Inspire…Educate…Transform.

Recurrent Neural Networks (RNN)

Dr. K. V. Dakshinamurthy
President, International School of Engineering

When context is important

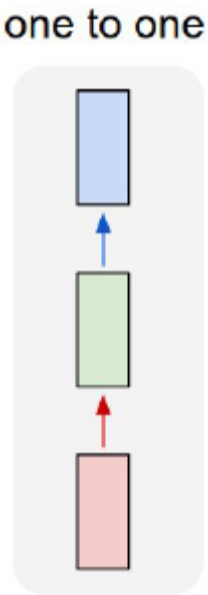
- Sometimes, very little context is needed for machine learning tasks but sometimes we need more context
  - Predicting next word
    - The clouds are in the _____ (sky and no further context is needed)
    - I fell in love with this French girl. My parents were against it initially as they were worried about the cultural differences. However, after they met her they realized how wonderful a person she is and agreed for our marriage. All that is left is convincing her parents. Here, I am booking my tickets to _____ (A lot of context is needed)

How to solve

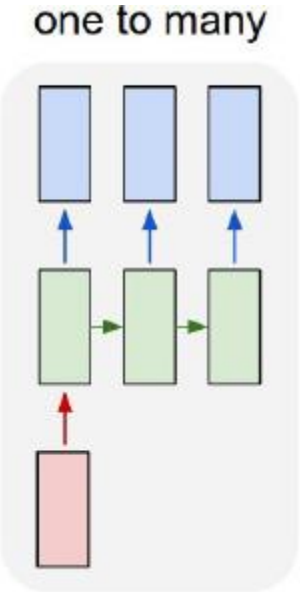
- A human carefully analyzes the passage and picks the right features to put in the memory. This might work for toy problems.

- A natural architecture that remembers the relevant past information. Recurrent neural networks is a powerful solution

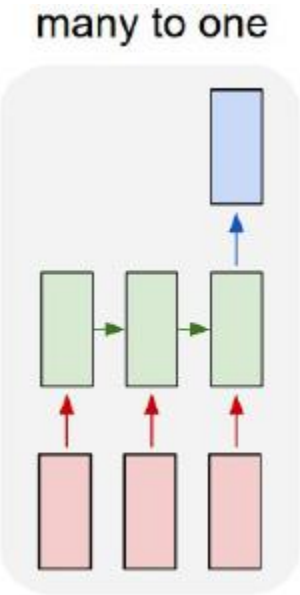
Tasks where context is useful



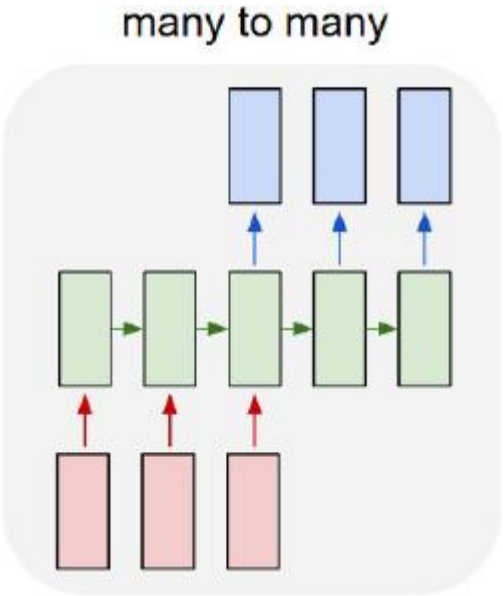
e.g. image classification



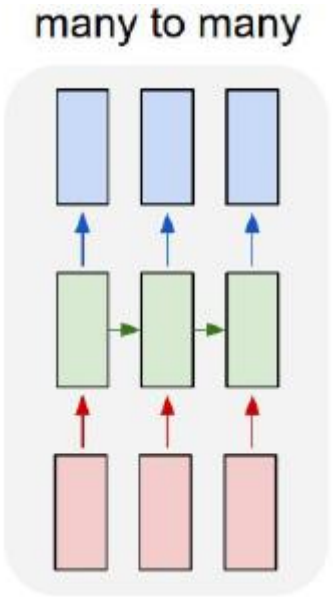
e.g. image captioning takes an image and outputs a sentence of words



e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment



Machine Translation

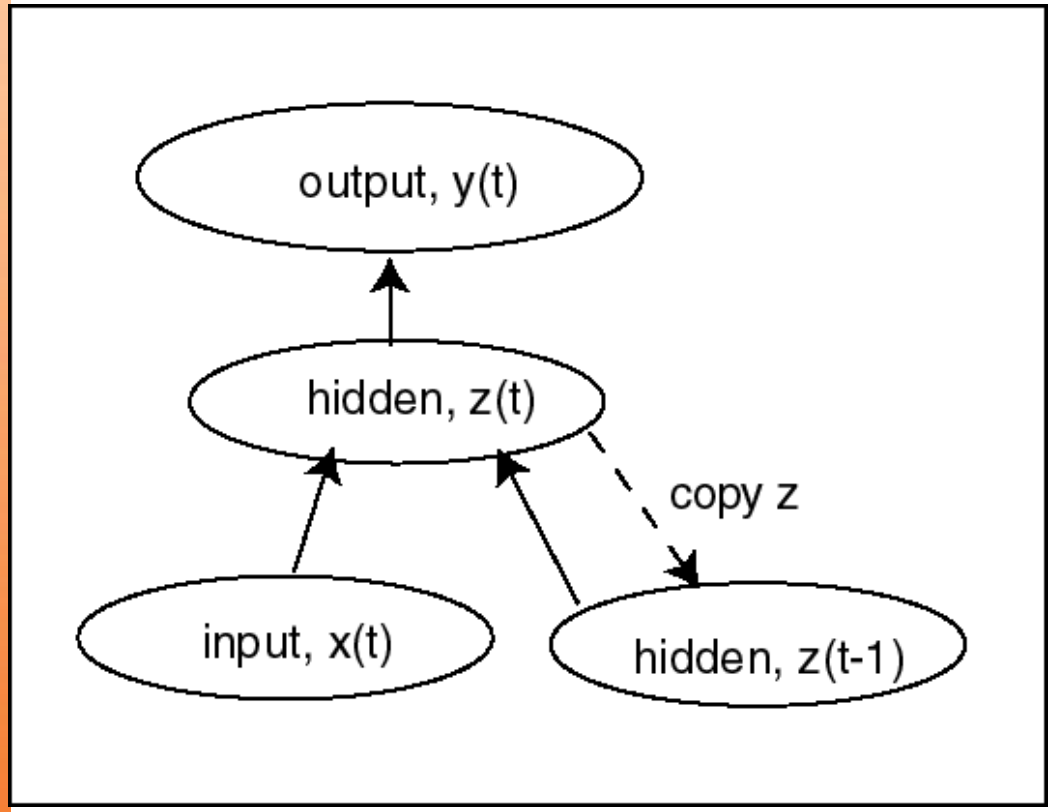


Video classification where we wish to label each frame of the video

RNNs

At the core, RNNs have a deceptively simple API: They accept an input vector x and give you an output vector y. However, crucially this output vector's contents are influenced not only by the input you just fed in, but also on the entire history of inputs you've fed in in the past.

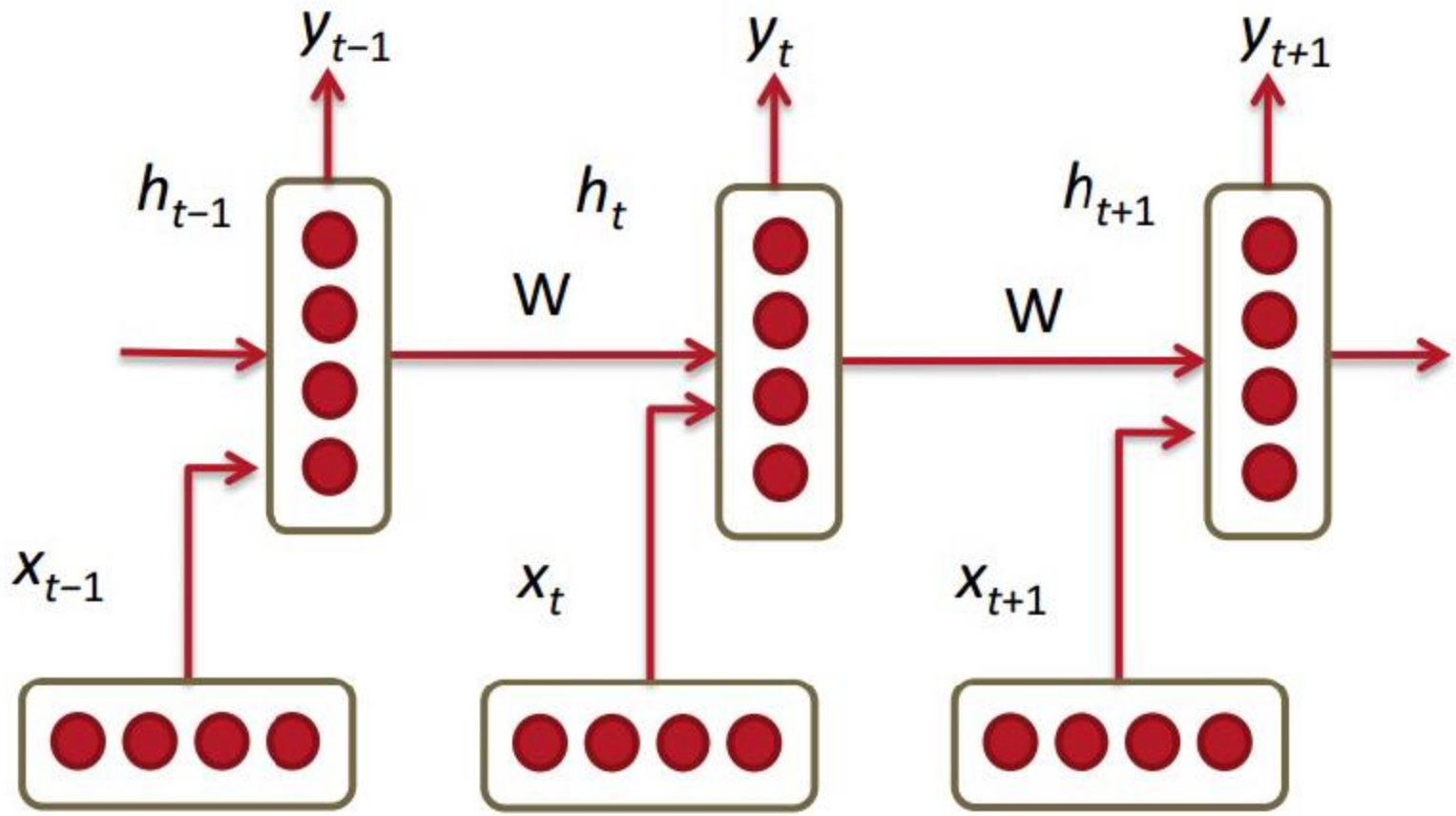
The Simple Recurrent Network



Elman network (after Jeff Elman, the originator), or as a **Simple Recurrent Network**. At each time step, a copy of the hidden layer units is made to a copy layer. Processing is done as follows:

1.Copy inputs for time *t* to the input units
2.Compute hidden unit activations using net input from input units and from copy layer
3.Compute output unit activations as usual
4.Copy new hidden unit activations to copy layer

A different perspective

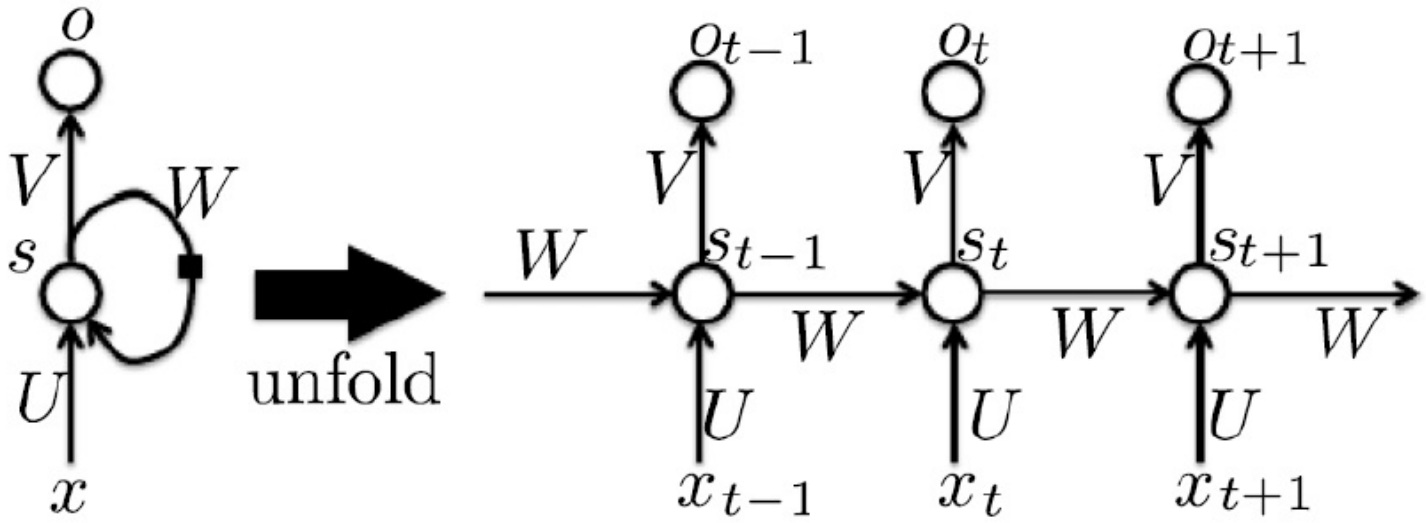


A generalization of this approach is to copy the input and hidden unit activations for a number of previous time steps.

The more context (copy layers) we maintain, the more history we are explicitly including in our gradient computation.

This approach has become known as **Back Propagation Through Time**. It can be seen as an approximation to the ideal of computing a gradient which takes into consideration not just the most recent inputs, but all inputs seen so far by the network

RNN



$$\mathbf{a}_t = \mathbf{b} + W\mathbf{s}_{t-1} + U\mathbf{x}_t$$

$$\mathbf{s}_t = tanh(\mathbf{a}_t)$$

$$\mathbf{o}_t = \mathbf{c} + V\mathbf{s}_t$$

$$\mathbf{p}_t = softmax(\mathbf{o}_t)$$

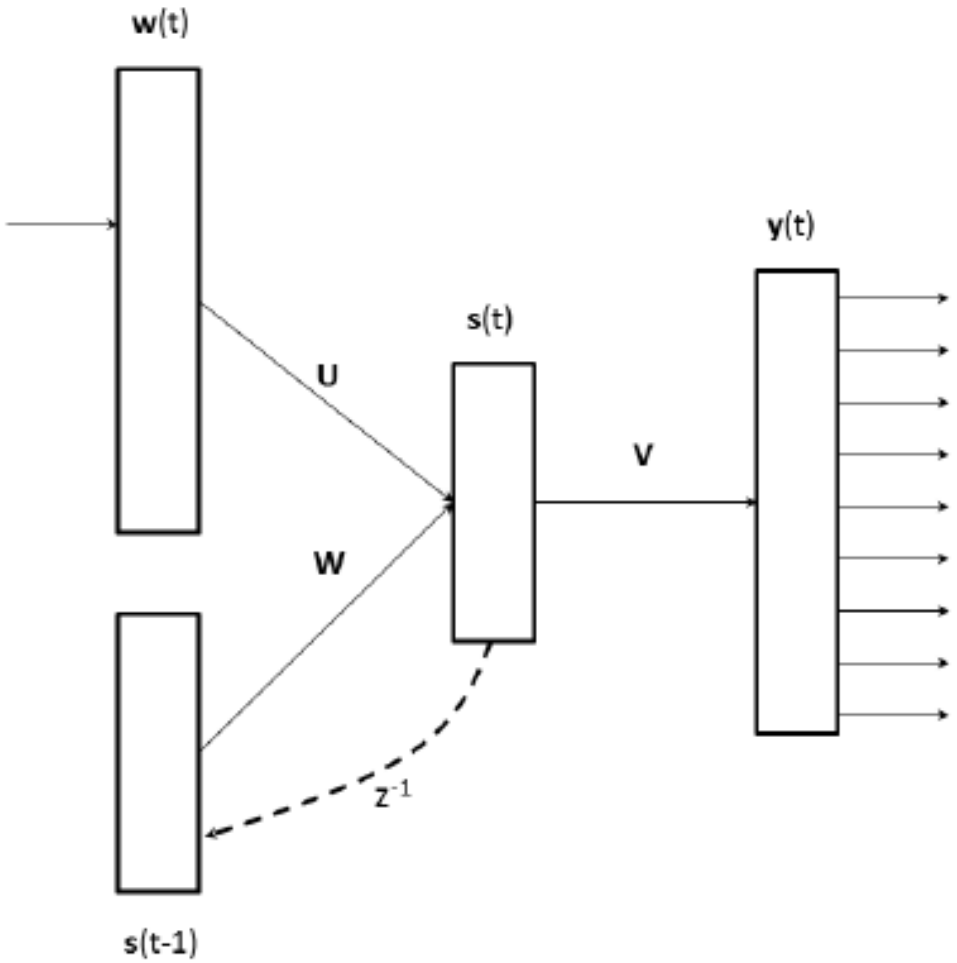
Parameter sharing

We can think of s_t as a summary of the past sequence of inputs up to t.

If we define a different function g_t for each possible sequence length, we would not get any generalization.

If the same parameters are used for any sequence length allowing much better generalization properties.

W2V through RNN



$$\mathbf{s}(t) = f\left(\mathbf{U}\mathbf{w}(t) + \mathbf{W}\mathbf{s}(t-1)\right)$$

$$\mathbf{y}(t) = g\left(\mathbf{V}\mathbf{s}(t)\right),$$

where

$$f(z) = \frac{1}{1+e^{-z}}, \quad g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}.$$

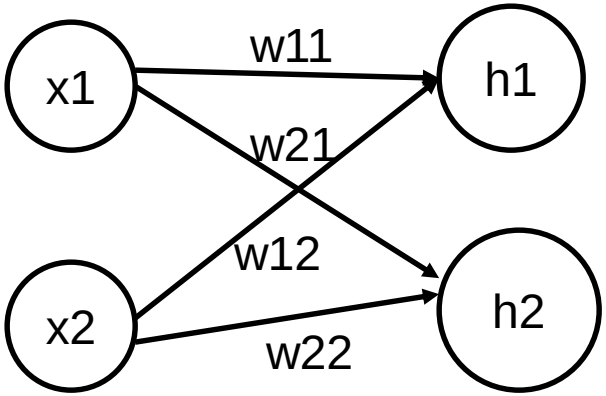
Training

- 100M words

- The RNN is trained with backpropagation to maximize the data log-likelihood under the model. The model itself has no knowledge of syntax or morphology or semantics.

The word representations

- The hidden layer s(t) maintains a representation of the sentence history. The input vector w(t) and the output vector y(t) have dimensionality of the vocabulary.



$$\begin{bmatrix} w11 & w12 \\ w21 & w22 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \end{bmatrix} = \begin{bmatrix} w11x1 + w12x2 \\ w21x1 + w22x2 \end{bmatrix}$$

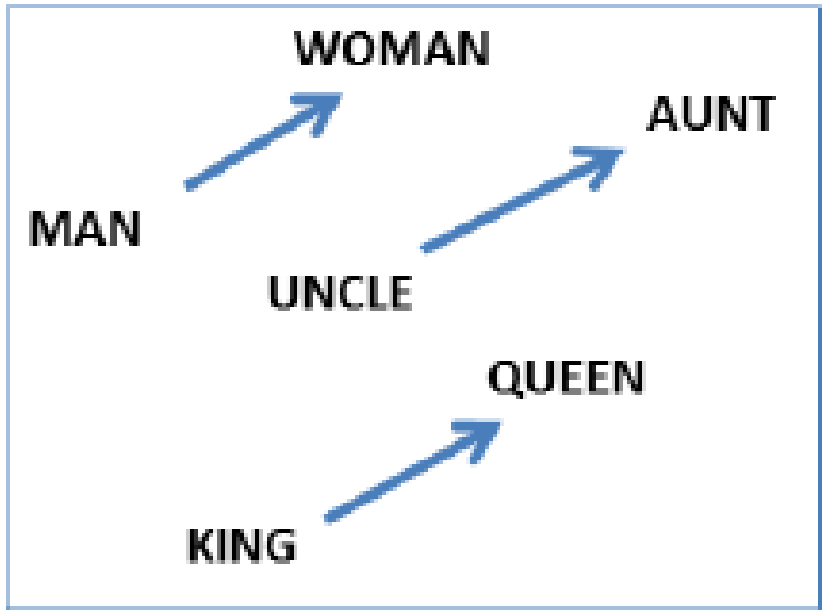
The word vectors

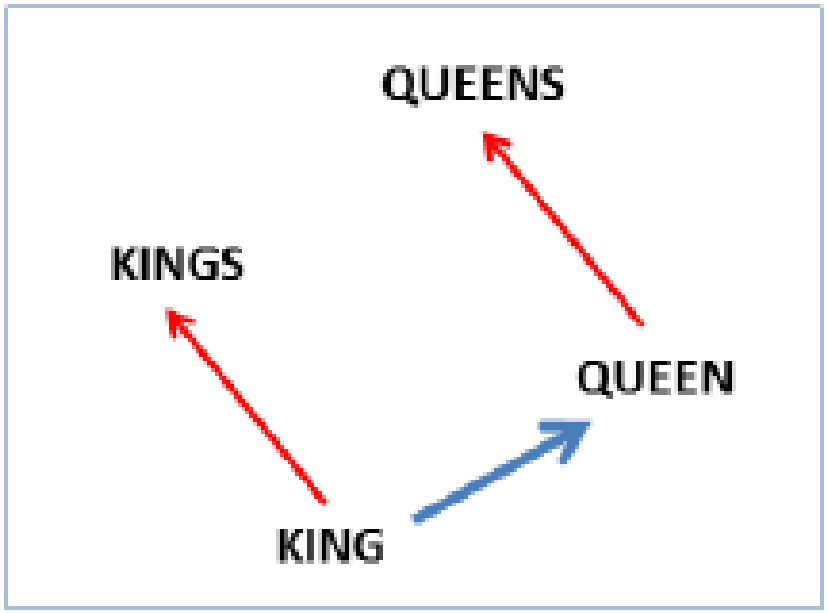
- the word representations are found in the columns of U, with each column representing a word.
- The dimensions of each word can be experimented
- The output is the probability distribution of words

In this model, to answer the analogy question a:b c:d where d is unknown, we find the embedding vectors x_a, x_b, x_c (all normalized to unit norm), and compute $y = x_b - x_a + x_c$. y is the continuous space representation of the word we expect to be the best answer. Of course, no word might exist at that exact position, so we then search for the word whose embedding vector has the greatest cosine similarity to y and output it:

$$w^* = argmax_w \frac{x_w y}{\|x_w\|\|y\|}$$

Semantics and Syntactics





Results

Method	Adjectives	Nouns	Verbs	All
LSA-80	9.2	11.1	17.4	12.8
LSA-320	11.3	18.1	20.7	16.5
LSA-640	9.6	10.1	13.8	11.3
RNN-80	9.3	5.2	30.4	16.2
RNN-320	18.2	19.0	45.0	28.5
RNN-640	21.0	25.2	54.8	34.7
RNN-1600	**23.9**	**29.2**	**62.2**	**39.6**

Table 2: Results for identifying syntactic regularities for different word representations. Percent correct.

Method	Spearman's ρ	MaxDiff Acc.
LSA-640	0.149	0.364
RNN-80	0.211	0.389
RNN-320	0.259	0.408
RNN-640	0.270	0.416
RNN-1600	**0.275**	**0.418**
CW-50	0.159	0.363
CW-100	0.154	0.363
HLBL-50	0.149	0.363
HLBL-100	0.146	0.362
UTD-NB	0.230	0.395

Table 4: Results in measuring relation similarity

ISSUE WITH PLAIN VANILLA RNNS

Exploding or vanishing products of jacobians

In recurrent nets (also in very deep nets), the final output is the composition of a large number of non-linear transformations.

Even if each of these non-linear transformations is smooth. Their composition might not be.

The derivatives through the whole composition will tend to be either very small or very large.

Simple example

Suppose: all the numbers in the product are scalar and have the same value α.

multiplying many numbers together tends to be either very large or very small.

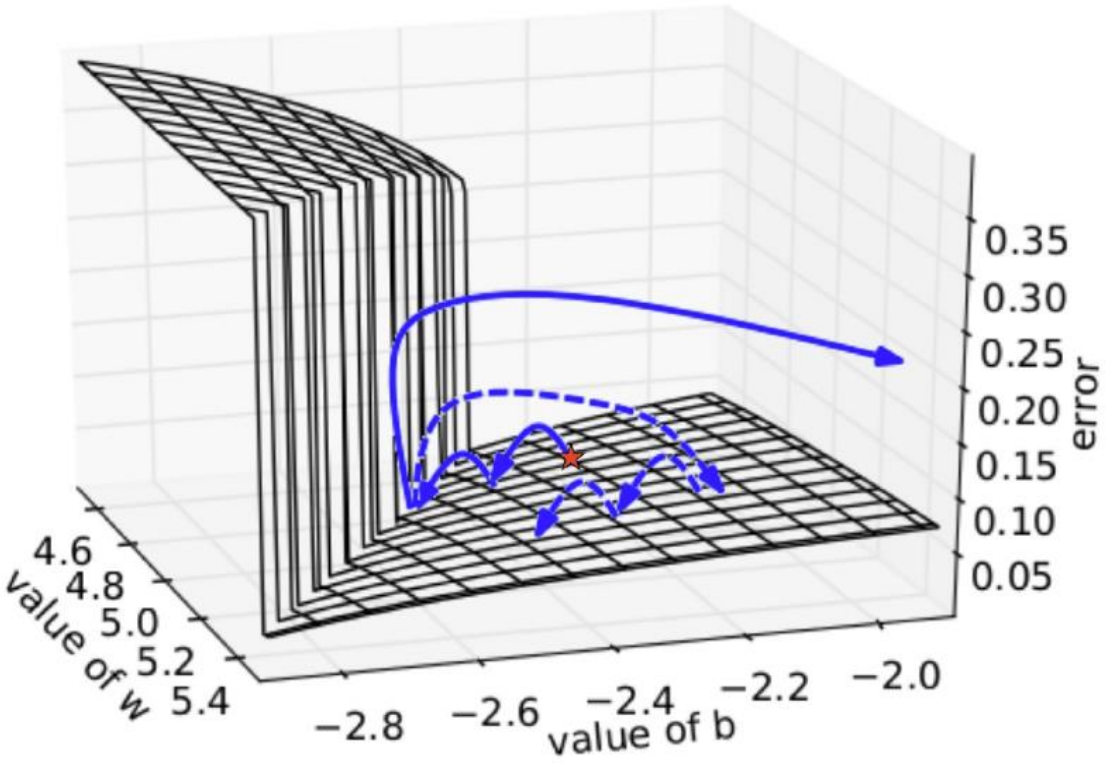
If T goes to ∞, then

α^T goes to ∞ if $\alpha > 1$

α^T goes to 0 if $\alpha < 1$

Now, let's say we started at the red star (using a random initialization of weights). You'll notice that as we use gradient descent, we get closer and closer to the local minimum on the surface.

But suddenly, when we slightly overreach the valley and hit the cliff, we are presented with a massive gradient in the opposite direction. This forces us to bounce extremely far away from the local minimum. And once we're in nowhere land, we quickly find that the gradients are so vanishingly small that coming close again will take a seemingly endless amount of time. This issue is called the problem of exploding and vanishing gradients.



Exploding gradients

Simple solution for clipping the gradient. (Mikolov, 2012; Pascanu et al., 2013):

Clip the parameter gradient from a mini batch element-wise (Mikolov, 2012) just before the parameter update.

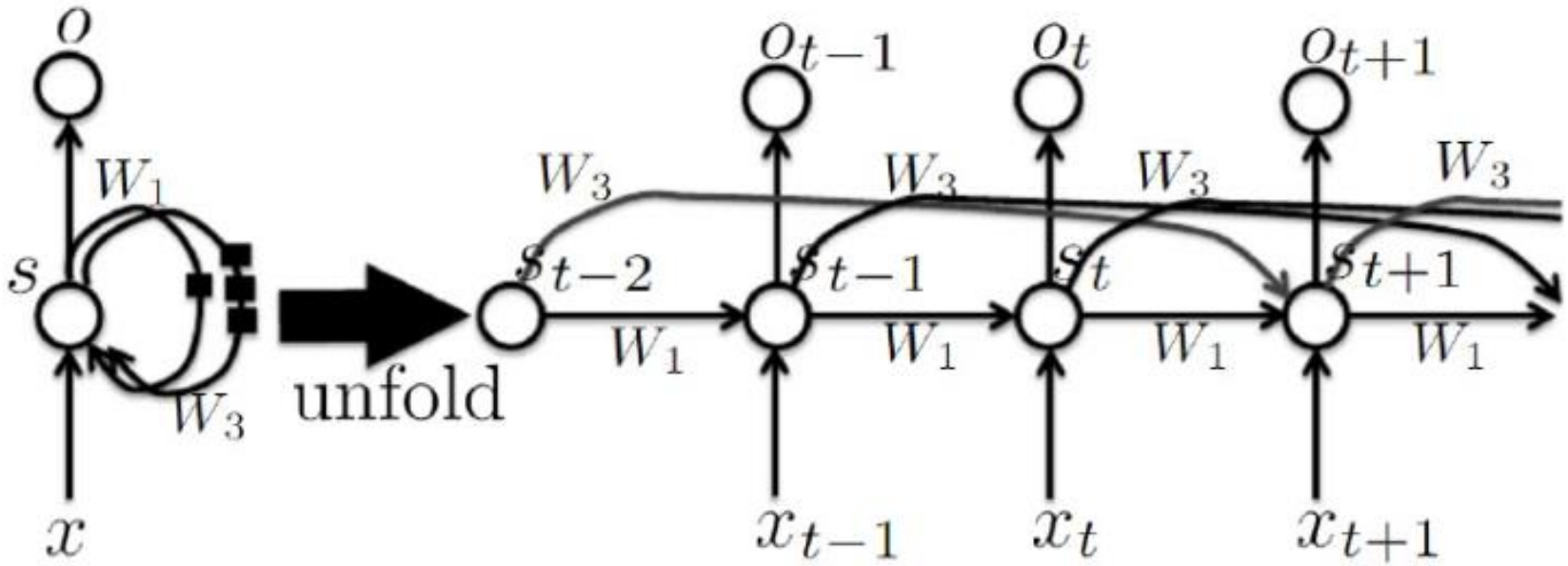
Clip the norm g of the gradient g (Pascanu et al., 2013a) just before the parameter update.

You can imagine perhaps controlling this issue by rescaling gradients to never exceed a maximal magnitude (see the dotted path after hitting the cliff), but this approach still doesn't perform spectacularly well, especially in more complex RNNs.

SOLUTIONS FOR VANISHING GRADIENTS

Long delay RNNs

Use recurrent connections with long delays.



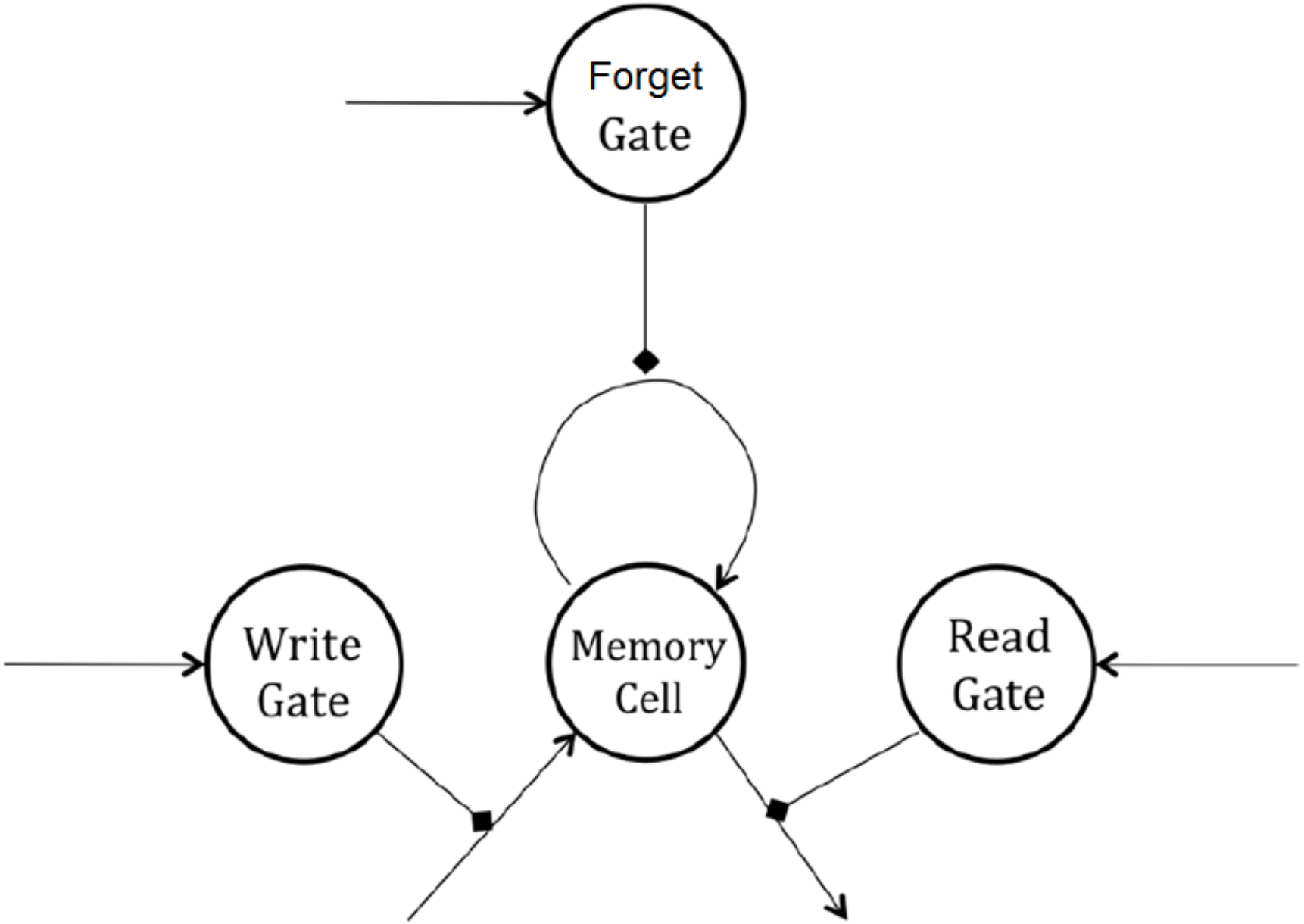
$In\ one\ step\ delay$, decrement after t steps is proportional to λ^t

$In\ d\ step\ delay, decrement\ after\ t\ steps\ is\ proportional\ to\ \lambda^{\frac{t}{d}}$

The gradient does not vanish or explode for longer terms.

LSTM (Long short term memory)

GATED RNNS

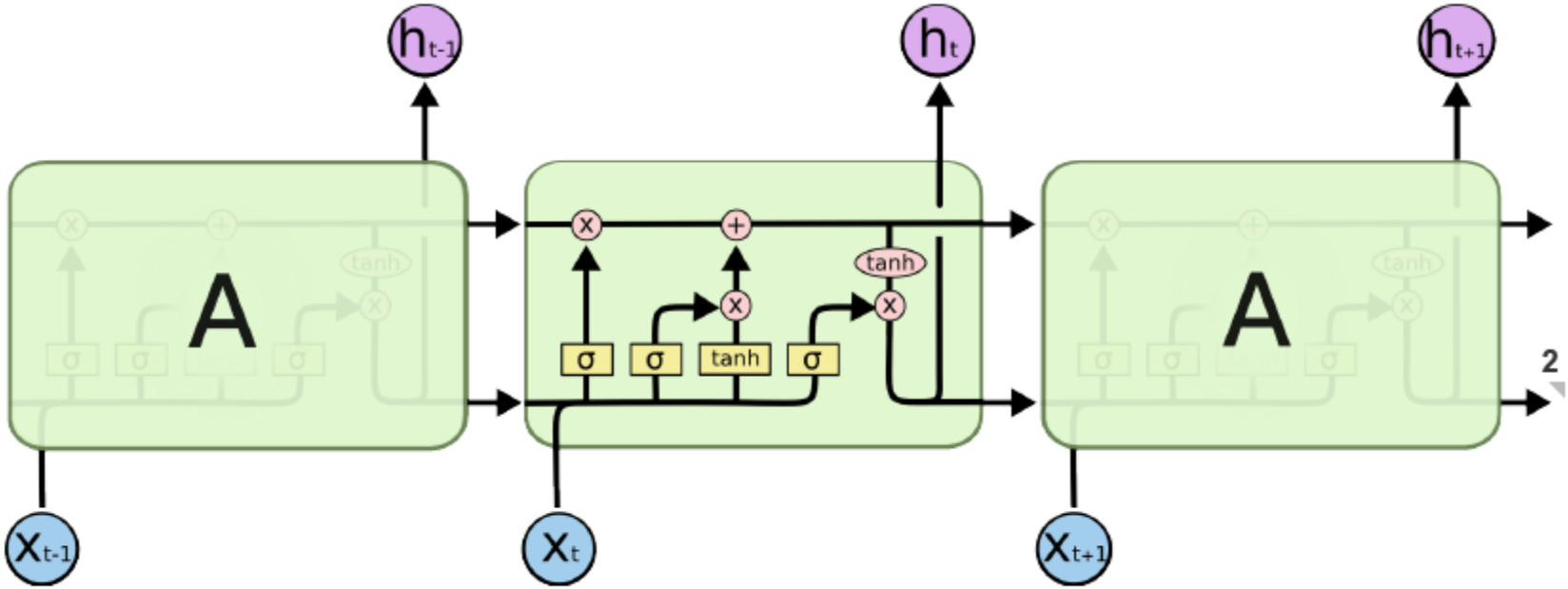

Forget Gate
Write Gate
Memory Cell
Read Gate

LSTMs

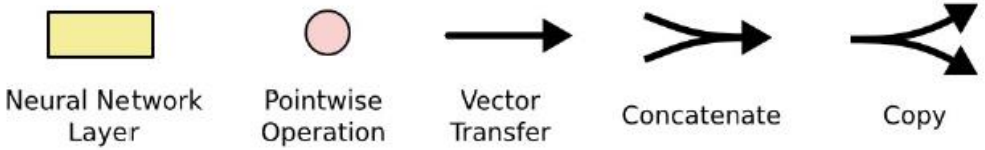
LSTMs help preserve the error that can be backpropagated through time and layers. By maintaining a more constant error, they allow recurrent nets to continue to learn over many time steps (over 1000), thereby opening a channel to link causes and effects remotely.

Information can be stored in, written to, or read from a cell, much like data in a computer's memory. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close.

Those gates act on the signals they receive, and similar to the neural network's nodes, they block or pass on information based on its strength and import, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the recurrent networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, backpropagating error, and adjusting weights via gradient descent.

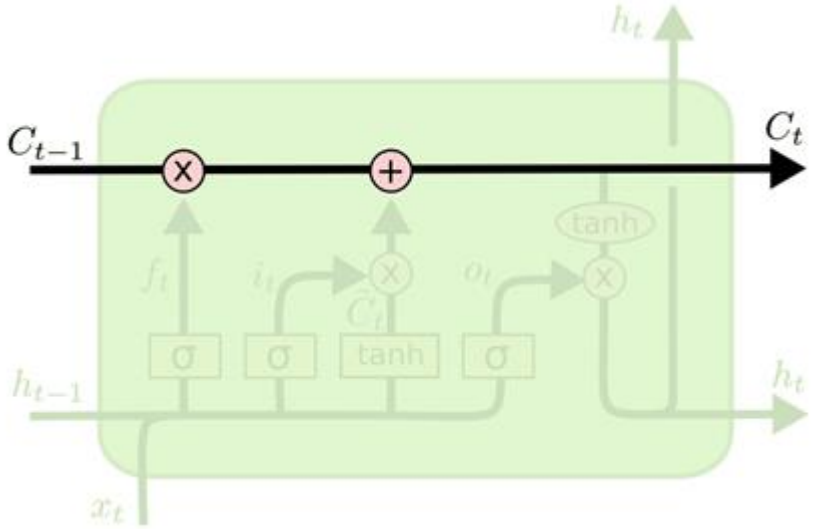


The repeating module in an LSTM contains four interacting layers.



Each line carries an entire vector, from the output of one node to the inputs of others.
The pink circles are pointwise operations, like vector addition,
The yellow boxes are learned neural network layers.
Lines merging denote concatenation,
Line forking denote its content being copied and the copies going to different locations.

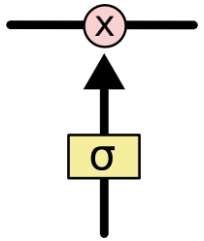
Cell state



The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

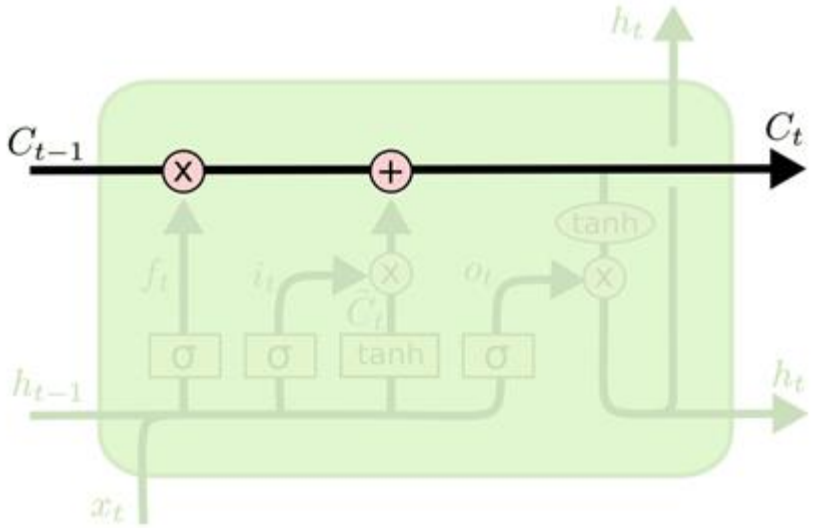
The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



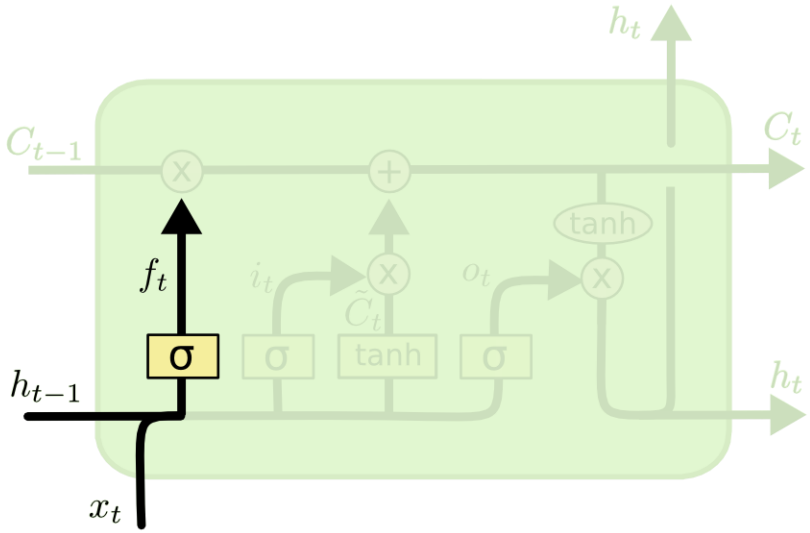
The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”

An LSTM has three of these gates, to protect and control the cell state.



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 ht−1 and xt, and outputs a number between 0 and 1 for each number in the cell state Ct−1.

1 represents "completely keep this" while a 0 represents "completely get rid of this."

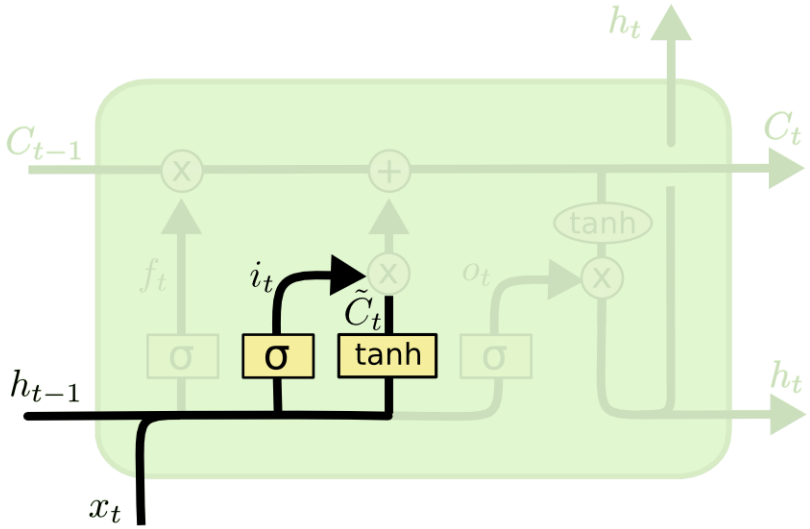


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

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.

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, Ct, 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.



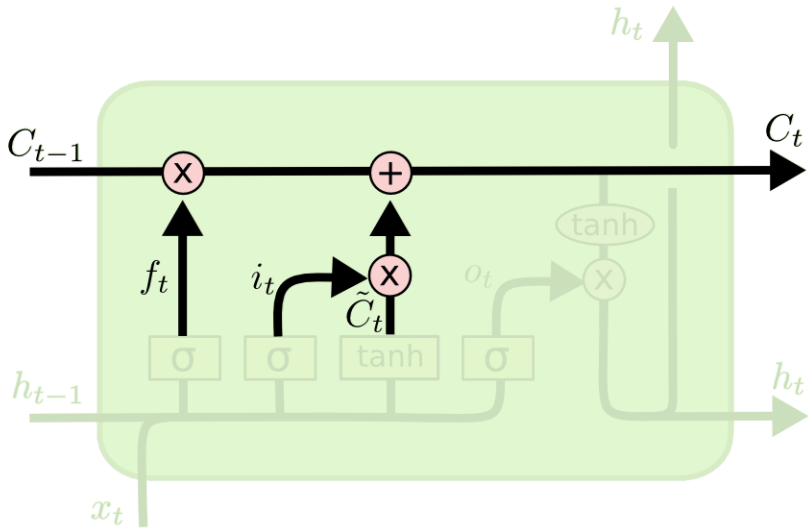
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

It’s now time to update the old cell state, Ct−1, into the new cell state Ct. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by ft, forgetting the things we decided to forget earlier. Then we add it∗C~t. This is the new candidate values, scaled by how much we decided to update each state value.

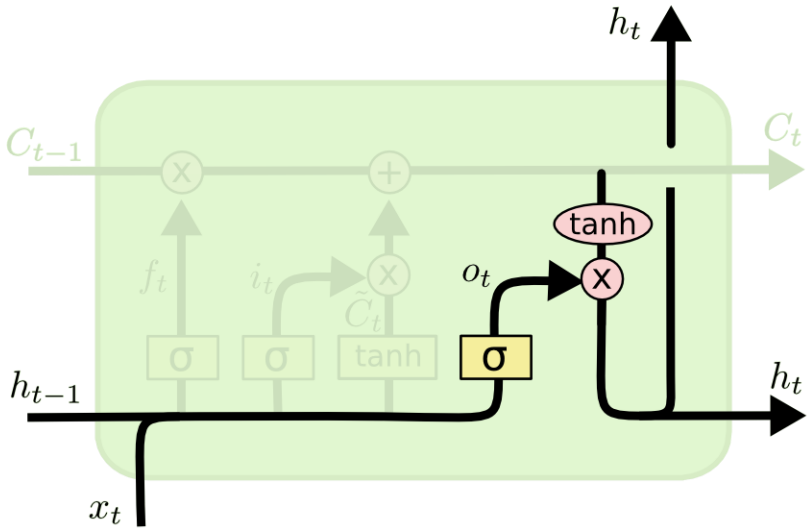
In the case of the language model, this is where we’d actually drop the information about the old subject’s gender and add the new information, as we decided in the previous steps.



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

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.



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

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

The architecture forces constant error flow (thus, neither exploding nor vanishing) through the internal state of special memory units.

LSTM operations in a sequence

Decide how much to input and how much to forget

Let us say current memory is c_{t-1}

New memory cell $\tilde{c}_t = tanh(W^{(c)}x_t + U^{(c)}h_{t-1})$

Input gate (current cell matters) $i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1})$

Forget (gate 0, forget past) $f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1})$

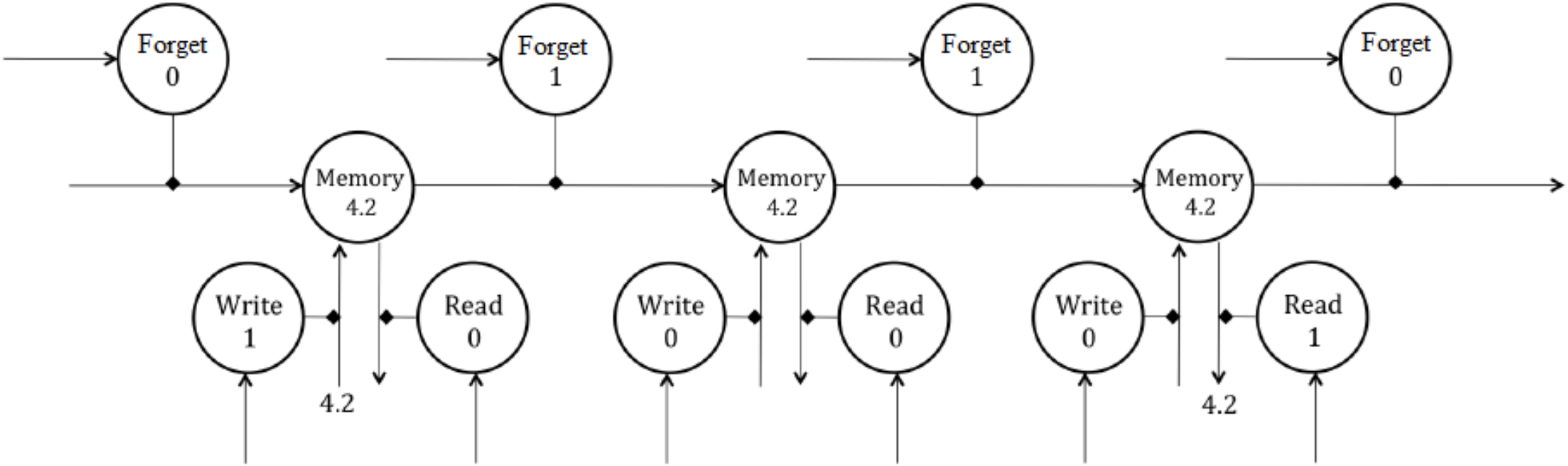
Final memory cell: $c_t = f_t \circ c_{t-1} + (i_t) \circ \tilde{c}_t$

LSTM operations: How much to send to next layers

Entire memory is not sent every time

Output (how much cell is exposed) $o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1})$

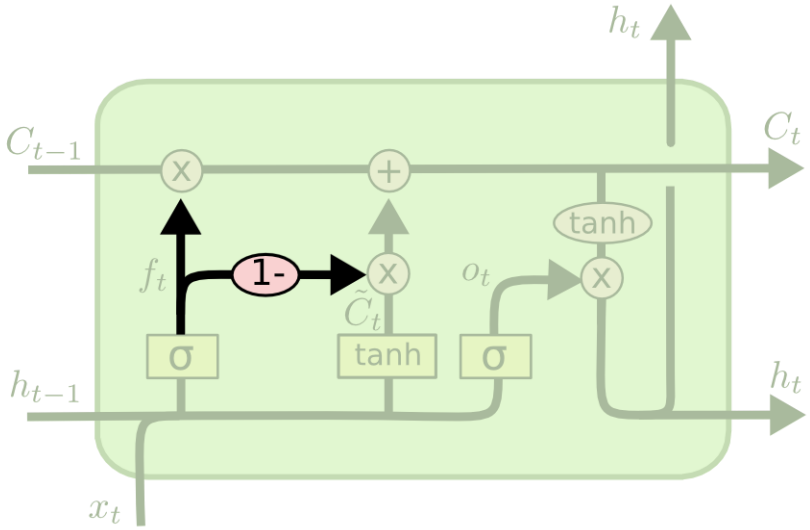
Final hidden state: $h_t = o_t \circ tanh(c_t)$


Forget
0
Forget
1
Forget
1
Forget
0
Memory
4.2
Memory
4.2
Memory
4.2
Write
1
Read
0
4.2
Write
0
Read
0
Write
0
Read
1
4.2

At first, the keep gate is set to 0 and the write gate is set to 1, which places 4.2 into the memory cell. This value is retained in the memory cell by a subsequent keep value of 1 and protected from read/write by values of 0. Finally, the cell is read and then cleared. Now we try to follow the backpropagation from the point of loading 4.2 into the memory cell to the point of reading 4.2 from the cell and its subsequent clearing. We realize that due to the linear nature of the memory neuron, the error derivative that we receive from the read point backpropagates with negligible change until the write point because the weights of the connections connecting the memory cell through all the time layers have weights approximately equal to 1 (approximate because of the logistic output of the keep gate). As a result, we can locally preserve the error derivatives over hundreds of steps without having to worry about exploding or vanishing gradients.

Many variants are available

One variation is to use coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we're going to input something in its place. We only input new values to the state when we forget something older.



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

Gated Recurrent Unit

A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit, or GRU, introduced by Cho, et al. (2014). The resulting simple model just computes new memory through a gate and adds is to existing memory through another gate.

Compute what new memory to use through a reset gate

$$\text{Reset gate: } r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

If reset gate is **0**, then this ignores previous memory and only stores the new information

$$\text{New memory content: } \tilde{h}_t = tanh(wx_t + r_t \circ Uh_{t-1})$$

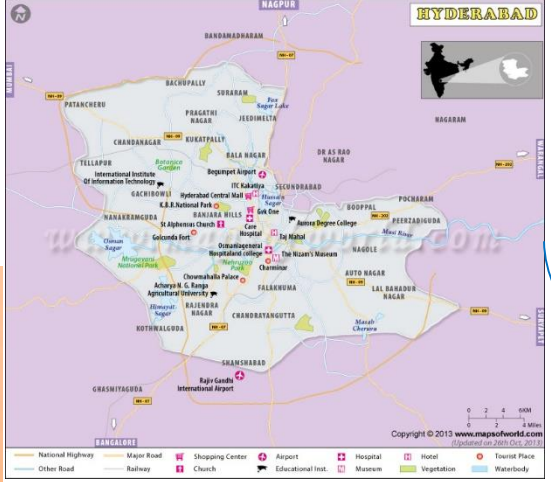
How much of new and how much of old is decided through an update gate

$$\text{Update gate: } z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

New memory is a combination of old memory and newly computed update

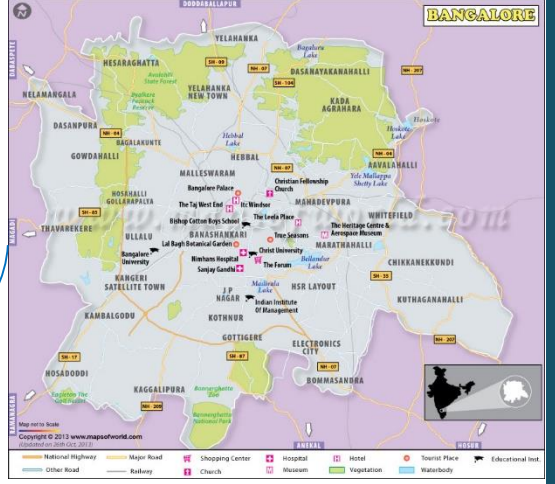
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- **RMSProp** / Adam / Adagrad (SGD can work too, but has much higher sensitivity to learning rate)
- **Clip gradients** (at 5.0 is a common value to use)
- **Initialize forget gates** with high bias (to encourage remembering at start) can help
- **L2 regularization** not very common, can even hurt sometimes
- **Dropout** always good along depth, but NOT along time (in the recurrent part). [*Zaremba et al.*]
- Typical training on good GPU: ~10mil params, 1-2 days



INSOFE

Inspire...Educate...Transform.



HYDERABAD

Office and Classrooms

Plot 63/A, Floors 1&2, Road # 13, Film Nagar,
Jubilee Hills, Hyderabad - 500 033
+91-9701685511 (Individuals)
+91-9618483483 (Corporates)

BENGALURU

Office

Incubex, #728, Grace Platina, 4th Floor, CMH Road,
Indira Nagar, 1st Stage, Bengaluru – 560038
+91-9502334561 (Individuals)
+91-9502799088 (Corporates)

Classroom

KnowledgeHut Solutions Pvt. Ltd., Reliable Plaza,
Jakkasandra Main Road, Teacher's Colony, 14th Main
Road, Sector – 5, HSR Layout, Bengaluru - 560102

Social Media

Web:	http://www.insofe.edu.in
Facebook:	https://www.facebook.com/insofe
Twitter:	https://twitter.com/Insofeedu
YouTube:	http://www.youtube.com/InsofeVideos
SlideShare:	http://www.slideshare.net/INSOFE
LinkedIn:	http://www.linkedin.com/company/international-school-of-engineering

This presentation may contain references to findings of various reports available in the public domain. INSOFE makes no representation as to their accuracy or that the organization subscribes to those findings.