MIDDLE EAST TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

# CENG 403
# Introduction to Deep Learning

## Week 12b

Sinan Kalkan

# ResNext

Previously on CENG403

**Aggregated Residual Transformations for Deep Neural Networks**

Saining Xie[1] Ross Girshick[2] Piotr Dollár[2] Zhuowen Tu[1] Kaiming He[2]
[1]UC San Diego [2]Facebook AI Research
{s9xie,ztu}@ucsd.edu {rbg,pdollar,kaiminghe}@fb.com

2017

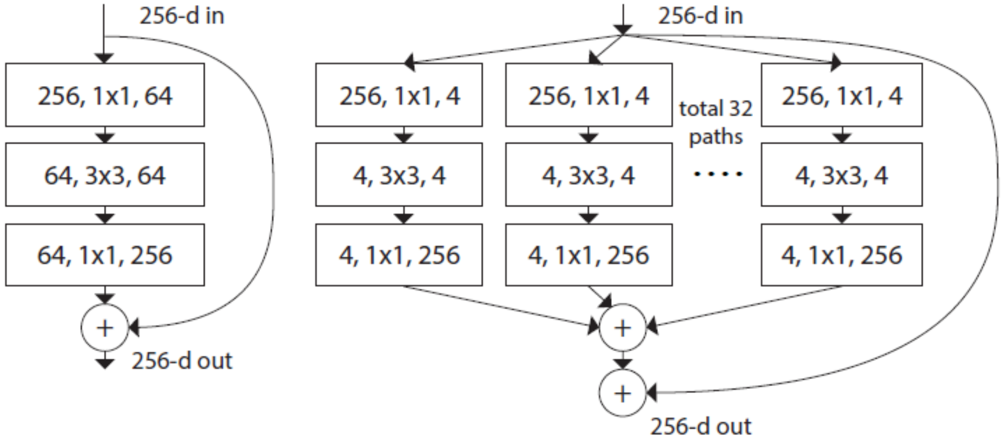
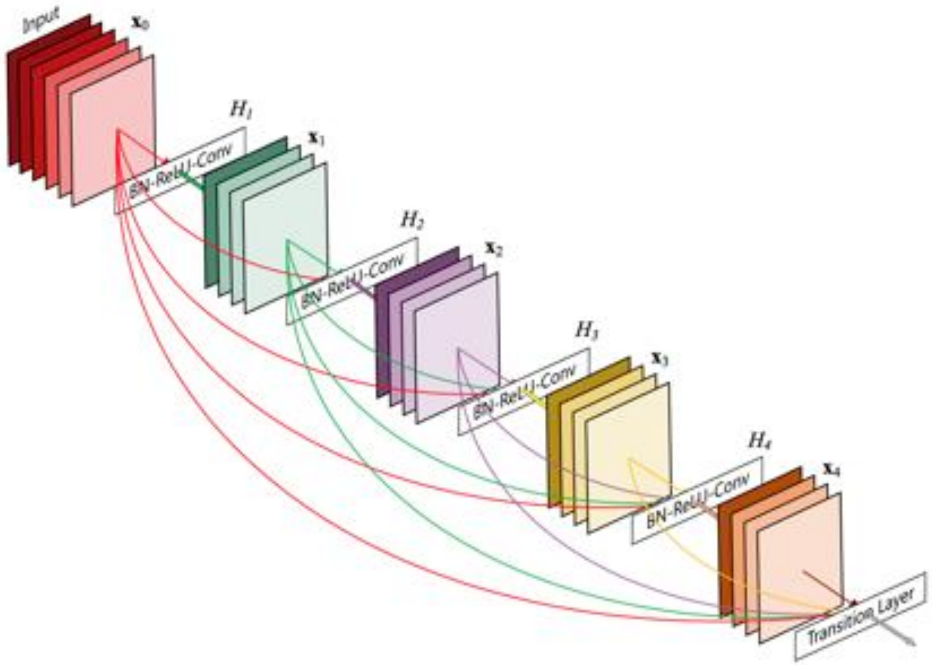



Figure 1. **Left**: A block of ResNet [14]. **Right**: A block of ResNeXt with cardinality = 32, with roughly the same complexity. A layer is shown as (# in channels, filter size, # out channels).

| | setting | top-1 err (%) | top-5 err (%) |
|---|---|---|---|
| *1× complexity references:* | | | |
| ResNet-101 | 1 × 64d | 22.0 | 6.0 |
| ResNeXt-101 | 32 × 4d | 21.2 | 5.6 |
| *2× complexity models follow:* | | | |
| ResNet-200 [15] | 1 × 64d | 21.7 | 5.8 |
| ResNet-101, wider | 1 × **100**d | 21.3 | 5.7 |
| ResNeXt-101 | **2** × 64d | 20.7 | 5.5 |
| ResNeXt-101 | **64** × 4d | **20.4** | **5.3** |

# DenseNet

Previously on CENG403




**Densely Connected Convolutional Networks**


Gao Huang* Cornell University gh349@cornell.edu

Zhuang Liu* Tsinghua University liuzhuang13@mails.tsinghua.edu.cn

Laurens van der Maaten Facebook AI Research lvdmaaten@fb.com

Kilian Q. Weinberger Cornell University kqw4@cornell.edu


2016;2018

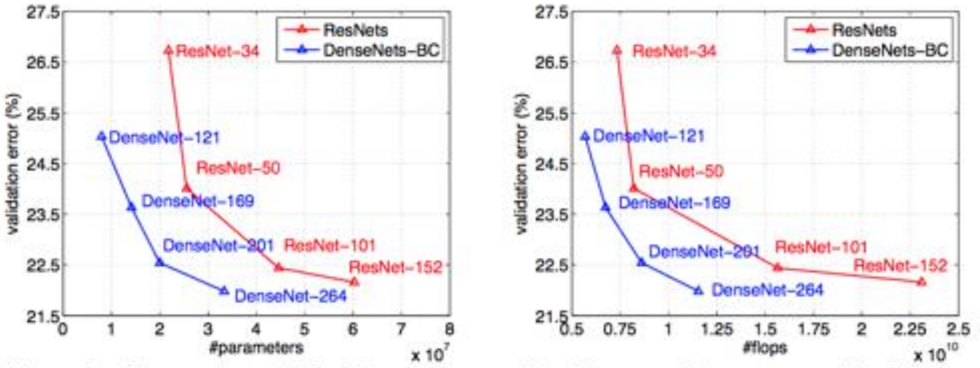
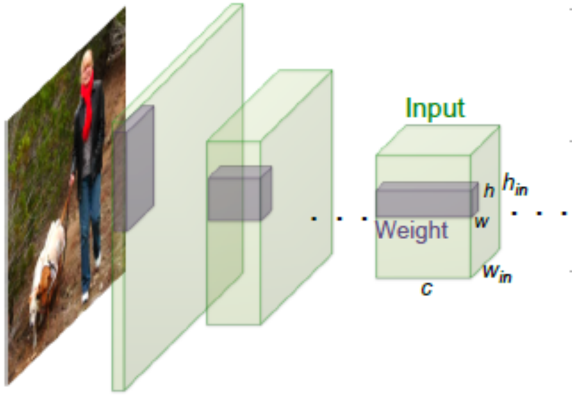



**Figure 3:** Comparison of the DenseNets and ResNets top-1 error rates (single-crop testing) on the ImageNet validation dataset as a function of learned parameters (*left*) and FLOPs during test-time (*right*).

Previously on CENG403

# Binary networks




| | Network Variations | | Operations used in Convolution | Memory Saving (Inference) | Time Saving on CPU (Inference) | Accuracy on ImageNet (AlexNet) |
|---|---|---|---|---|---|---|
| Standard Convolution | Real-Value Inputs<br>0.11 -0.21 ... -0.34<br>-0.25 0.61 ... 0.52 | Real-Value Weights<br>0.12 -1.2 ... 0.41<br>-0.2 0.5 ... 0.68 | +, −, × | 1x | 1x | %56.7 |
| Binary Weight | Real-Value Inputs<br>0.11 -0.21 ... -0.34<br>-0.25 0.61 ... 0.52 | Binary Weights<br>1 -1 ... 1<br>-1 1 ... 1 | +, − | ~32x | ~2x | %53.8 |
| BinaryWeight Binary Input **(XNOR-Net)** | Binary Inputs<br>1 -1 ... -1<br>-1 1 ... 1 | Binary Weights<br>1 -1 ... 1<br>-1 1 ... 1 | XNOR, bitcount | ~32x | ~58x | %44.2 |

Fig. 1: We propose two efficient variations of convolutional neural networks. **Binary-Weight-Networks**, when the weight filters contains binary values. **XNOR-Networks**, when both weigh and input have binary values. These networks are very efficient in terms of memory and computation, while being very accurate in natural image classification. This offers the possibility of using accurate vision techniques in portable devices with limited resources.

Previously on CENG403

# Different types of sequence learning / recognition problems

- Sequence Classification
  - A sequence to a label
  - E.g., recognizing a single spoken word
  - Length of the sequence is fixed
  - Why RNNs then? Because sequential modeling provides robustness against translations and distortions.
- Segment Classification
  - Segments in a sequence correspond to labels
- Temporal Classification
  - General case: sequence (input) to sequence (label) modeling.
  - May not have clue about where input or label starts.

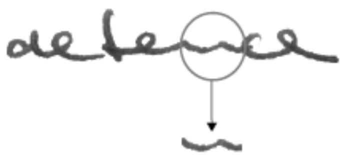
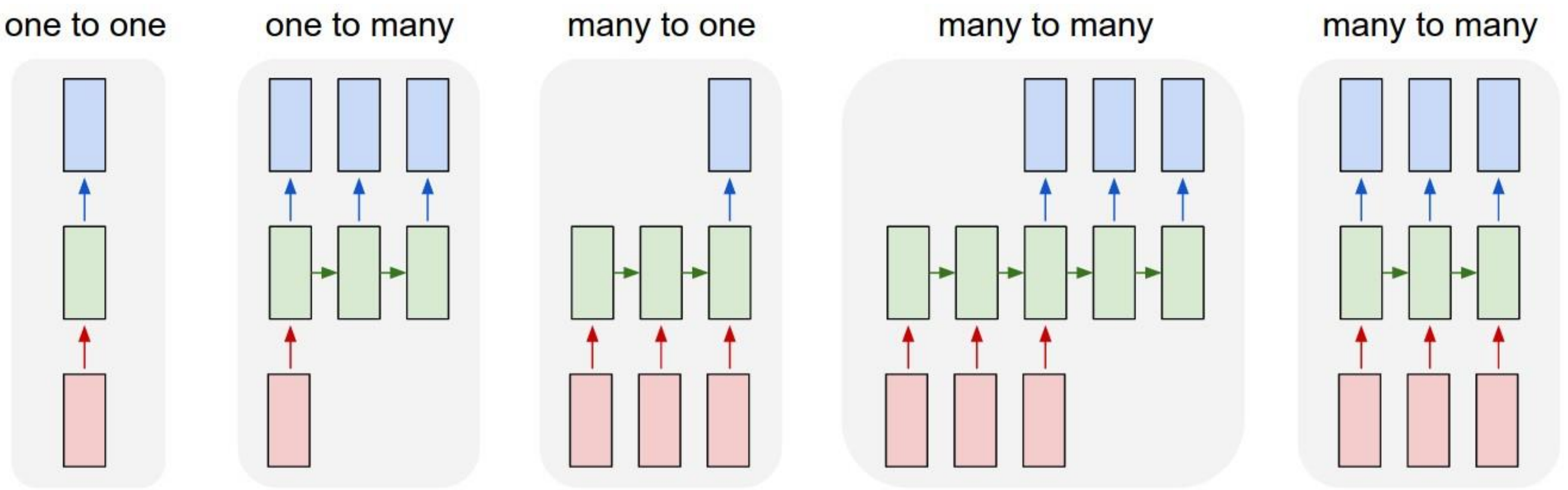


**Fig. 2.3 Importance of context in segment classification.** The word 'defence' is clearly legible. However the letter 'n' in isolation is ambiguous.

A. Graves, "Supervised Sequence Labelling with Recurrent Neural Networks", 2012.

Previously on CENG403

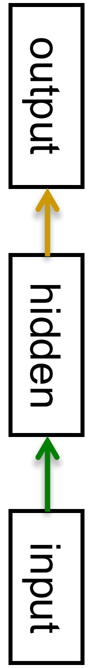
# Different types of sequence learning / recognition problems



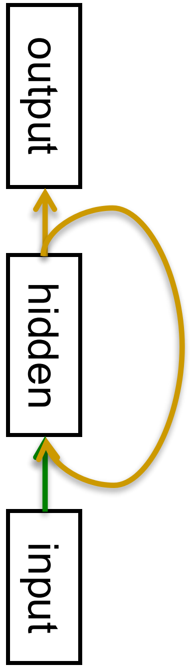

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

# Recurrent Neural Networks (RNNs)

Previously on CENG403




Feed-forward networks




Recurrent networks

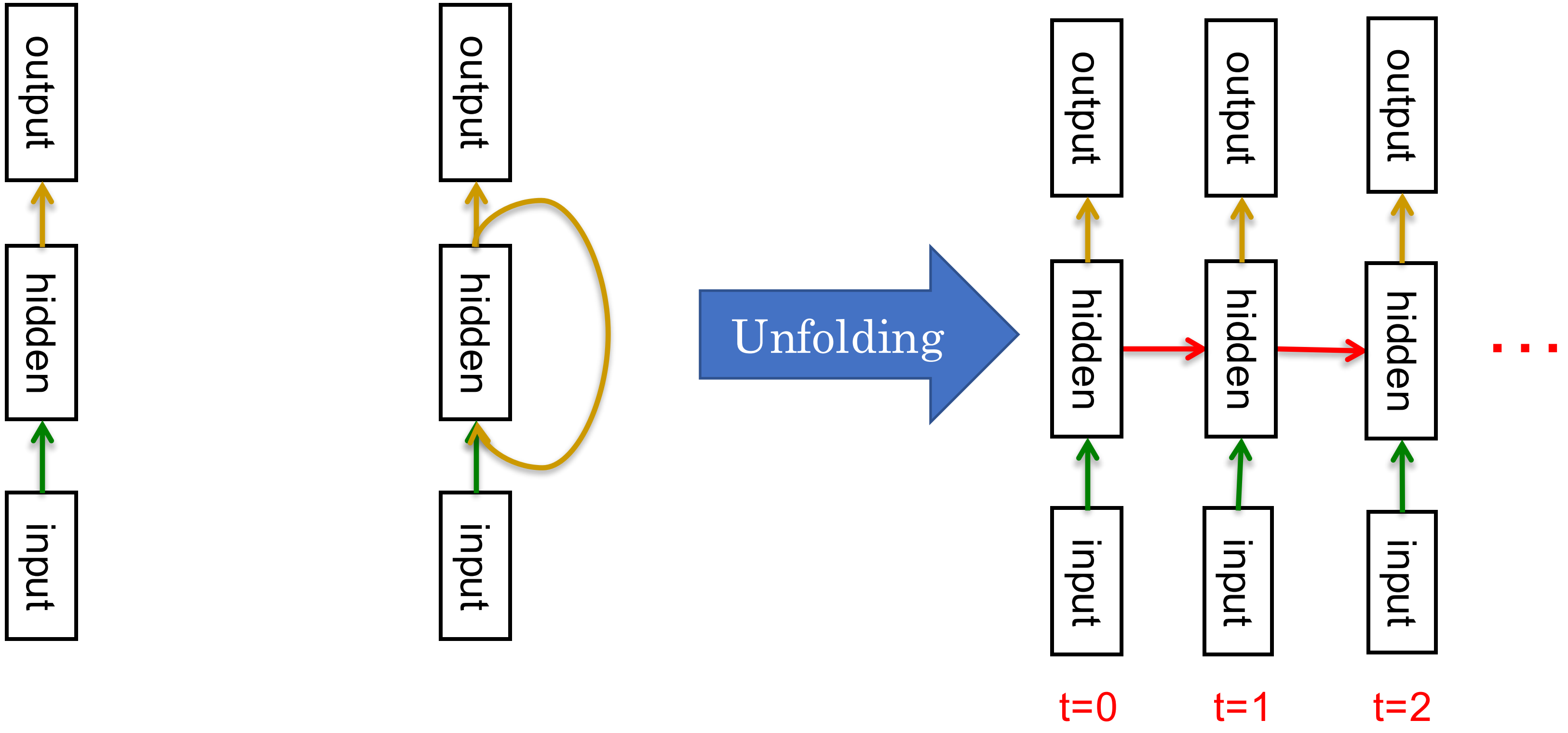
- RNNs are very powerful because:
  - Distributed hidden state that allows them to store a lot of information about the past efficiently.
  - Non-linear dynamics that allows them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by your computer.
- More formally, RNNs are Turing complete.



Adapted from Hinton

Previously on CENG403

# Unfolding




Feed-forward networks

Recurrent networks

time →

# Today

- Recurrent Neural Networks (RNNs)
  - Backpropagation Through Time
  - Problems with RNNs
  - LSTM networks
  - Language modeling

# Back-propagation Through Time

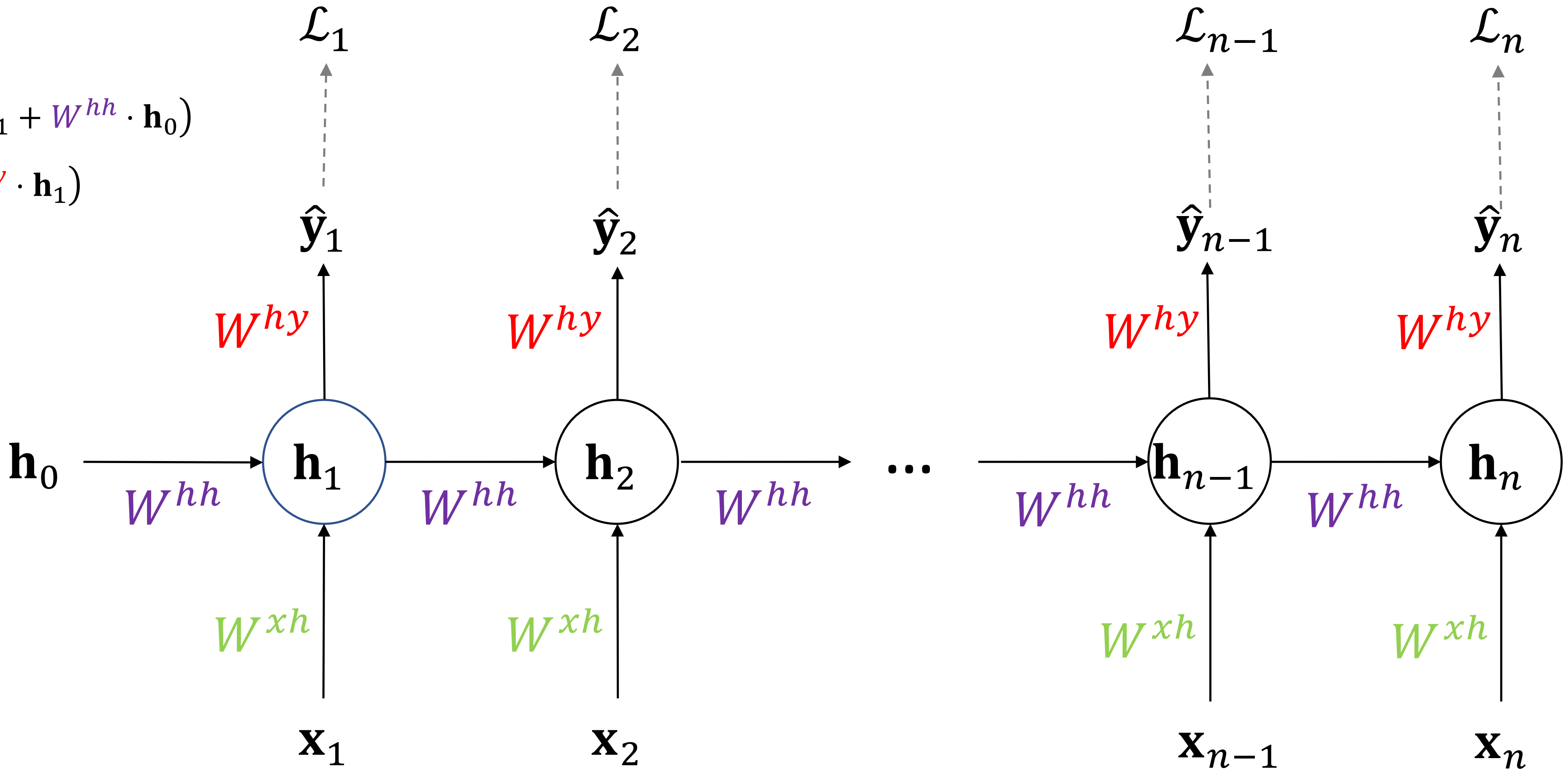
# Feedforward through Vanilla RNN

# Feedforward through Vanilla RNN

$\mathbf{h}_1 = tanh(W^{xh} \cdot \mathbf{x}_1 + W^{hh} \cdot \mathbf{h}_0)$

$\hat{\mathbf{y}}_1 = softmax(W^{hy} \cdot \mathbf{h}_1)$

$\mathcal{L}_1 = CE(\hat{\mathbf{y}}_1, \mathbf{y}_1)$



# Feedforward through Vanilla RNN

**The Vanilla RNN Model**

First time-step ($t = 1$):

$$\mathbf{h}_1 = tanh\left(W^{xh} \cdot \mathbf{x}_1 + W^{hh} \cdot \mathbf{h}_0\right)$$

$$\hat{\mathbf{y}}_1 = softmax\left(W^{hy} \cdot \mathbf{h}_1\right)$$

$$\mathcal{L}_1 = CE(\hat{\mathbf{y}}_1, \mathbf{y}_1)$$

In general:

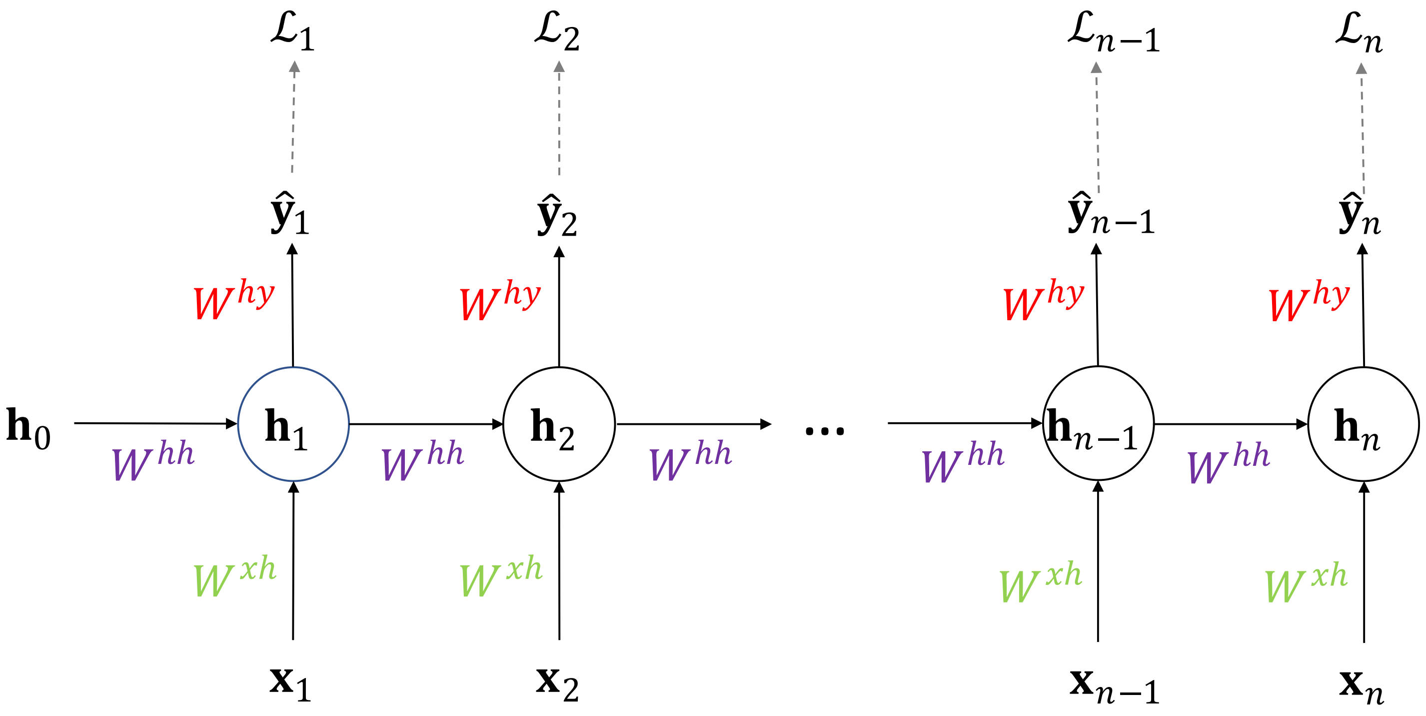
$$\mathbf{h}_t = tanh\left(W^{xh} \cdot \mathbf{x}_t + W^{hh} \cdot \mathbf{h}_{t-1}\right)$$

$$\hat{\mathbf{y}}_t = softmax\left(W^{hy} \cdot \mathbf{h}_t\right)$$

$$\mathcal{L}_t = CE(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

In total:

$$\mathcal{L} = \sum_t \mathcal{L}_t$$



# Backpropagation through Vanilla RNN

**The Vanilla RNN Model**

In general:

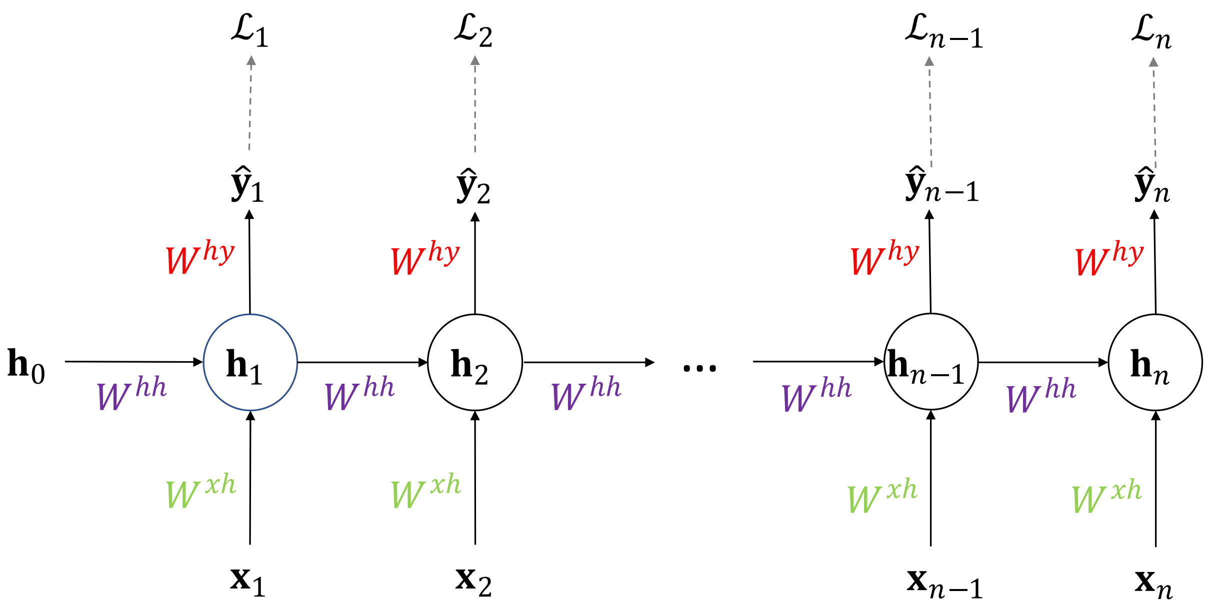
$$\mathbf{h}_t = tanh\left(W^{xh} \cdot \mathbf{x}_t + W^{hh} \cdot \mathbf{h}_{t-1}\right)$$

$$\hat{\mathbf{y}}_t = softmax\left(W^{hy} \cdot \mathbf{h}_t\right)$$

$$\mathcal{L}_t = CE(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

In total:

$$\mathcal{L} = \sum_t \mathcal{L}_t$$



# Backpropagation through Vanilla RNN

**The Vanilla RNN Model**

In general:

$$\mathbf{h}_t = tanh\left(W^{xh} \cdot \mathbf{x}_t + W^{hh} \cdot \mathbf{h}_{t-1}\right)$$

$$\hat{\mathbf{y}}_t = softmax\left(W^{hy} \cdot \mathbf{h}_t\right)$$

$$\mathcal{L}_t = CE(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

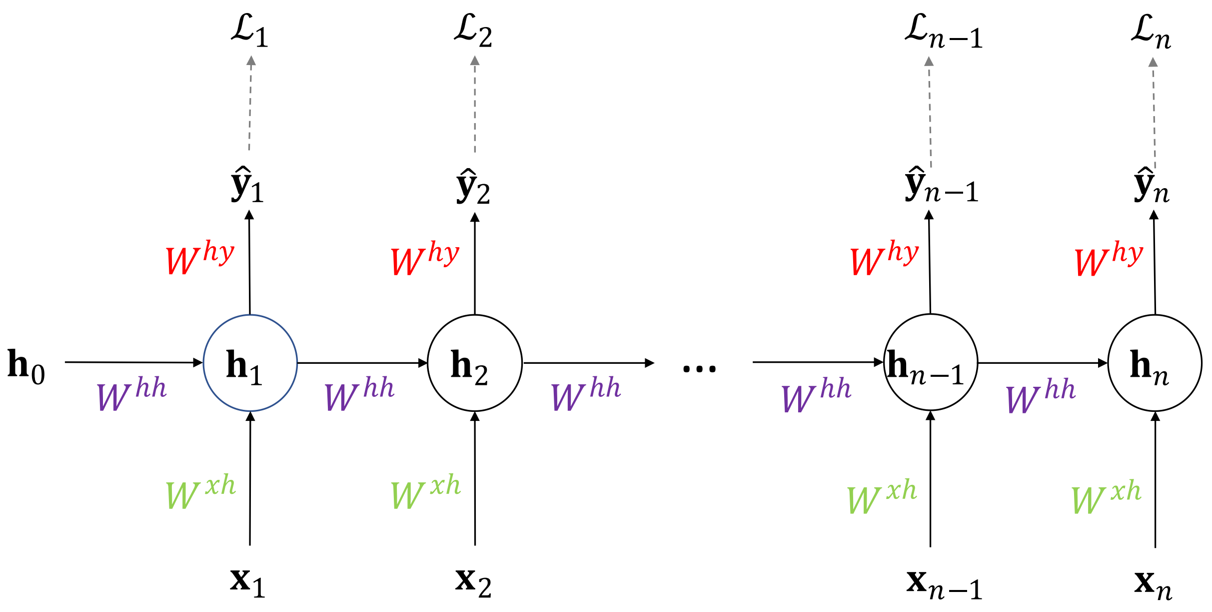
In total:

$$\mathcal{L} = \sum_t \mathcal{L}_t$$

$$\frac{\partial \mathcal{L}}{\partial W^{hy}} = ?$$

$$= \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_n} \frac{\partial \hat{\mathbf{y}}_n}{\partial W^{hy}} + \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_{n-1}} \frac{\partial \hat{\mathbf{y}}_{n-1}}{\partial W^{hy}} + \cdots + \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_1} \frac{\partial \hat{\mathbf{y}}_1}{\partial W^{hy}}$$

$$= \sum_{t=1..n} \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial W^{hy}}$$



# Backpropagation through Vanilla RNN

$$\frac{\partial \mathcal{L}}{\partial W^{hh}} = ?$$

$$= \frac{\partial \mathcal{L}}{\partial \mathbf{h}_n}\frac{\partial \mathbf{h}_n}{\partial W^{hh}} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{n-1}}\frac{\partial \mathbf{h}_{n-1}}{\partial W^{hh}} + \cdots + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W^{hh}}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}}\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$$

**The Vanilla RNN Model**

In general:

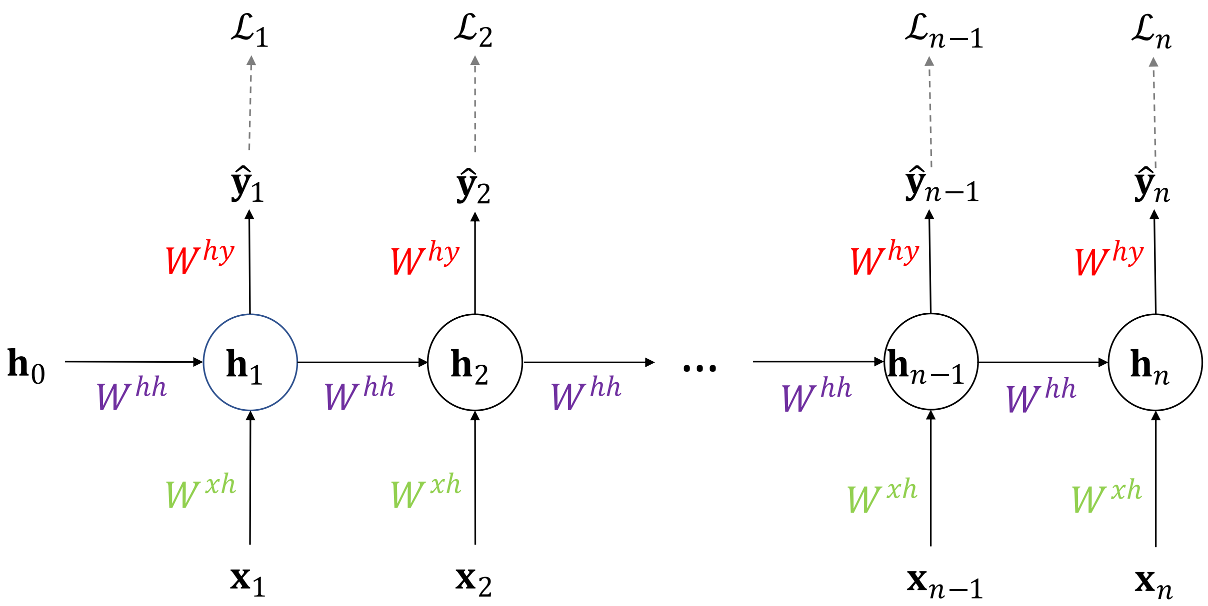
$$\mathbf{h}_t = tanh(W^{xh}\cdot \mathbf{x}_t + W^{hh}\cdot \mathbf{h}_{t-1})$$

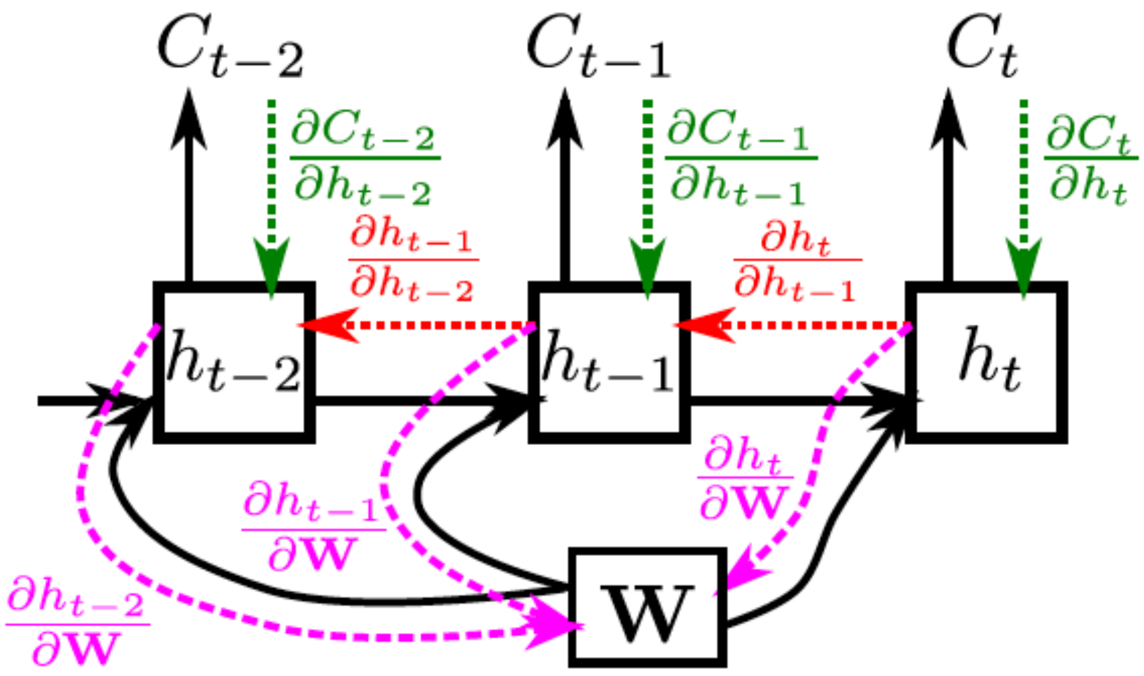
$$\hat{\mathbf{y}}_t = softmax(W^{hy}\cdot \mathbf{h}_t)$$

$$\mathcal{L}_t = CE(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

In total:

$$\mathcal{L} = \sum_t \mathcal{L}_t$$






$$\frac{\partial C_t}{\partial \mathbf{W}} = \sum_{t'=1}^{t} \frac{\partial C_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial h_{t'}}{\partial \mathbf{W}}, \quad \text{where} \quad \frac{\partial h_t}{\partial h_{t'}} = \prod_{k=t'+1}^{t} \frac{\partial h_k}{\partial h_{k-1}}$$

# Backpropagation through Vanilla RNN

$$\frac{\partial \mathcal{L}}{\partial W^{xh}} = ?$$

$$= \frac{\partial \mathcal{L}}{\partial \mathbf{h}_n}\frac{\partial \mathbf{h}_n}{\partial W^{xh}} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{n-1}}\frac{\partial \mathbf{h}_{n-1}}{\partial W^{xh}} + \cdots + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W^{xh}}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}}\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$$

(calculated before)

**The Vanilla RNN Model**

In general:

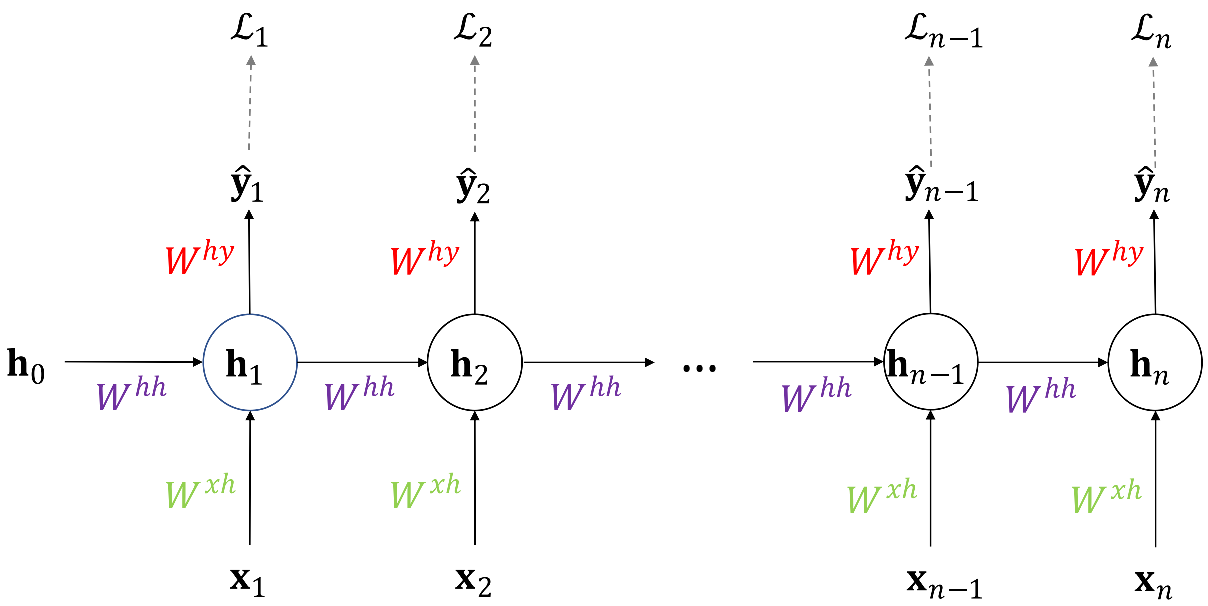
$$\mathbf{h}_t = tanh\left(W^{xh} \cdot \mathbf{x}_t + W^{hh} \cdot \mathbf{h}_{t-1}\right)$$

$$\hat{\mathbf{y}}_t = softmax\left(W^{hy} \cdot \mathbf{h}_t\right)$$

$$\mathcal{L}_t = CE(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

In total:

$$\mathcal{L} = \sum_t \mathcal{L}_t$$



# Initial hidden state

- We need to specify the initial activity state of all the hidden units.
- We could just fix these initial states to have some default value like 0.5.
- But it is better to treat the initial states as learned parameters.
- We learn them in the same way as we learn the weights.
  - Start off with an initial random guess for the initial states.
  - At the end of each training sequence, backpropagate through time all the way to the initial states  to get the gradient of the error function with respect to each initial state.
  - Adjust the initial states by following the negative gradient.



Slide: Hinton

# Initializing parameters

- Since an unfolded RNN is a deep MLP, we can use Xavier initialization.

# The problem of exploding or vanishing gradients

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}}\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$$

- What happens to the magnitude of the gradients as we backpropagate through many layers?
  - If the weights are small, the gradients shrink exponentially.
  - If the weights are big the gradients grow exponentially.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily explode or vanish.
  - We can avoid this by initializing the weights very carefully.
- Even with good initial weights, its very hard to detect that the current target output depends on an input from many time-steps ago.
  - So RNNs have difficulty dealing with long-range dependencies.



Slide: Hinton

# Exploding and vanishing gradients problem

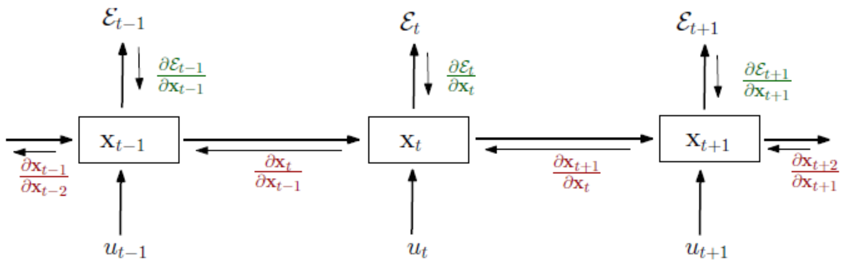
- Solution 1: Gradient clipping for exploding gradients:

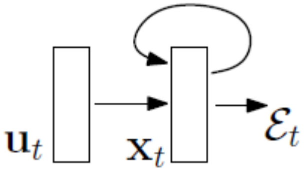
**Algorithm 1** Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
**if** $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
**end if**

  - For vanishing gradients: Regularization term that penalizes changes in the magnitudes of back-propagated gradients

$$\Omega = \sum_k \Omega_k = \sum_k \left( \frac{\left\| \frac{\partial \mathcal{E}}{\partial \mathbf{x}_{k+1}} \frac{\partial \mathbf{x}_{k+1}}{\partial \mathbf{x}_k} \right\|}{\left\| \frac{\partial \mathcal{E}}{\partial \mathbf{x}_{k+1}} \right\|} - 1 \right)^2$$




On the difficulty of training recurrent neural networks


**Razvan Pascanu** PASCANUR@IRO.UMONTREAL.CA
Université de Montréal, 2920, chemin de la Tour, Montréal, Québec, Canada, H3T 1J8
**Tomas Mikolov** T.MIKOLOV@GMAIL.COM
Speech@FIT, Brno University of Technology, Brno, Czech Republic
**Yoshua Bengio** YOSHUA.BENGIO@UMONTREAL.CA
Université de Montréal, 2920, chemin de la Tour, Montréal, Québec, Canada, H3T 1J8


2012

# Exploding and vanishing gradients problem

- Solution 2:
  - Use methods like LSTM

# Long Short-Term Memory (LSTM)

LONG SHORT-TERM MEMORY

NEURAL COMPUTATION 9(8):1735–1780, 1997

Sepp Hochreiter
Fakultät für Informatik
Technische Universität München
80290 München, Germany
hochreit@informatik.tu-muenchen.de
http://www7.informatik.tu-muenchen.de/~hochreit

Jürgen Schmidhuber
IDSIA
Corso Elvezia 36
6900 Lugano, Switzerland
juergen@idsia.ch
http://www.idsia.ch/~juergen

**Abstract**

Learning to store information over extended time intervals via recurrent backpropagation takes a very long time, mostly due to insufficient, decaying error back flow. We briefly review Hochreiter's 1991 analysis of this problem, then address it by introducing a novel, efficient, gradient-based method called "Long Short-Term Memory" (LSTM). Truncating the gradient where this does not do harm, LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing *constant* error flow through "constant error carrousels" within special units. Multiplicative gate units learn to open and close access to the constant error flow. LSTM is local in space and time; its computational complexity per time step and weight is $O(1)$. Our experiments with artificial data involve local, distributed, real-valued, and noisy pattern representations. In comparisons with RTRL, BPTT, Recurrent Cascade-Correlation, Elman nets, and Neural Sequence Chunking, LSTM leads to many more successful runs, and learns much faster. LSTM also solves complex, artificial long time lag tasks that have never been solved by previous recurrent network algorithms.

# RNN

- Basic block diagram

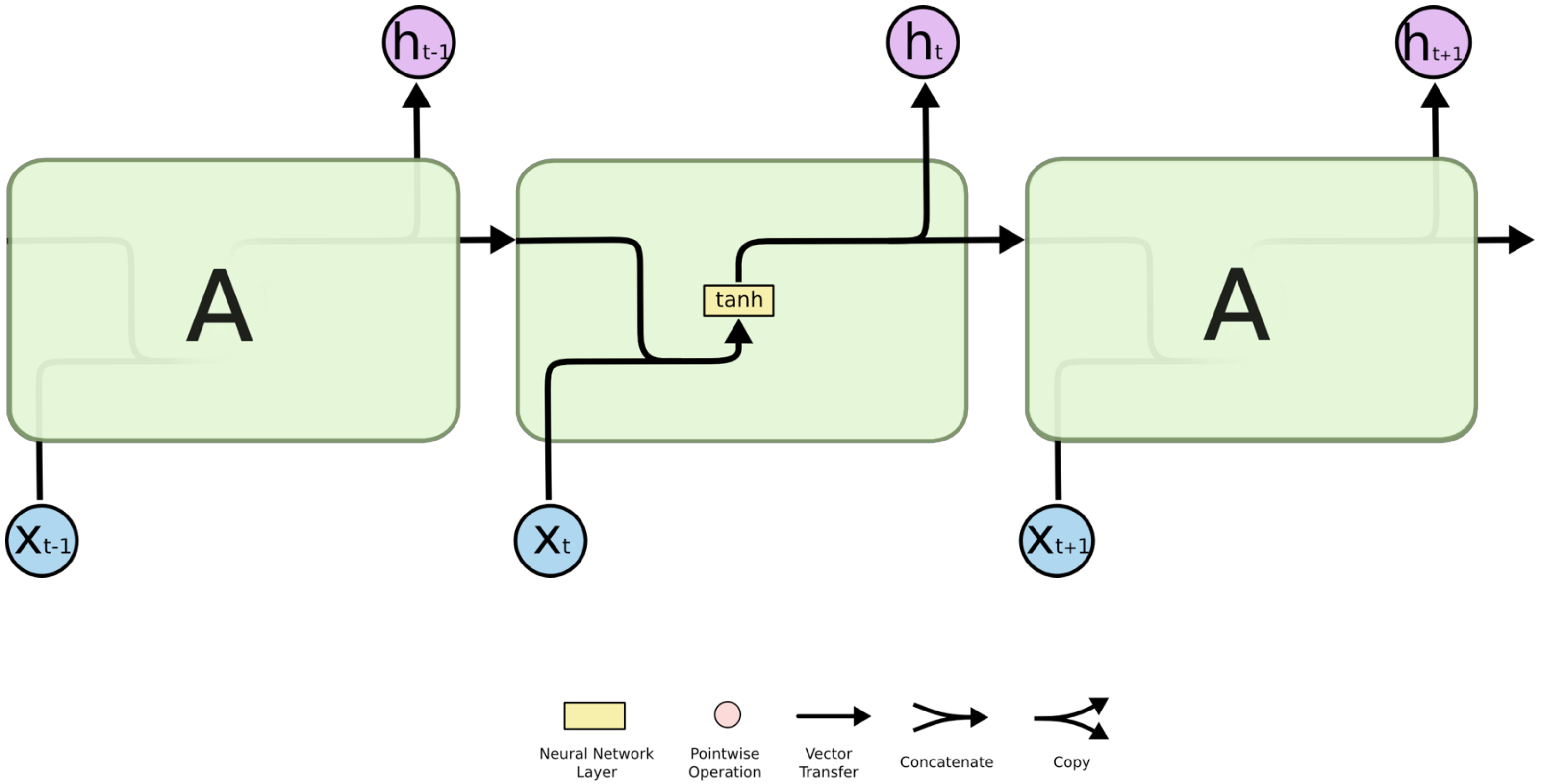





Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# Key Problem

- Learning long-term dependencies is hard

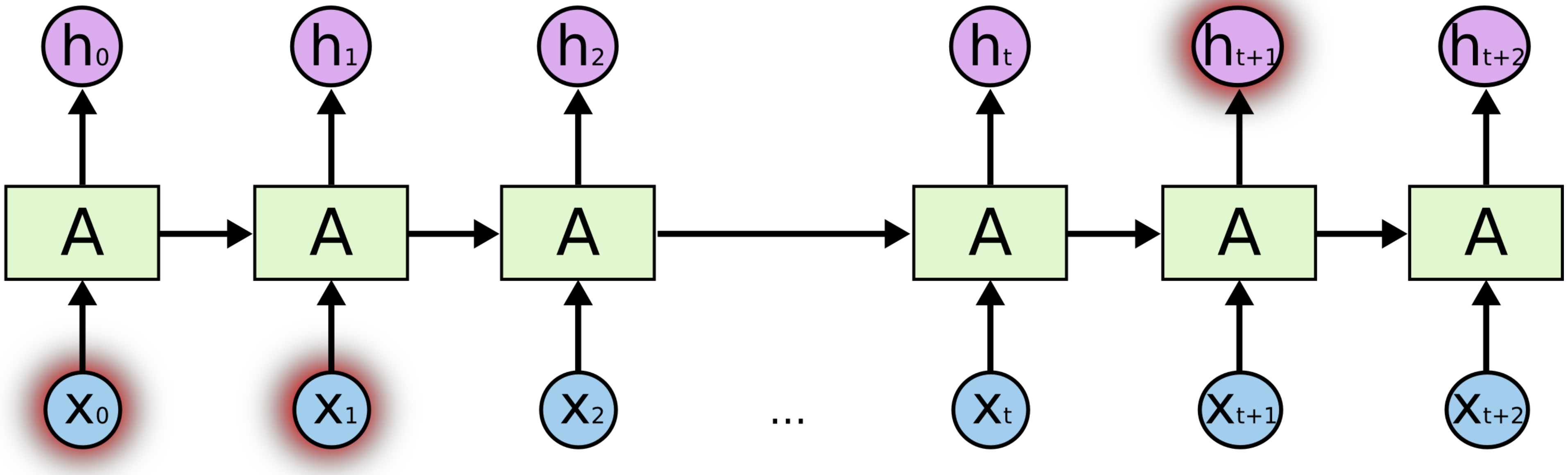





Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# Long Short Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
- They designed a memory cell using logistic and linear units with multiplicative interactions.

- Information gets into the cell whenever its "write" gate is on.
- The information stays in the cell so long as its "keep" gate is on.
- Information can be read from the cell by turning on its "read" gate.

# Meet LSTMs

- How about we explicitly encode memory?

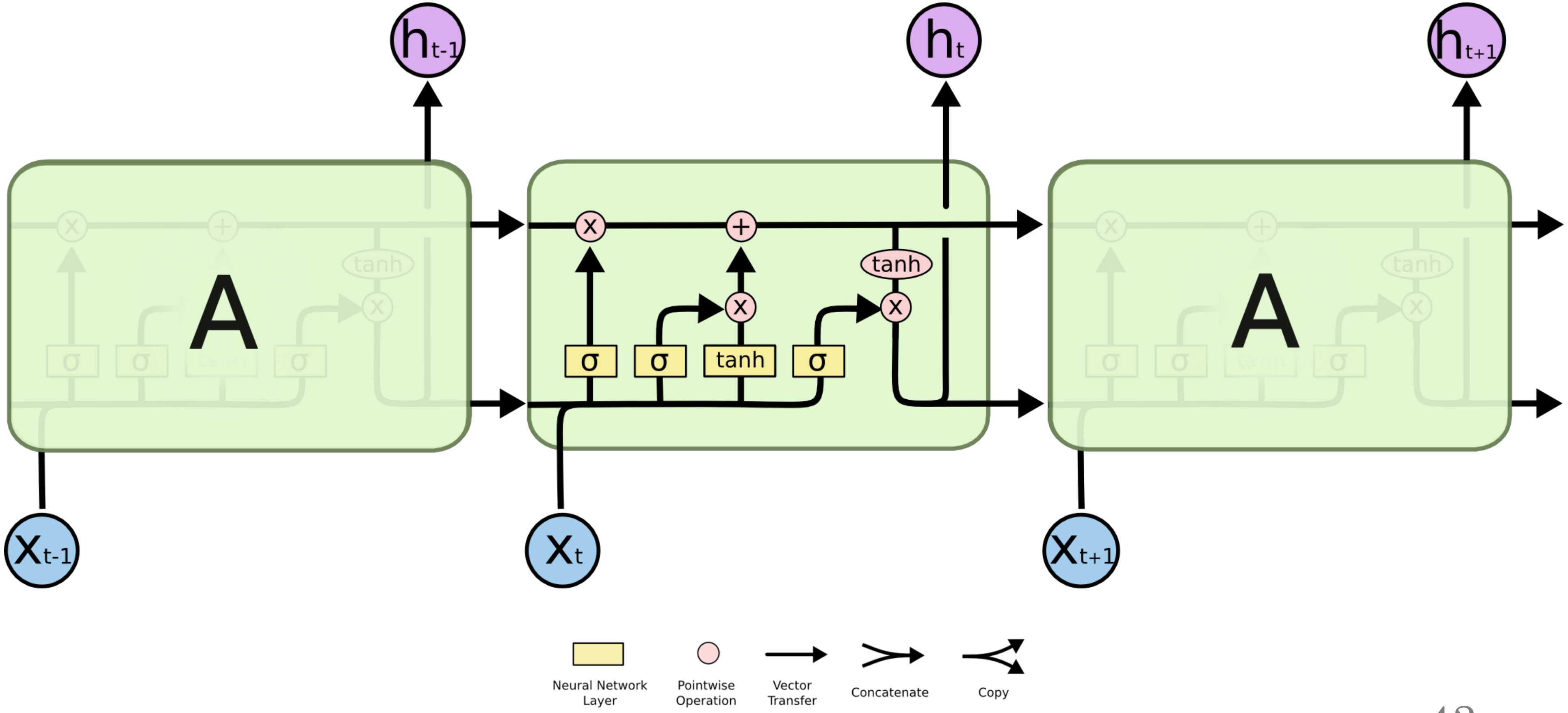




Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

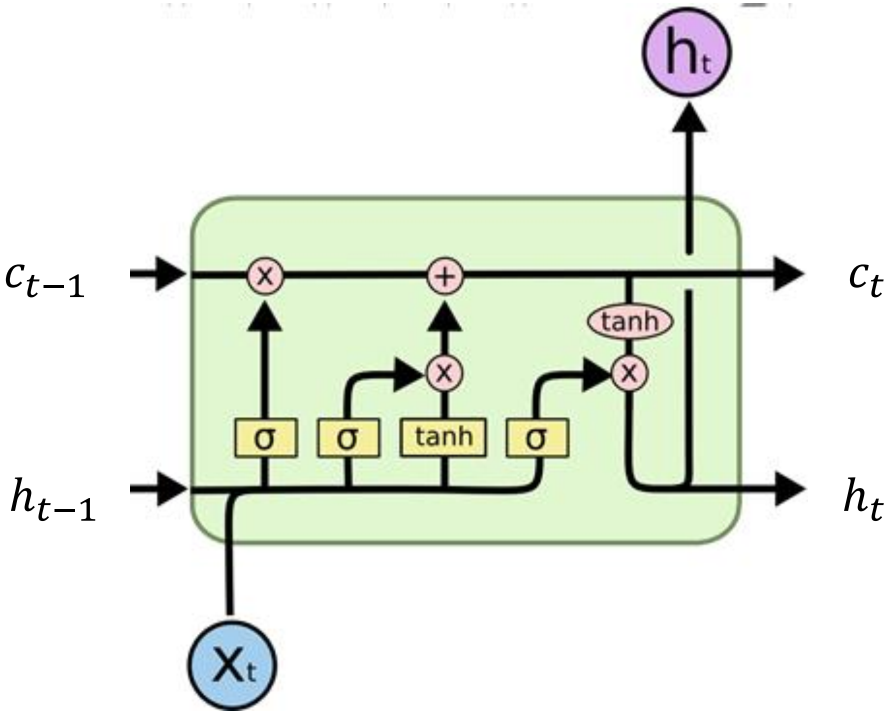





Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# LSTM in detail

- We first compute an activation vector, $a$:
  $$a = W_x x_t + W_h h_{t-1} + b$$
- Split this into four vectors of the same size:
  $$a_f, a_i, a_g, a_o \leftarrow a$$
- We then compute the values of the gates:
  $$f = \sigma(a_f) \quad i = \sigma(a_i) \quad g = \tanh(a_g) \quad o = \sigma(a_o)$$
  where $\sigma$ is the sigmoid.
- The next cell state $c_t$ and the hidden state $h_t$:
  $$c_t = f \odot c_{t-1} + i \odot g$$
  $$h_t = o \odot \tanh(c_t)$$
  where $\odot$ is the element-wise product of vectors

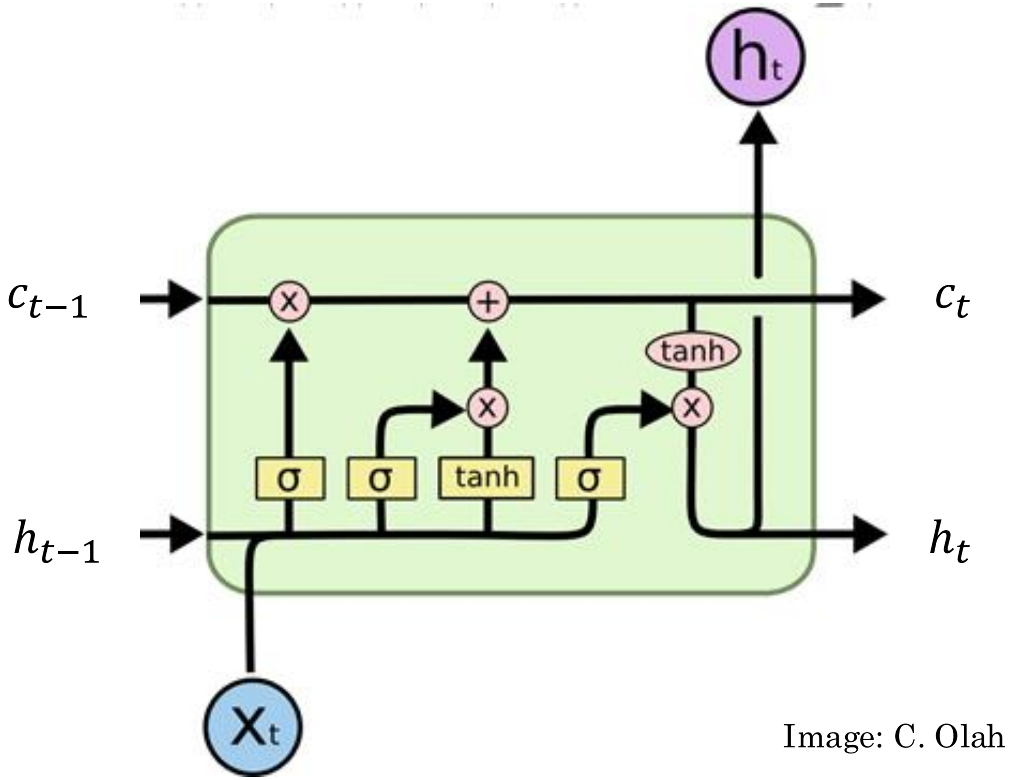



Image: C. Olah

Alternative formulation:

$$i_t = g(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$
$$f_t = g(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$
$$o_t = g(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

Eqs: Karpathy

# LSTMs Intuition: Memory

- Cell State / Memory

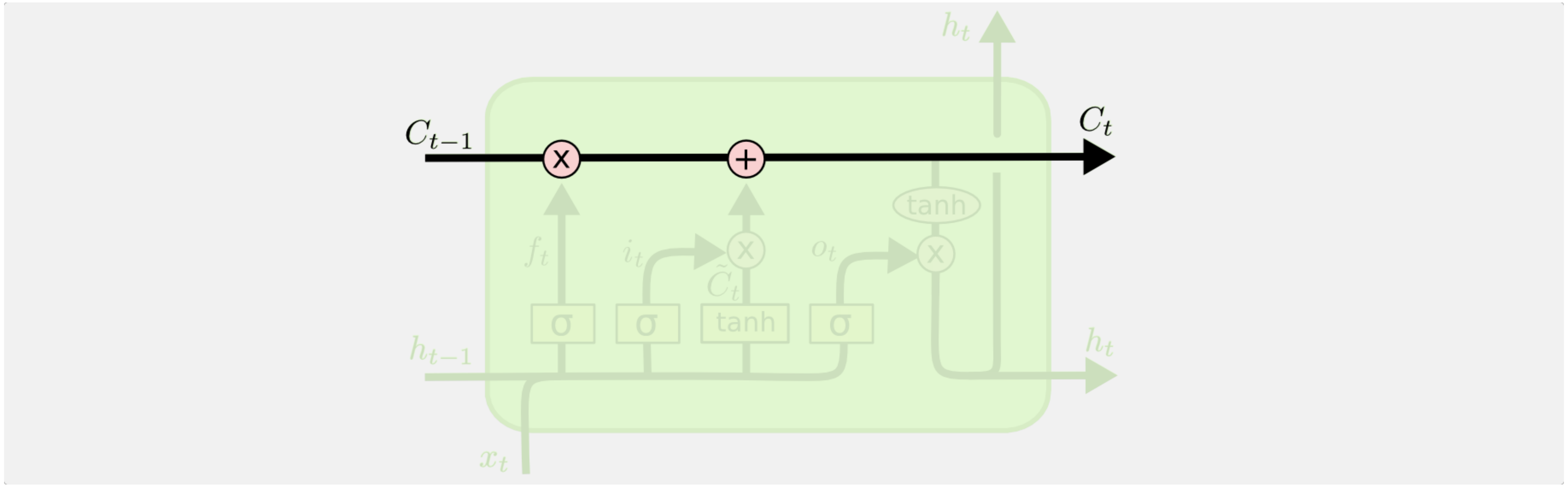
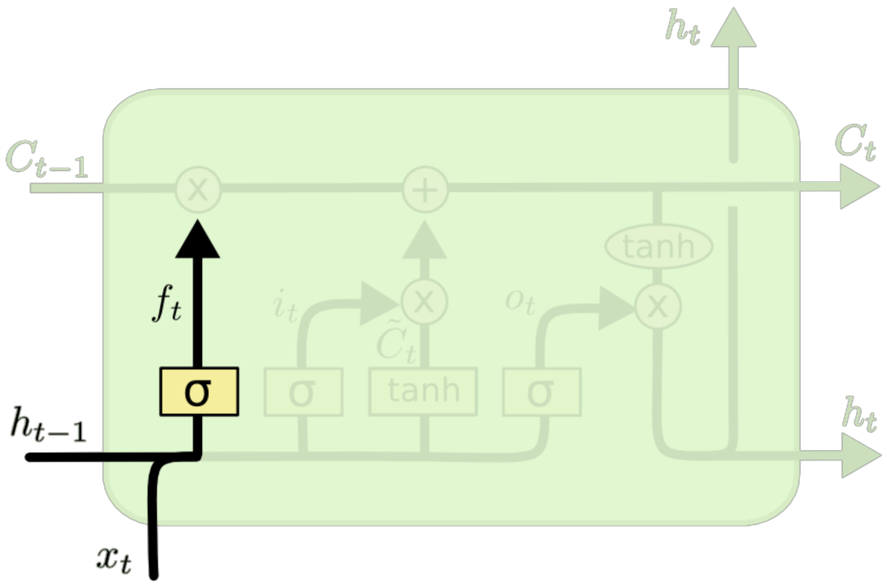



Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# LSTMs Intuition: Forget Gate

- Should we continue to remember this “bit” of information or not?

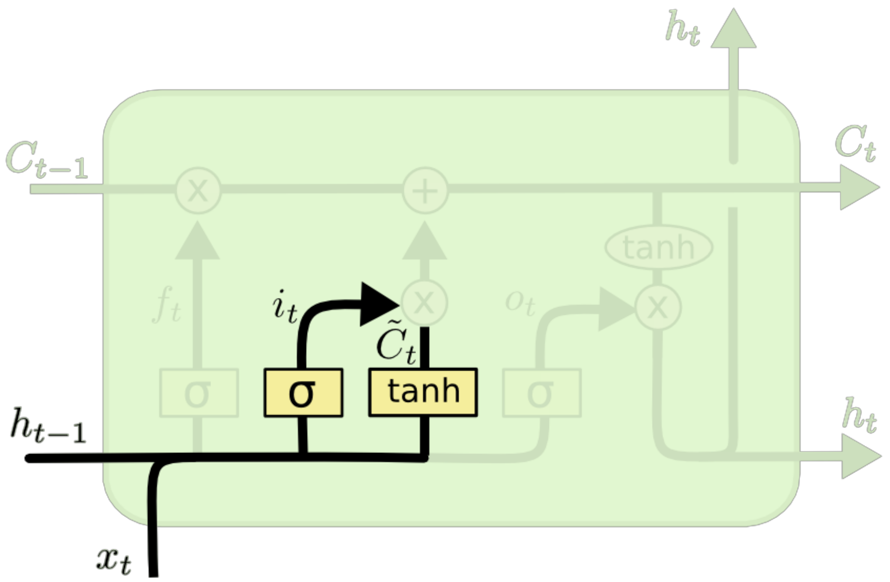



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$


Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# LSTMs Intuition: Input Gate

- Should we update this “bit” of information or not?
  - If so, with what?




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

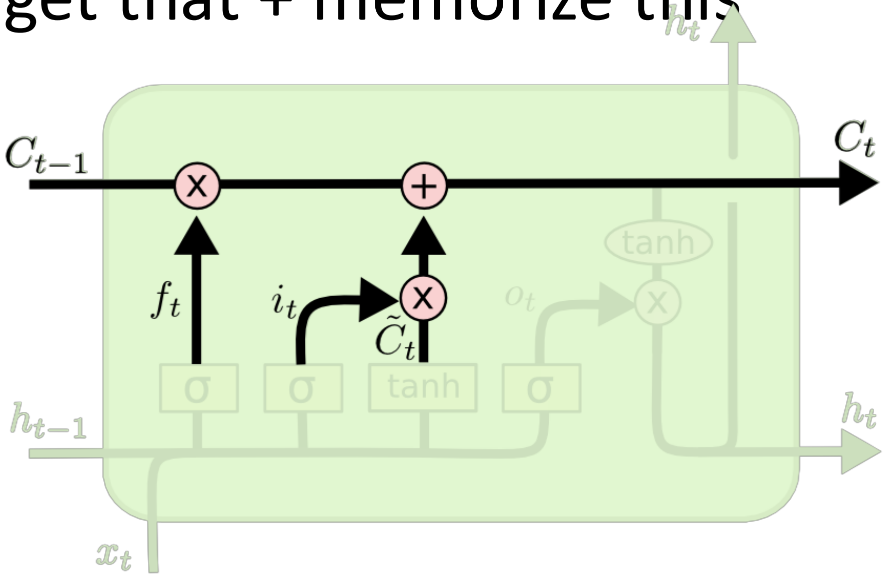
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$



Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# LSTMs Intuition: Memory Update

- Forget that + memorize this

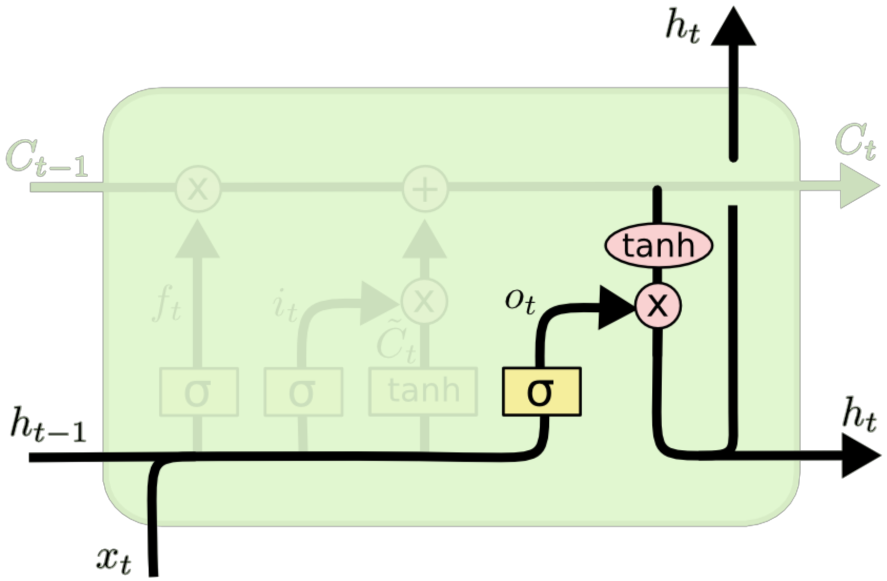



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$


Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# LSTMs Intuition: Output Gate

- Should we output this “bit” of information to “deeper” layers?




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


Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# LSTMs

- A pretty sophisticated cell

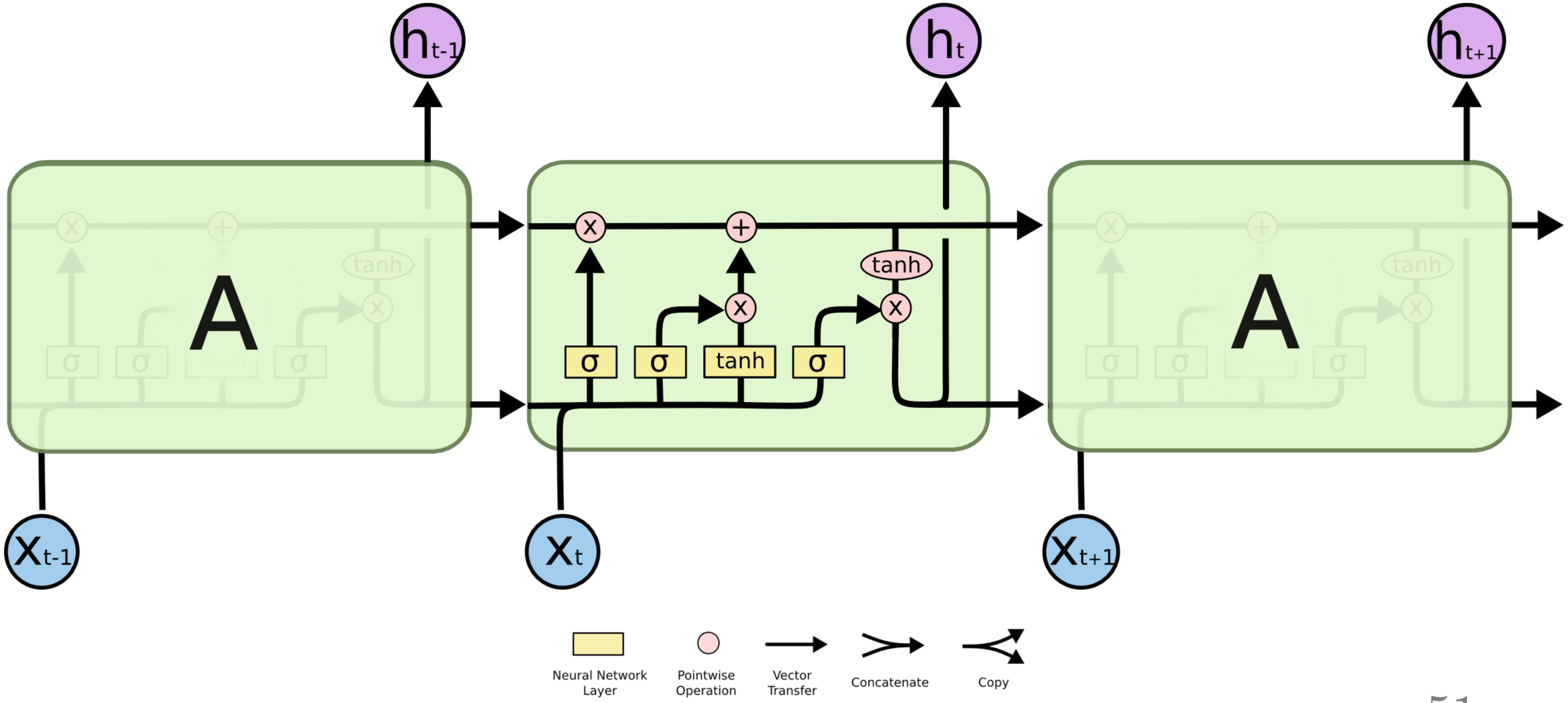
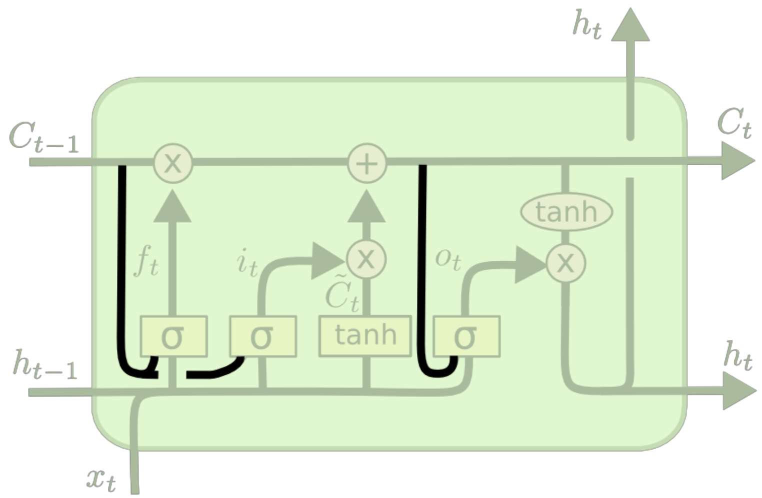





Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# LSTM Variants #1: Peephole Connections

- Let gates see the cell state / memory



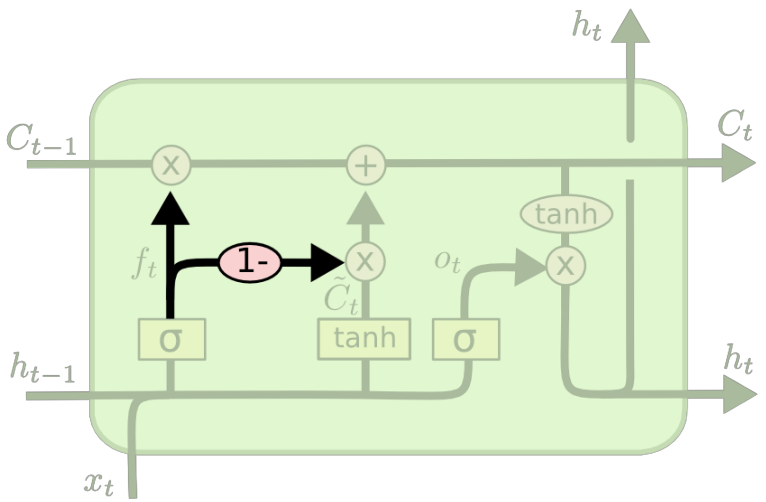

$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \ + \ b_o\right)$$



Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# LSTM Variants #2: Coupled Gates

- Only memorize new if forgetting old



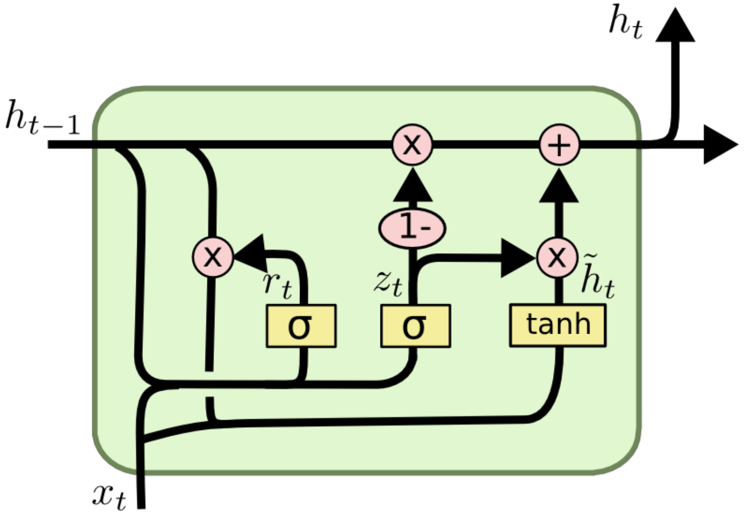

$$C_t = f_t * C_{t-1} + (1 - f_t) * \tilde{C}_t$$



Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# LSTM Variants #3: Gated Recurrent Units

- Changes:
  - No explicit memory; memory = hidden output
  - Z = memorize new and forget old




$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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



Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

# LSTM vs. GRU

Accepted as a short paper in ACL 2018

**On the Practical Computational Power of Finite Precision RNNs for Language Recognition**

**Gail Weiss** Technion, Israel
**Yoav Goldberg** Bar-Ilan University, Israel
**Eran Yahav** Technion, Israel

{sgailw,yahave}@cs.technion.ac.il
yogo@cs.biu.ac.il

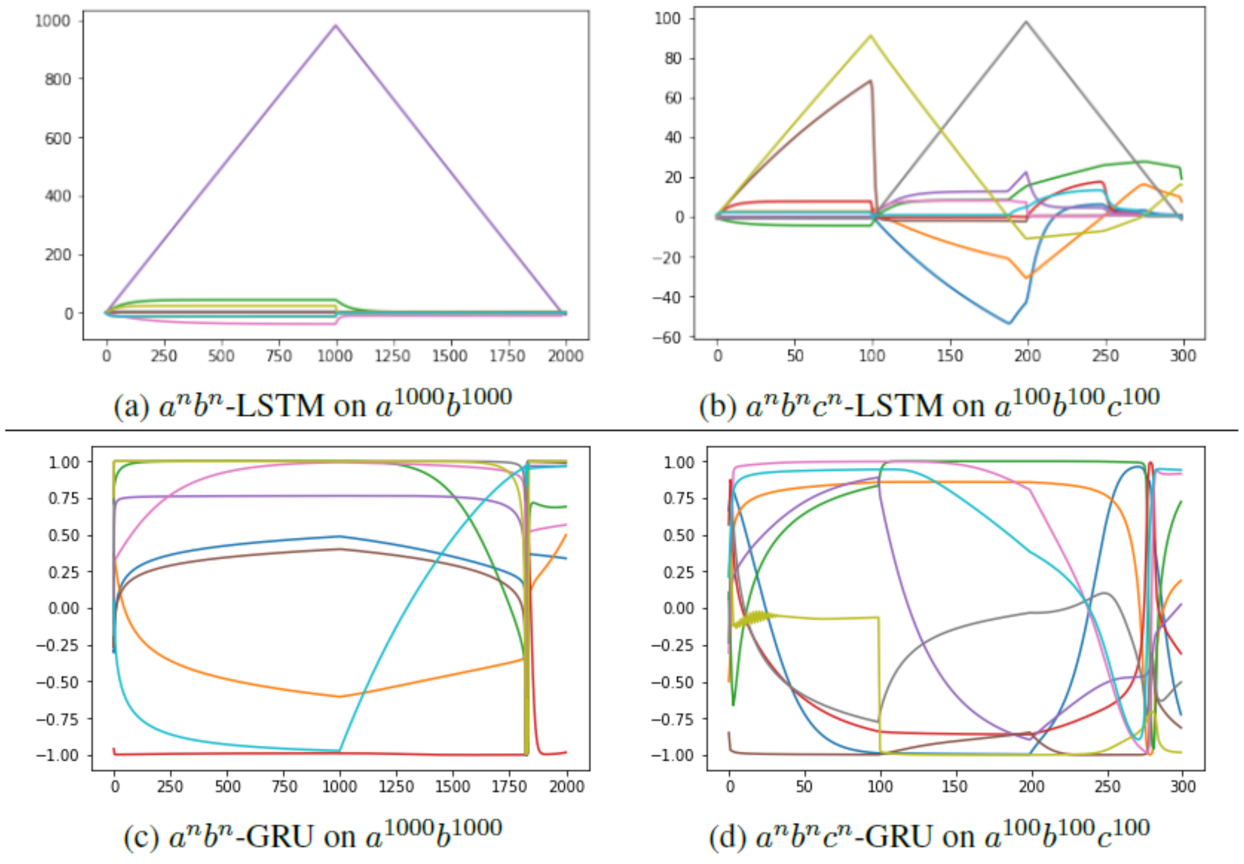
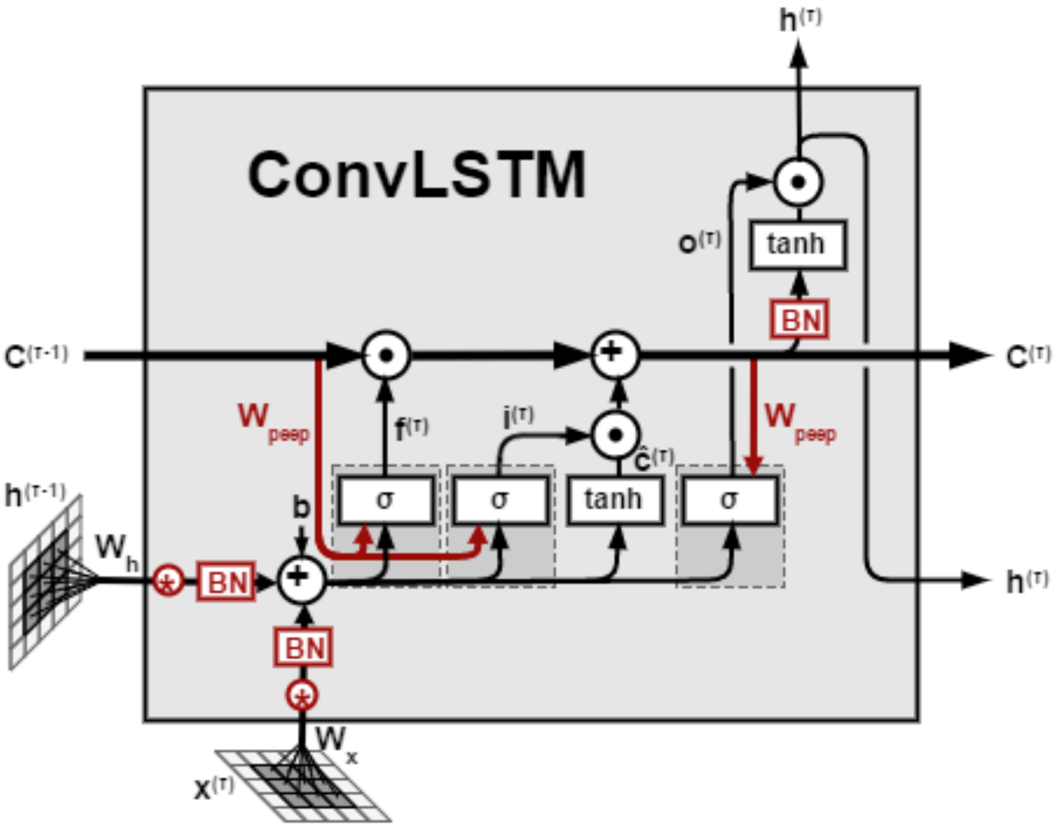



Figure 1: Activations — c for LSTM and h for GRU — for networks trained on $a^nb^n$ and $a^nb^nc^n$. The LSTM has clearly learned to use an explicit counting mechanism, in contrast with the GRU.

# ConvLSTM




https://medium.com/neuronio/an-introduction-to-convlstm-55c9025563a7

**Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting**


**Xingjian Shi Zhourong Chen Hao Wang Dit-Yan Yeung**
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
{xshiab,zchenbb,hwangaz,dyyeung}@cse.ust.hk

**Wai-kin Wong Wang-chun Woo**
Hong Kong Observatory
Hong Kong, China
{wkwong,wcwoo}@hko.gov.hk


2015

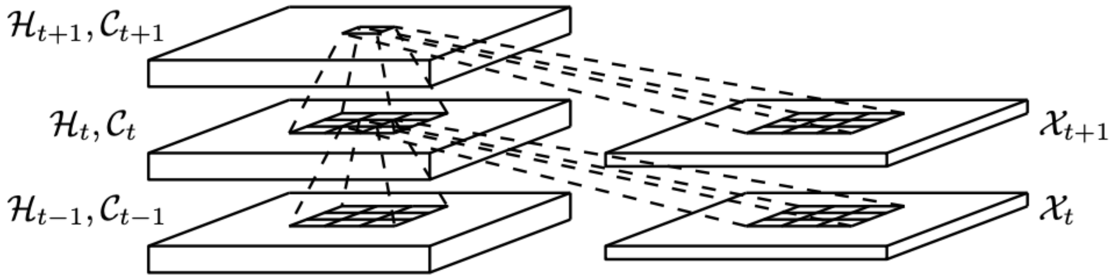



Figure 2: Inner structure of ConvLSTM

# Reference

- A very detailed explanation with nice figures

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

# CNN vs RNN

**An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling**

Shaojie Bai[1] J. Zico Kolter[2] Vladlen Koltun[3]

4 Mar 2018

**Abstract**

For most deep learning practitioners, sequence modeling is synonymous with recurrent networks. Yet recent results indicate that convolutional architectures can outperform recurrent networks on tasks such as audio synthesis and machine translation. Given a new sequence modeling task or dataset, which architecture should one use? We conduct a systematic evaluation of generic convo-

chine translation (van den Oord et al., 2016; Kalchbrenner et al., 2016; Dauphin et al., 2017; Gehring et al., 2017a;b). This raises the question of whether these successes of convolutional sequence modeling are confined to specific application domains or whether a broader reconsideration of the association between sequence processing and recurrent networks is in order.

We address this question by conducting a systematic empirical evaluation of convolutional and recurrent architectures

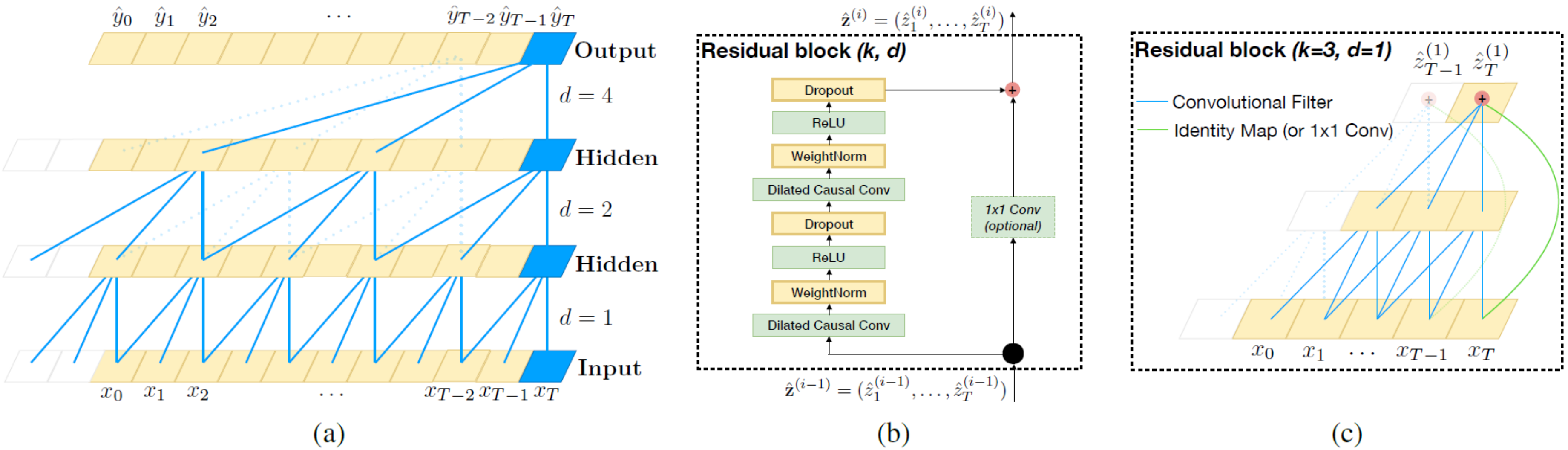



*Figure 1.* Architectural elements in a TCN. (a) A dilated causal convolution with dilation factors $d = 1, 2, 4$ and filter size $k = 3$. The receptive field is able to cover all values from the input sequence. (b) TCN residual block. An 1x1 convolution is added when residual input and output have different dimensions. (c) An example of residual connection in a TCN. The blue lines are filters in the residual function, and the green lines are identity mappings.