Deep Learning (CS324)

6. Recurrent Neural Networks*

Jianguo Zhang

SUSTech

*Based on http://www.deeplearningbook.org chapter 10 + sources mentioned in slides

Learning over sequences

- A lot of data naturally comes in the form of sequences

Videos

Source: https://www.eltima.com/images/upload/elmedia/articles/frame/sample.jpg

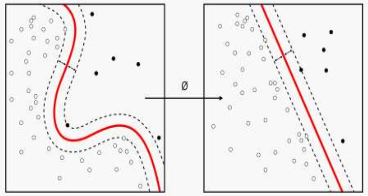
Text (and speech)

A **recurrent neural network** (**RNN**) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition[1] or speech recognition.[2][3]

The term "recurrent neural network" is used indiscriminately to refer to two broad classes of networks with a similar general structure, where one is finite impulse and the other is infinite impulse. Both classes of networks exhibit temporal dynamic behavior.[4] A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that can not be unrolled.

Both finite impulse and infinite impulse recurrent networks can have additional stored state, and the storage can be under direct control by the neural network. The storage can also be replaced by another network or graph, if that incorporates time delays or has feedback loops. Such controlled states are referred to as gated state or gated memory, and are part of long short-term memory networks (LSTMs) and gated recurrent units.This is also called Feedback Neural Network.

Machine learning and data mining



Problems [show]

Supervised learning [show]
(**classification** • **regression**)

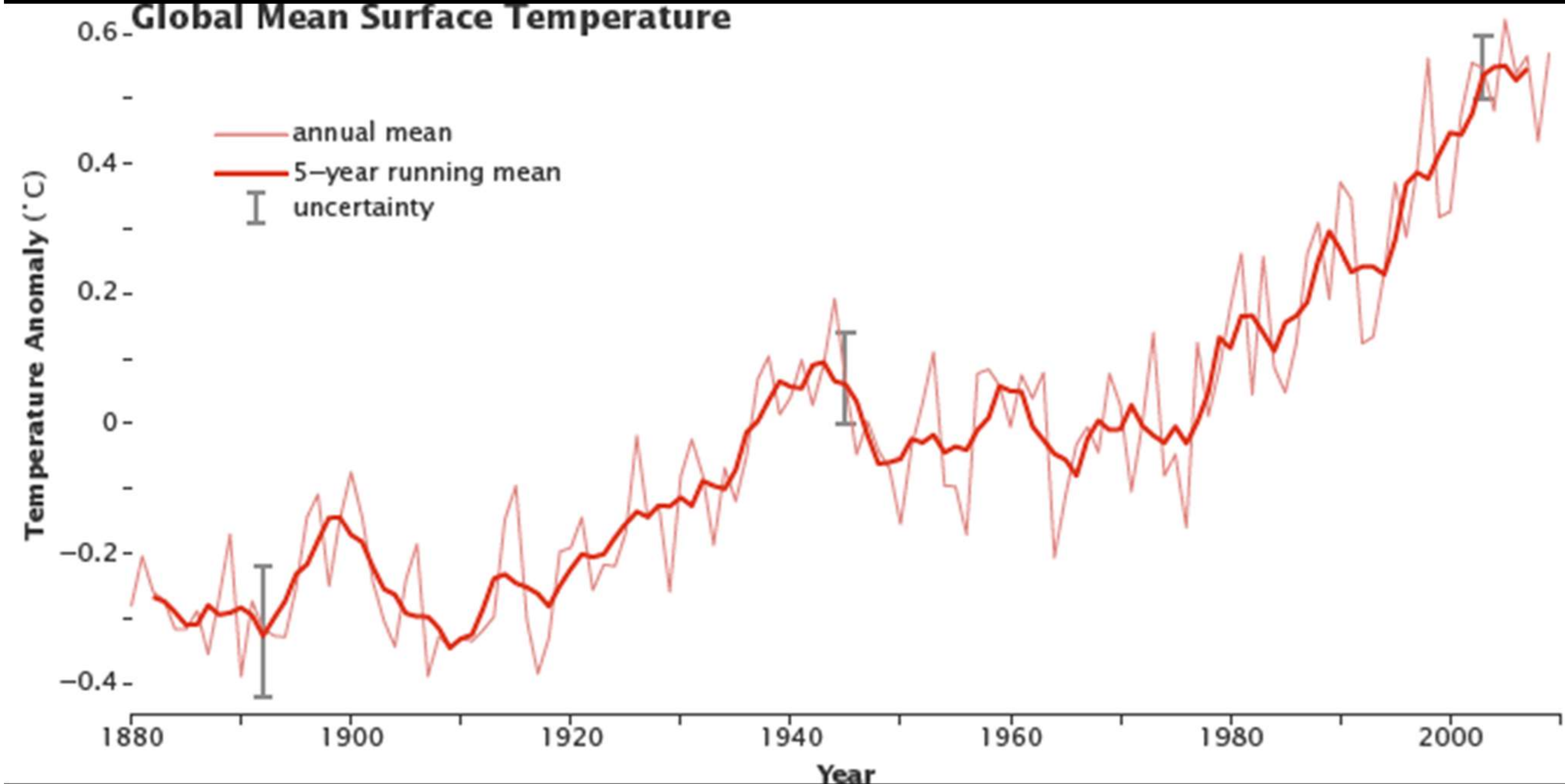
Clustering [show]

Dimensionality reduction [show]

Structured prediction [show]

Source: https://en.wikipedia.org/wiki/Recurrent_neural_network

Climate measurements



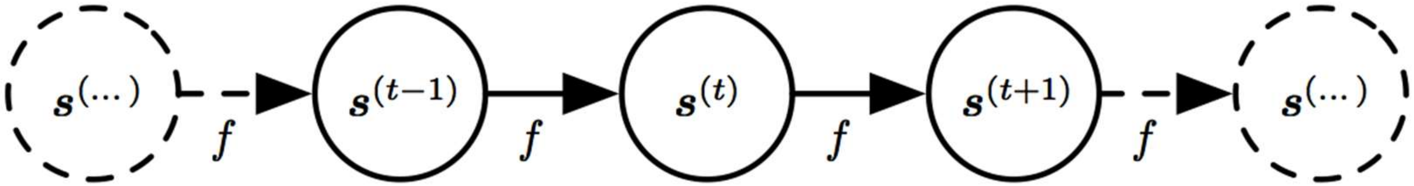
Source: https://earthobservatory.nasa.gov/features/GlobalWarming/page2.php

Music

Source: https://www.everynote.com/goods.pic/Beet_OdeJoy_ClPia.gif

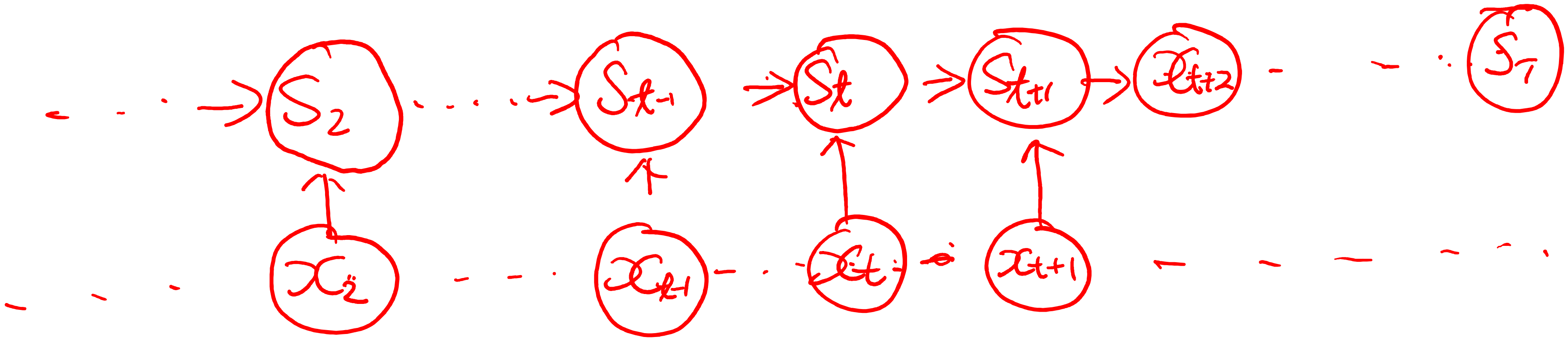
Learning over sequences

- As with images, we need to take structure (normally time temporal dependencies) into account, but note that in sequences **future and past** are not treated "symmetrically" (but could be, e.g., bidirectional …)
- Also, we can have very **long temporal dependencies** and **inputs/outputs** of arbitrary length, even **infinite length**



Learning over sequences

- Transition matrix: $\mathrm{S_t} = f(S_{t-1})$
  - 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
- However, it can't model the sequence below (Example (noisy):
- 1.1, 2.2, 3.1, 1.3, 2.2, 3.3, 1.4, 2.4, 3.3, 1.1 ,2.0, 3.1, 1.2, 2.1, 3.0, 1.1, 2.3, 3.4, ...
  - we need to use $\mathrm{S_t} = f(S_{t-1}, X_t)$



S_2
S_{t-1}
S_t
S_{t+1}
x_{t+2}
S_T
x_2
x_{t-1}
x_t
x_{t+1}

Learning over sequences

- Consider the word ‘mountain’
- Given the string ‘mountai’, there is strong probability that the next character will be ‘n’
- This is precisely what we want our neural network to be able to model
- More in general, we need to somehow **model context and memory**

Text classification: Examples

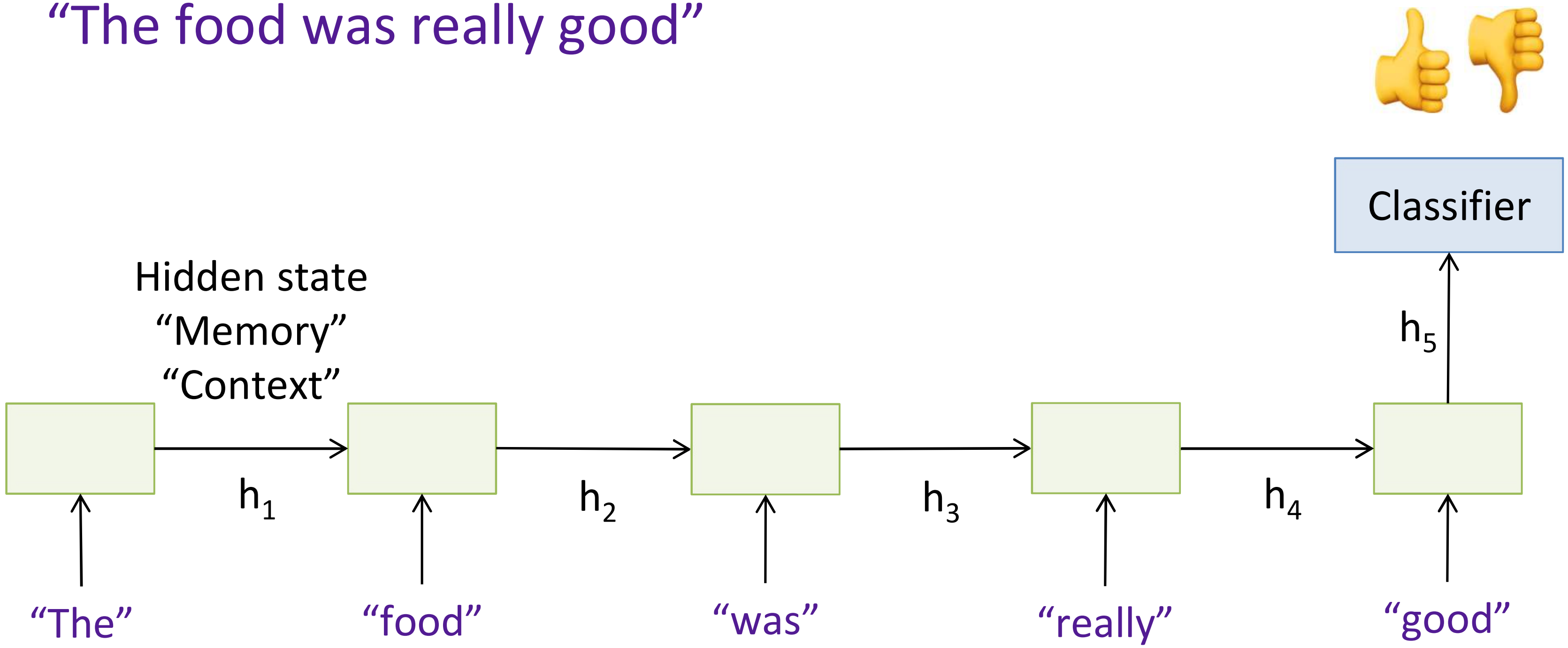
- **Sentiment classification:** classify a restaurant or movie or product review as positive or negative

    – "The food was really good"
    – "The vacuum cleaner broke within two weeks"
    – "The movie had slow parts, but overall was worth watching"

- What feature representation or predictor structure can we use for this problem?

Sentiment classification

- “The food was really good”



Recurrent Neural Network (RNN)

Language Modeling

- Character RNN

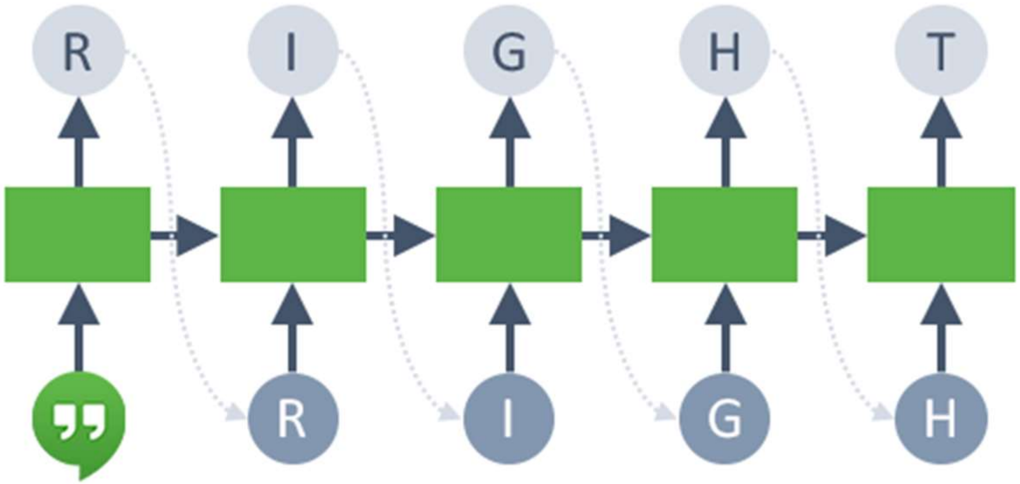


Image source

Input representation

- In the following we assume our inputs are either characters (e.g., the network's task it to predict the next one) or words (e.g., the network's task is to predict the next word)
- In general, we will have a sequence of symbols in input that can be drawn from a larger vocabulary

Input representation

- We start with a one-hot vector representation

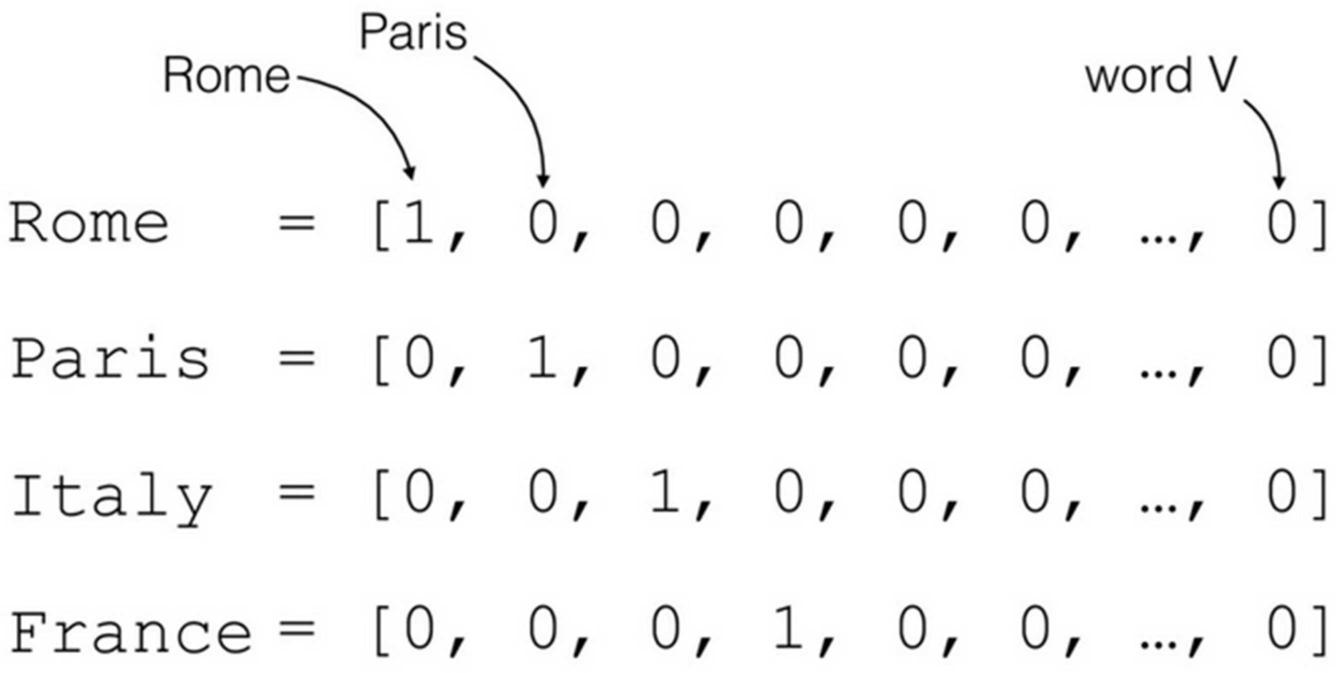
Rome Paris word V

```
Rome   = [1, 0, 0, 0, 0, 0, …, 0]
Paris  = [0, 1, 0, 0, 0, 0, …, 0]
Italy  = [0, 0, 1, 0, 0, 0, …, 0]
France = [0, 0, 0, 1, 0, 0, …, 0]
```

Source: https://medium.com/@athif.shaffy/one-hot-encoding-of-text-b69124bef0a7

Input representation

- We start with a one-hot vector representation

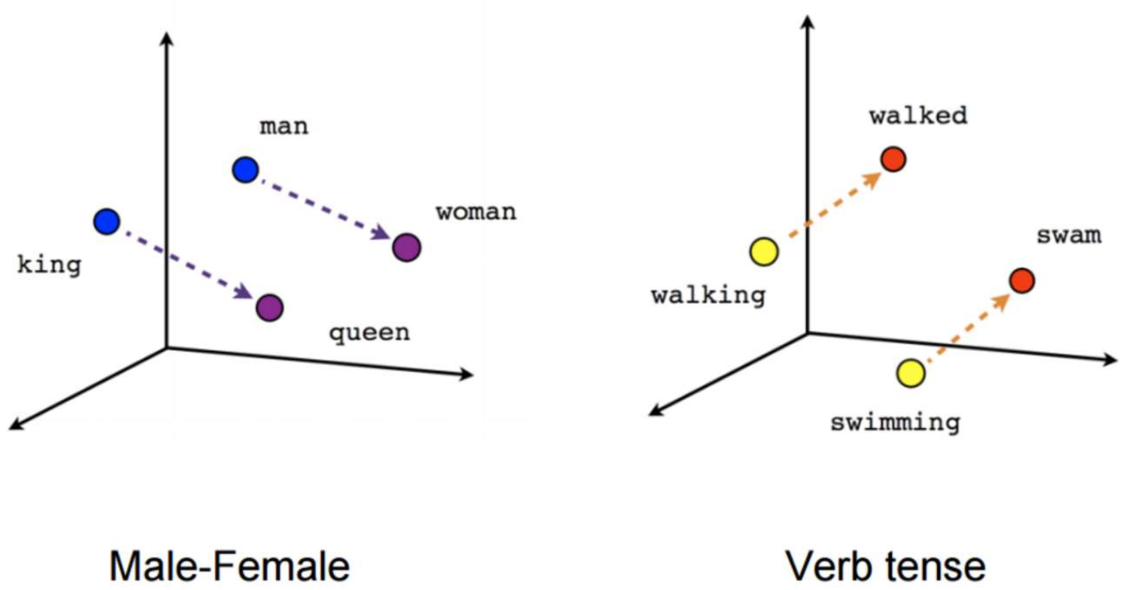


Note: we can do the same if out inputs are characters and one-hot encode them

Source: https://medium.com/@athif.shaffy/one-hot-encoding-of-text-b69124bef0a7

Input representation

- We start with a one-hot vector representation
- If we're dealing with words, after the one-hot vector, we apply an embedding (e.g., Word2Vec)



Source: https://towardsdatascience.com/word-embedding-with-word2vec-and-fasttext-a209c1d3e12c

Character RNN

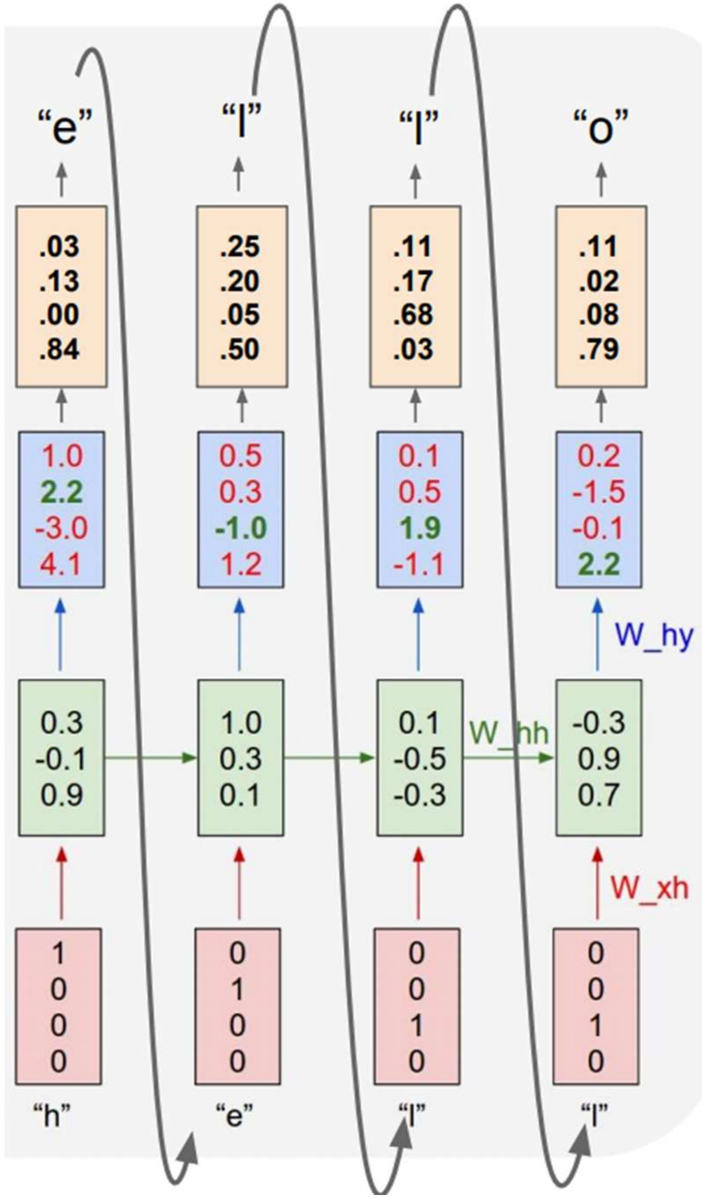
Output symbol y_i

Output layer (linear transformation + softmax)

Hidden state h_i

One-hot encoding x_i

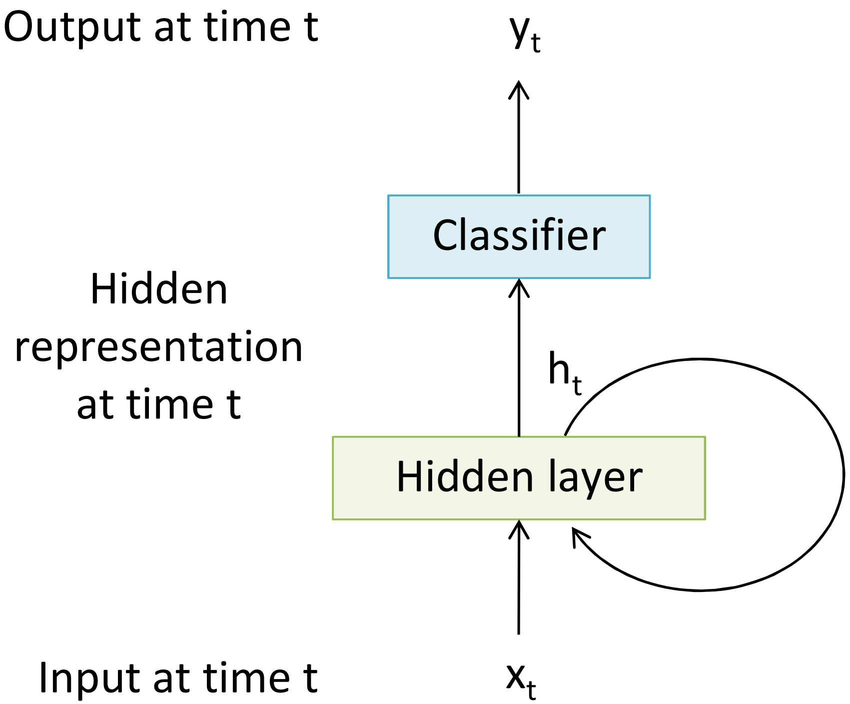
Input symbol



$$p(y_1, y_2, \dots, y_n) = \prod_{i=1}^{n} p(y_i | y_1, \dots, y_{i-1}) \approx \prod_{i=1}^{n} P_W(y_i | h_i)$$

Image source

Recurrent Neural Network (RNN)

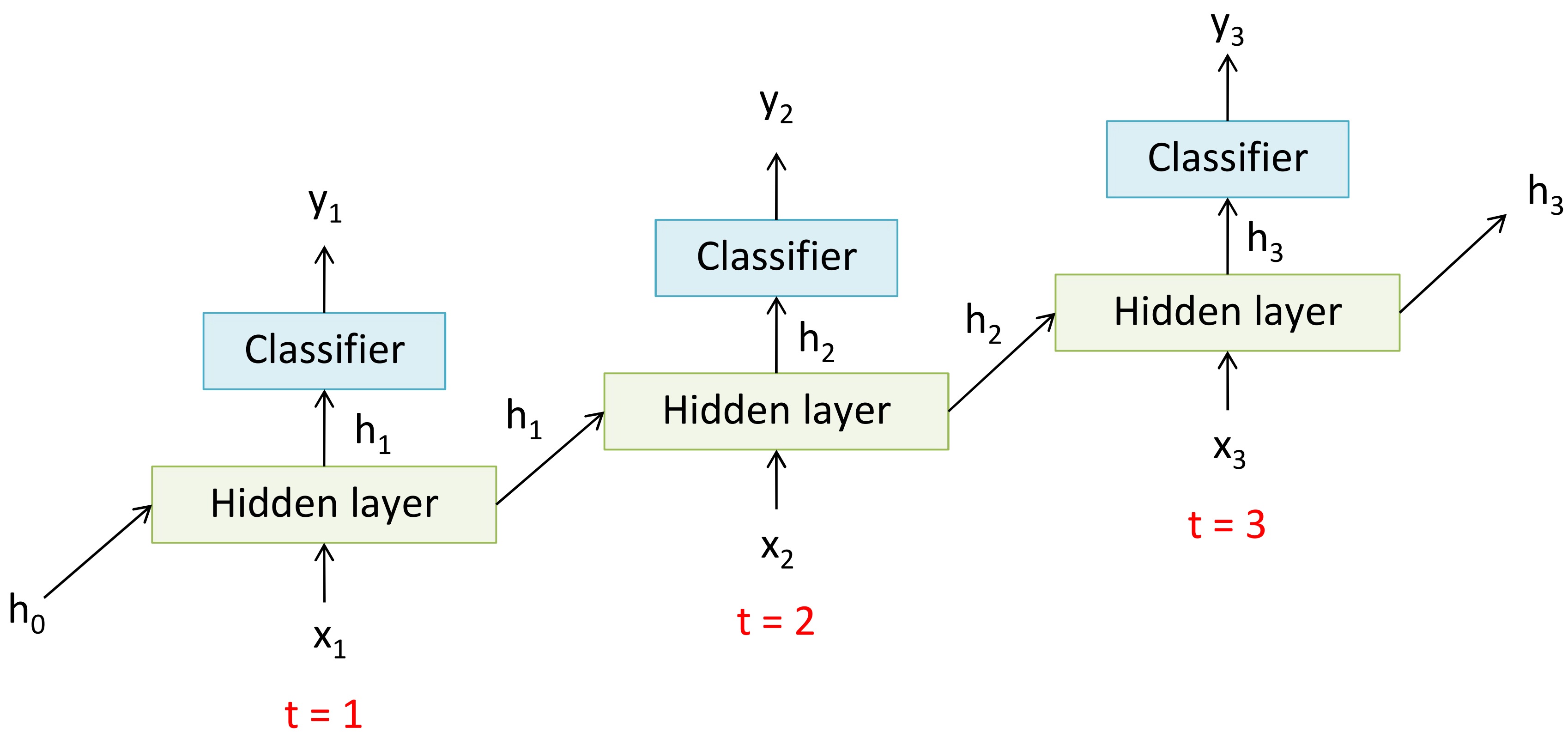


Recurrence:

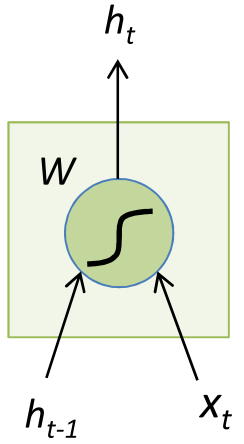
$$h_t = f_W(x_t, h_{t-1})$$

new state | function of W | input at time t | old state

Unrolling the RNN



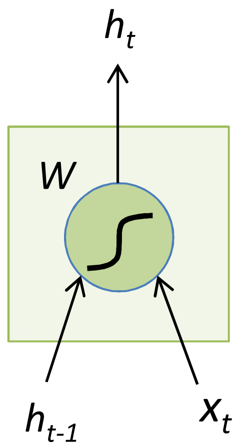
Vanilla RNN Cell



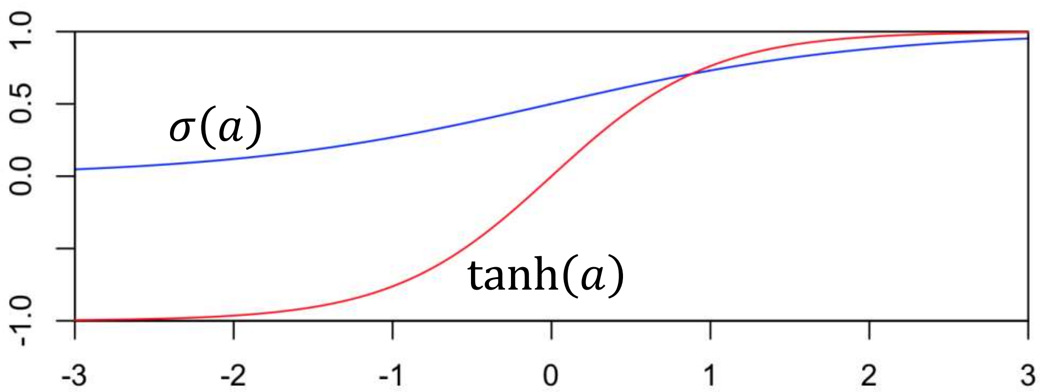
$$h_t = f_W(x_t, h_{t-1})$$
$$= \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

J. Elman, [Finding structure in time](), Cognitive science 14(2), pp. 179–211, 1990

Vanilla RNN Cell



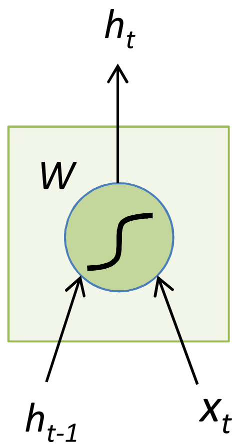
$$h_t = f_W(x_t, h_{t-1})$$
$$= \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$



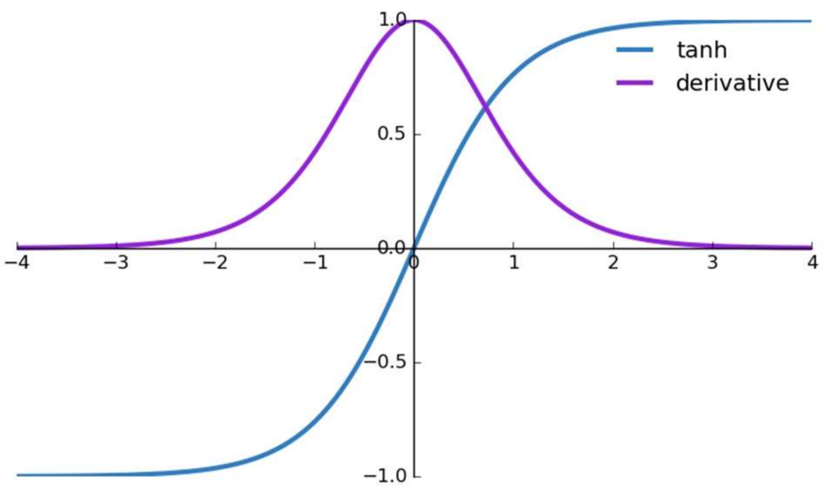
$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$
$$= 2\sigma(2a) - 1$$

Image source

Vanilla RNN Cell



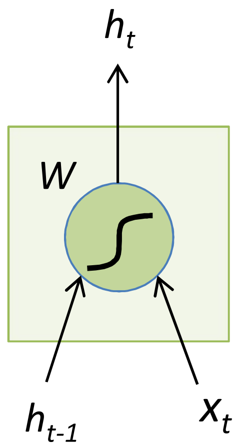
$$h_t = f_W(x_t, h_{t-1})$$
$$= \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$



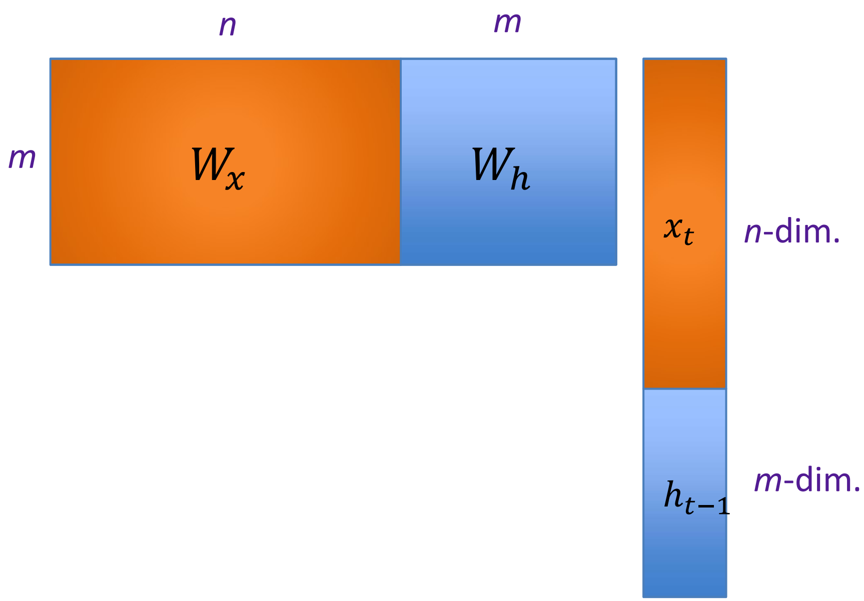
$$\frac{d}{da}\tanh(a) = 1 - \tanh^2(a)$$

Image source

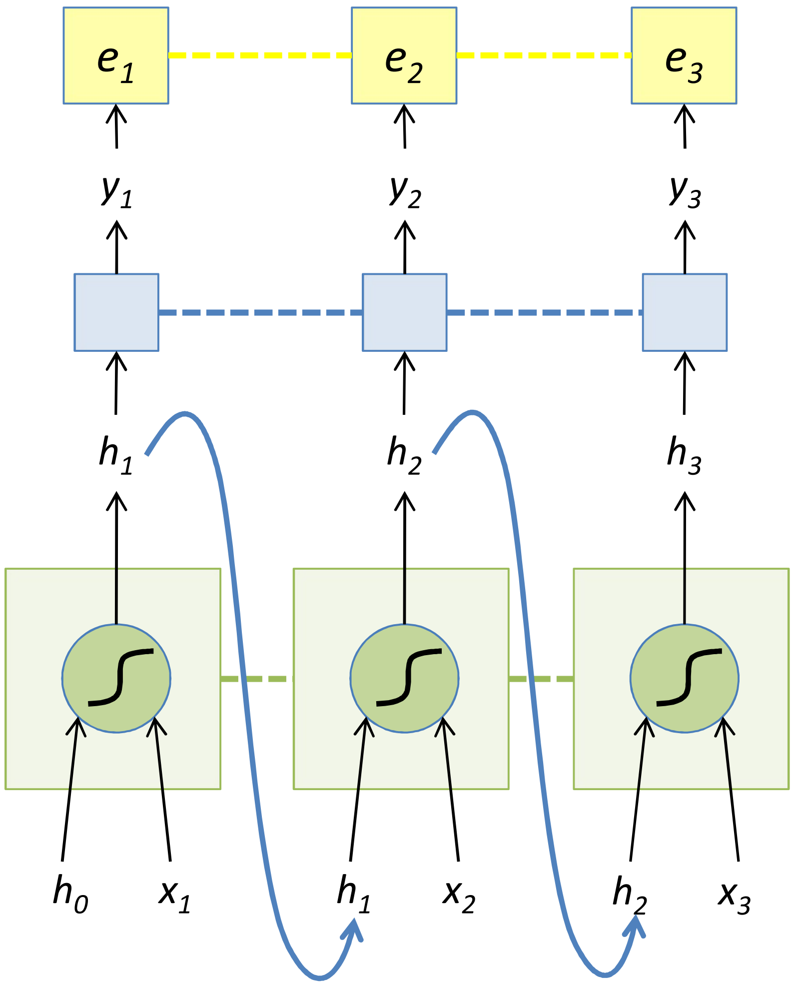
Vanilla RNN Cell



$$
\begin{aligned}
h_t &= f_W(x_t, h_{t-1}) \\
&= \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} \\
&= \tanh(W_x x_t + W_h h_{t-1})
\end{aligned}
$$



RNN Forward Pass



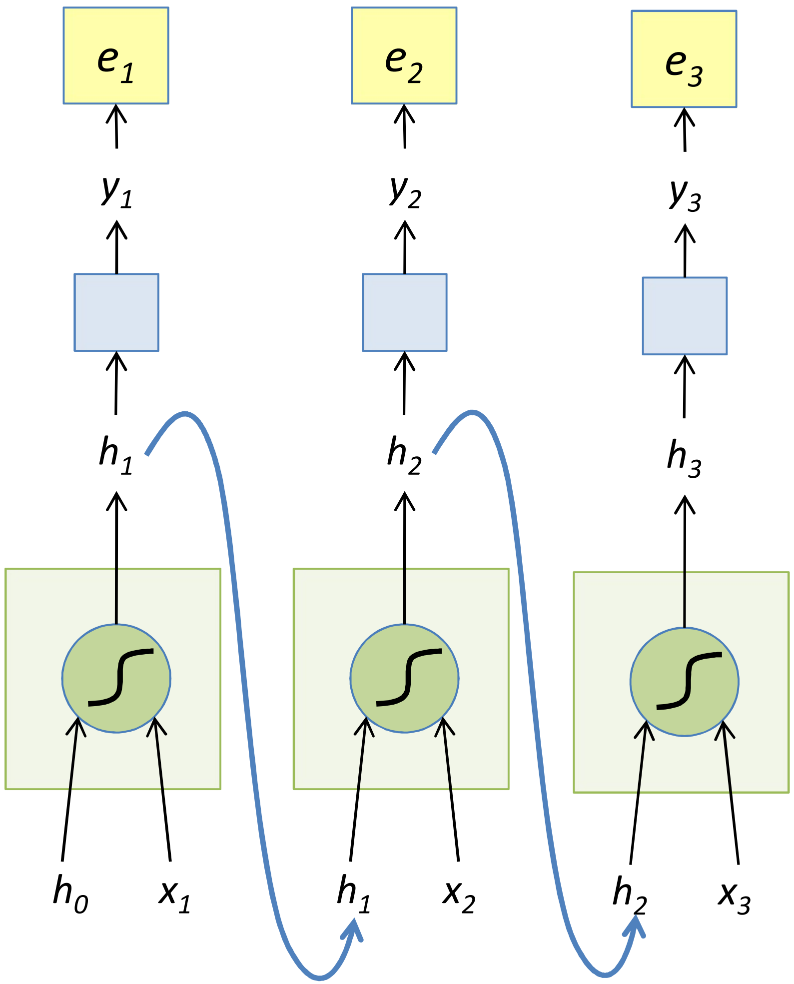
$$e_t = -\log(y_t(GT_t))$$

$$y_t = \mathrm{softmax}(W_y h_t)$$

$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

------ shared weights

Unfolded RNN Forward Pass

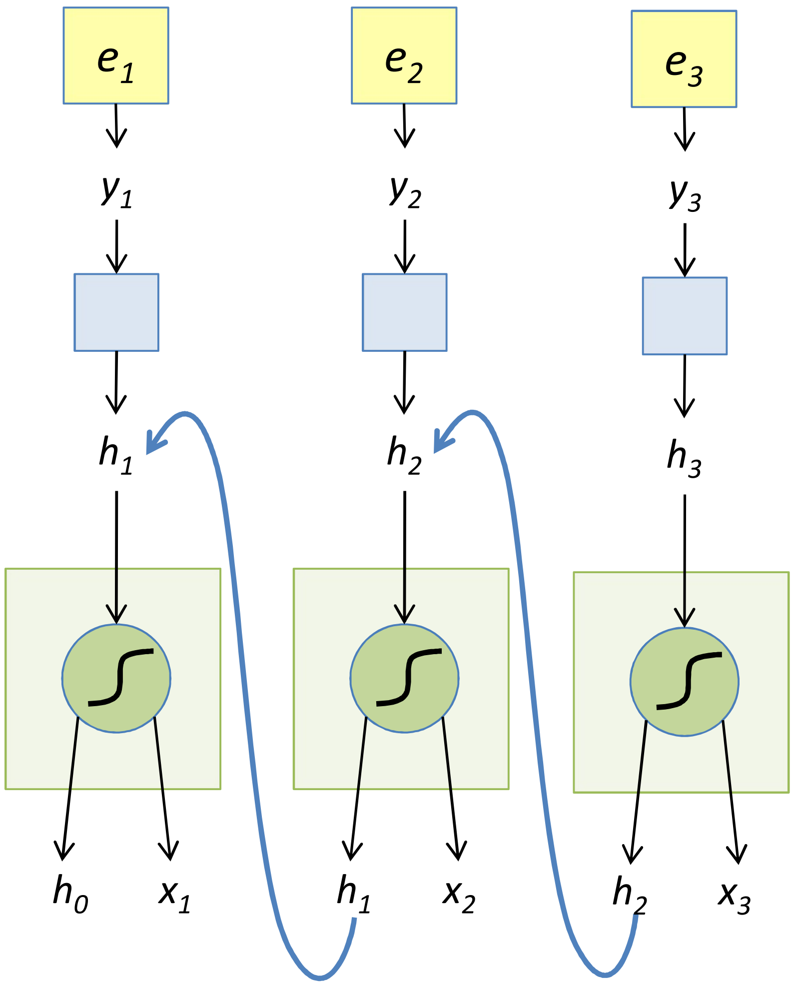


$$e_t = -\log(y_t(GT_t))$$

$$y_t = \mathrm{softmax}(W_y h_t)$$

$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

Unfolded RNN Backward Pass



$$e_t = -\log(y_t(GT_t))$$

$$y_t = \text{softmax}(W_y h_t)$$

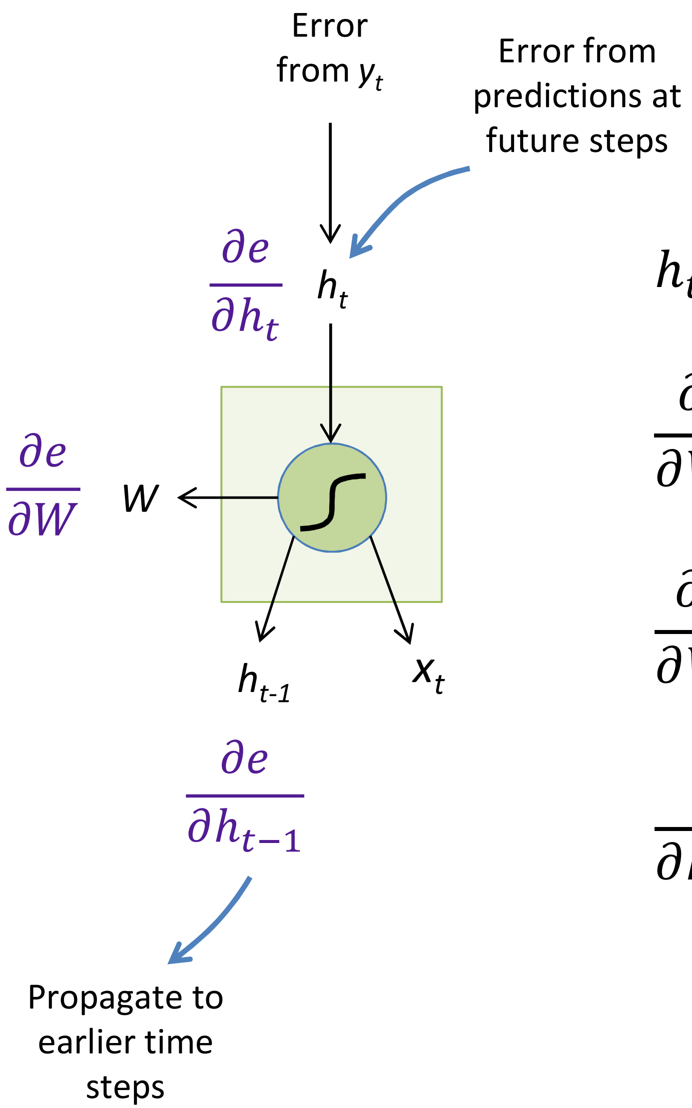
$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

Backpropagation Through Time (BPTT)

- Most common method used to train RNNs
- The unfolded network (used during forward pass) is treated as one big feed-forward network that accepts the whole time series as input
- The weight updates are computed for each copy in the unfolded network, then summed (or averaged) and applied to the RNN weights
- In practice, *truncated* BPTT is used: run the RNN forward k_1 time steps, propagate backward for k_2 time steps

https://machinelearningmastery.com/gentle-introduction-backpropagation-time/
http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf

RNN Backward Pass



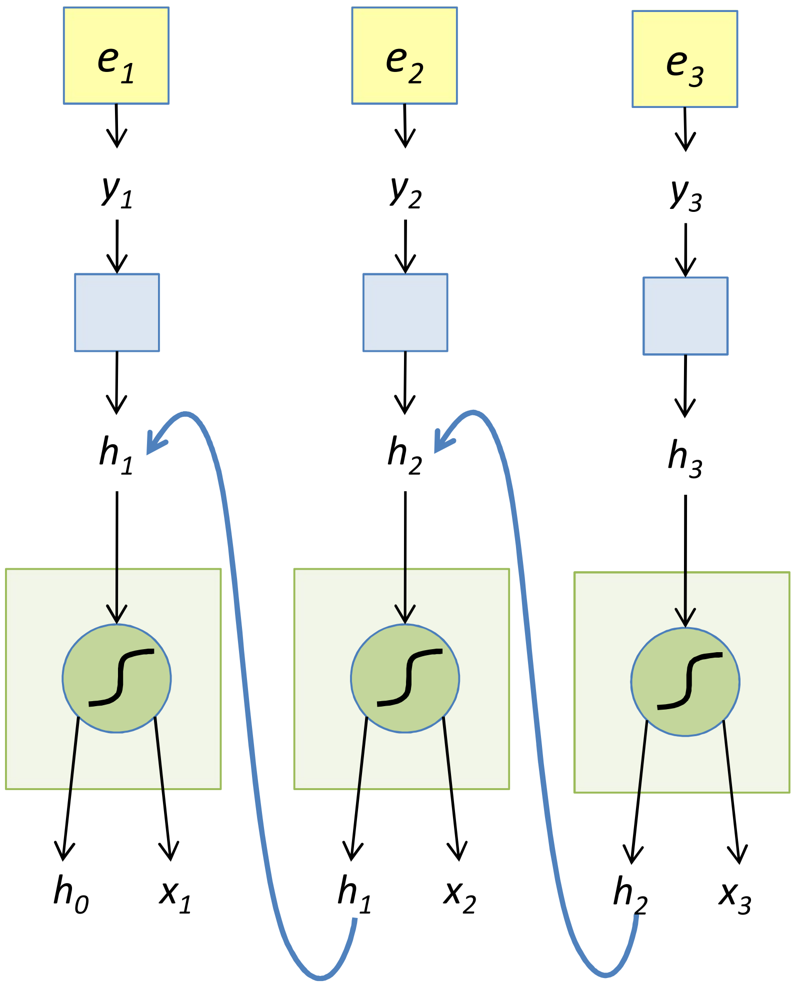
$$h_t = \tanh(W_x x_t + W_h h_{t-1})$$

$$\frac{\partial e}{\partial W_h} = \frac{\partial e}{\partial h_t} \odot \left(1 - \tanh^2(W_x x_t + W_h h_{t-1})\right) h_{t-1}^T$$

$$\frac{\partial e}{\partial W_x} = \frac{\partial e}{\partial h_t} \odot \left(1 - \tanh^2(W_x x_t + W_h h_{t-1})\right) x_t^T$$

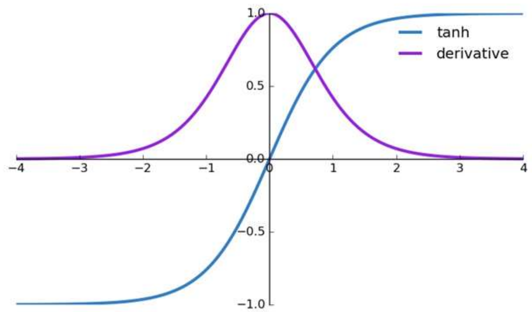
$$\frac{\partial e}{\partial h_{t-1}} = W_h^T \left(1 - \tanh^2(W_x x_t + W_h h_{t-1})\right) \odot \frac{\partial e}{\partial h_t}$$

RNN Backward Pass



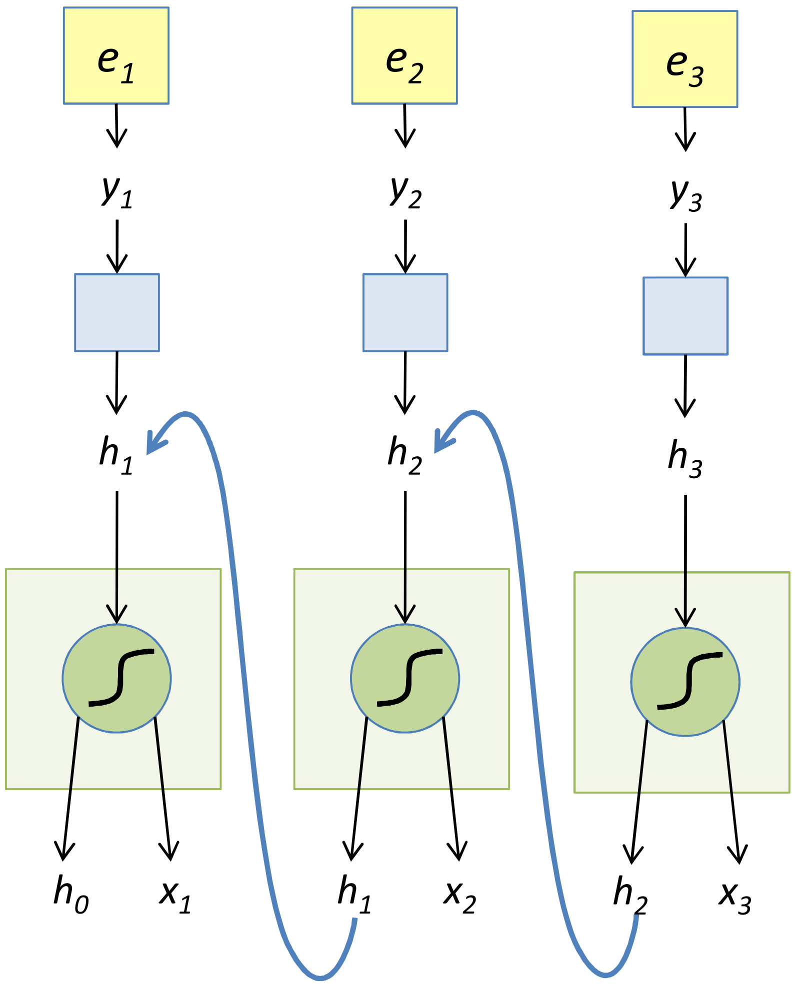
$$\frac{\partial e}{\partial h_{t-1}} = W_h^T\left(1 - \tanh^2(W_x x_t + W_h h_{t-1})\right) \odot \frac{\partial e}{\partial h_t}$$

Large tanh activations will give small gradients



Consider $\frac{\partial e_n}{\partial h_k}$ for $k \ll n$

RNN Backward Pass



Consider $\frac{\partial e_n}{\partial h_k}$ for $k \ll n$

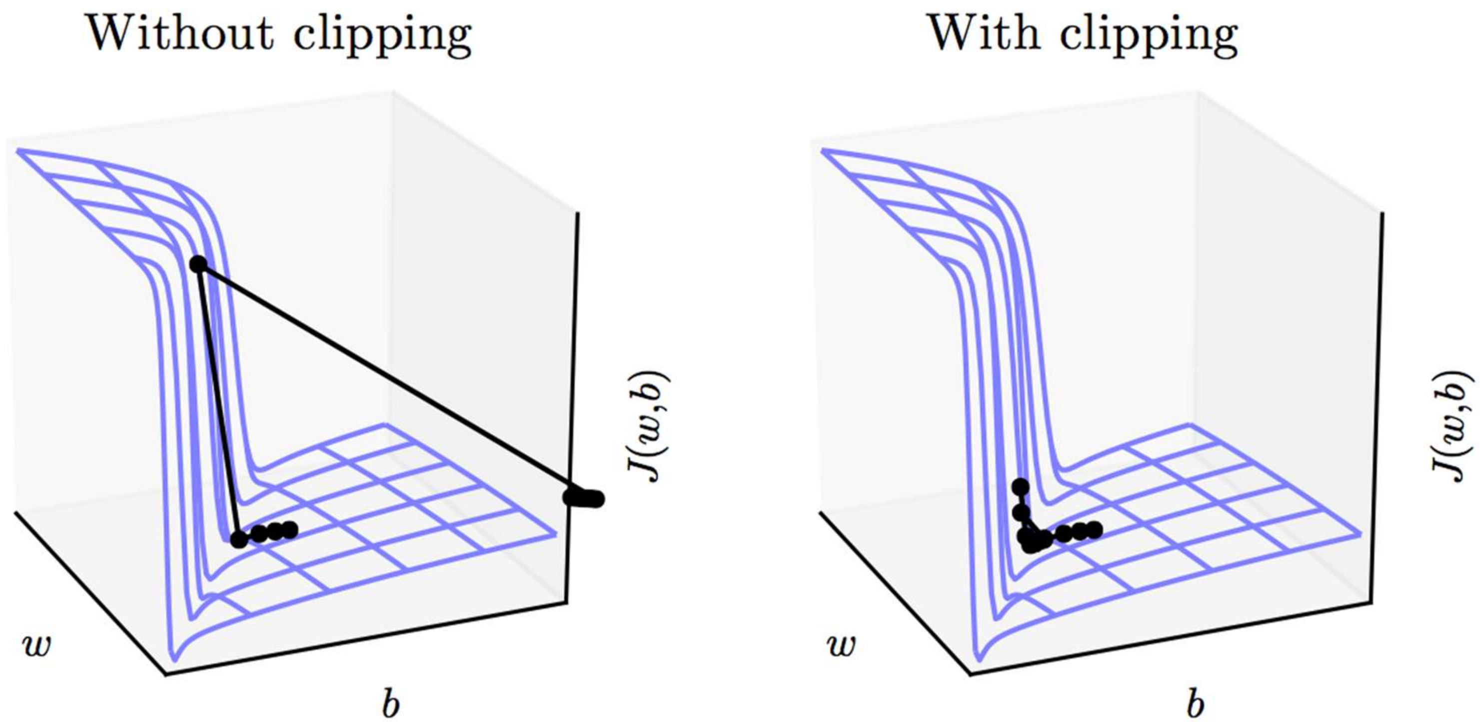
$$\frac{\partial e}{\partial h_{t-1}} = W_h^T \big(1 - \tanh^2(W_x x_t + W_h h_{t-1})\big) \odot \frac{\partial e}{\partial h_t}$$

Gradients will vanish if largest singular value of W_h is less than 1

Backpropagation through time

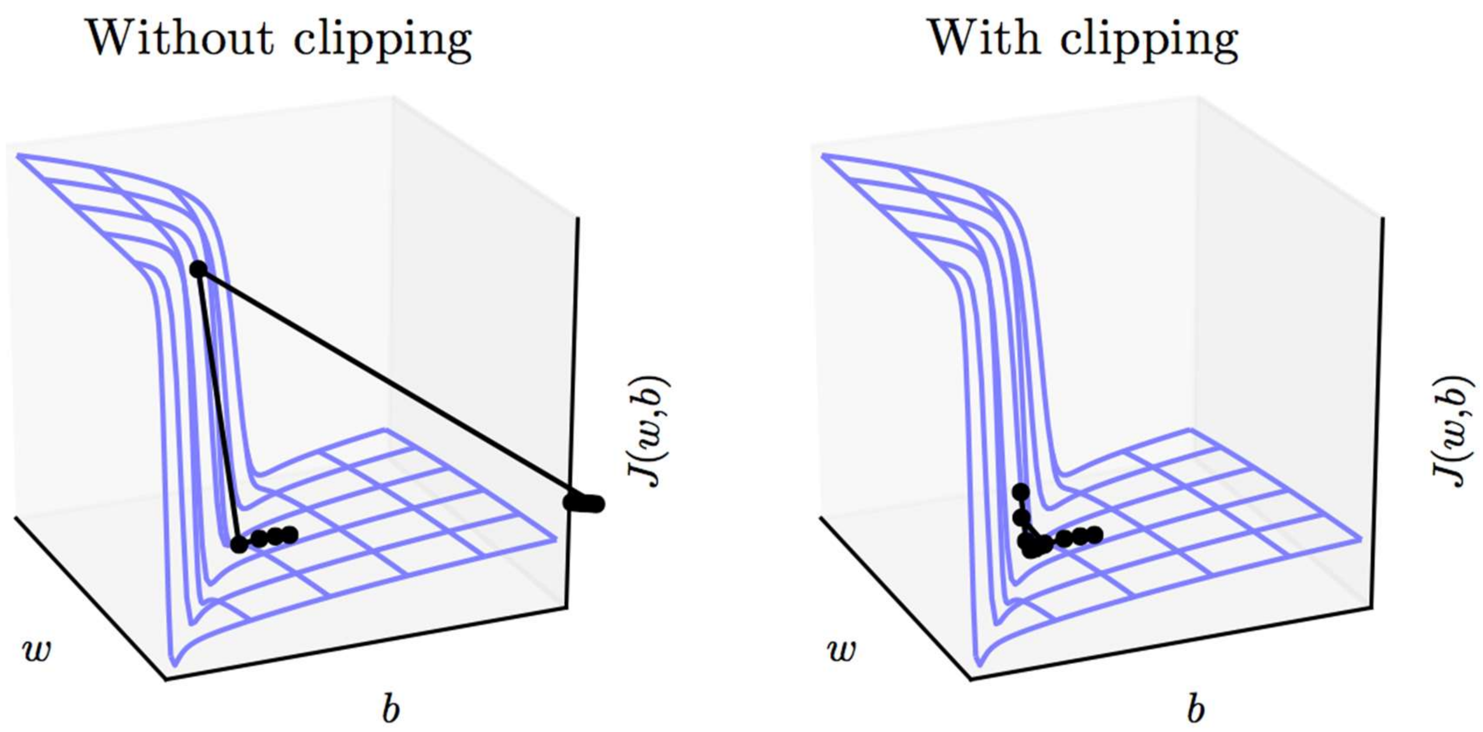
- Backpropagation is done similarity to MLP/CNN
- However the layers now correspond to different steps/times, so we call it **backpropagation through time** (section 10.2.2. in the book)
- As with very deep networks we need to multiply together many different gradients
- For long sequences, this is a problem
  - Vanishing/exploding gradients

Gradient clipping



Exploding gradients are easily solved by rescaling or clipping the gradient

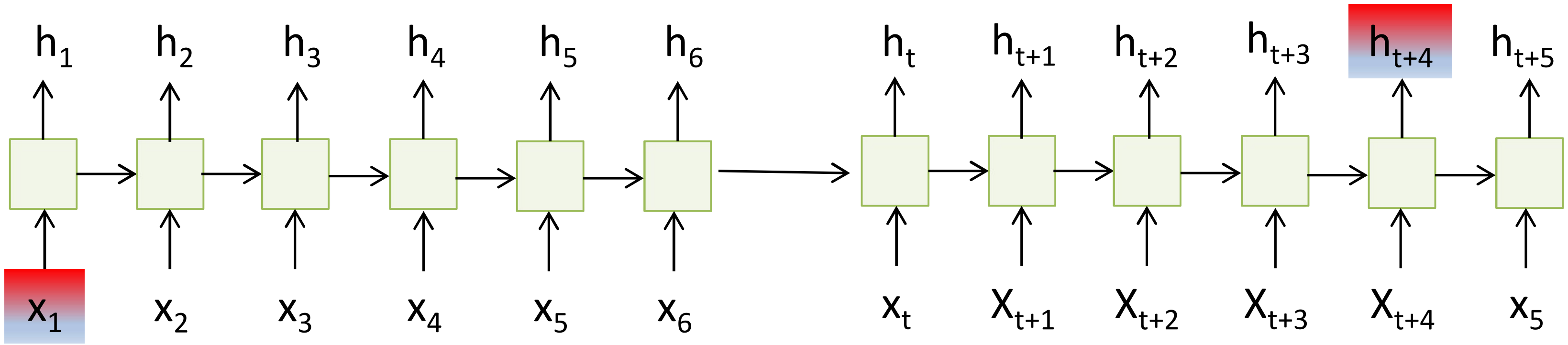
Gradient clipping



But what about vanishing gradients?
There is no gradient to begin with, what should we clip/rescale?
Actually, there’s even another problem…

Long-term dependencies

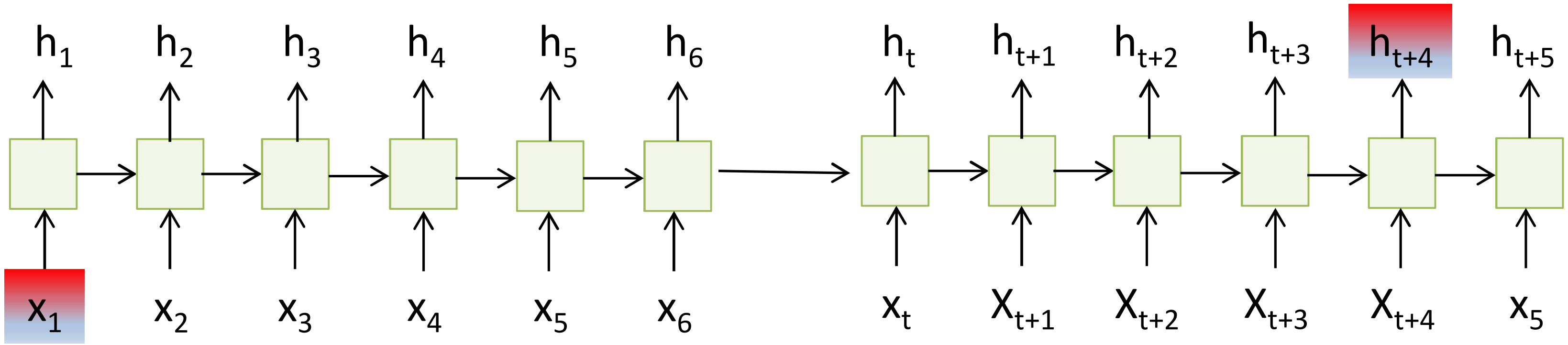
Simplified version of RNN (omitting y)



In theory RNNs should be able to capture long-term dependencies
but in practice they're not!

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-term dependencies

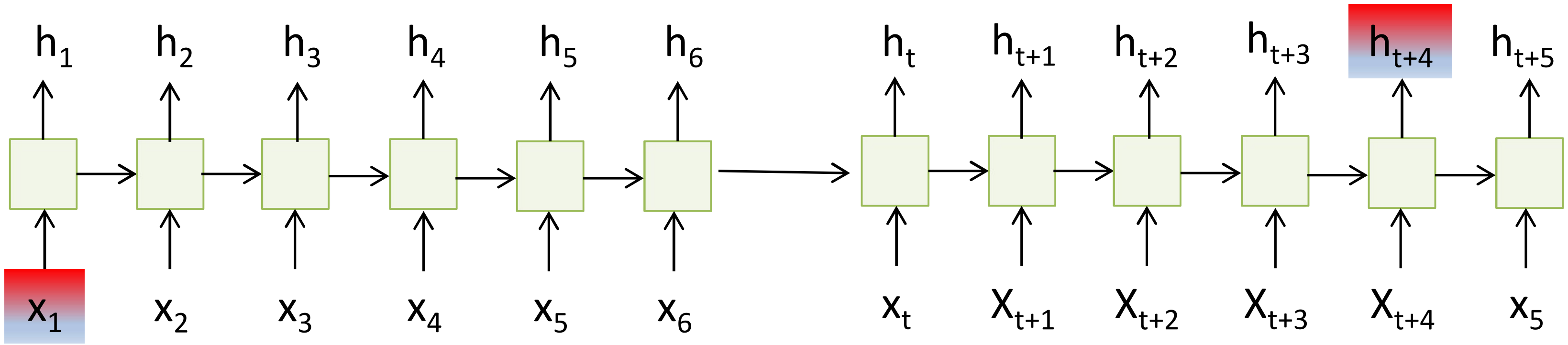


$$\boldsymbol{h}^{(t)} = \boldsymbol{W}^{\top}\boldsymbol{h}^{(t-1)}$$

Considering a very simply RNN described by this recurrence relation

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-term dependencies

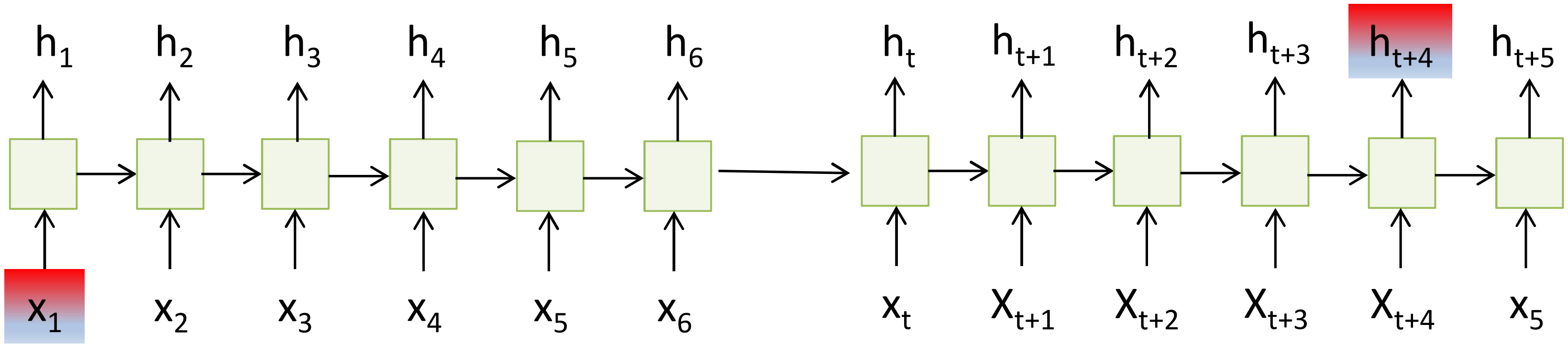


$$\boldsymbol{h}^{(t)} = \left(\boldsymbol{W}^{t}\right)^{\top} \boldsymbol{h}^{(0)}$$

This can be simplified as shown above

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-term dependencies



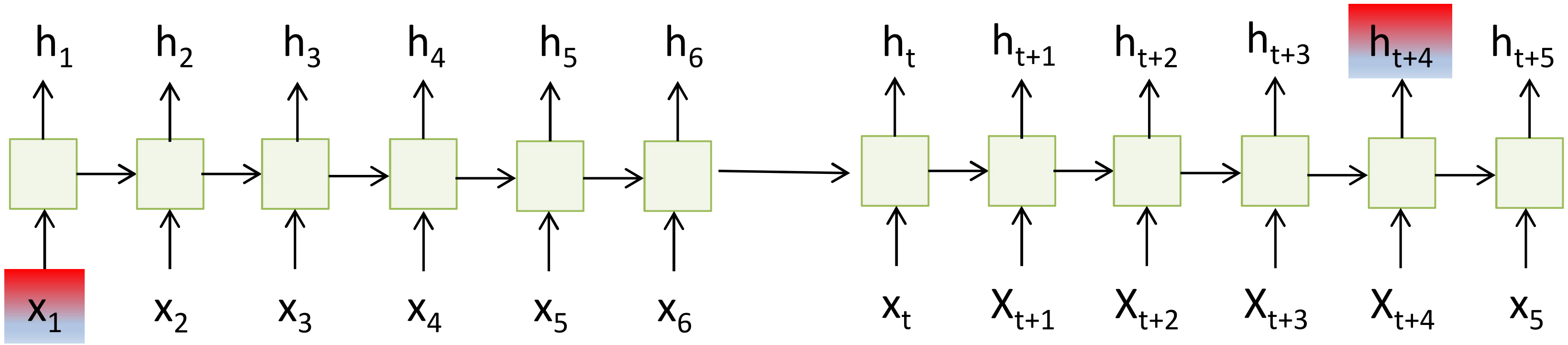
$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^{\top} \boldsymbol{\Lambda}^{t} \boldsymbol{Q} \boldsymbol{h}^{(0)}$$

$$\boldsymbol{W} = \boldsymbol{Q} \boldsymbol{\Lambda} \boldsymbol{Q}^{\top}$$

Given the matrix decomposition of W, we get the formula above

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-term dependencies



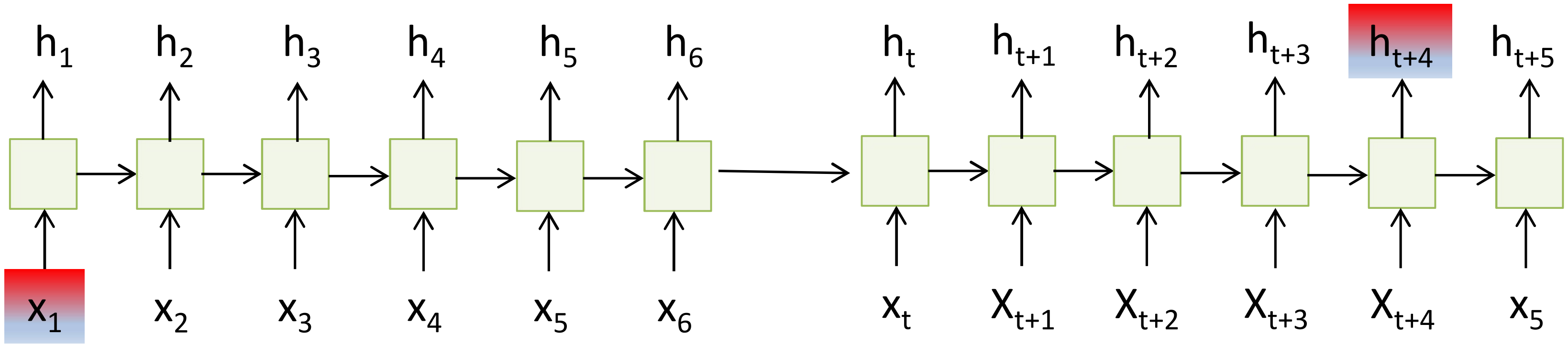
$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^\top \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}^{(0)}$$

$$\boldsymbol{W} = \boldsymbol{Q} \boldsymbol{\Lambda} \boldsymbol{Q}^\top$$

eigenvalue with magnitude < 1 vanish, those with magnitude > 1 explode

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-term dependencies



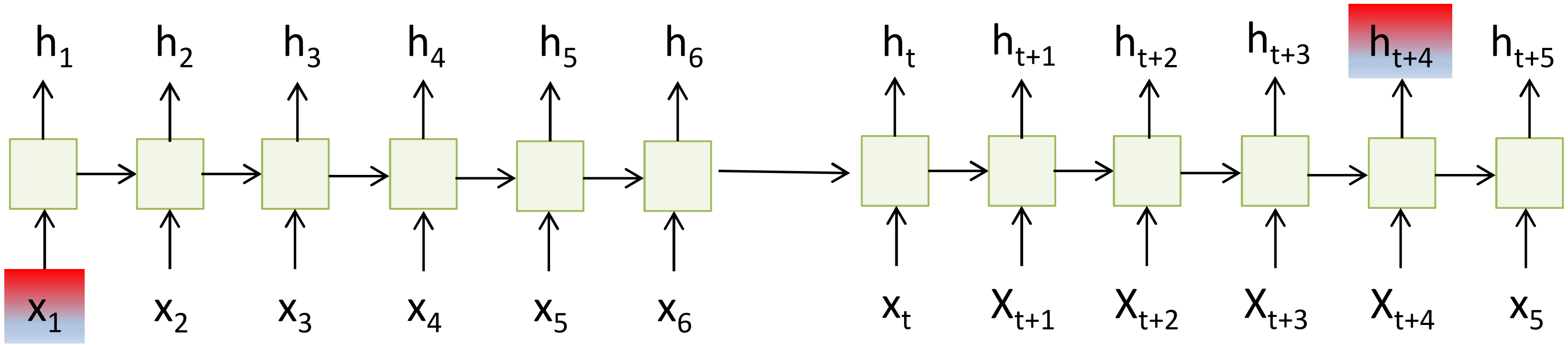
$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^\top \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}^{(0)}$$

$$\boldsymbol{W} = \boldsymbol{Q} \boldsymbol{\Lambda} \boldsymbol{Q}^\top$$

Any component of h(0) not aligned with max eigenvector of W is eventually discarded

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-term dependencies



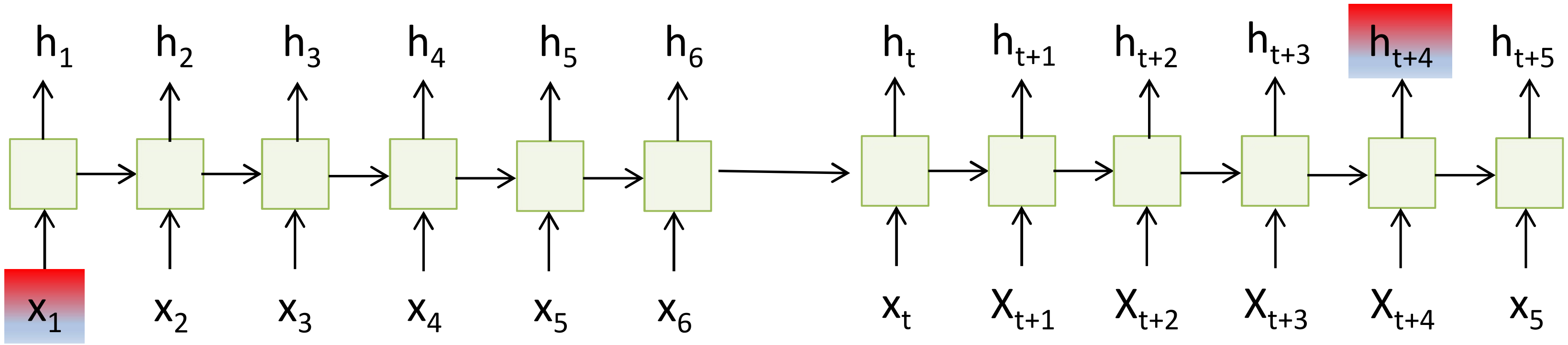
$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^\top \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}^{(0)}$$

$$\boldsymbol{W} = \boldsymbol{Q} \boldsymbol{\Lambda} \boldsymbol{Q}^\top$$

Note that this problem is particular of RNNs. In MLPs W is not shared by all layers.

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-term dependencies

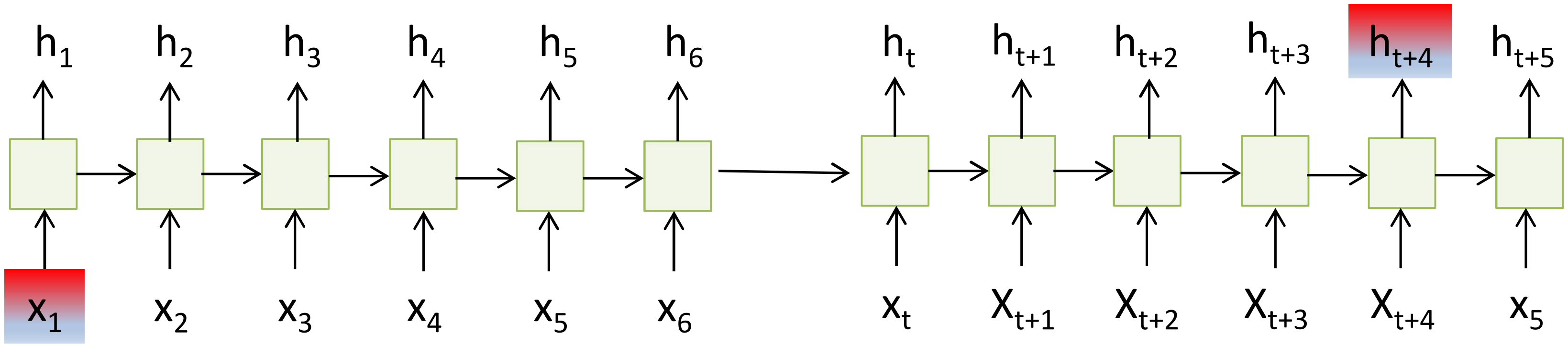


$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^\top \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}^{(0)}$$

Unfortunately it turns out that we can't avoid regions of the parameters space with exploding/vanishing gradients. It's there we need to go to learn long-term dependencies…

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-term dependencies

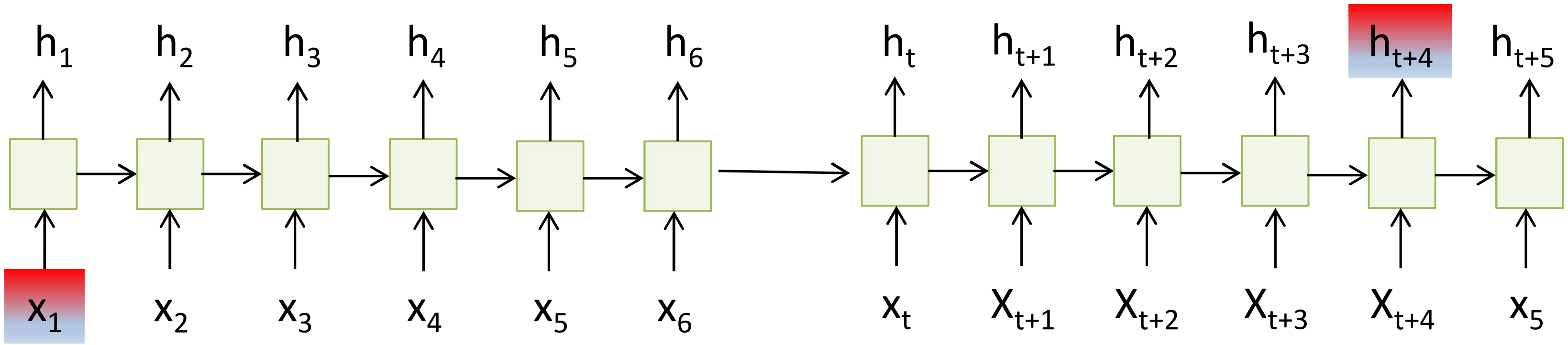


$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^{\top} \boldsymbol{\Lambda}^{t} \boldsymbol{Q} \boldsymbol{h}^{(0)}$$

There the training doesn't stop but becomes unsustainably slow.
In practice, we can't train RNNs for sequences longer than ~10

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-term dependencies



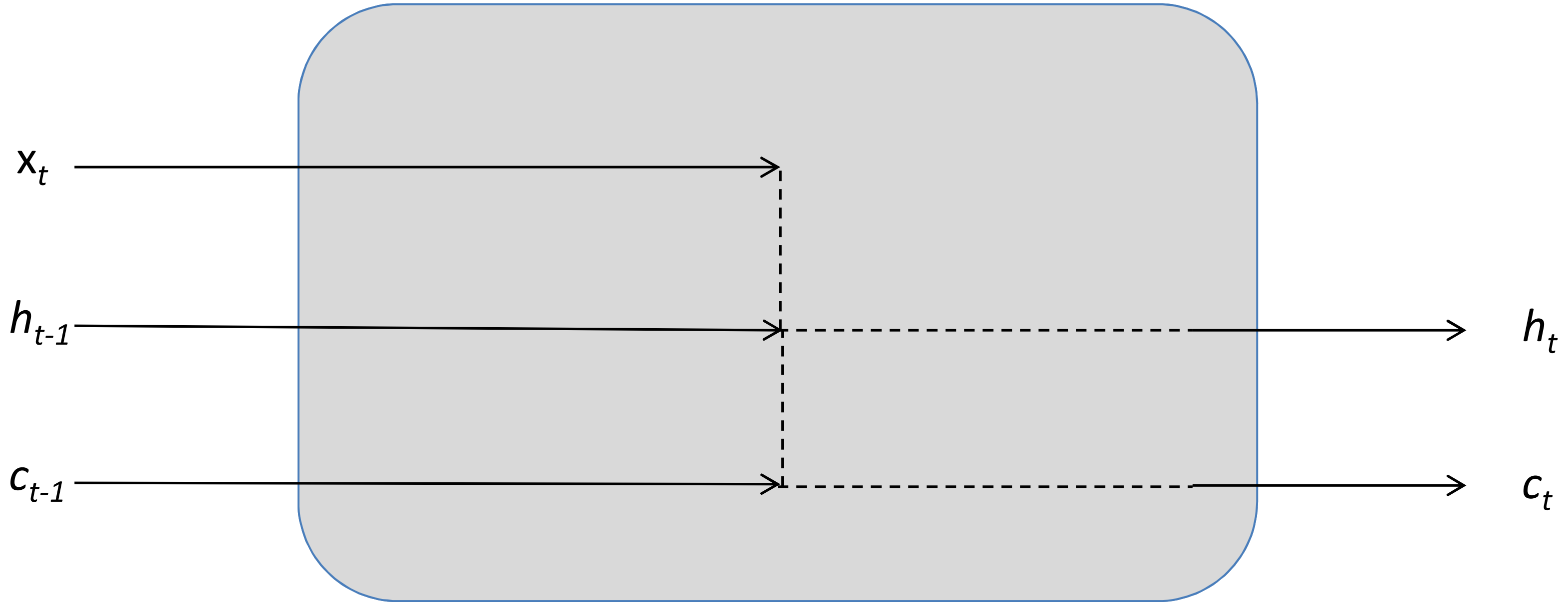
$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^{\top} \boldsymbol{\Lambda}^{t} \boldsymbol{Q} \boldsymbol{h}^{(0)}$$

But there may be another way…

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

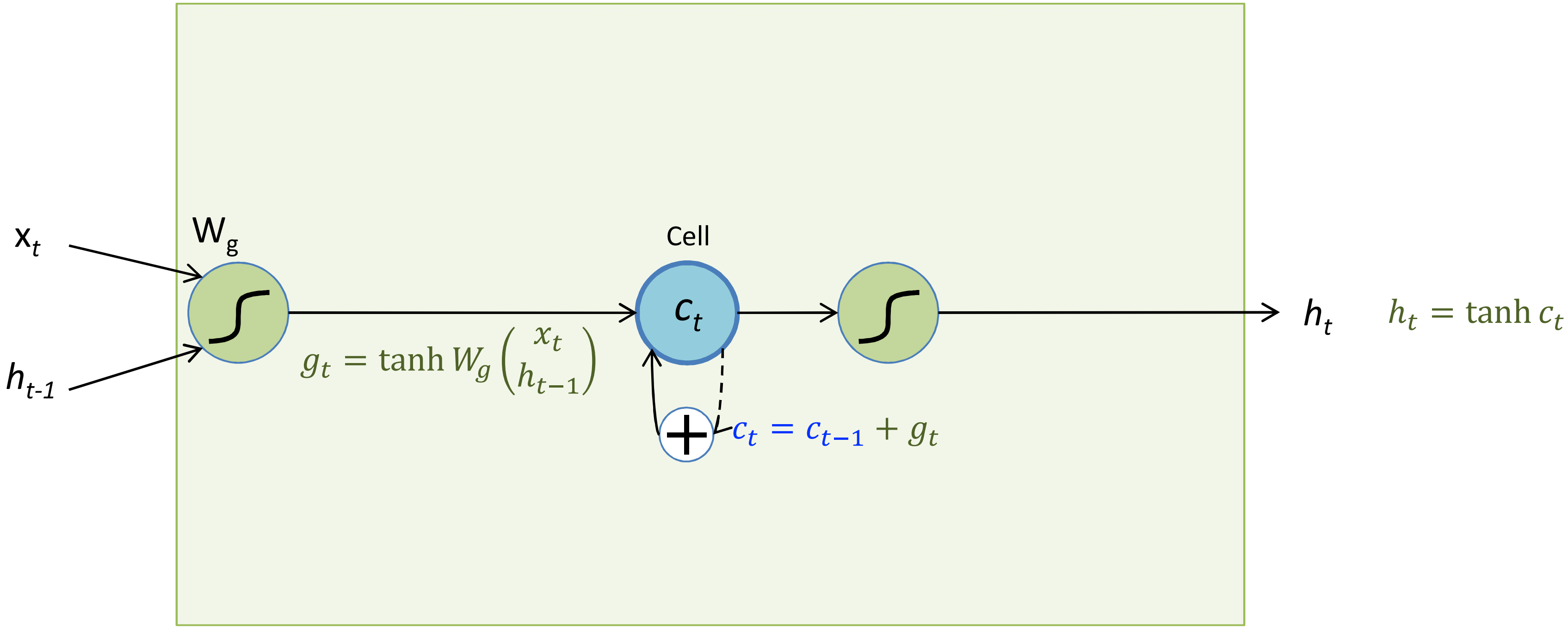
Long Short-Term Memory (LSTM)

- Add a *memory cell* that is not subject to matrix multiplication or squishing, thereby avoiding gradient decay



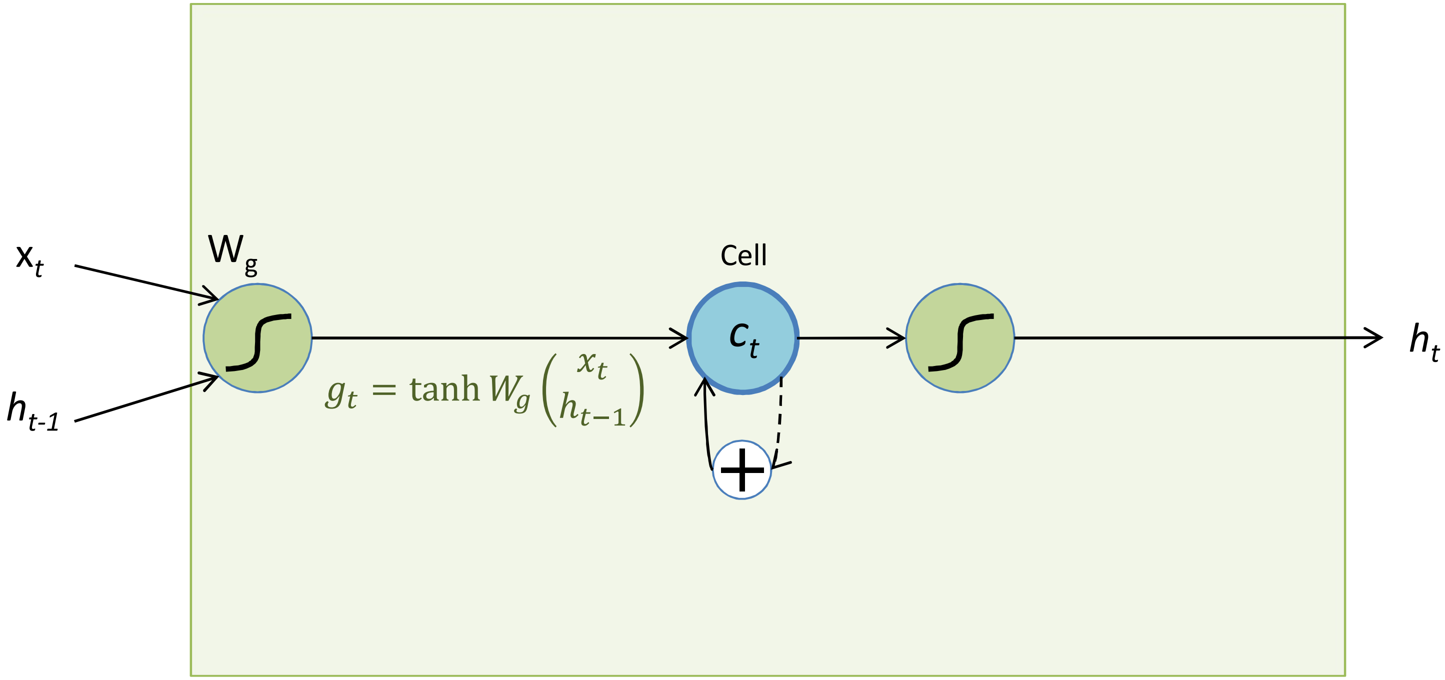
S. Hochreiter and J. Schmidhuber, [Long short-term memory](), Neural Computation 9 (8), pp. 1735–1780, 1997

The LSTM Cell

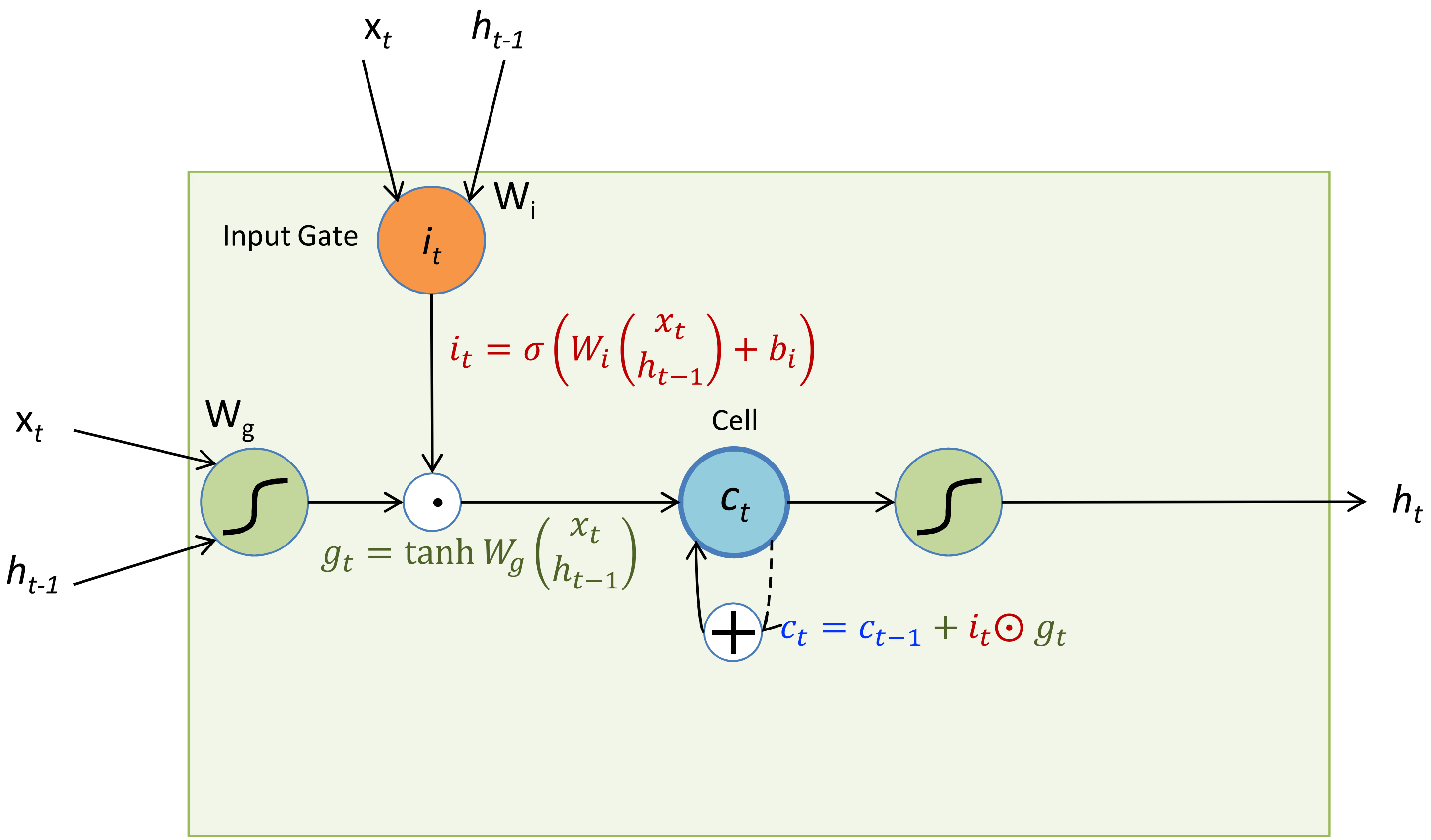


* Dashed line indicates time-lag

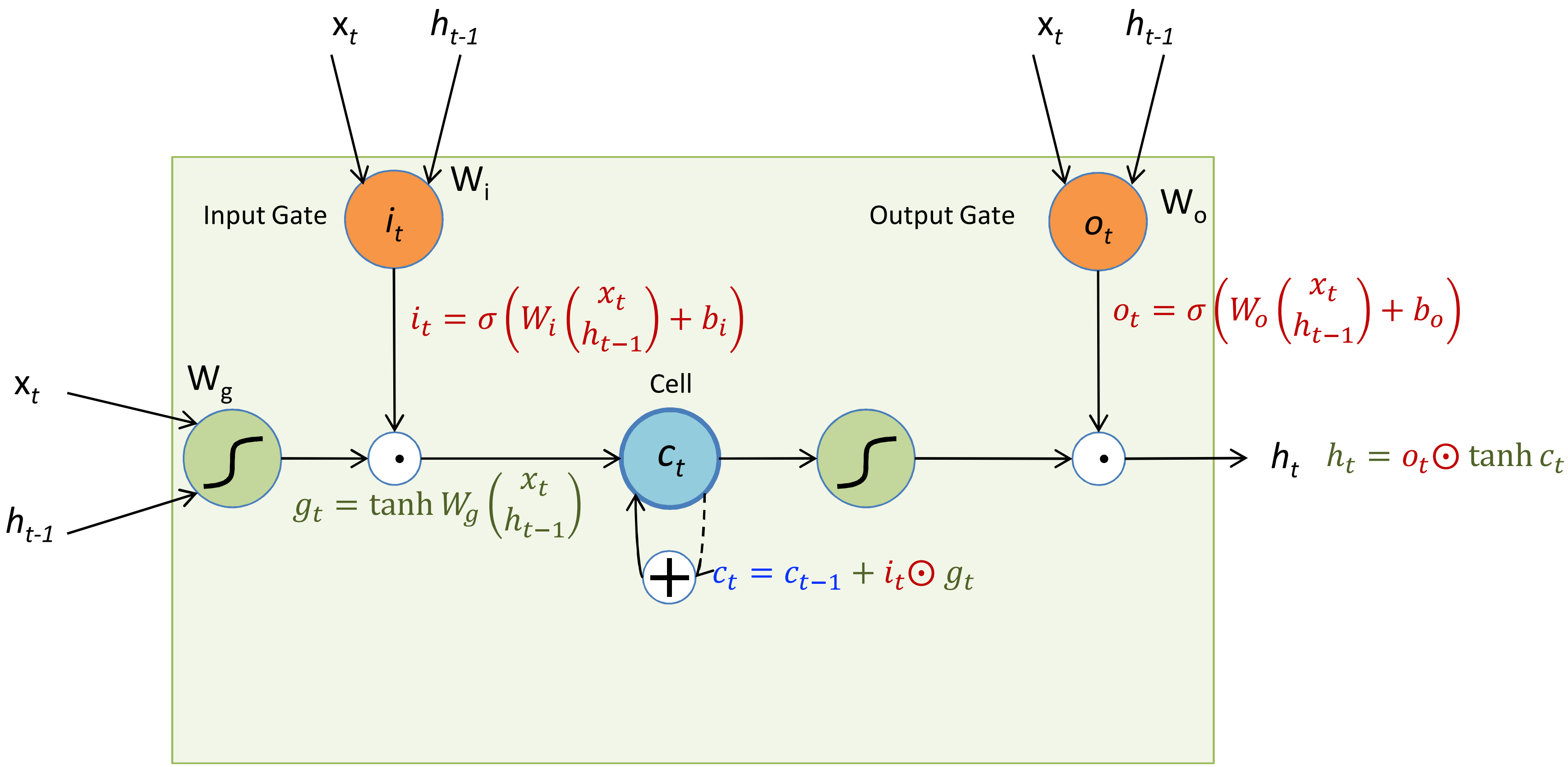
The LSTM Cell



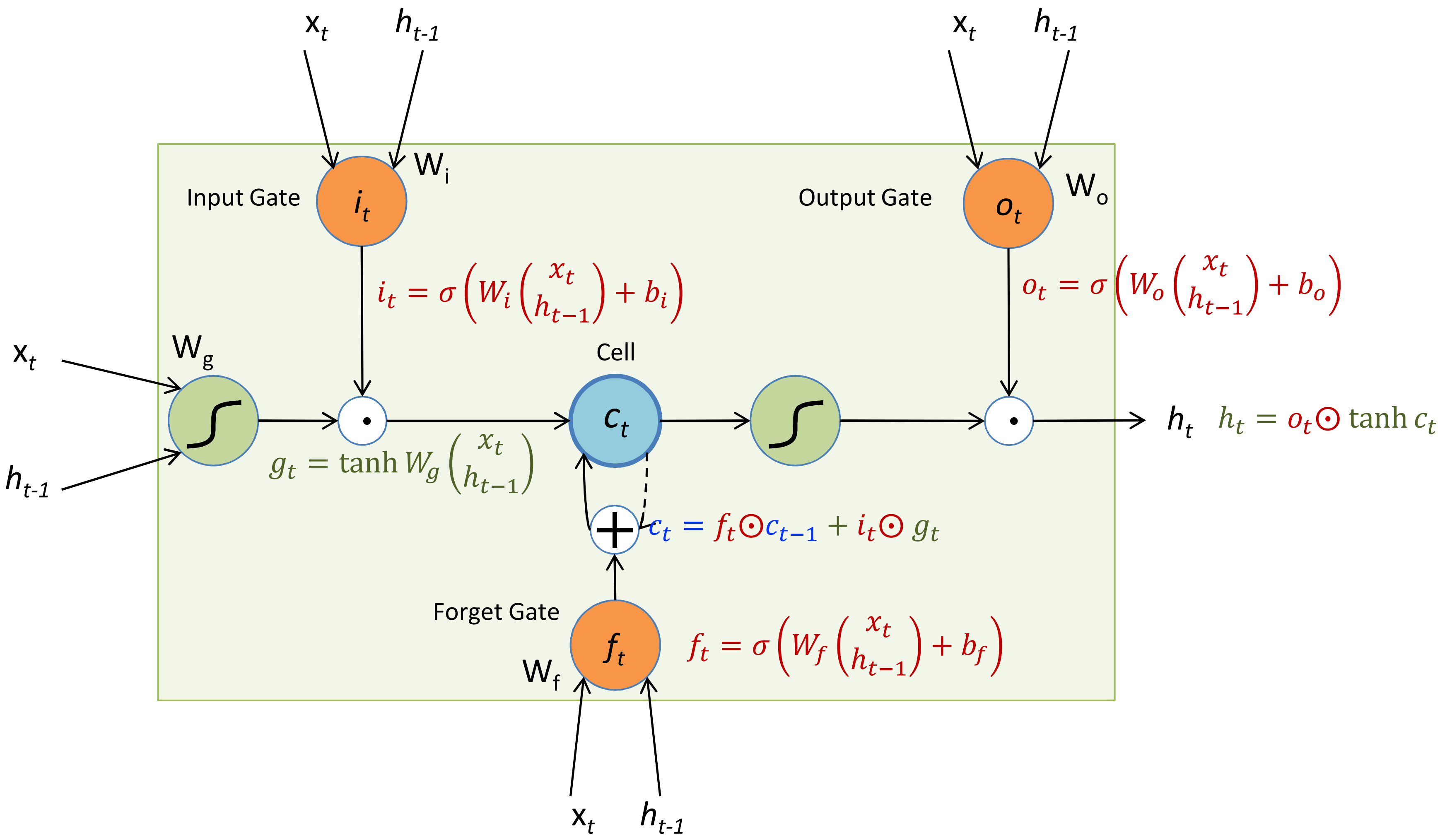
The LSTM Cell



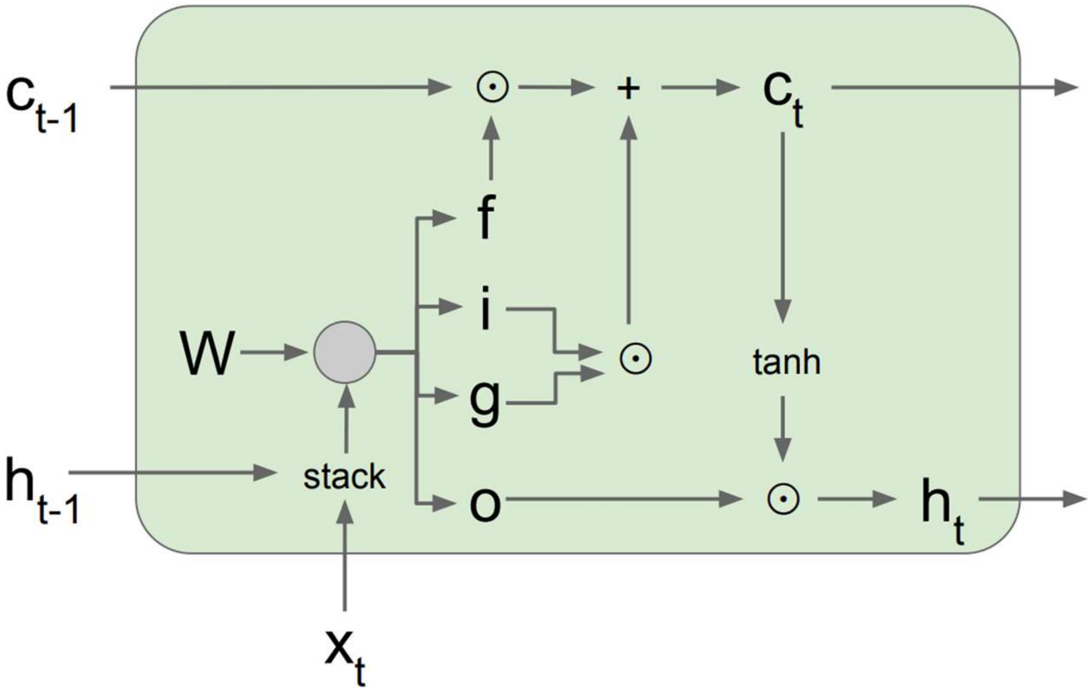
The LSTM Cell



The LSTM Cell



LSTM Forward Pass Summary



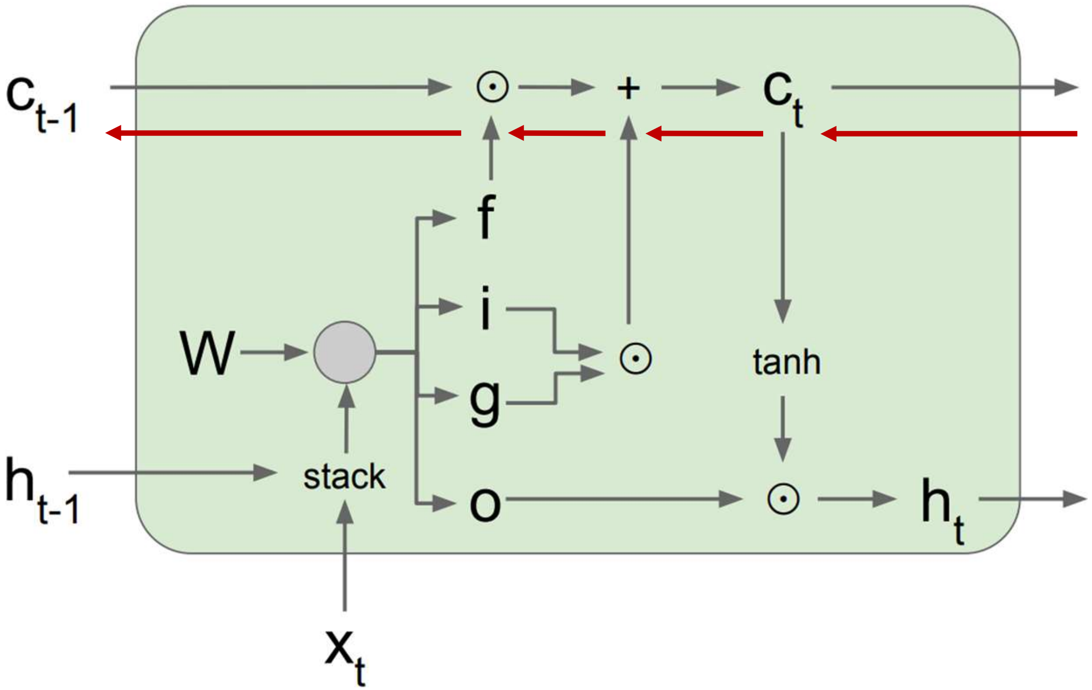
$$\begin{pmatrix} g_t \\ i_t \\ f_t \\ o_t \end{pmatrix} = \begin{pmatrix} \tanh \\ \sigma \\ \sigma \\ \sigma \end{pmatrix} \begin{pmatrix} W_g \\ W_i \\ W_f \\ W_o \end{pmatrix} \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot \tanh c_t$$

Figure source

LSTM Backward Pass



Gradient flow from c_t to c_{t-1} only involves back-propagating through addition and elementwise multiplication, not matrix multiplication or tanh

For complete details: [Illustrated LSTM Forward and Backward Pass]

[Figure source]

- Next Week:
  - Continue: RNN – Gated Recurrent Unit
  - different applications and architecture of RNN
  - Manifold learning and Autoencoders