LG전자 Deep Learning 과정

RNN

Gunhee Kim

Computer Science and Engineering

서울대학교
SEOUL NATIONAL UNIVERSITY

Outline

- Recurrent Neural networks
- Backpropagation Through Time
- LSTM Recurrent Networks

(Credits: Stanford CS231n Lecture Notes)

Modeling Sequence Data

Limitation of previous neural networks

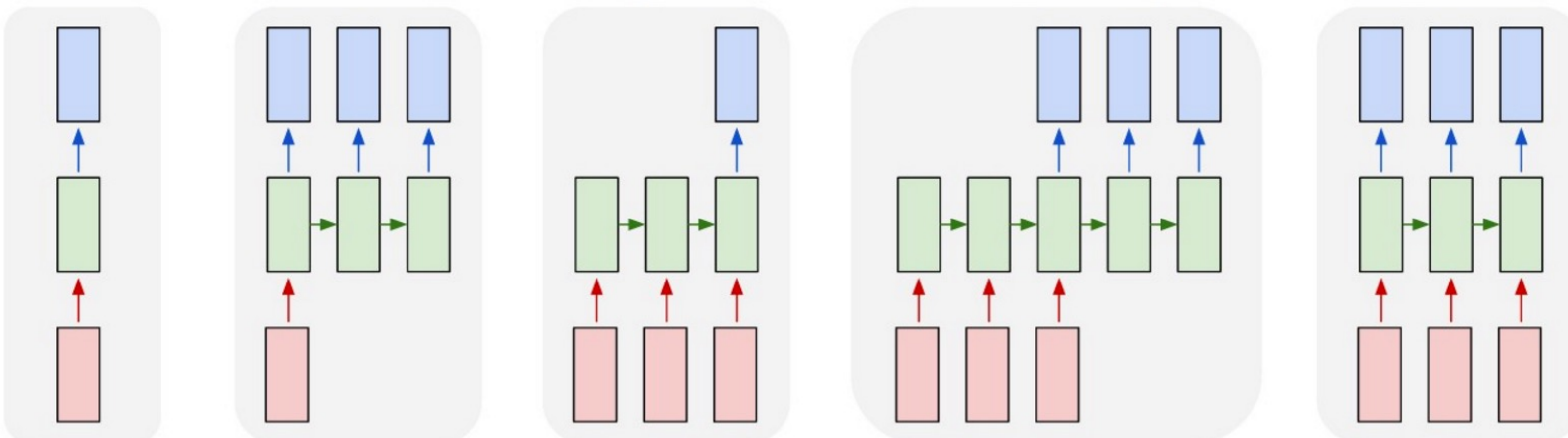
- Input and output are fixed-sized vectors (e.g. an image and probabilities of different classes)
- A fixed amount of computational steps (e.g. the number of layers in the model)

Can we handle variable length input and output?

- Sequence of words in a sentence
- Sequence of acoustic features at time frames in speech
- Successive frames in video classification

RNN Model Types

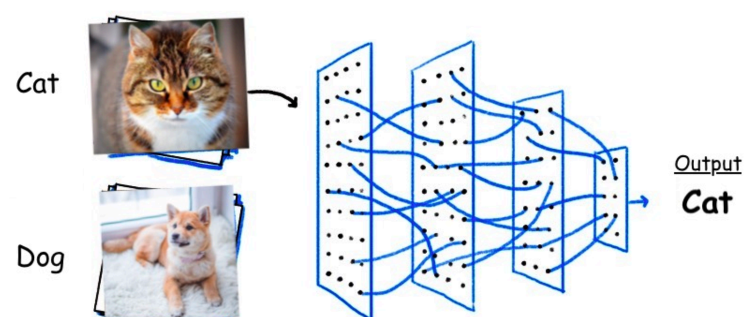
one-to-one one-to-many many-to-one many-to-many many-to-many



(1) (2) (3) (4) (5)

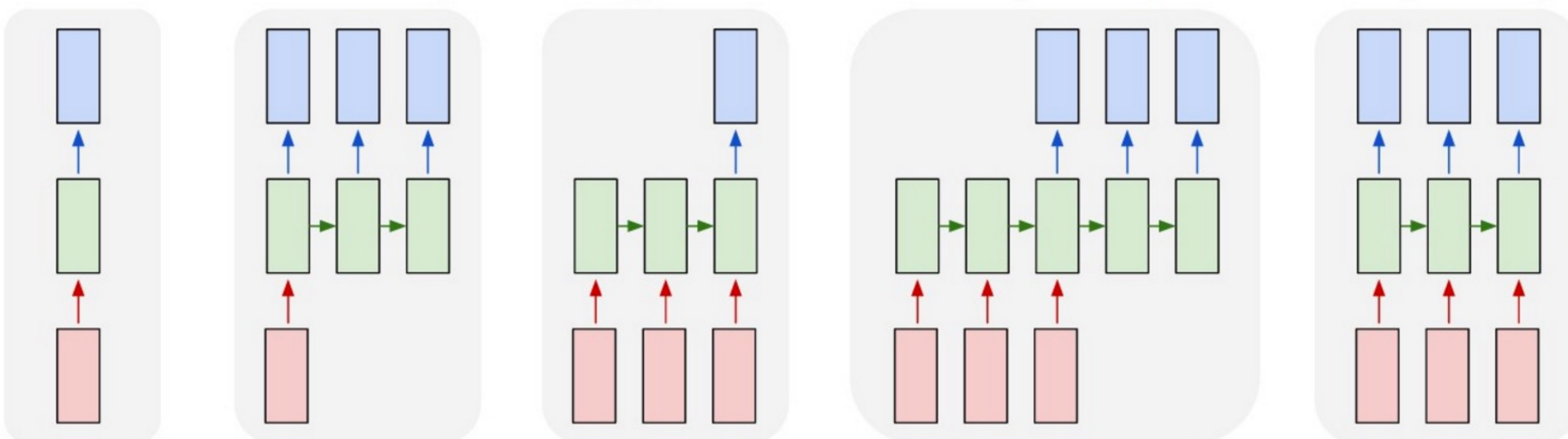
(1) Fixed-sized input and output

- e.g. image classification to classify an image into a fixed set of labels



RNN Model Types

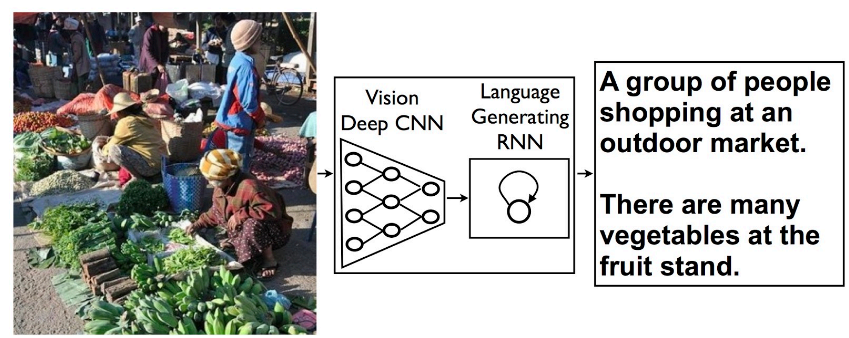
one-to-one | one-to-many | many-to-one | many-to-many | many-to-many



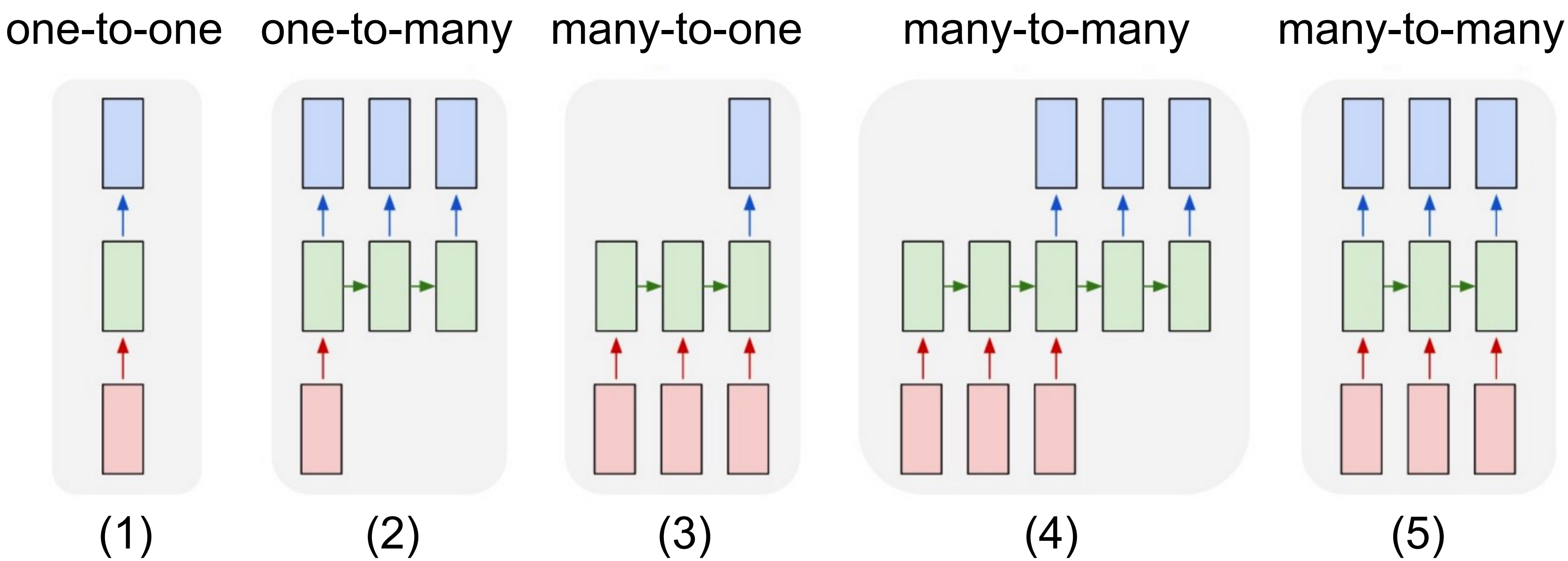
(1) (2) (3) (4) (5)

(2) Sequence output

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

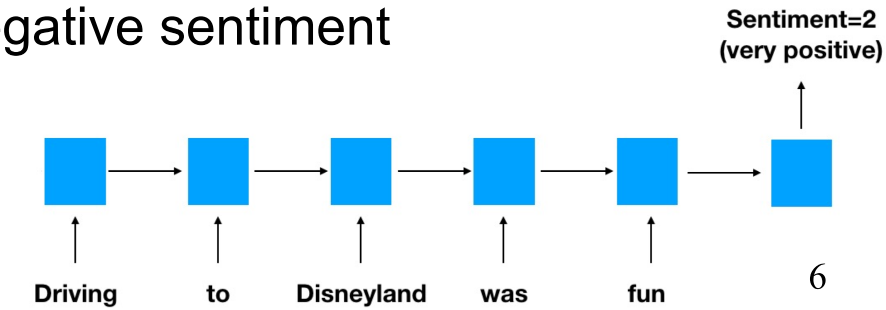


RNN Model Types

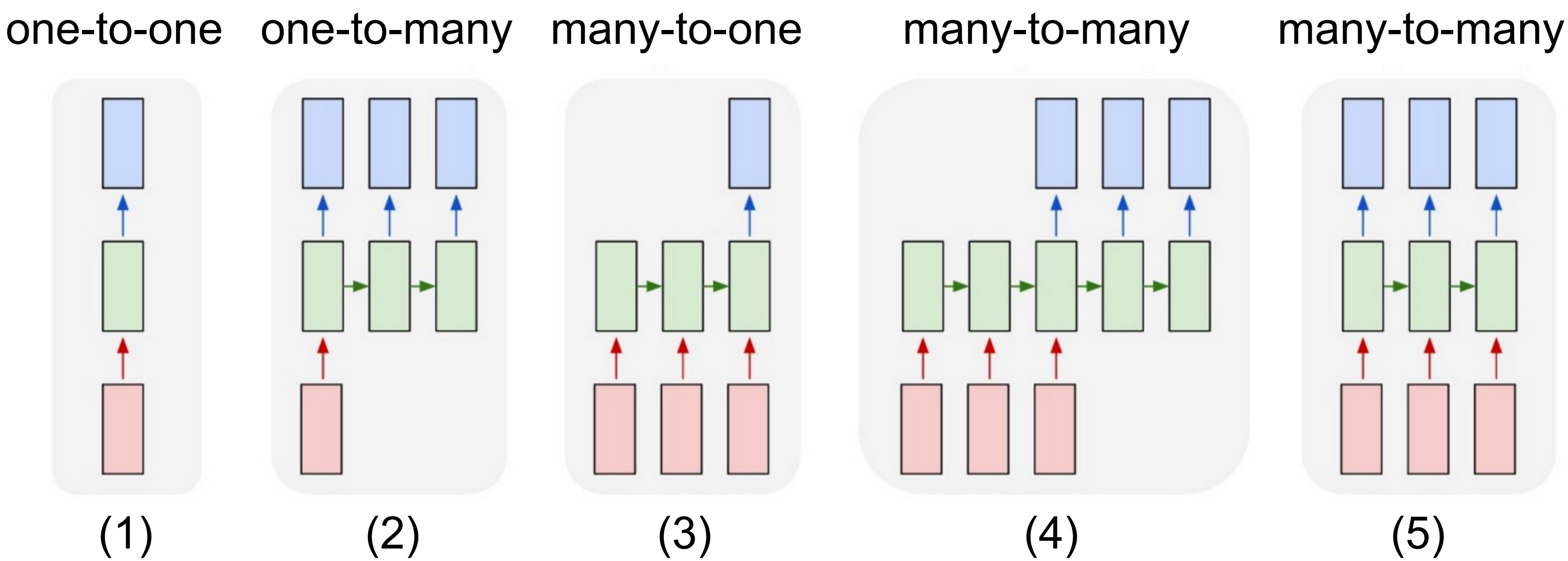


(3) Sequence of input

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

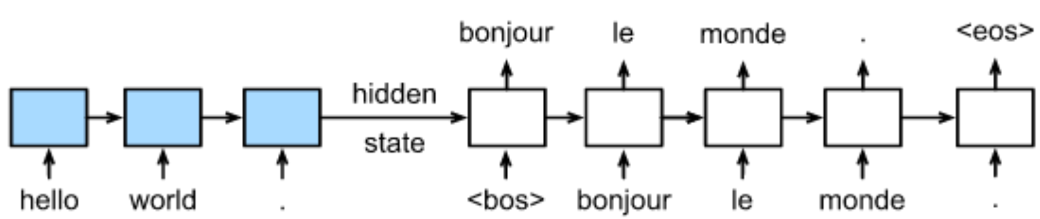


RNN Model Types



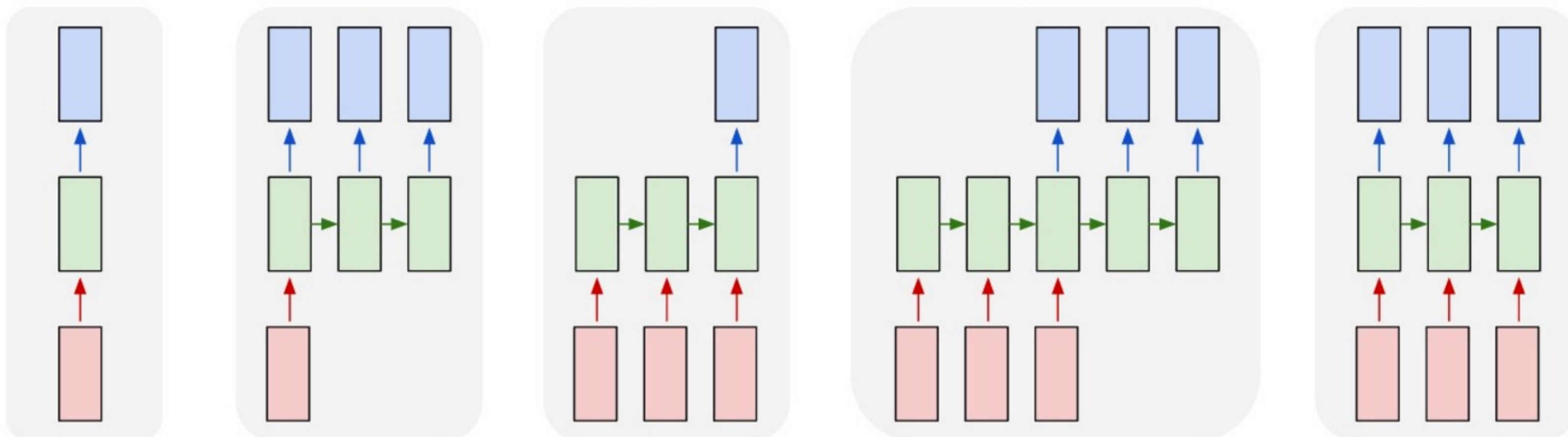
(4) Sequence input and sequence output

- e.g. machine translation from English to French sentence



RNN Model Types

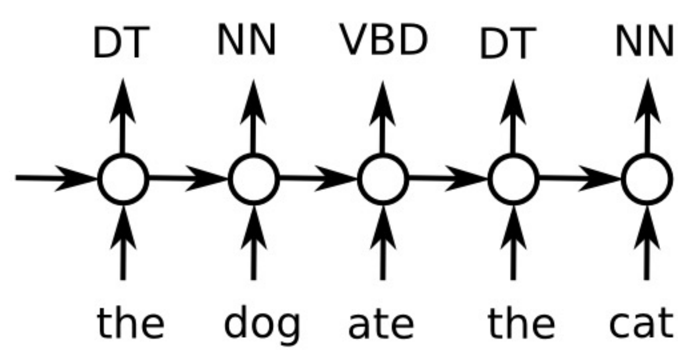
one-to-one one-to-many many-to-one many-to-many many-to-many



(1) (2) (3) (4) (5)

(5) Synced sequence input and output

- e.g. part-op-speech tagging and video classification to label each frame of the video

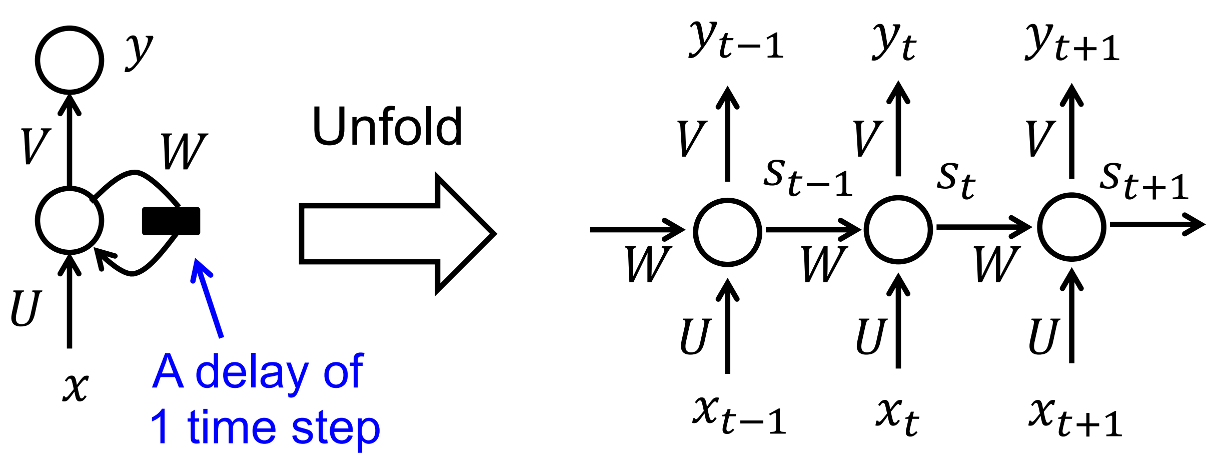


Recurrent Neural Network

A generative model for a dynamic system

- Recurrent: run the same task for every element of a sequence
- x_t: the input at time step t (e.g. word vector)
- s_t: the hidden state at time step t
- y_t: the output at time step t

Unfolding (unrolloing)



Recurrent Neural Network

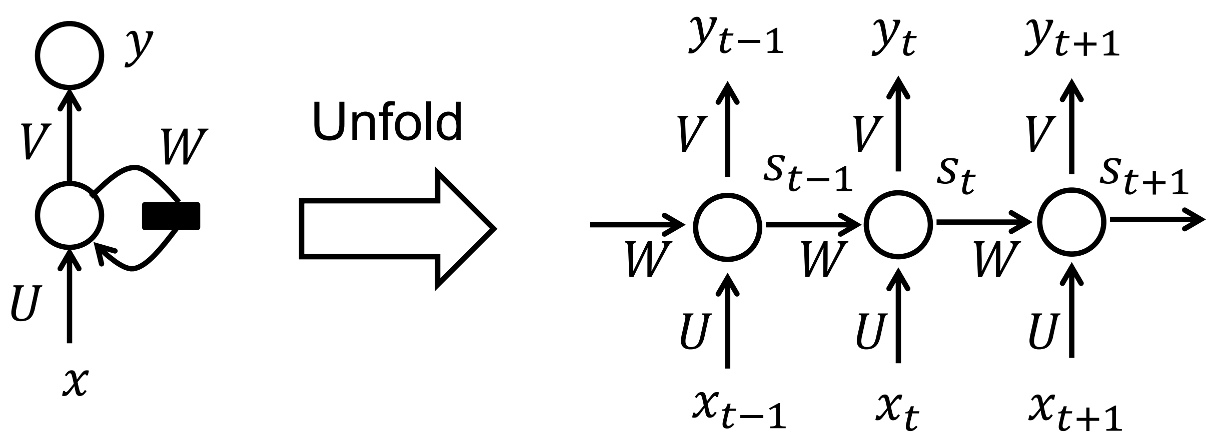
A generative model for a dynamic system

- The hidden state depends on the previous hidden and input

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

- f is a nonlinear function (hyperbolic tangent (tanh), or ReLU)
- If the output is the prediction of a word

$$y_t = \mathrm{softmax}(Vs_t + c)$$



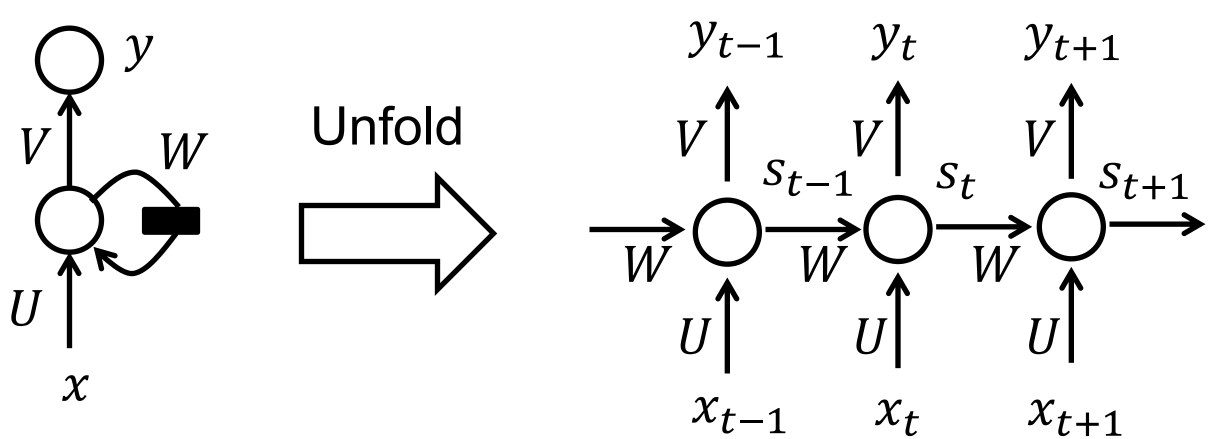
Recurrent Neural Network

The hidden state s_t contains information about the whole past sequence

- Lossy summary from an arbitrary length sequence $x_1, \ldots, x_{t-1}$ to a fixed length vector s_t

RNN shares the same parameters (U, V, W)

- Perform the same task at each step, just with different inputs
- Greatly reduces the total number of parameters to learn



RNN Extension – Bidirectional RNN

In vanilla RNN, the output at time t only depends on the previous elements in the sequence

- Let's consider both previous and future elements
- Combines a forward-going RNN and a backward-going RNN

Equations

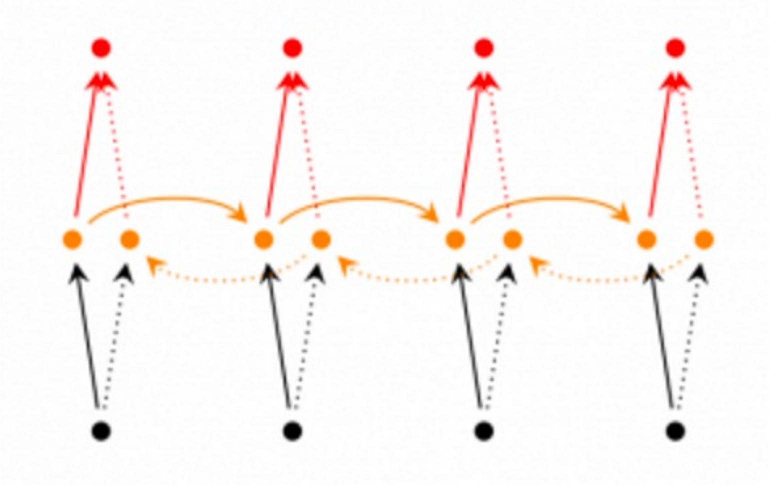
$$s_t = f(Ux_t + Ws_{t-1} + b)$$

$$y_t = \text{softmax}(Vs_t + c)$$

$$s_t^f = f(U^f x_t + W^f s_{t-1}^f + b^f)$$

$$s_t^b = f(U^b x_t + W^b s_{t+1}^b + b^b)$$

$$y_t = \text{softmax}(V^f s_t^f + V^b s_t^b + c)$$

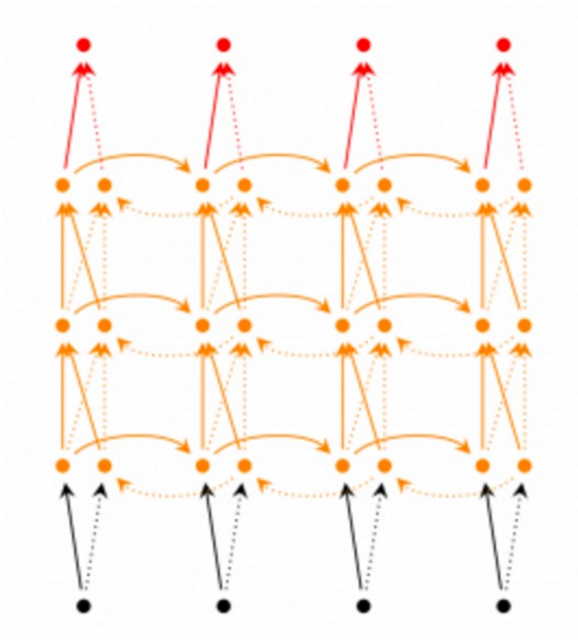


RNN Extension – Deep (Bidirectional) RNN

Multiple layers per time step

- Deep in both time and space
- Higher learning capacity (but need more training data)

Equations



$$s_t^{f(1)} = f(U^{f(1)}x_t + W^{f(1)}s_{t-1}^{f(1)} + b^{f(1)})$$

$$s_t^{b(1)} = f(U^{b(1)}x_t + W^{b(1)}s_{t+1}^{b(1)} + b^{b(1)})$$

$$s_t^{f(l)} = f(U^{ff(l)}s_t^{f(l-1)} + U^{fb(l)}s_t^{b(l-1)} + W^{f(l)}s_{t-1}^{f(l)} + b^{f(l)})$$

$$s_t^{b(l)} = f(U^{bf(l)}s_t^{f(l-1)} + U^{bb(l)}s_t^{b(l-1)} + W^{b(l)}s_{t-1}^{b(l)} + b^{b(l)})$$

$$y_t = \mathrm{softmax}(V^{f(L)}s_t^{f(L)} + V^{b(L)}s_t^{b(L)} + c)$$

Outline

- Recurrent Neural networks
- Backpropagation Through Time
- LSTM Recurrent Networks

(Credits: WildML RNN Tutorial)

Loss (Error)

Begin with a Vanilla RNN model

- One training example = a sequence of vectors (words)
- n: # of training samples, o: dim of output vector

$$s_t = f(Ux_t + Ws_{t-1} + b) \qquad y_t = \mathrm{softmax}(Vs_t + c)$$

Learning objective: find out the best U, W, V, b, c that minimize the loss (error)

- Sum of squared error (SSE) (or use MSE)

$$L = \frac{1}{2}\sum_{i=1}^{n}\sum_{t=1}^{T}(d_{it} - y_{it})^2$$

- d_{it}/y_{it}: GT/prediction at time step t of training data i

Loss (Error)

Cross-entropy loss: often a better choice

$$L = -\sum_{i=1}^{n}\sum_{t=1}^{T} d_{it} \ln y_{it}$$

- Only consider how well the model predicts the true label
- e.g. for 5-class classification, suppose a label $d_{it} = [1\ 0\ 0\ 0\ 0]$,
- If a prediction is $y_{it} = [0.7\ 0.2\ 0\ 0\ 0.1]$, $L = -\ln 0.7 = 0.3566$
- If $y_{it} = [0.1\ 0.2\ 0\ 0\ 0.7]$, $L = -\ln 0.1 = 1$
- If $y_{it} = [0\ 0.3\ 0\ 0\ 0.7]$, $L = -\ln 0 = \infty$ A very big number
- If $y_{it} = [1.0\ 0\ 0\ 0\ 0]$, $L = -\ln 1 = 0$ A very small number

Loss (Error)

Which one do we have to use?

- Nether is better… depends on the task
- Generally, if the target is continuous and normally distributed (e.g. regression), use MSE
- If the target is discrete and multinomially distributed (e.g. classification), use cross-entropy loss
- Why? Think of maximizing the likelihood

Gradient Descent

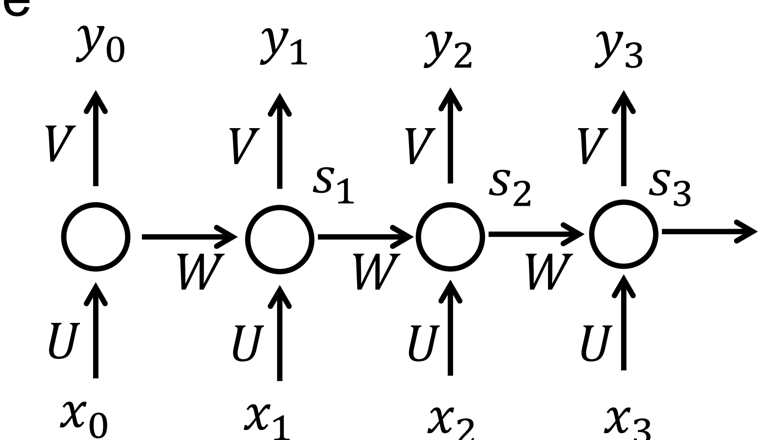
Vanilla RNN model

$$s_t = f(Ux_t + Ws_{t-1} + b) \qquad y_t = \mathrm{softmax}(Vs_t + c)$$

Gradient for one training example

$$\frac{\partial L}{\partial W} = \sum_t \frac{\partial L_t}{\partial W} \quad \text{where } L_t = \mathrm{loss}(y_t, \mathrm{GT}_t)$$

- Sum up the gradients at each time step for one training example



Gradient Descent

Vanilla RNN model

$$s_t = f(Ux_t + Ws_{t-1} + b) \qquad y_t = \mathrm{softmax}(Vs_t + c)$$

$$L_t = \mathrm{loss}(y_t, \mathrm{GT}_t)$$

Let's take an example of L_3 at $t = 3$

- Gradient for V

Derivative of softmax

$$\frac{\partial L_3}{\partial V} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial V} = \boxed{\frac{\partial L_3}{\partial y_3}}\boxed{\frac{\partial y_3}{\partial z_3}}\boxed{\frac{\partial z_3}{\partial V}}$$

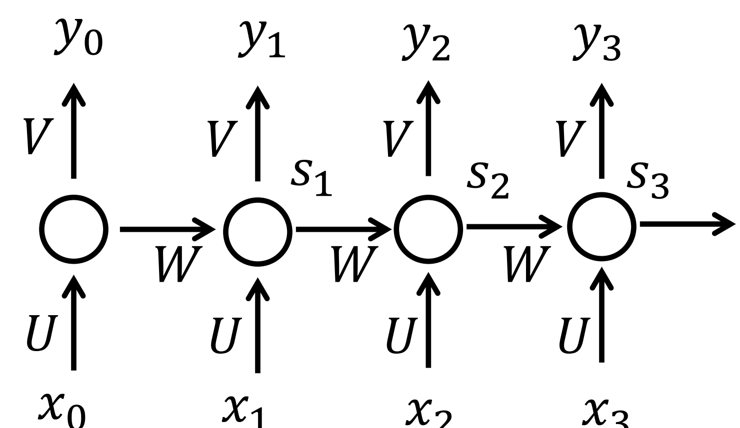
Derivative of loss function

s_3

- e.g. cross-entropy+softmax

$$= -(d_3 - y_3) \otimes s_3$$

cf. outer product: $u \otimes v = uv^T$



Gradient Descent

Vanilla RNN model

$$s_t = f(Ux_t + Ws_{t-1} + b) \qquad y_t = \mathrm{softmax}(Vs_t + c)$$

$$L_t = \mathrm{loss}(y_t, \mathrm{GT}_t)$$

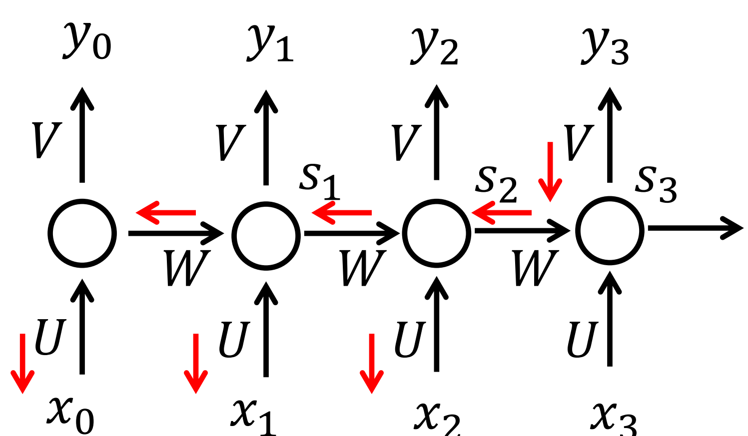
Let's take an example of L_3 at $t = 3$

- Gradient for W

$$\frac{\partial L_3}{\partial W} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial s_3}\boxed{\frac{\partial s_3}{\partial W}} \quad \left(\frac{\partial y_3}{\partial s_3} = \frac{\partial y_3}{\partial z_3}\frac{\partial z_3}{\partial s_3}\right)$$

However, the function of s_t involves s_{t-1}!

(Cannot simply treat as a constant because W is used in every step)



Gradient Descent

Vanilla RNN model

$$s_t = f(Ux_t + Ws_{t-1} + b) \qquad y_t = \mathrm{softmax}(Vs_t + c)$$

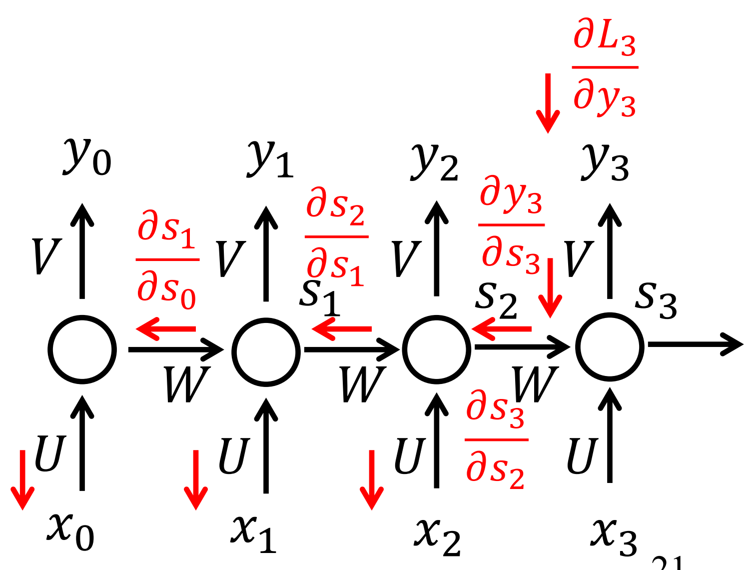
$$L_t = \mathrm{loss}(y_t, \mathrm{GT}_t)$$

Let's take an example of L_3 at $t = 3$

- Gradient for W

$$\frac{\partial L_3}{\partial W} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial s_3}\frac{\partial s_3}{\partial W}$$

$$= \sum_{k=0}^{3} \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$



BPTT Algorithm

BackPropagation Through Time

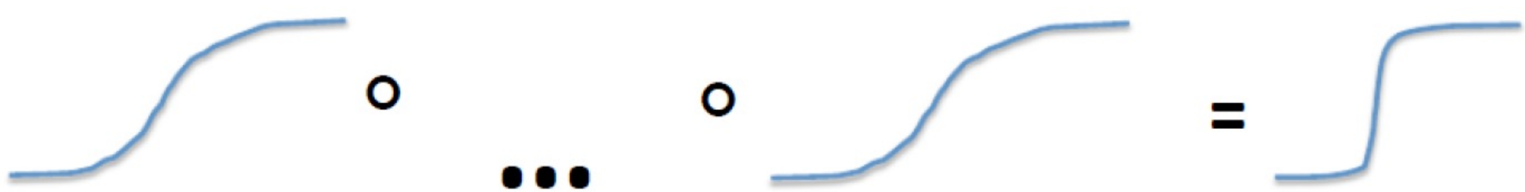
- One of the methods used to train RNNs
- The unfolded network is treated as one big feed-forward network
- This big network takes in entire sequence as an input
- Compute gradients through the usual backpropagation
- Update shared weights

Vanishing and (Exploding) Gradient Problem

RNNs have difficulties learning long-range dependencies

In very deep nets and recurrent nets, the final output is composed of many non-linear transformations

- Although each non-linear state may be smooth enough, their composition is going to be very steep



- Multiplying the same number multiple times tends to be either very large or very small
- Gradients become either very small or very large

[Xiaogang Wang]

Vanishing and (Exploding) Gradient Problem

Take a look at previous gradient

$$\frac{\partial L_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$

$$= \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial s_3}\frac{\partial s_3}{\partial W} + \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial W} + \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial s_3}\frac{\partial s_3}{\partial s_1}\frac{\partial s_1}{\partial W} + \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial s_3}\frac{\partial s_3}{\partial s_0}\frac{\partial s_0}{\partial W}$$

- Involve a chain rule
- e.g. for a term with $k = 1$, $\frac{\partial s_3}{\partial s_1} = \frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}$

$$= \sum_{k=0}^{3} \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial s_3}\left(\prod_{j=k+1}^{3} \frac{\partial s_3}{\partial s_j}\right)\frac{\partial s_k}{\partial W}$$

- We may need to multiply the derivative of f multiple times

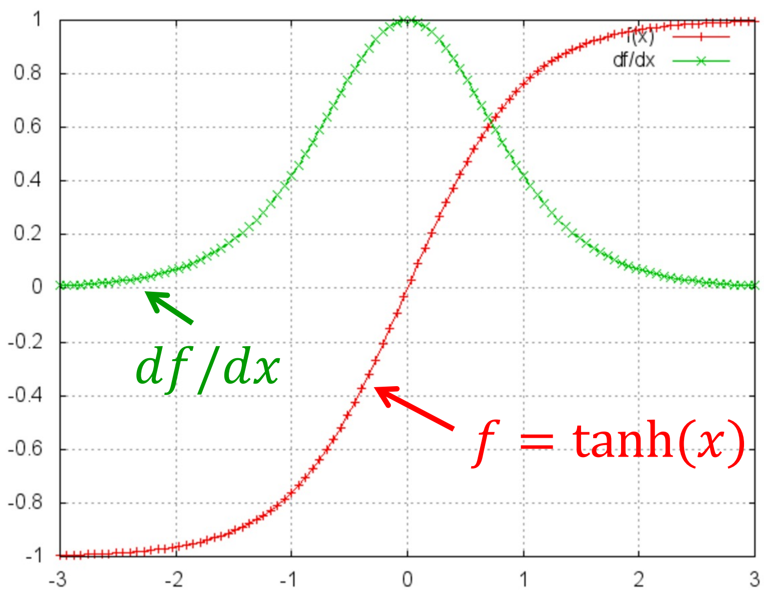
$s_t = f(Ux_t + Ws_{t-1} + b)$

Vanishing and (Exploding) Gradient Problem

Take a look at previous gradient

$$\frac{\partial C_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial C_3}{\partial y_3} \frac{\partial y_3}{\partial s_3} \left(\prod_{j=k+1}^{3} \frac{\partial s_3}{\partial s_j} \right) \frac{\partial s_k}{\partial W}$$

- Ex. $\tanh$ function



- $\tanh$ and sigmoid functions have derivatives of 0 at both ends (i.e. a flat line)
- Gradient contributions from *far away* steps become 0, and the state at those steps doesn't contribute

Vanishing and (Exploding) Gradient Problem

Exploding gradient is also problematic but get less attention

- Easy to detection (e.g. finding NaN)
- Easy to remedy (e.g. using *clipping* by a pre-defined threshold)

Use Long Short-Term Memory (LSTM)

- Same architecture with RNN but different functions for hidden states
- Design a cell and gates to keep in memory

Transformers are popular these days!

- e.g. BERT, GPT-3 to name a few

Outline

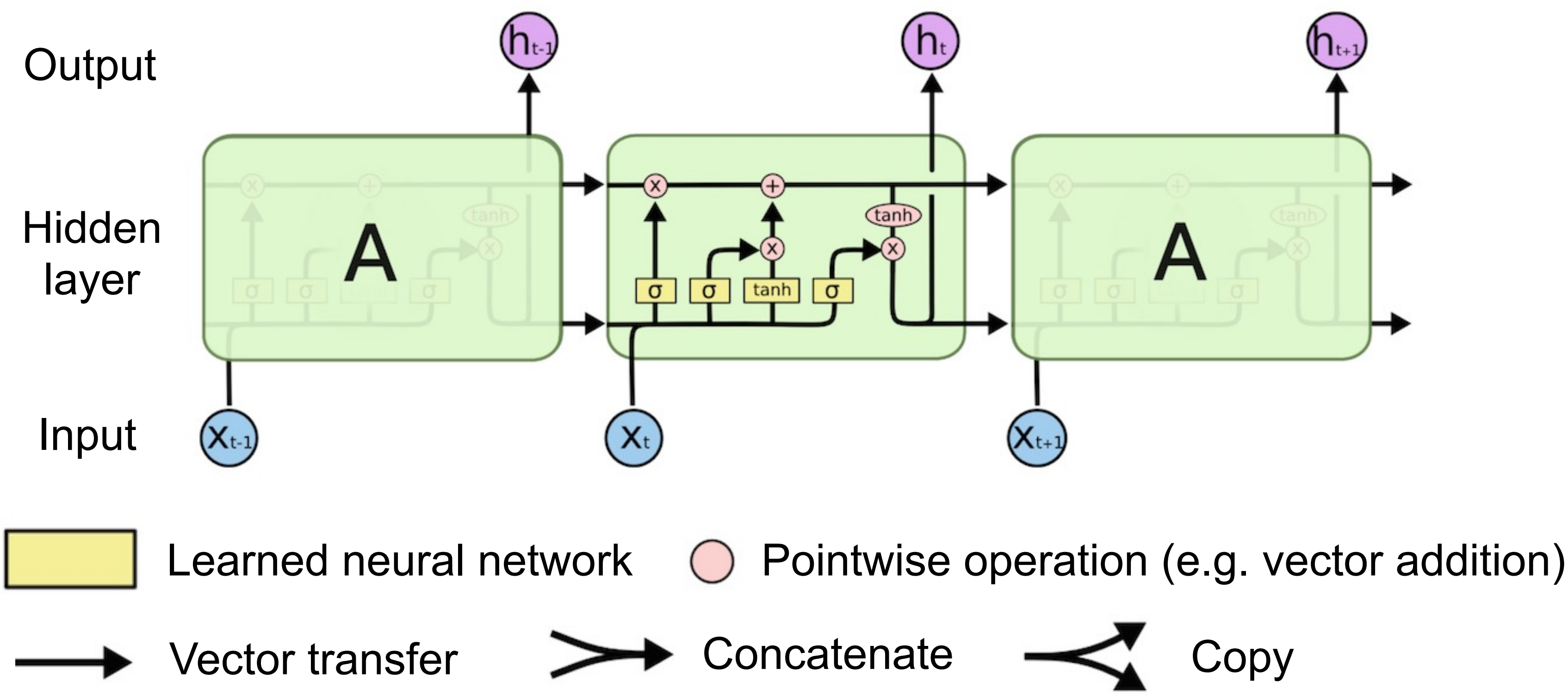
- Recurrent Neural networks
- Backpropagation Through Time
- LSTM Recurrent Networks

(Credits: colah's blog)

LSTM Networks

Designed to avoid the long-term dependency problem

- How? We will see it later
- Have the same chain structure with RNNs, but the repeating hidden module has different structure

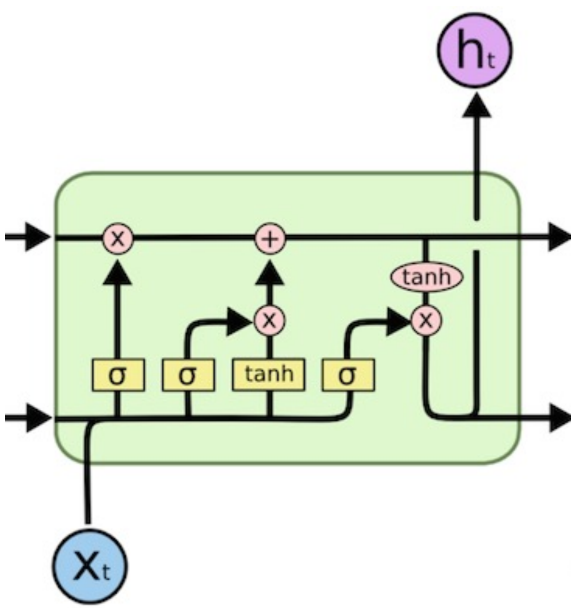


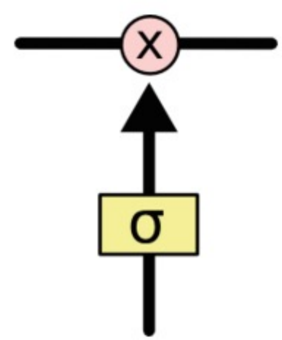
S. Hochreiter & H. Schmidhuber, Long Short-term Memory, Neural Computation, 1997

LSTM Networks

A cell and three gates in an LSTM unit

- Each gate protects and controls the cell state
- Each gate regulates the information flow
- Consists of a sigmoid (or tanh) layer and pointwise multiplication
- Sigmoid output ranges [0, 1] (turn off/on)



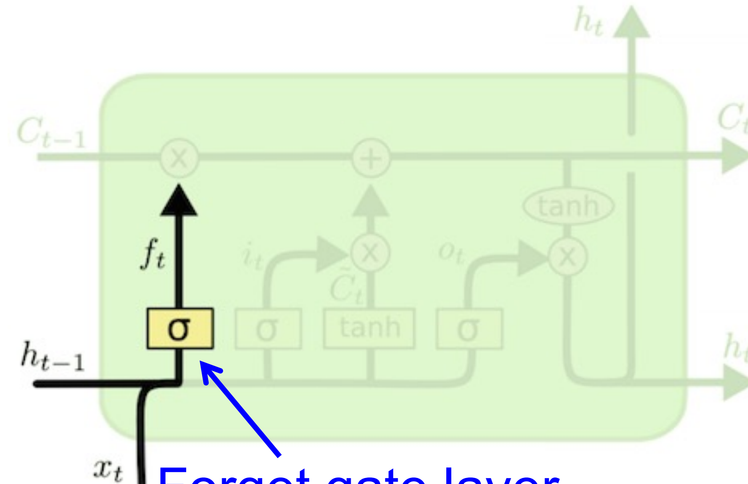


Step-by-Step LSTM Walk Through

First step

- Look at previous output h_{t-1} and current input x_t
- Decide whether previous cell state C_{t-1} is forgotten or not

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



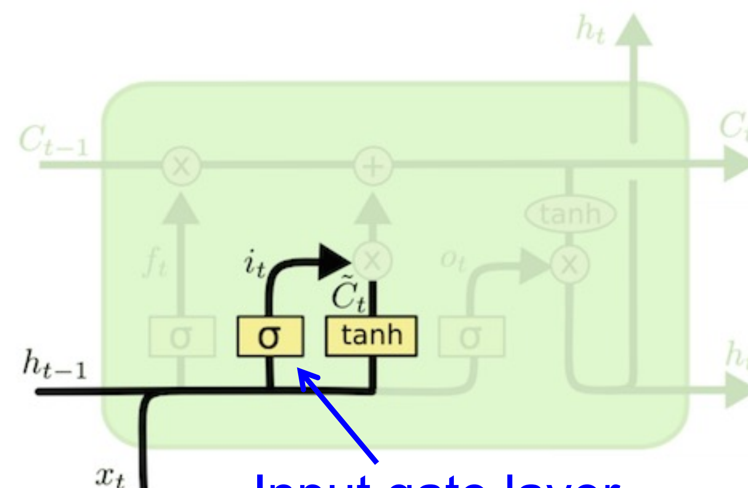
Forget gate layer

Second step

- The sigmoid layer decides which values are updated
- The tanh layer creates new values to be added to the state

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

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



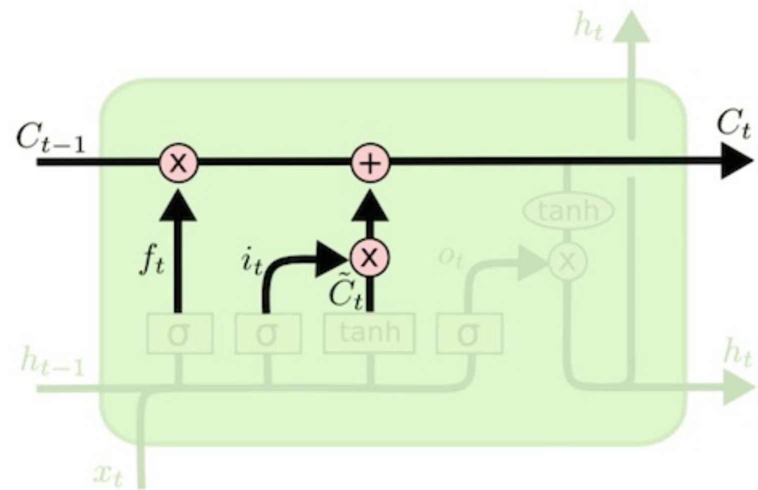
Input gate layer

Step-by-Step LSTM Walk Through

Third step

- Old state C_{t-1} multiplied by f_t (forgetting factor)
- New candidate values multiplied by i_t (update factor)

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

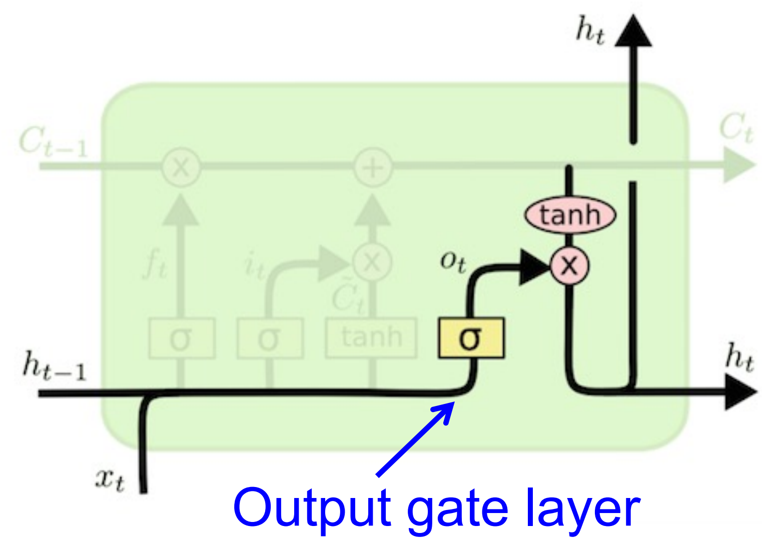


Fourth step

- The sigmoid works like a filter
- New cell state C_t goes through $\tanh$ (push values within [-1,1])

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

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



Summary of LSTM Networks

LSTM cell can be defined with following equations

- Gates (input / forget / output)

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

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

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

- Input transform

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

- State update

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C_t}$$

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

How LSTM Handles Long-Term Dependency

Networks can retain/discard the information for long time

- We can regulate f_t, i_t to control how much previous cell and new input information

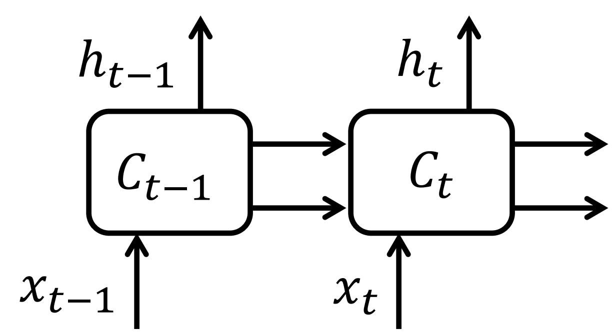
$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t$$

Prevent vanishing/exploding gradient problem

- Suppose the error at t is $\dfrac{\partial L}{\partial C_t}$
- By chain rule,

$$\frac{\partial L}{\partial C_{t-1}} = \frac{\partial L}{\partial C_t}\frac{\partial C_t}{\partial C_{t-1}} = f_t * \frac{\partial L}{\partial C_t}$$

$$\frac{\partial L}{\partial C_{t-1}} = \frac{\partial L}{\partial C_{t-1}} + f_t * \frac{\partial L}{\partial C_t}$$



- If $f_t \sim 1$, it is not vanishing; since $f_t < 1$, it is not exploding

Variants on LSTM

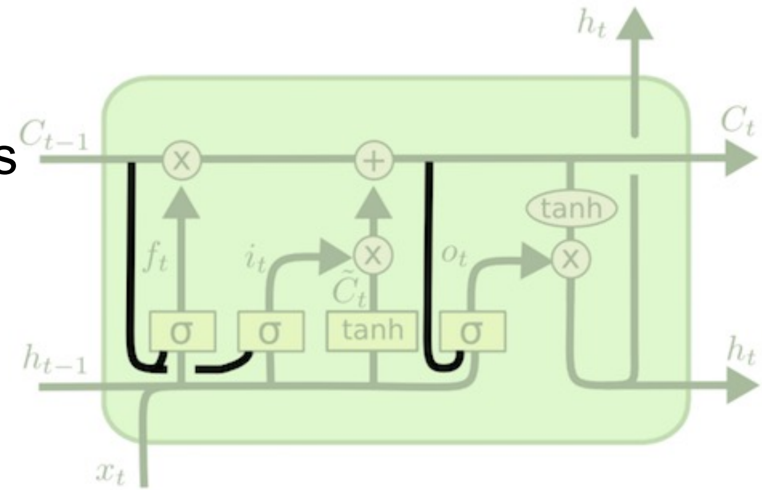
Add peepholes to all the gates

- Let all gates look at the cell states

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

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

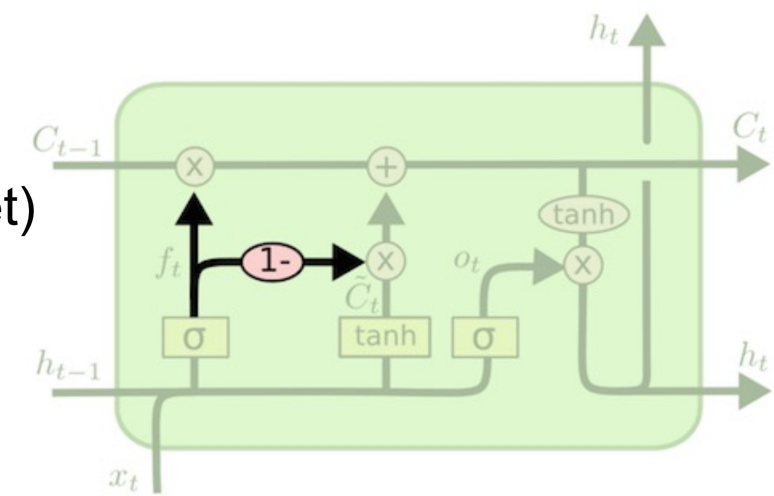
$$o_t = \sigma(W_o \cdot [C_{t-1}, h_{t-1}, x_t] + b_o)$$



Couple forget and input gates

- Replace input gate by (1 – forget)

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



F. A. Gers & H. Schmidhuber, Recurrent Nets that Time and Count, IJCNN 2000

Variants on LSTM

GRU (Gated Recurrent Unit)

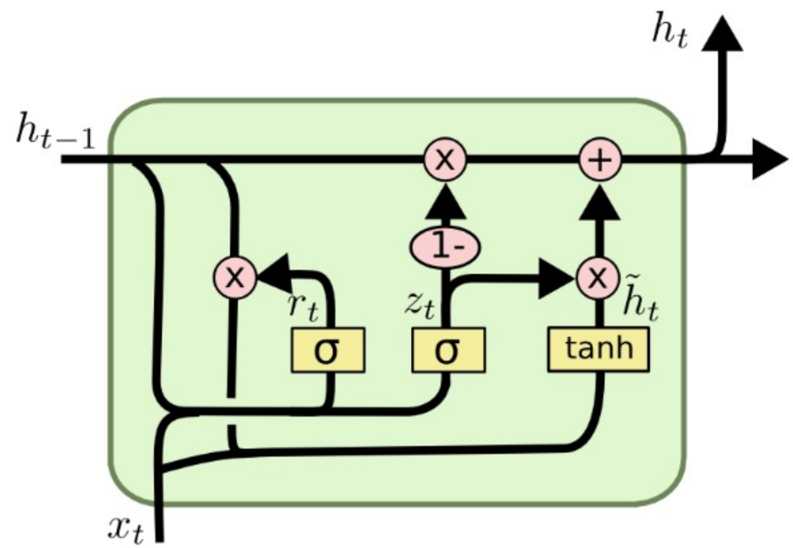
- Much simpler than LSTM
- Merges the cell state and hidden state
- Combines the forget and input gates into a single *update gate*
- And other minor modifications

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

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

$$\widetilde{h_t} = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \widetilde{h_t}$$

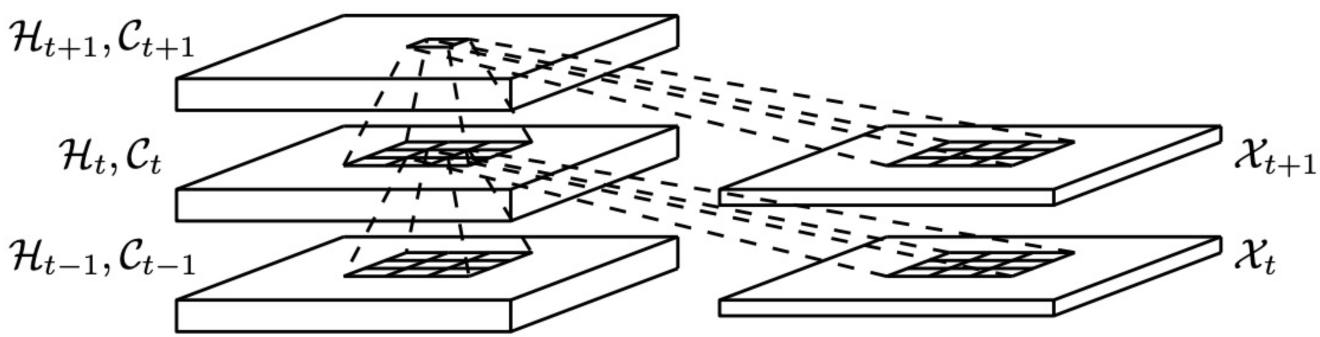


K. Cho et al., Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation, EMNLP 2014

CNN + RNN

ConvLSTM

- A type of RNN for spatio-temporal prediction
- Use convolutional structure inside the functions (in both input-to-state and state-to-state transitions)
- When using input and state, apply Conv to capture its local neighbor



X. Shi et al., Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting, NeurIPS 2015

CNN + RNN

ConvLSTM

- LSTM equations are changed as follows
- $*$: convolution, $\odot$: Hadamard (element-wise) product

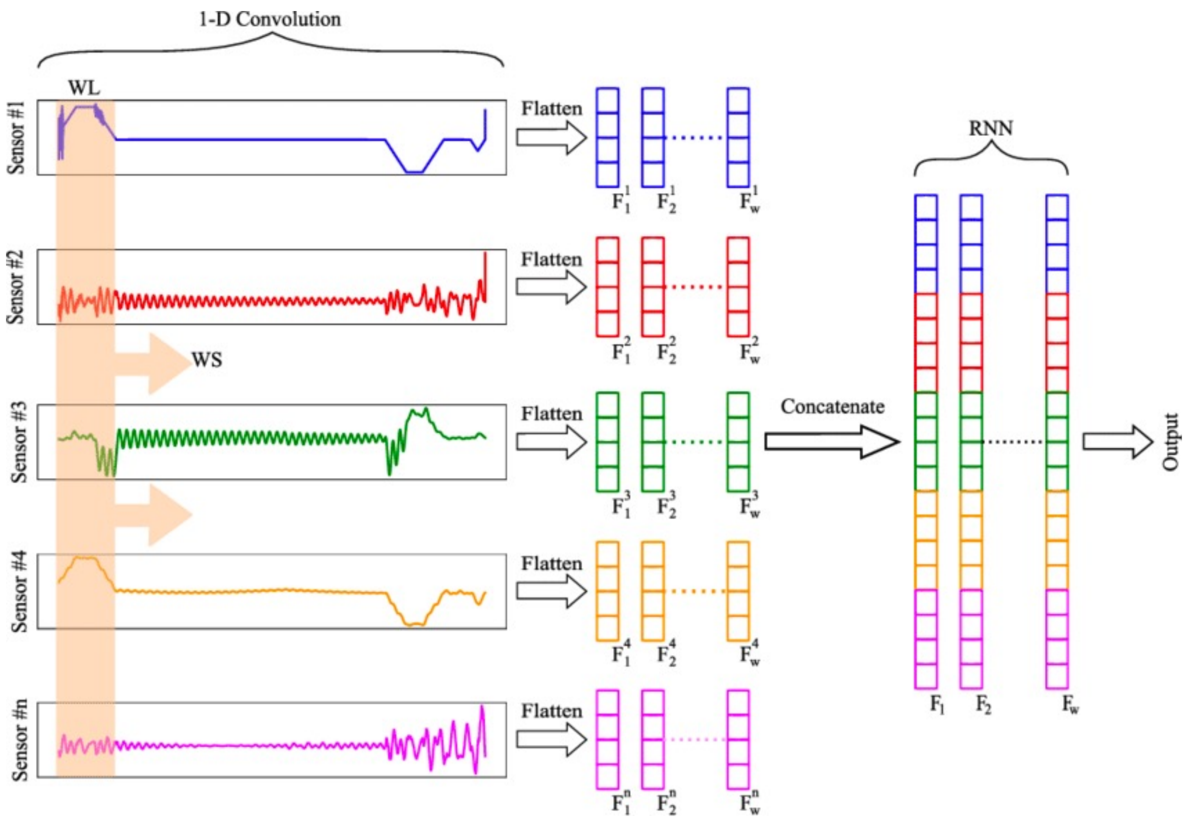
LSTM		ConvLSTM
$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$		$i_t = \sigma(W_{xi} * x_t + W_{hi} * h_{t-1} + W_{ci} \odot C_{t-1} + b_i)$
$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$		$f_t = \sigma(W_{xf} * x_t + W_{hf} * h_{t-1} + W_{cf} \odot C_{t-1} + b_f)$
$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$	$\Rightarrow$	$o_t = \sigma(W_{xo} * x_t + W_{ho} * h_{t-1} + W_{co} \odot C_t + b_o)$
$\widetilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$		$\widetilde{C}_t = \tanh(W_{xC} * x_t + W_{hC} * h_{t-1} + b_c)$
$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t$		$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t$
$h_t = o_t * \tanh(C_t)$		$h_t = o_t * \tanh(C_t)$

X. Shi et al., Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting, NeurIPS 2015

CNN + RNN

Many variants on CNN-RNN models

- First apply CNN to obtain representation and then apply RNN for prediction



- Each CNN head independently processes a time series from a sensor
- The feature maps are concatenated and used as an input to RNN

M. Canizo et al., Multi-Head CNN-RNN for Multi-Time Series Anomaly Detection: An industrial case study, Neurocomputing 2019