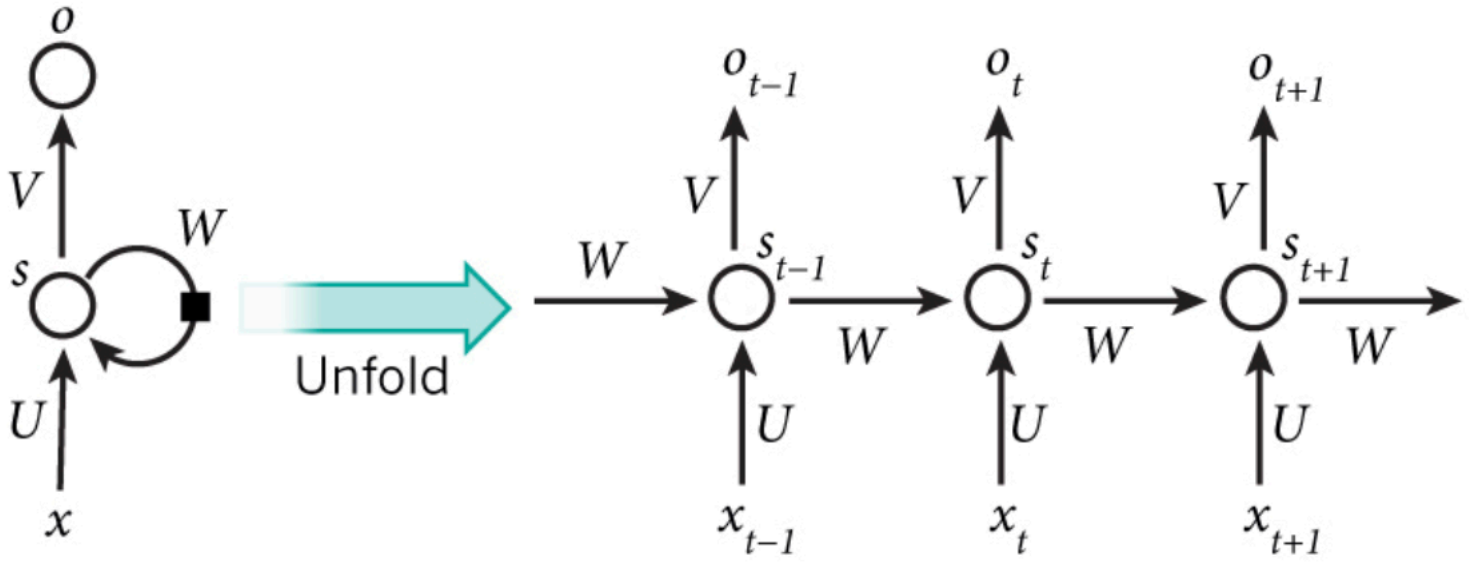
- What is RNN?
- RNN: the Architecture and Forward Computation
- RNN with Progapagation algorithm and Vanishing problem
- Long-Short Term Memory

What is Recurrent Neural Network (RNN)?

- Recurrent Neural Networks (RNNs)
- The idea behind RNNs is to make use of sequential information.
- RNNs are called *recurrent* because they perform the same task for every element of a sequence.
- RNNs have a “memory” which captures information about what has been calculated so far.

What is Recurrent Neural Network (RNN)?

- The idea behind RNNs is to make use of sequential information.
- RNNs are called *recurrent* because they perform the same task for every element of a sequence.
- RNNs have a “memory” which captures information about what has been calculated so far.

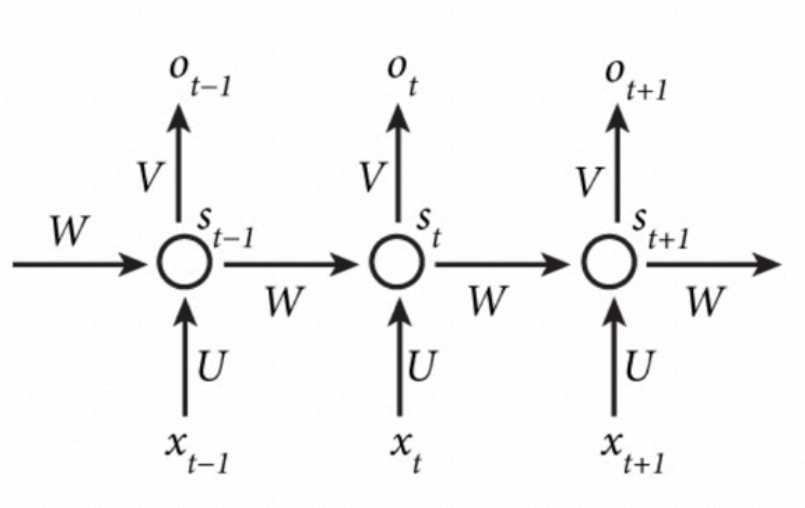


Applications of RNNs

- Language Modeling and Generating Text
- Machine Translation
- Speech Recognition
- Generating Image Descriptions
- Human Activity Recognition
- Predict Stock Prices
- …

Forward Computation in RNNs

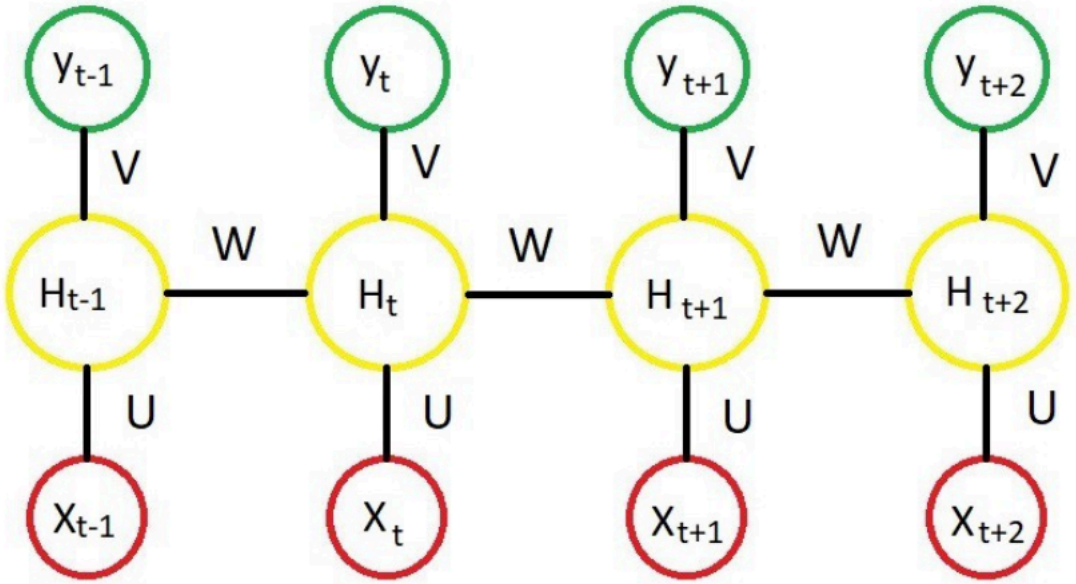
- x_t is the input at time step t. For example, x_1 could be a one-hot vector corresponding to the second word of a sentence.
- s_t is the hidden state at time step t. It's the "memory" of the network. s_t is calculated based on the previous hidden state and the input at the current step: $s_t = f(Ux_t + Ws_{t-1})$. 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. $o_t = \text{softmax}(Vs_t)$.



$$s_t = f(Ux_t + Ws_{t-1}).$$

$$o_t = \text{softmax}(Vs_t).$$

Notations and computation



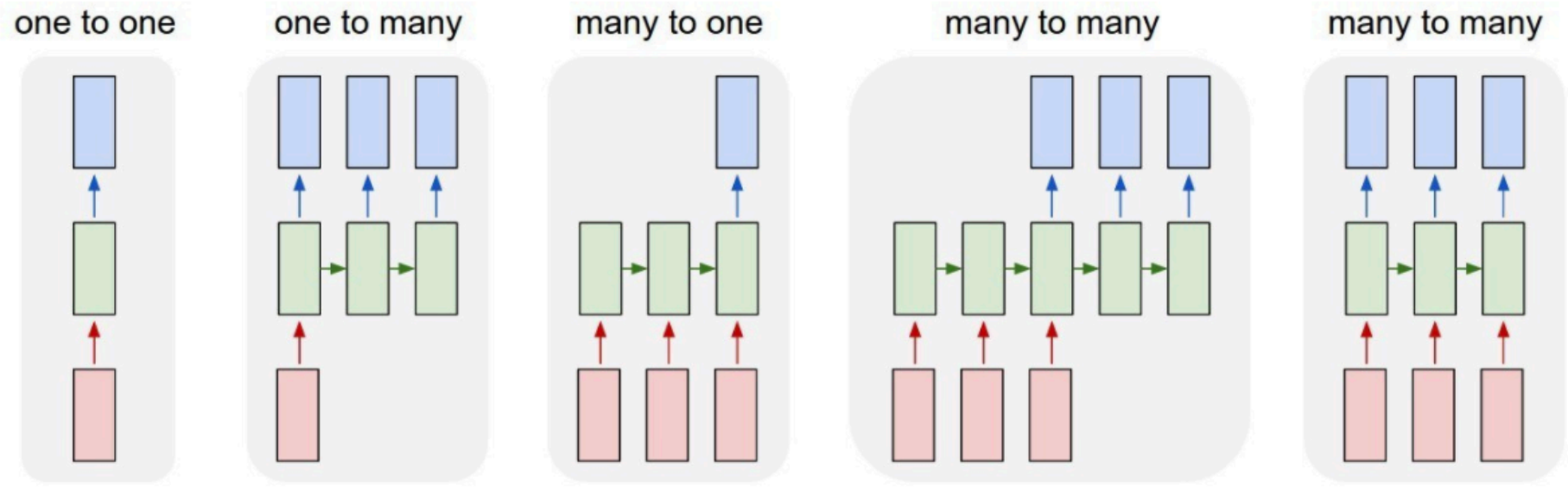
U = Weight vector for Hidden layer
V = Weight vector for Output layer
W = Same weight vector for different Timesteps
X = Word vector for Input word
y = Word vector for Output word

At Timestep (t)

$Ht = \sigma (U * Xt + W * Ht-1)$

$yt = Softmax (V * Ht)$

RNN architectures



Applications of RNNs

1. One to Many

Image

RNN

A person is sitting in a beach holding guitar.

Sequence of text

2.Many to One

This movie is not good

Sequence of text

RNN

Negative

Sentiment

3.Many to Many

Main tenu pyar karda haan

Punjabi,India

RNN

I love you

English, World

Exercise

- Suppose the input *x* is a vector of *k* dimensions; and the output y is a vector of m dimentions.

Give a shape design for matrices U, V, W and length for the state s

1	0	0	0
0	1	0	0
0	0	1	1
0	0	0	0
h	e	l	l

wxh			
0.287027	0.84606	0.572392	0.486813
0.902874	0.871522	0.691079	0.18998
0.537524	0.09224	0.558159	0.491528

×

1
0
0
0
h

=

0.287027
0.902874
0.537524

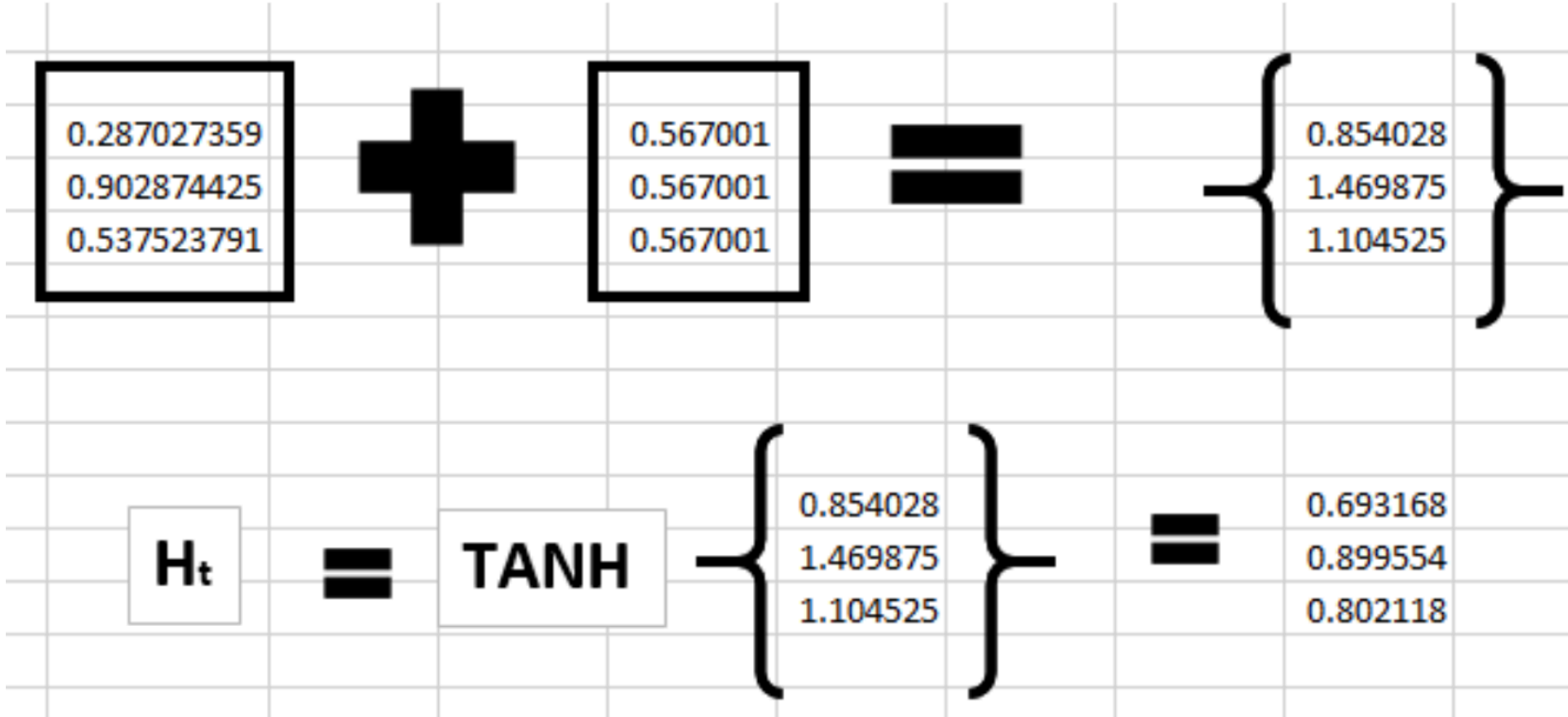
Weight(whh)	bias
0.427043	0.567001

×

0
0
0
h_{t-1}

=

0.567001
0.567001
0.567001


0.287027359
0.902874425
0.537523791
0.567001
0.567001
0.567001
0.854028
1.469875
1.104525
Ht
TANH
0.854028
1.469875
1.104525
0.693168
0.899554
0.802118

Continue in

- https://www.analyticsvidhya.com/blog/2017/12/introduction-to-recurrent-neural-networks/

Backpropagation Through Time (BPTT)

The *loss*, or error, to be the cross entropy loss, given by:

$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t$$

$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$

$$= -\sum_t y_t \log \hat{y}_t$$

$$s_t = \tanh(Ux_t + Ws_{t-1})$$

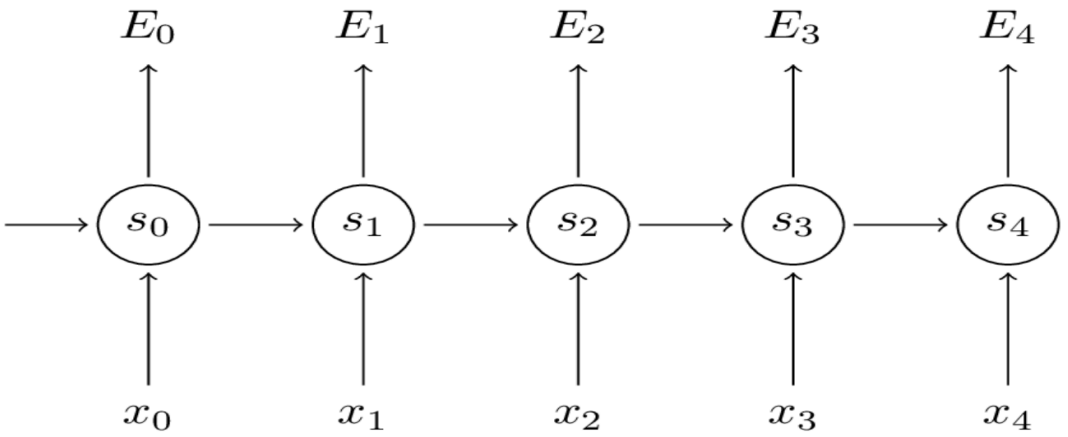
$$\hat{y}_t = \mathrm{softmax}(Vs_t)$$

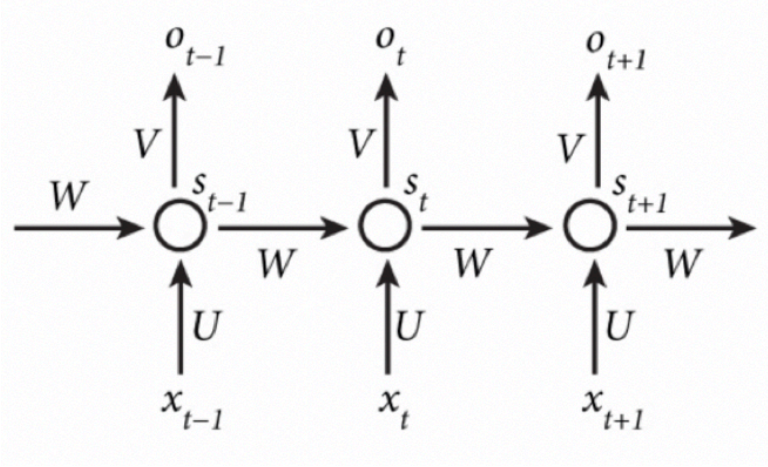
The cross entropy formula takes in two distributions, $p(x)$, the true distribution, and $q(x)$, the estimated distribution, defined over the discrete variable x and is given by

$$H(p, q) = -\sum_{\forall x} p(x) \log(q(x))$$

Backpropagation Through Time (BPTT)

Remember that our goal is to calculate the gradients of the error with respect to our parameters U, V and W and then learn good parameters using Stochastic Gradient Descent. Just like we sum up the errors, we also sum up the gradients at each time step for one training example: $\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$.



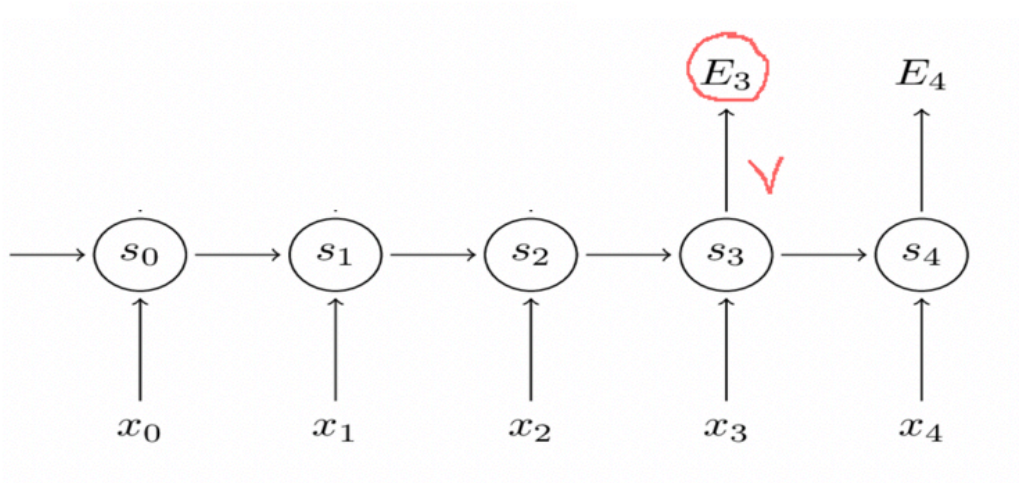


$$s_t = \tanh(Ux_t + Ws_{t-1})$$

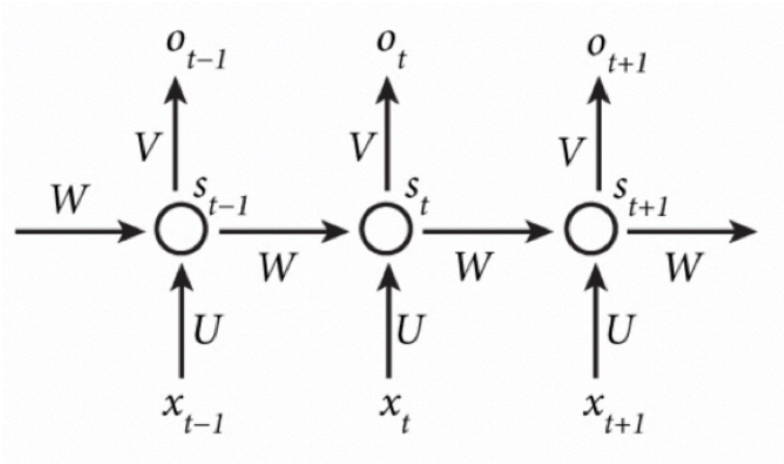
$$\hat{y}_t = \text{softmax}(Vs_t)$$

$$z_3 = Vs_3,$$

$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t$$



$$\frac{\partial E_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial V}$$
$$= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial z_3}\frac{\partial z_3}{\partial V}$$
$$= (\hat{y}_3 - y_3) \otimes s_3$$



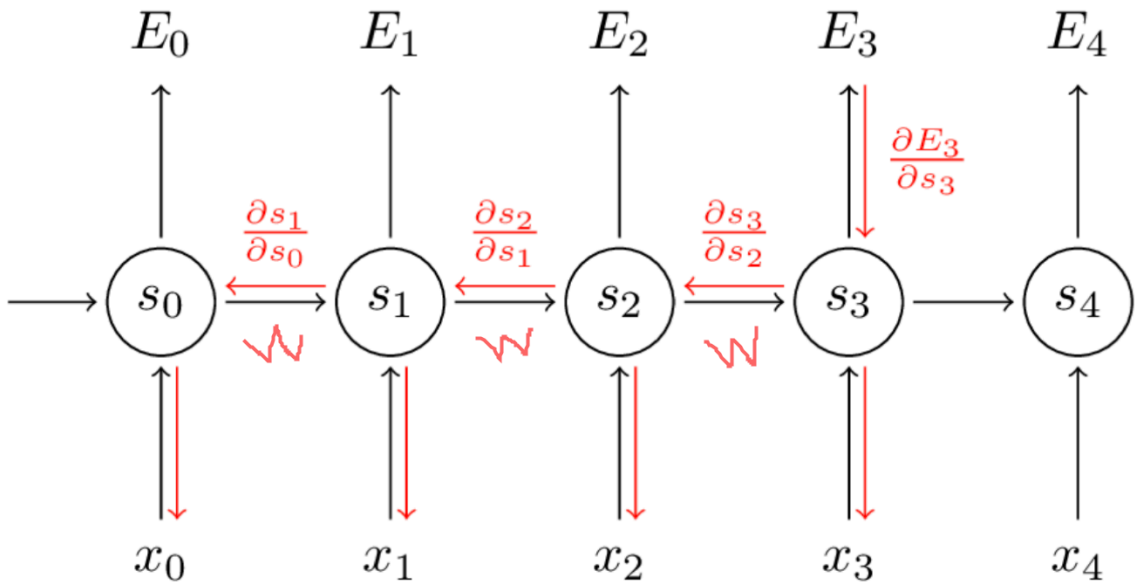
$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$

$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$\hat{y}_t = \text{softmax}(Vs_t)$$

$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t$$

$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$



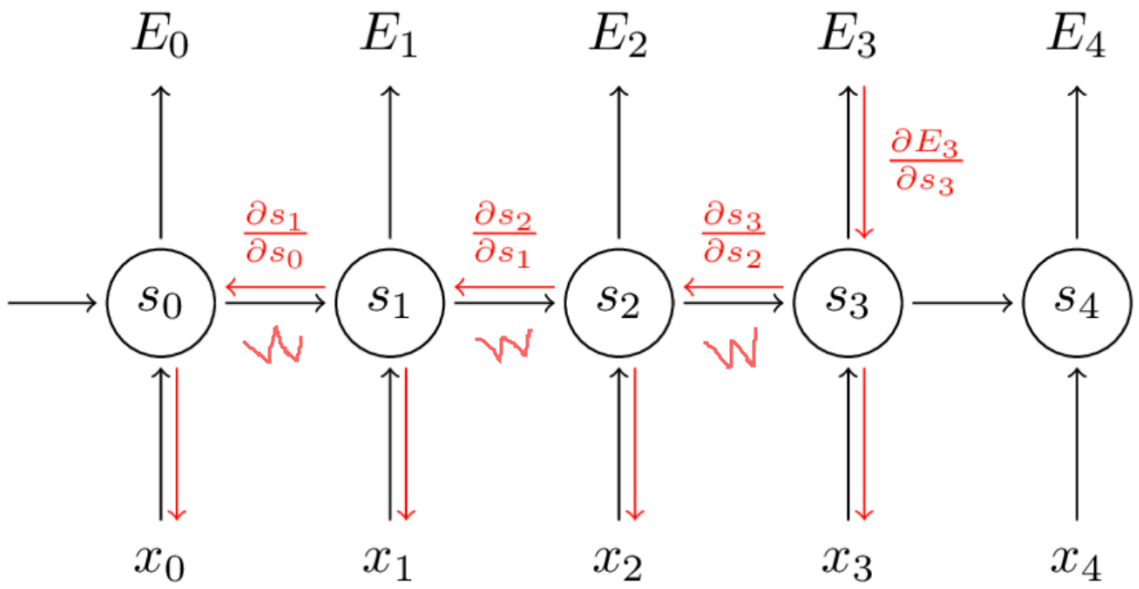
Note that $\frac{\partial s_3}{\partial s_k}$ is a chain rule in itself! For example,

$$\frac{\partial s_3}{\partial s_1} = \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1}.$$

$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$\hat{y}_t = \text{softmax}(Vs_t)$$

$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t$$

$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$



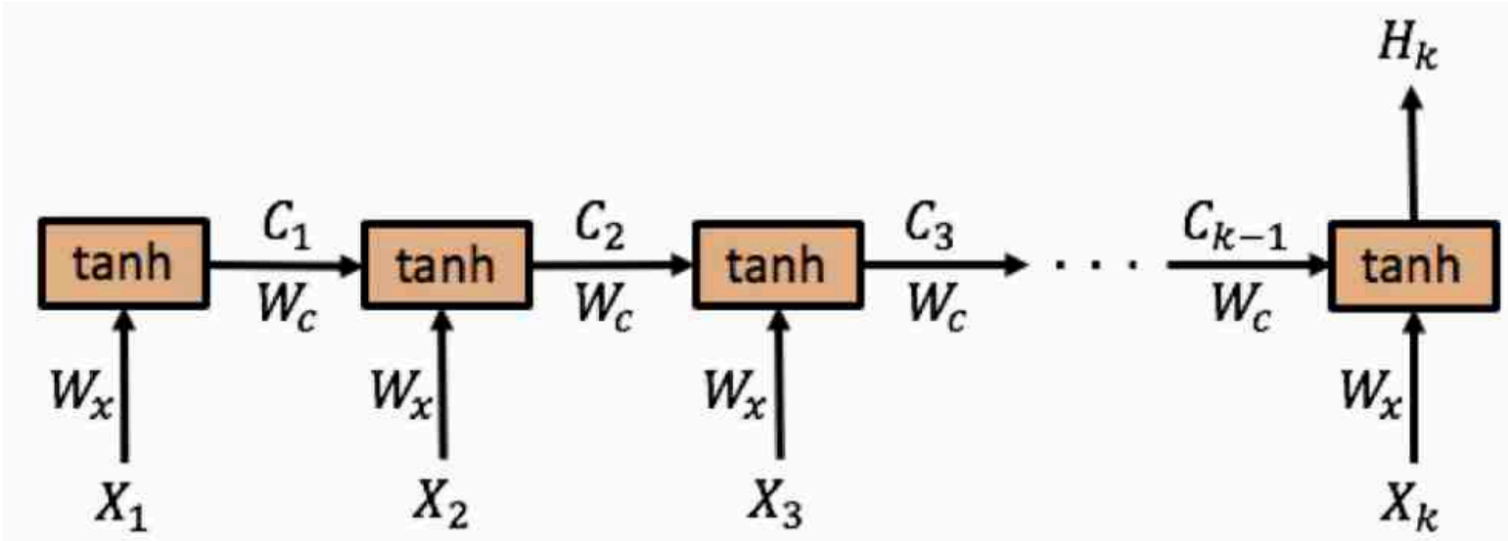
$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \left(\prod_{j=k+1}^{3} \frac{\partial s_j}{\partial s_{j-1}} \right) \frac{\partial s_k}{\partial W}$$

Exercises

- Derive the formula for $\frac{\partial E}{\partial u}$

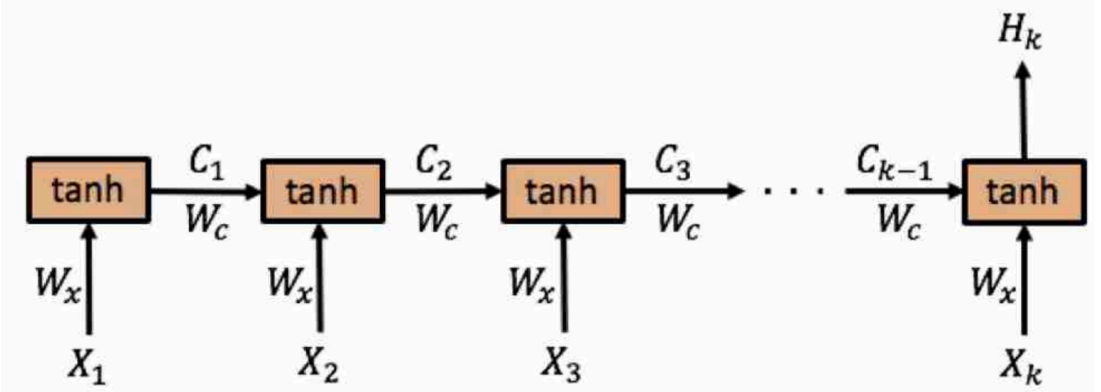
Vanishing Gradients



W = [Wc,Wx]

$$W \leftarrow W - \alpha \frac{\partial E_k}{\partial W}$$

Vanishing Gradients

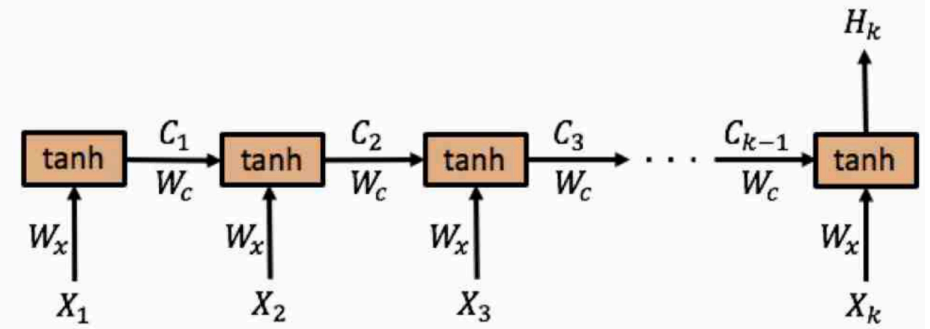


$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\frac{\partial C_k}{\partial C_{k-1}}\cdots\frac{\partial C_2}{\partial C_1}\frac{\partial C_1}{\partial W} =$$

$$\frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\left(\prod_{t=2}^{k}\frac{\partial C_t}{\partial C_{t-1}}\right)\frac{\partial C_1}{\partial W}$$

$$C_t = tanh(W_c C_{t-1} + W_x X_t)$$

Vanishing Gradients



$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\frac{\partial C_k}{\partial C_{k-1}}\cdots\frac{\partial C_2}{\partial C_1}\frac{\partial C_1}{\partial W} =$$

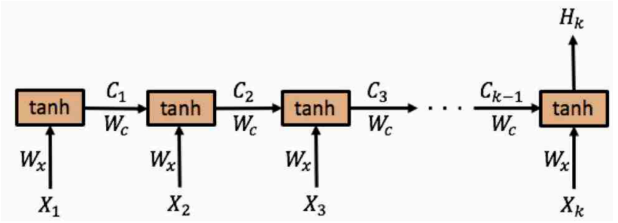
$$\frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\left(\prod_{t=2}^{k}\frac{\partial C_t}{\partial C_{t-1}}\right)\frac{\partial C_1}{\partial W}$$

$$C_t = tanh(W_c C_{t-1} + W_x X_t)$$

$$\frac{\partial C_t}{\partial C_{t-1}} = tanh'(W_c C_{t-1} + W_x X_t)\cdot\frac{d}{dC_{t-1}}[W_c C_{t-1} + W_x X_t] =$$

$$tanh'(W_c C_{t-1} + W_x X_t)\cdot W_c$$

Vanishing Gradients



$$C_t = tanh(W_c C_{t-1} + W_x X_t)$$

$$\frac{\partial C_t}{\partial C_{t-1}} = tanh'(W_c C_{t-1} + W_x X_t) \cdot \frac{d}{dC_{t-1}}[W_c C_{t-1} + W_x X_t] =$$

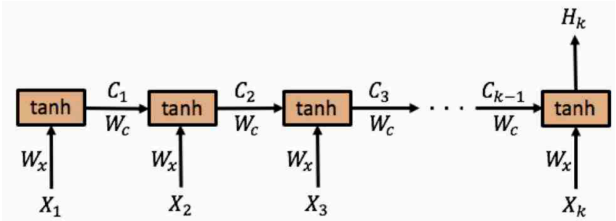
$$tanh'(W_c C_{t-1} + W_x X_t) \cdot W_c$$

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\frac{\partial C_k}{\partial C_{k-1}} \cdots \frac{\partial C_2}{\partial C_1}\frac{\partial C_1}{\partial W} =$$

$$\frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\left(\prod_{t=2}^{k}\frac{\partial C_t}{\partial C_{t-1}}\right)\frac{\partial C_1}{\partial W}$$

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\left(\prod_{t=2}^{k} tanh'(W_c C_{t-1} + W_x X_t) \cdot W_c\right)\frac{\partial C_1}{\partial W}$$

Vanishing Gradients



$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\left(\prod_{t=2}^{k} tanh'(W_c C_{t-1} + W_x X_t)\cdot W_c\right)\frac{\partial C_1}{\partial W}$$

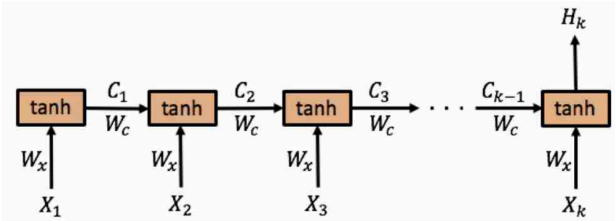
The last expression **tends to vanish when k is large**, this is due to the derivative of the activation function tanh which is smaller or equal 1.

$$\prod_{t=2}^{k} tanh'(W_c C_{t-1} + W_x X_t)\cdot W_c \longrightarrow 0,\ so\ \frac{\partial E_k}{\partial W} \longrightarrow 0$$

$$W \leftarrow W - \alpha\frac{\partial E_k}{\partial W} \approx W$$

vướng

Vanishing Gradients



$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\left(\prod_{t=2}^{k} tanh'(W_c C_{t-1} + W_x X_t)\cdot W_c\right)\frac{\partial C_1}{\partial W}$$

The last expression **tends to vanish when k is large**, this is due to the derivative of the activation function tanh which is smaller or equal 1.

$$\prod_{t=2}^{k} tanh'(W_c C_{t-1} + W_x X_t)\cdot W_c \longrightarrow 0, \; so \; \frac{\partial E_k}{\partial W} \longrightarrow 0$$

$$W \leftarrow W - \alpha\frac{\partial E_k}{\partial W} \approx W$$

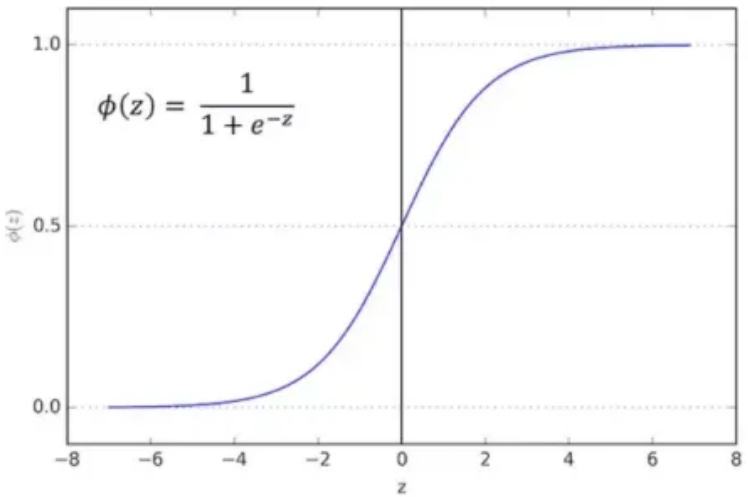
W_C ~ small -> vanishing
W_C ~ large -> exploding

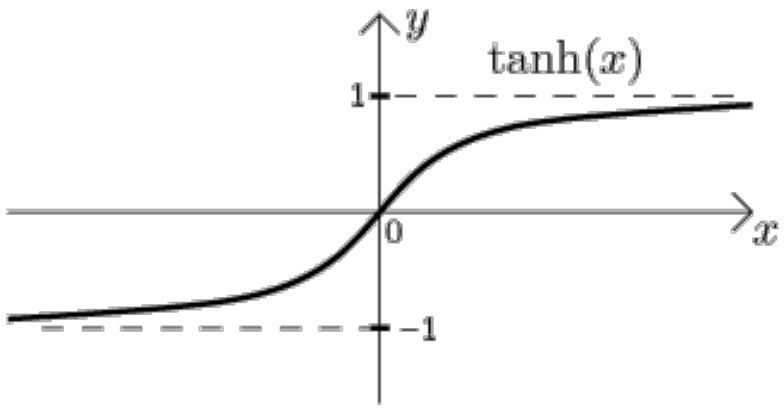
vanishing

- https://www.jefkine.com/general/2018/05/21/2018-05-21-vanishing-and-exploding-gradient-problems/

- https://stats.stackexchange.com/questions/185639/how-does-lstm-prevent-the-vanishing-gradient-problem

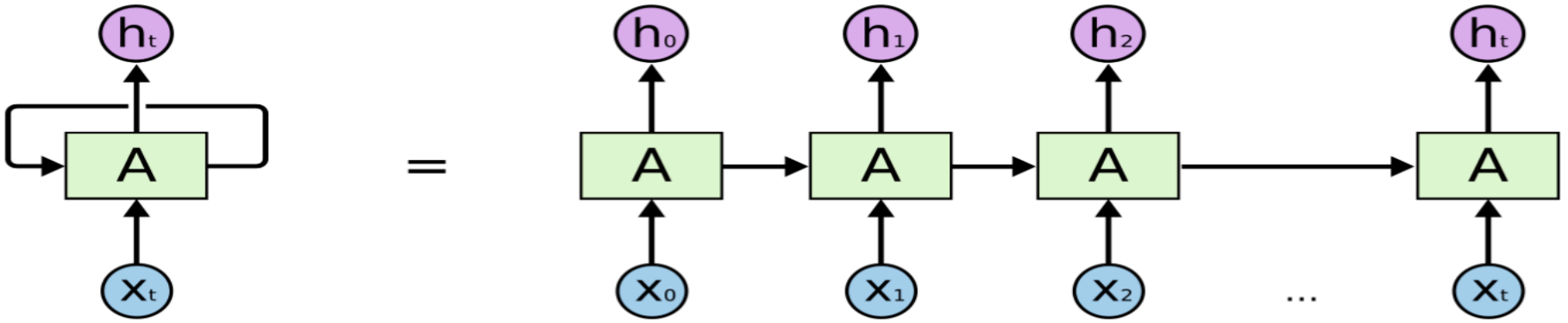
Sigmoid and Tanh funtions





$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$

Limitation of RNN



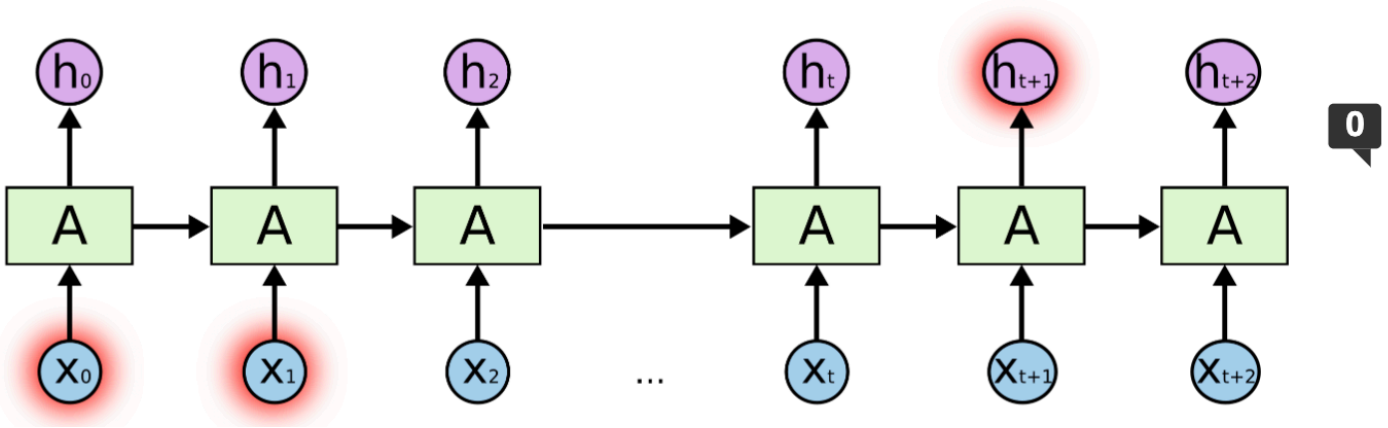
An unrolled recurrent neural network.

In theory, RNNs are absolutely capable of handling such “long-term dependencies.” A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don’t seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) [German] and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.

Limitation of RNN

But there are also cases where we need more context. 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.

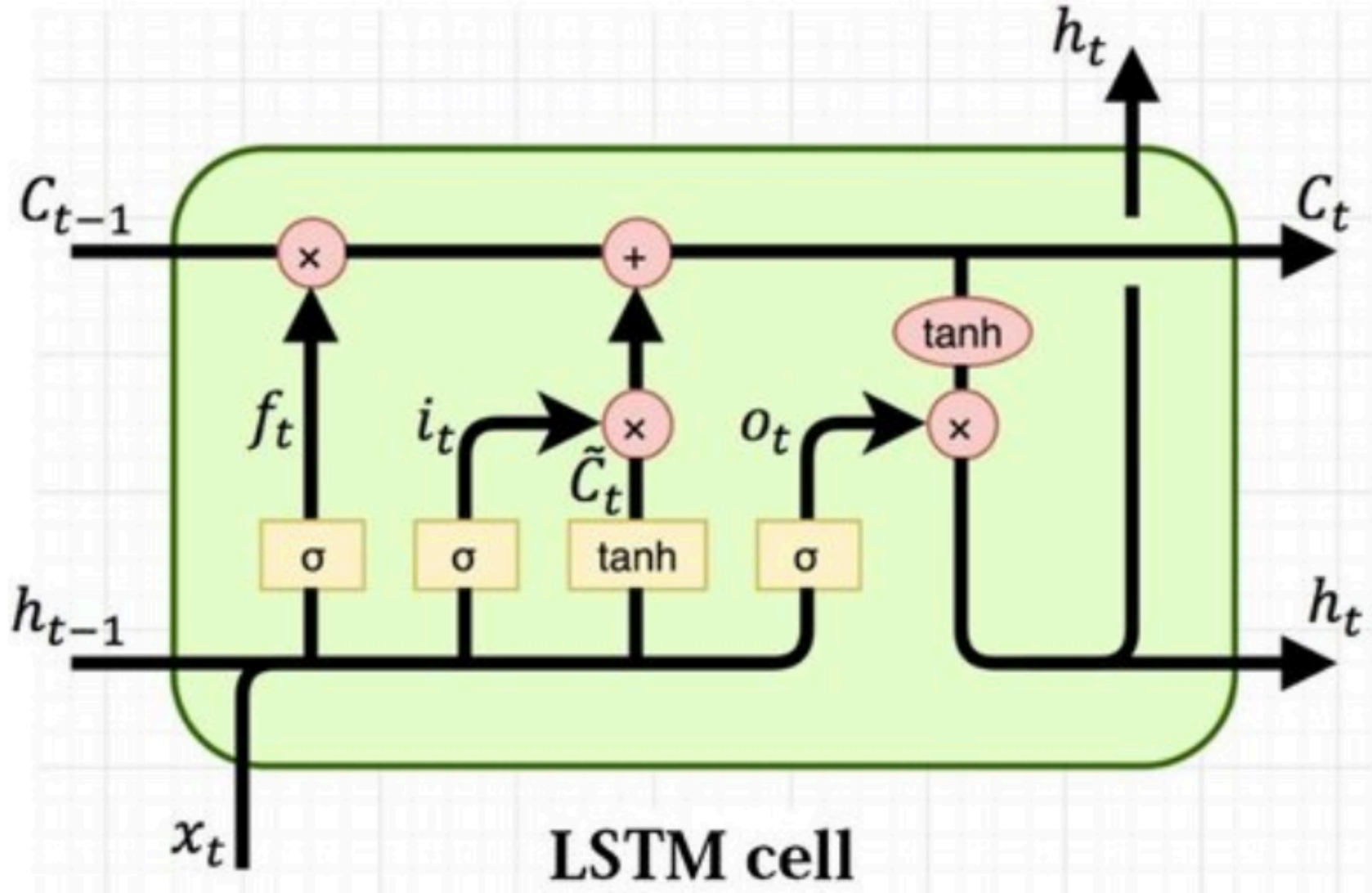


LSTM

- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- http://blog.echen.me/2017/05/30/exploring-lstms/
- https://mc.ai/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients/

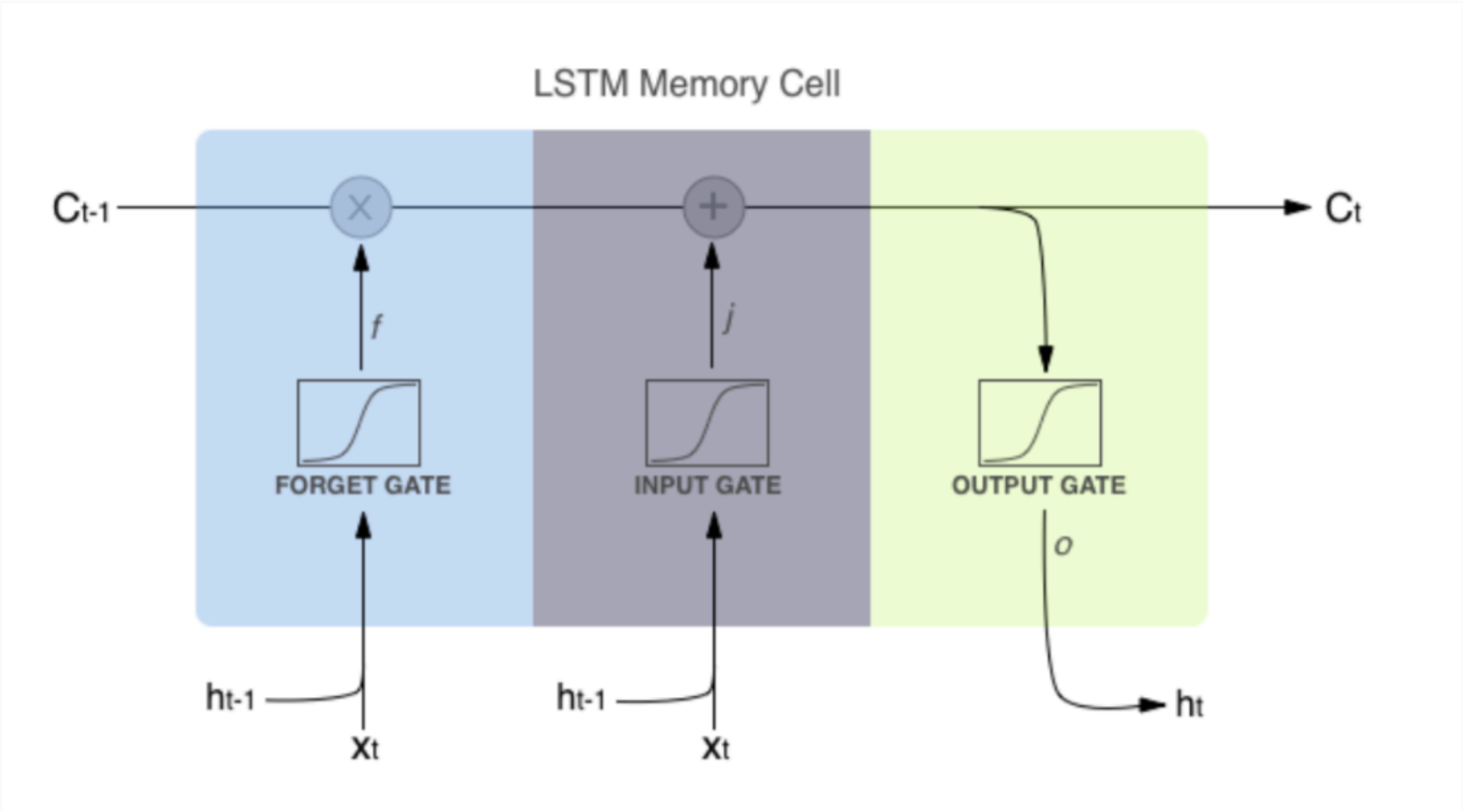
LSTM Networks

- Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work.[1] They work tremendously well on a large variety of problems, and are now widely used.
- LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

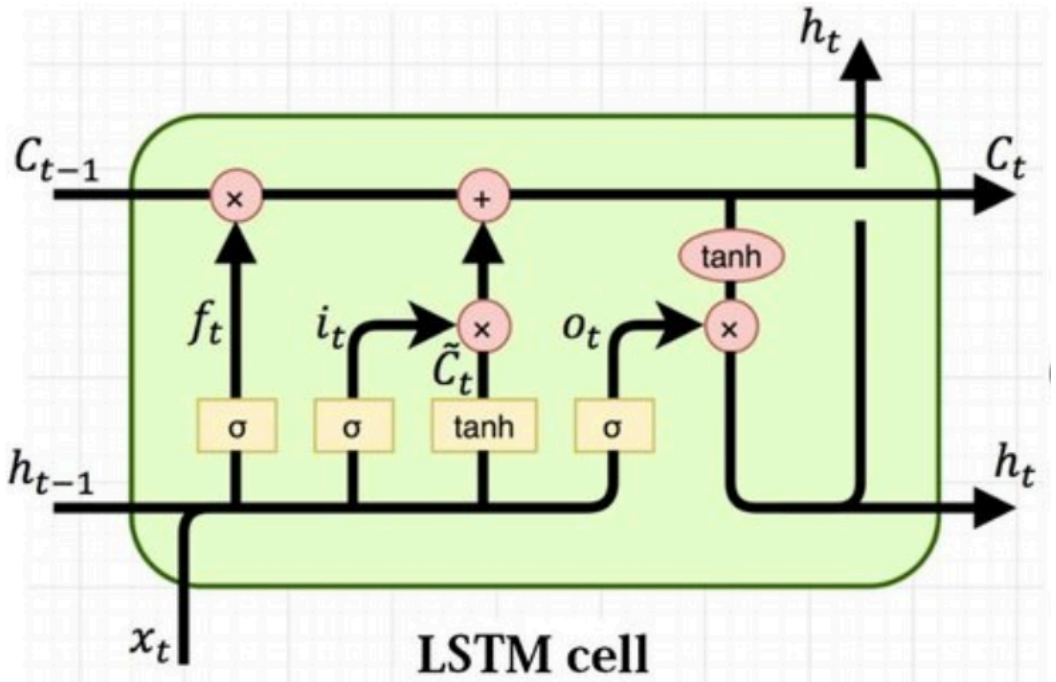


LSTM cell

LSTM Memory Cell



$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \sigma_h(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$



where the initial values are $c_0 = 0$ and $h_0 = 0$ and the operator $\circ$ denotes the Hadamard product (element-wise product). The subscript t indexes the time step.

https://en.wikipedia.org/wiki/Long_short-term_memory

Variables [edit]

- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in \mathbb{R}^h$: forget gate's activation vector
- $i_t \in \mathbb{R}^h$: input/update gate's activation vector
- $o_t \in \mathbb{R}^h$: output gate's activation vector
- $h_t \in \mathbb{R}^h$: hidden state vector also known as output vector of the LSTM unit
- $\tilde{c}_t \in \mathbb{R}^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector
- $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$: weight matrices and bias vector parameters which need to to be learned during training

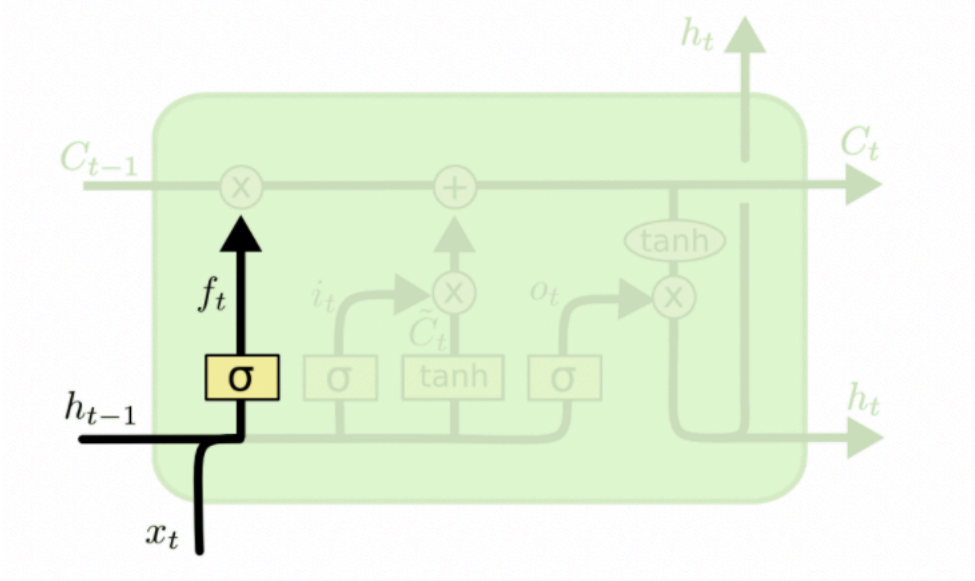
where the superscripts d and h refer to the number of input features and number of hidden units, respectively.

Activation functions [edit]

- σ_g: sigmoid function.
- σ_c: hyperbolic tangent function.
- σ_h: hyperbolic tangent function or, as the peephole LSTM paper[29][30] suggests, $\sigma_h(x) = x$.

Forget gate

Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, 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.

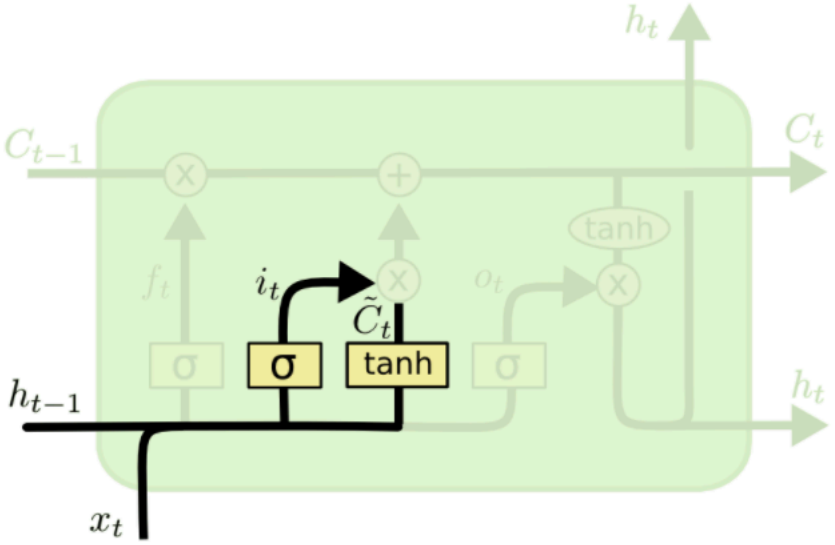


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Infor Gate

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, $\tilde{C}_t$, 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.



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

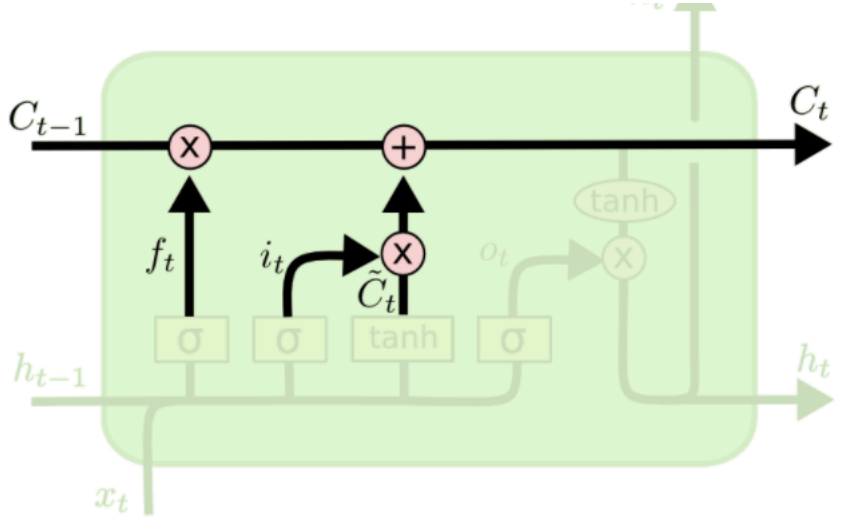
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Update Cell

It's now time to update the old cell state, C_{t-1} into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$. 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.

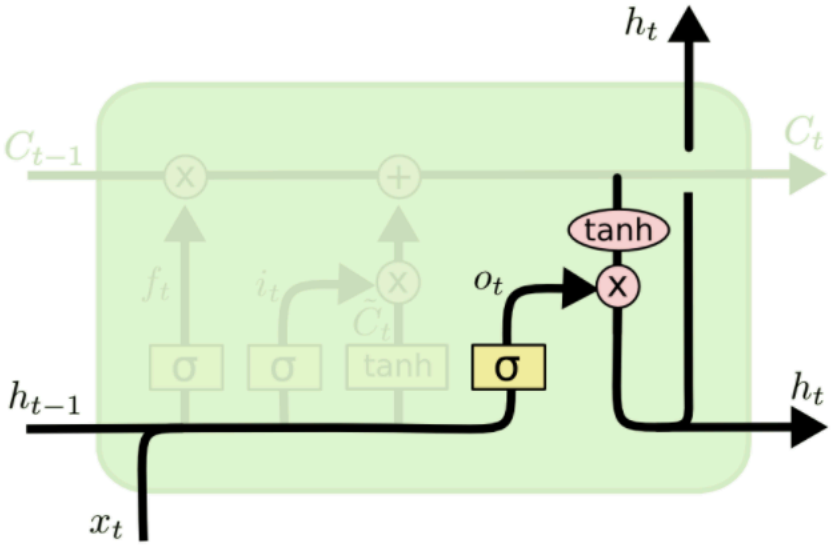


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Output

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. 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** (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

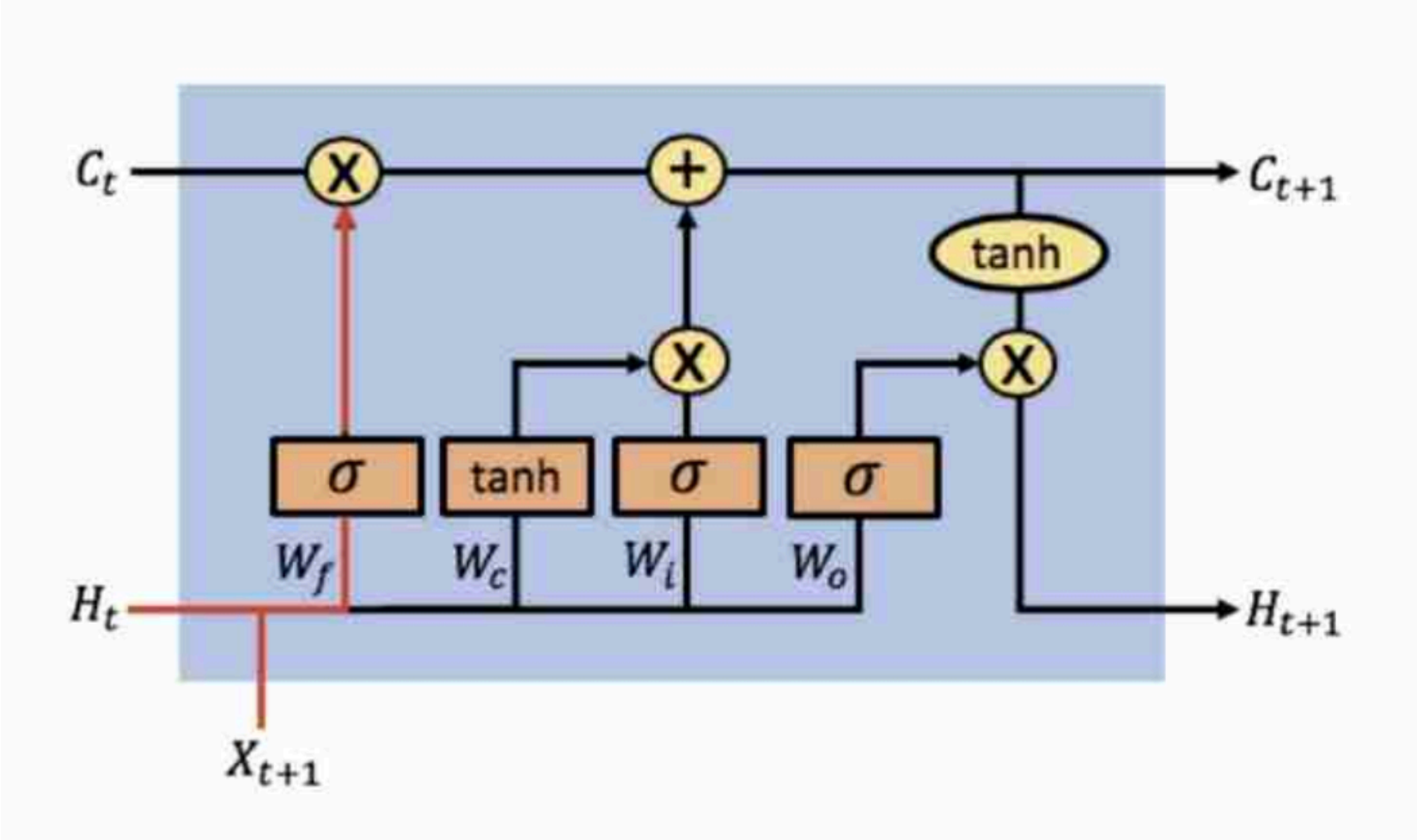
For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.



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

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

How LSTM avoid Vanishing



How LSTM avoid Vanishing

The LSTM forgate gate update of the cell state Ct. Notice the forget gate's output which is given by

$$\sigma(W_f \cdot [H_t, X_{t+1}])$$

The LSTM input gate update of the cell state Ct. The input gate's output has the form:

$$tanh(W_c \cdot [H_t, X_{t+1}]) \otimes \sigma\,(W_i \cdot [H_t, X_{t+1}])$$

Backpropogating through time for gradient computation

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\frac{\partial C_k}{\partial C_{k-1}}\cdots\frac{\partial C_2}{\partial C_1}\frac{\partial C_1}{\partial W} =$$

$$\frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\left(\prod_{t=2}^{k}\frac{\partial C_t}{\partial C_{t-1}}\right)\frac{\partial C_1}{\partial W}$$

In an LTSM, the state vector Ct, has the form:

$$C_t = C_{t-1} \otimes \sigma\left(W_f \cdot [H_{t-1}, X_t]\right) \oplus$$

$$tanh\left(W_c \cdot [H_{t-1}, X_t]\right) \otimes \sigma\left(W_i \cdot [H_{t-1}, X_t]\right)$$

Backpropogating through time for gradient computation

In an LTSM, the state vector Ct, has the form:

$$C_t = C_{t-1} \otimes \sigma\left(W_f \cdot [H_{t-1}, X_t]\right) \oplus \tanh\left(W_c \cdot [H_{t-1}, X_t]\right) \otimes \sigma\left(W_i \cdot [H_{t-1}, X_t]\right)$$

$$\frac{\partial C_t}{\partial C_{t-1}} = \sigma(W_f \cdot [H_{t-1}, X_t]) + \frac{d}{dC_{t-1}}(\tanh(W_c \cdot [H_{t-1}, X_t]) \otimes \sigma(W_i \cdot [H_{t-1}, X_t]))$$

Backpropogating through time for gradient computation

$$C_t = C_{t-1} \otimes \sigma\left(W_f \cdot [H_{t-1}, X_t]\right) \oplus$$

$$tanh\left(W_c \cdot [H_{t-1}, X_t]\right) \otimes \sigma\left(W_i \cdot [H_{t-1}, X_t]\right)$$

$$\frac{\partial C_t}{\partial C_{t-1}} = \sigma(W_f \cdot [H_{t-1}, X_t]) + \frac{d}{dC_{t-1}}(tanh(W_c \cdot [H_{t-1}, X_t]) \otimes \sigma(W_i \cdot [H_{t-1}, X_t]))$$

For simplicity, we leave out the computation of:

$$\frac{d}{dC_{t-1}}(tanh(W_c \cdot [H_{t-1}, X_t]) \otimes \sigma(W_i \cdot [H_{t-1}, X_t]))$$

So we just write:

$$\frac{\partial C_t}{\partial C_{t-1}} \approx \sigma(W_f \cdot [H_{t-1}, X_t])$$

Backpropogating through time for gradient computation

$$\frac{\partial C_t}{\partial C_{t-1}} \approx \sigma(W_f \cdot [H_{t-1}, X_t]) \qquad (2)$$

Now, notice equation (2) means that **the gradient behaves similarly to the forget gate**, and if the forget gate decides that a certain piece of information should be remembered, it will be open and have values closer to 1 to allow for information flow.
For simplicity, we can think of the forget gate's action as:

And the gradients do not vanish!

$$\sigma(W_f \cdot [H_{t-1}, X_t]) \approx \vec{1}$$

So we get:

$$\frac{\partial C_t}{\partial C_{t-1}} \nrightarrow 0$$

Finally:

$$\frac{\partial E_k}{\partial W} \approx \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\left(\prod_{t=2}^{k} \sigma(W_f \cdot [H_{t-1}, X_t])\right)\frac{\partial C_1}{\partial W} \nrightarrow 0$$

And the gradients do not vanish!

Exercise

- https://towardsdatascience.com/understanding-lstm-and-its-quick-implementation-in-keras-for-sentiment-analysis-af410fd85b47

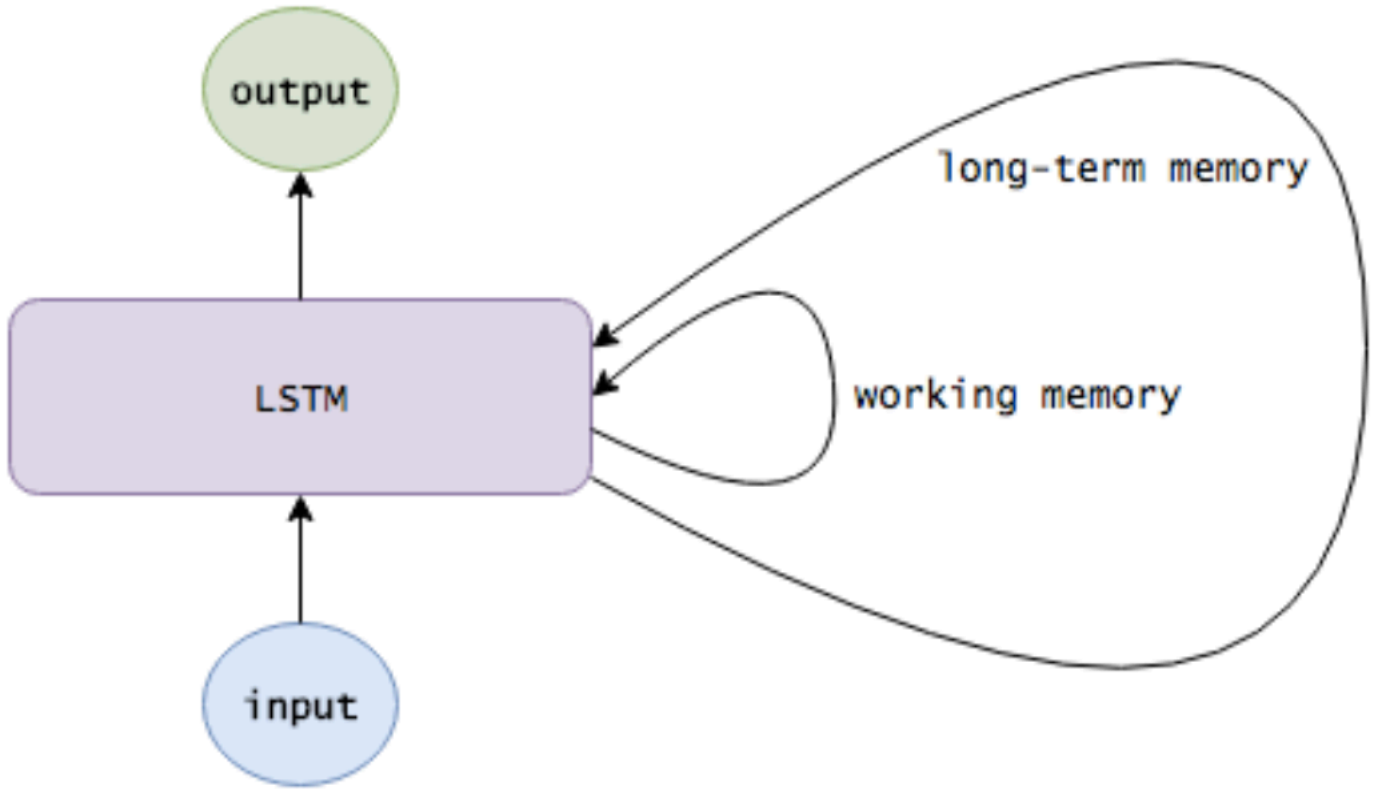
- https://www.kdnuggets.com/2018/11/keras-long-short-term-memory-lstm-model-predict-stock-prices.html

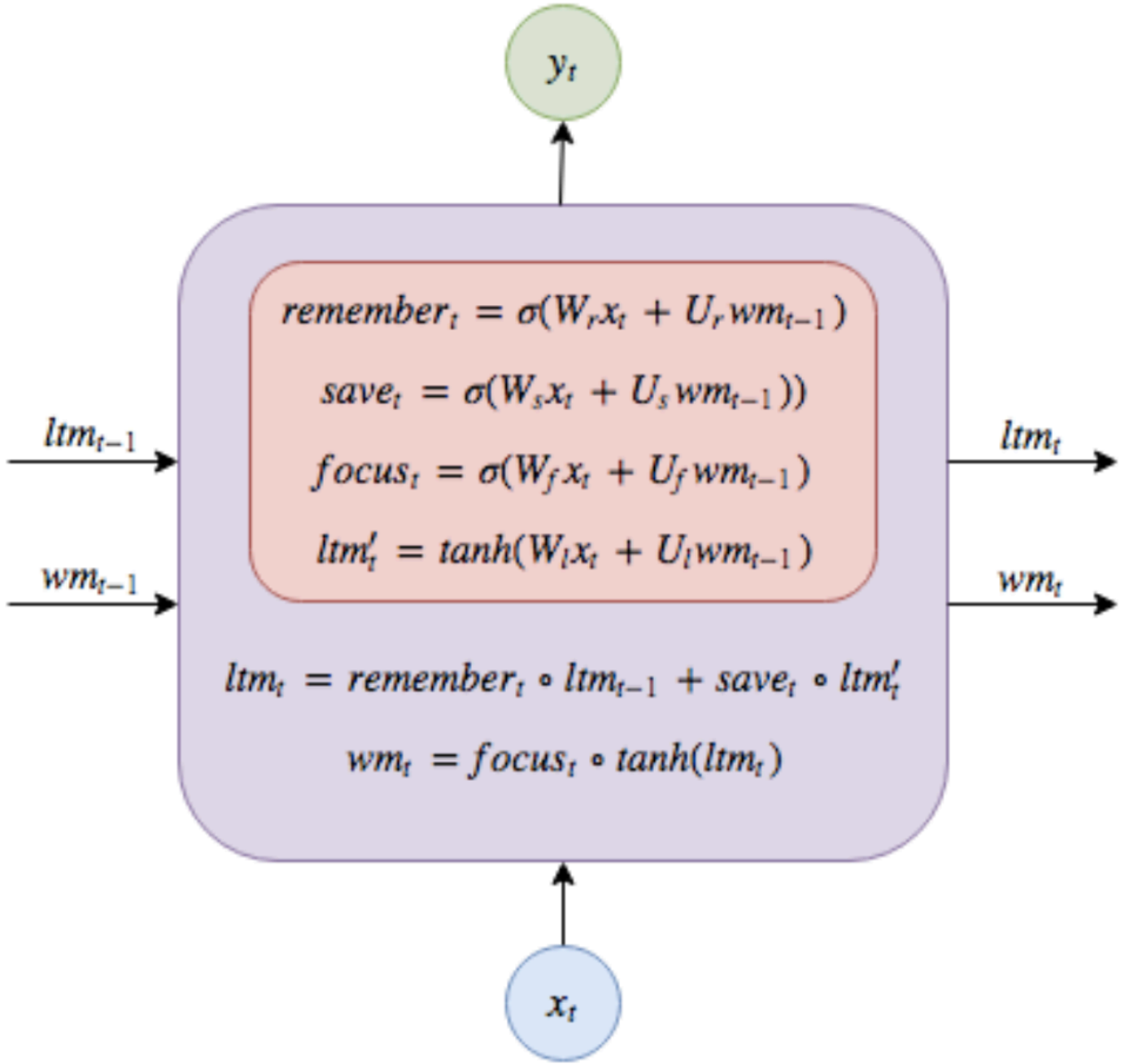
Longer Memories through LSTMs

- **Adding a forgetting mechanism.** If a scene ends, for example, the model should forget the current scene location, the time of day, and reset any scene-specific information; however, if a character dies in the scene, it should continue remembering that he's no longer alive. Thus, we want the model to learn separate *forgetting/remembering* mechanism: when new inputs come in, it needs to know which beliefs to keep or throw away.
- **Adding a saving mechanism.** When the model sees a new image, it needs to learn whether any information about the image is worth using and saving. Maybe your mom sent you an article about the Kardashians, but who cares?

Longer Memories through LSTMs

- So when a new input comes in, the model first forgets any long-term information it decides it no longer needs. Then it learns which parts of the new input are worth using, and saves them into its long-term memory.
- **Focusing long-term memory into working memory.** Finally, the model needs to learn which parts of its long-term memory are immediately useful. For example, Bob's age may be a useful piece of information to keep in the long term (children are more likely to be crawling, adults are more likely to be working), but is probably irrelevant if he's not in the current scene. So instead of using the full long-term memory all the time, it learns which parts to focus on instead.


output
long-term memory
LSTM
working memory
input



$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \sigma_h(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

Describe the LSTM additions mathematically

We'll start with our long-term memory. First, we need to know which pieces of long-term memory to continue remembering and which to discard, so we want to use the new input and our working memory to learn a **remember gate** of n numbers between 0 and 1, each of which determines how much of a long-term memory element to keep. (1 means to keep it, a 0 means to forget it entirely.)
Naturally, we can use a small neural network to learn this remember gate:

$$remember_t = \sigma(W_r x_t + U_r wm_{t-1})$$

Describe the LSTM additions mathematically

We'll start with our long-term memory. know which pieces of long-term memory to continue remembering and which to discard, so we want to use the new input and our working memory to learn a **remember gate**

$$remember_t = \sigma(W_r x_t + U_r wm_{t-1})$$

Next, we need to compute the information we can learn from x_t, i.e., a **candidate addition to our long-term memory**:

$$ltm'_t = \phi(W_l x_t + U_l wm_{t-1})$$

Describe the LSTM additions mathematically

We want to use the new input and our working memory to learn a **remember gate**

$$remember_t = \sigma(W_r x_t + U_r wm_{t-1})$$

Next, we need to compute the information we can learn from x_t, i.e., a **candidate addition to our long-term memory**:

$$ltm'_t = \phi(W_l x_t + U_l wm_{t-1})$$

Before we add the candidate into our memory, though, we want to learn **which parts of it are actually worth using and saving**:

$$save_t = \sigma(W_s x_t + U_s wm_{t-1})$$

Describe the LSTM additions mathematically

We want to use the new input and our working memory to learn a **remember gate**

$$remember_t = \sigma(W_r x_t + U_r wm_{t-1})$$

Next, we need to compute the information we can learn from x_t, i.e., a **candidate addition to our long-term memory**:

$$ltm'_t = \phi(W_l x_t + U_l wm_{t-1})$$

Before we add the candidate into our memory, though, we want to learn **which parts of it are actually worth using and saving**:

$$save_t = \sigma(W_s x_t + U_s wm_{t-1})$$

After forgetting memories we don't think we'll ever need again and saving useful pieces of incoming information, we have our **updated long-term memory**:

$$ltm_t = remember_t \circ ltm_{t-1} + save_t \circ ltm'_t$$

Describe the LSTM additions mathematically

Next, let's update our working memory. We want to learn how to focus our long-term memory into information that will be *immediately* useful. (Put differently, we want to learn what to move from an *external hard drive* onto our *working laptop*.) So we learn a **focus/attention vector**:

$$focus_t = \sigma(W_f x_t + U_f wm_{t-1})$$

Our **working memory** is then

$$wm_t = focus_t \circ \phi(ltm_t)$$

To summarize, whereas a vanilla RNN uses one equation to update its hidden state/memory:

$$h_t = \phi(Wx_t + Uh_{t-1})$$

An LSTM uses several:

$$ltm_t = remember_t \circ ltm_{t-1} + save_t \circ ltm'_t$$

$$wm_t = focus_t \circ tanh(ltm_t)$$

where each memory/attention sub-mechanism is just a mini brain of its own:

$$remember_t = \sigma(W_r x_t + U_r wm_{t-1})$$

$$save_t = \sigma(W_s x_t + U_s wm_{t-1})$$

$$focus_t = \sigma(W_f x_t + U_f wm_{t-1})$$

$$ltm'_t = tanh(W_l x_t + U_l wm_{t-1})$$